Chapter 1 Introduction to System Programming

> "UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity." - Dennis Ritchie, 1941 - 2011.

Concepts Covered

The kernel and kernel API,
System calls and libraries,
Processes, logins and shells,
Environments, man pages,
Users, the root, and groups,
Authentication,
File system, file hierarchy,
Files and directories,
Device special files,
UNIX standards, POSIX,
System programming,
Terminals and ANSI escape sequences,
History of UNIX,
syscall, getpid, ioctl

1.1 Introduction

A modern software application typically needs to manage both private and system resources. Private resources are its own data, such as the values of its internal data structures. System resources are things such as files, screen displays, and network connections. An application may also be written as a collection of cooperating threads or sub-processes that coordinate their actions with respect to shared data. These threads and sub-processes are also system resources.

Modern operating systems prevent application software from managing system resources directly, instead providing interfaces that these applications can use for managing such resources. For example, when running on modern operating systems, applications cannot draw to the screen directly or read or write files directly. To perform screen operations or file I/O they must use the interface that the operating system defines. Although it may seem that functions from the C standard library such as `getc()` or `fprintf()` access files directly, they do not; they make calls to system routines that do the work on their behalf.

The interface provided by an operating system for applications to use when accessing system resources is called the operating system's *application programming interface* (*API*). An API typically consists of a collection of function, type, and constant definitions, and sometimes variable definitions as well. The API of an operating system in effect defines the means by which an application can utilize the services provided by that operating system.

It follows that developing a software application for any *platform*[1] requires mastery of that platform's API. Therefore, aside from designing the application itself, the most important task for the application developer is to master the system level services defined in the operating system's API. A program that uses these system level services directly is called a *system program*, and the type of programming that uses these services is called *system programming*. System programs make requests for resources and services directly from the operating system and may even access the system

[1] We use the term platform to mean a specific operating system running on a specific machine architecture.

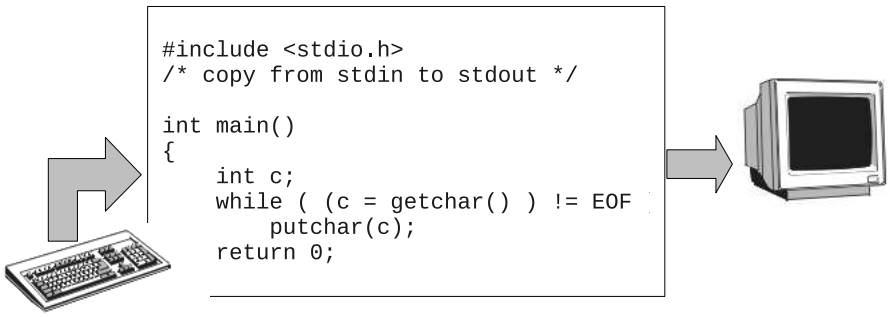


Figure 1.1: Simple I/O model used by beginning programmer.

resources directly. System programs can sometimes be written to extend the functionality of the operating system itself and provide functions that higher level applications can use.

These lecture notes specifically concern system programming using the API of the UNIX operating system. They do not require any prior programming experience with UNIX. They also include tutorial information for those readers who have little experience with UNIX as a user, but this material can be skipped by the experienced UNIX users.

In the remainder of these notes, a distinction will be made between the user's view of UNIX and the *programmer's view* of UNIX. The user's view of UNIX is limited to a subset of commands that can be entered at the command-line and parts of the file system. Some commands and files are not available to all users, as will be explained later. The programmer's view includes the programming language features of the kernel API, the functions, types, and constants in all of the libraries, the various header files, and the various files used by the system. Familiarity with basic C programming is assumed.

1.2 A Programming Illusion

A beginning programmer typically writes programs that follow the simple I/O model depicted in Figure 1.1: the program gets its input from the keyboard or a disk file, and writes its output to the display screen or to a file on disk. Such programs are called *console applications*. because the keyboard and display screen are part of the console device. Listings 1.1 and 1.2 contain examples of such a program, one using the C Standard I/O Library, and the other, the C++ stream library. Both get input from the keyboard and send output to the display device, which is some sort of a console window on a monitor.

The comment in Listing1.1 states that the program copies from stdin to stdout. In UNIX[2], every process has access to abstractions called the standard input device and the standard output device. When a process is created and loaded into memory, UNIX automatically creates the standard input and standard output devices for it, opens them, and makes them ready for reading and writing respectively[3]. In C (and C++), `stdin` and `stdout` are variables defined in the `<stdio.h>`

[2]In fact, every POSIX-compliant operating system must provide both a standard input and standard output stream.

[3]It also creates a standard error device that defaults to the same device as standard output.

header file, that refer to the standard input and standard output device[4] respectively. By default, the keyboard and display of the associated terminal are the standard input and output devices respectively.

Listing 1.1: C program using simple I/O model.

```
#include <stdio.h>
/* copy from stdin to stdout */
int main()
{
    int c;
    while ( (c = getchar() ) != EOF )
        putchar(c);
    return 0;
}
```

Listing 1.2: Simple C++ program using simple I/O model.

```
#include <iostream>
using namespace std;
/* copy from stdin to stdout using C++ */
int main()
{
    char c;
    while ( (c = cin.get() ) && !cin.eof() )
        cout.put(c);
    return 0;
}
```

These programs give us the illusion that they are directly connected to the keyboard and the display device via C library functions `getchar()` and `putchar()` and the C++ `iostream` member functions `get()` and `put()`. Either of them can be run on a single-user desktop computer or on a multi-user, time-shared workstation in a terminal window, and the results will be the same. If you build and run them as console applications in Windows, they will have the same behavior as if you built and ran them from the command-line in a UNIX system.

On a personal computer running in single-user mode, this illusion is not far from reality in the sense that the keyboard is indirectly connected to the input stream of the program, and the monitor is indirectly connected to the output stream. This is not the case in a multi-user system.

In a multi-user operating system, several users may be logged in simultaneously, and programs belonging to different users might be running at the same time, each receiving input from a different keyboard and sending output to a different display. For example, on a UNIX computer on a network into which you can login, many people may be connected to a single computer via a network program such as SSH, and several of them will be able to run the above program on the same computer at the same time, sending their output to different terminal windows on physically different computers, and each will see the same output as if they had run the program on a single-user machine.

As depicted in Figure 1.2, UNIX ensures, in a remarkably elegant manner, that each user's *processes* have a logical connection to their keyboard and their display. (The process concept will be explained

[4] In C and C++, stderr is the variable associated with the standard error device.

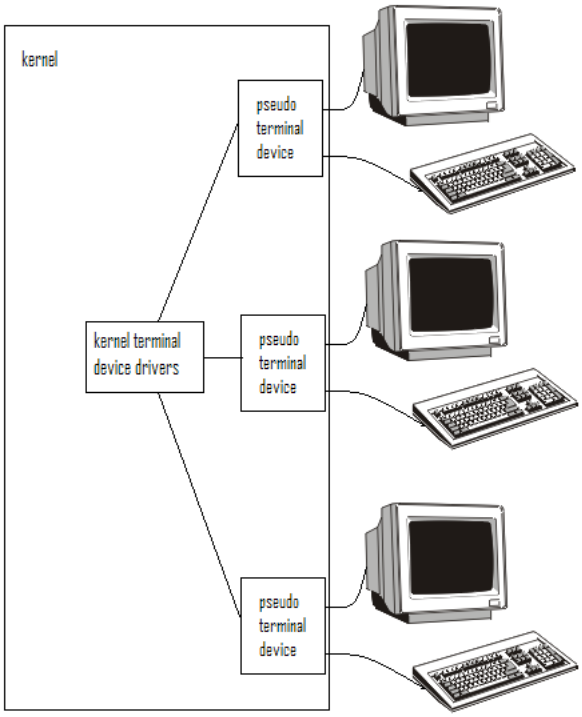


Figure 1.2: Connecting multiple users to a UNIX system.

shortly.) Programs that use the model of I/O described above do not have to be concerned with the complexities of connecting to monitors and keyboards, because the operating system hides that complexity, presenting a simplified interface for dealing with I/O. To understand how the operating system achieves this, one must first understand several cornerstone concepts of the UNIX operating system: files, processes, users and groups, privileges and protections, and environments.

1.3 Cornerstones of UNIX

From its beginning, UNIX was designed around a small set of clever ideas, as Ritchie and Thompson [2] put it:

> "The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system."

Those "fertile ideas" included the design of its file system, its process concept, the concept of privileged and unprivileged programs, the concepts of user and groups, a programmable shell, environments, and device independent input and output. In this section we describe each of these briefly.

1.3.1 Files and the File Hierarchy

Most people who have used computers know what a file is, but as an exercise, try explaining what a file is to your oldest living relative. You may know what it is, but knowing how to define it is another matter. In UNIX, the traditional definition of a file was that "it is the *smallest unit of external storage*." "External storage" has always meant non-volatile storage, not in primary memory, but on media such as magnetic, optical, and electronic disks, tapes and so on. (Internal storage is on memory chips.) The contemporary definition of a file in UNIX is that it is an object that can be written to, or read from, or both. There is no requirement that it must reside on external storage. We will use this definition of a file in the remainder of these notes.

UNIX organizes files into a tree-like hierarchy that most people erroneously call the *file system*. It is more accurately called the *file hierarchy*, because a file system is something slightly different. The internal nodes of the hierarchy are called *directories*. Directories are special types of files that, from the user perspective, appear to contain other files, although they do not contain files any more than a table of contents in a book contains the chapters of the book themselves. To be precise, a directory is a file that contains *directory entries*. A directory entry is an object that associates a *filename* to a file[5]. Filenames are not the same things as files. The root of the UNIX file system is a directory known in the UNIX world as the *root directory*, however it is not named "`root`" in the file system; it is named "`/`". When you need to refer to this directory, you call it "root", not "slash". More will be said about files, filenames, and the file hierarchy in Section 1.8.

1.3.2 Processes

A *program* is an executable file, and a *process* is an instance of a running program. When a program is run on a computer, it is given various resources such as a primary memory space, both physical and logical, secondary storage space, mappings of various kinds[6], and privileges, such as the right to read or write certain files or devices. As a result, at any instant of time, associated to a process is the collection of all resources allocated to the running program, as well as any other properties and settings that characterize that process, such as the values of the processor's registers. Thus, although the idea of a process sounds like an abstract idea, it is, in fact, a very concrete thing.

UNIX assigns to each process a unique number called its *process-id* or *pid*. For example, at a given instant of time, several people might all be running the Gnu C compiler, `gcc`. Each separate execution instance of `gcc` is a process with its own unique pid. The `ps` command can be used to display which processes are running, and various options to it control what it outputs.

At the programming level, the function `getpid()` returns the process-id of the process that invokes it. The program in Listing 1.3 does nothing other than printing its own process-id, but it illustrates how to use it. Shortly we will see that `getpid()` is an example of a *system call*.

Listing 1.3: A program using `getpid()`.

```
#include <stdio.h>
#include <unistd.h>
int main()
```

[5] In practice a directory entry is an object with two components: the name of a file and a pointer to a structure that contains the attributes of that file.

[6] For example, a map of how its logical addresses map to physical addresses, and a map of where the pieces of its logical address space reside on secondary storage.

```
{
    printf("I am the process with process-id %d\n",
            getpid());

    return 0;
}
```

1.3.3 Users and Groups

Part of the security of UNIX rests on the principle that every user of the system must be *authenticated*. Authentication is a form of security clearance, like passing through a metal detector at an airport.

In UNIX, a *user* is a person[7] who is authorized to use the system. The only way to use a UNIX system is to log into it[8]. UNIX maintains a list of names of users who are allowed to login[9]. These names are called *user-names*. Associated with each user-name is a unique non-negative number called the *user-id*, or *uid* for short. Each user also has an associated password. UNIX uses the user-name/password pair to authenticate a user attempting to login. If that pair is not in its list, the user is rejected. Passwords are stored in an encrypted form in the system's files.

A *group* is a set of users. Just as each user has a user-id, each group has unique integer *group-id*, or *gid* for short. UNIX uses groups to provide a form of resource sharing. For example, a file can be associated with a group, and all users in that group would have the same access rights to that file. Since a program is just an executable file, the same is true of programs; an executable program can be associated with a group so that all members of that group will have the same right to run that program. Every user belongs to at least one group, called the primary group of that user. The `id` command can be used to print the user's user-id and user-name, and the group-id and group-name of all groups to which the user belongs. The `groups` command prints the list of groups to which the user belongs.

In UNIX, there is a distinguished user called the *superuser*, whose user-name is root, one of a few predefined user-names in all UNIX systems. The superuser has the ability to do things that ordinary users cannot do, such as changing a person's user-name or modifying the operating system's configuration. Being able to login as `root` in UNIX confers absolute power to a person over that system. For this reason, all UNIX systems record every attempt to login as `root`, so that the system administrator can monitor and catch break-in attempts.

Every process has an associated (real) user-id and, as we will see later, an effective user-id that might be different from the real user-id. In simplest case, when a user starts up a program, the resulting process has that user's uid as both its real and effective uid. The privileges of the process are the same as those of the user[10]. When the superuser (root) runs a process, that process runs

[7] A user may not be an actual person. It can also be an abstraction of a person. For example, `mail`, `lp`, and `ftp` are each users in a UNIX system, but they are actually programs.

[8] To "login" to a system is to "log into" it. Remember that logging means recording something in a logbook, as a sea captain does. The term "login" conveys the idea that the act is being recorded in a logbook. In UNIX, logins are recorded in a special file that acts like a logbook.

[9] We take this word for granted. We use "login" as a single word only because it has become a single word on millions of "login screens" around the world. To login, as a verb, really means "to log into" something; it requires an indirect object.

[10] To be precise, the privileges are those of user with the process's effective user-id.

with the superuser's privileges. Processes running with user privileges are called user processes. At the programming level, the function `getuid()` returns the real user-id of the process that calls it, and the `getgid()` function returns the real group-id of the process that calls it.

1.3.4 Privileged and Non-Privileged Instructions

In order to prevent ordinary user processes from accessing hardware and performing other operations that may corrupt the state of the computer system, UNIX requires that the processor support two modes of operation, known as *privileged* and *unprivileged* mode[11]. Privileged instructions are those that can alter system resources, directly or indirectly. Examples of privileged instructions include:

- acquiring more memory;
- changing the system time;
- raising the priority of the running process;
- reading from or writing to the disk;
- entering privileged mode.

Only the operating system is allowed to execute privileged instructions. User processes can execute only the unprivileged instructions. The security and reliability of the operating system depend upon this separation of powers.

1.3.5 Environments

When a program is run in UNIX, one of the steps that the operating system takes prior to running the program is to make available to that program an array of name-value pairs called the *environment*. Each name-value pair is a string of the form

```
name=value
```

where `value` is a `NULL`-terminated C string. The `name` is called an *environment variable* and the pair `name=value` is called an *environment string*. The variables by convention contain only uppercase letters, digits, and underscores, but this is not required[12]. The only requirement is that the name does not contain the "=" character. For example, `LOGNAME` is an environment variable that stores the user-name of the current user, and `COLUMNS` is a variable that stores the number of columns in the current console window[13]. Even though it is a number, it is stored as a string.

When a user logs into a UNIX system, the operating system creates the environment for the user, based on various files in the system. From that point forward, whenever a new program runs, it is given a copy of that environment. This will be explained in greater depth later.

[11] These modes are also known as *supervisor mode* and *user mode*.

[12] Environment variable names used by the utilities in the Shell and Utilities volume of POSIX.1-2008 consist solely of uppercase letters, digits, and the underscore ('_') and do not begin with a digit.

[13] If the user defines a value for `COLUMNS` in a start-up script, then terminal windows will have that many columns. If the user does not define it, or sets it to the `NULL` string, the size of terminal windows is determined by the operating system software.

The `printenv` command displays the values of all environment variables as does the `env` command. Within a program the `getenv()` function can be used to retrieve a particular environment string, as in

```
char* username = getenv("LOGNAME");
printf("The user's user-name is %s\n, username);
```

The operating system also makes available to every running program an external global variable

```
extern char **environ;
```

which is a pointer to the start of the array of the name-value pairs in the running program's environment. Programs can read and modify these variables if they choose. For example, a program that needs to know how many columns are in the current terminal window will query the `COLUMNS` variable, whereas other programs may just ignore it.

1.3.6 Shells

The kernel provides services to processes, not to users; users interact with UNIX through a command-line interface called a *shell*. The word "shell" is the UNIX term for a particular type of *command-line-interpreter*. Command-line interpreters have been in operating systems since they were first created. *DOS* uses a command-line-interpreter, as is the *Command* window of Microsoft Windows, which is simply a DOS emulator. The way that DOS and the Command window are used is similar to the way that UNIX is used[14]: you type a command and press the *Enter* key, and the command is executed, after which the prompt reappears. The program that displays the prompt, reads the input you type, runs the appropriate programs in response and re-displays the prompt is the command-line-interpreter, which in UNIX is called a shell.

In UNIX, a shell is much more than a command-line-interpreter. It can do more than just read simple commands and execute them. A shell is also programming language interpreter; it allows the user to define variables, evaluate expressions, use conditional control-of-flow statements such as `while-` and `if`-statements, and make calls to other programs. A sequence of shell commands can be saved into a file and executed as a program by typing the name of the file. Such a sequence of shell commands is called a *shell script*. When a shell script is run, the operating system starts up a shell process to read the instructions and execute them.

1.3.7 Online Documentation: The Man Pages

Shortly after Ritchie and Thompson wrote the first version of UNIX, at the insistence of their manager, Doug McIlroy, in 1971, they wrote the *UNIX Programmer's Manual*. This manual was initially a single volume, but in short course it was extended into a set of seven volumes, organized by topic. It existed in both printed form and as formatted files for display on an ordinary character display device. Over time it grew in size. Every UNIX distribution comes with this set of manual pages, called "manpages" for short. Appendix B.4 contains a brief description of the structure of the manpages, and Chapter 2 provides more detail about how to use them.

[14]This is not a coincidence. Long before Microsoft wrote MS-DOS, they wrote a version of UNIX for the PC called *Xenix*, whose rights they sold to Santa Cruz Operations in 1987

1.4 The UNIX Kernel API

A multi-user operating system such as UNIX must manage and protect all of the system's resources and provide an operating environment that allows all users to work efficiently, safely, and happily. It must prevent users and the processes that they invoke from accessing any hardware resources directly. In other words, if a user's process wants to read from or write to a disk, it must ask the operating system to do this on its behalf, rather than doing it on its own. The operating system will perform the task and transfer any data to or from the user's process. To see why this is necessary, consider what would happen if a user's process could access the hard disk directly. A user could write a program that tried to acquire all disk space, or even worse, tried to erase the disk.

A program such as the ones in Listings 1.1 and 1.2, may look like it does not ask the operating system to read or write any data, but that is not true. Both `getchar()` and `putchar()`, are library functions in the C Standard I/O Library (whose header file is `<stdio.h>`), and they do, in fact, "ask" the operating system to do the work for the calling program. The details will be explained later, but take it on faith that one way or another, the operating system has intervened in this task.

The operating system must protect users from each other and protect itself from users. However, while providing an operating environment for all users, a multi-user operating system gives each user the impression that he or she has the computer entirely to him or herself. This is precisely the illusion underlying the execution of the program in Figure 1.1. Somehow everyone is able to write programs that look like they have the computer all to themselves, and run them as if no one else is using the machine. The operating system creates this illusion by creating data paths between user processes and devices and files. The data paths connect user processes and devices in the part of memory reserved for the operating system itself. And that is the first clue – physical memory is divided into two regions, one in which ordinary user programs are loaded, called *user space*, and one where the operating system itself is stored, called *system space*.

How does UNIX create this illusion? We begin with a superficial answer, and gradually add details in later chapters.

The UNIX operating system is called the *kernel*. The kernel defines the application programming interface and provides all of UNIX's services, whether directly or indirectly. The kernel is a program, or a collection of interacting programs, depending on the particular implementation of UNIX, with many *entry points*[15]. Each of these entry points provides a service that the kernel performs. If you are used to thinking of programs as always starting at their first line, this may be disconcerting. The UNIX kernel, like many other programs, can be entered at other points. You can think of these entry points as functions that can be called by other programs. These functions do things such as opening, reading, and writing files, creating new processes, allocating memory, and so on. Each of these functions expects a certain number of arguments of certain types, and produces well-defined results. The collection of kernel entry points makes up a large part of UNIX's API. You can think of the kernel as a collection of separate functions, bundled together into a large package, and its API as the collection of signatures or prototypes of these functions.

When UNIX boots, the kernel is loaded into the portion of memory called system space and stays there until the machine is shut down. User processes are not allowed to access system space. If they do, they are terminated by the kernel.

[15] An entry point is an instruction in a program at which execution can begin. In the programs that you have probably written, there has been a single entry point – `main()` –, but in other programs, you can specify that the code can be entered at any of several entry points. Software libraries are code modules with multiple entries points. In the Windows world, dynamically linked libraries (DLLs) are examples of code modules with multiple entry points.

The kernel has full access to all of the hardware attached to the computer. User programs do not; they interact with the hardware indirectly through the kernel. The kernel maintains various system resources in order to provide these services to user programs. These system resources include many different data structures that keep track of I/O, memory, and device usage for example. In Section 1.4.1 this is explained in more detail.

Summarizing, if a user process needs data from the disk for example, it has to "ask" the kernel to get it. If a user process needs to write to the display, it has to "ask" the kernel to do this too. All processes gain access to devices and resources through the kernel. The kernel uses its resources to provide these services to user processes.

1.4.1 System Resources

The kernel provides many services to user programs, including

- process scheduling and management,
- I/O handling,
- physical and virtual memory management,
- device management,
- file management,
- signaling and inter-process communication,
- multi-threading,
- multi-tasking,
- real-time signaling and scheduling, and
- networking services.

Network services include protocols such as HTTP, NIS, NFS, X.25, SSH, SFTP, TCP/IP, and Java. Exactly which protocols are supported is not important; what is important is for you to understand that the kernel provides the means by which a user program can make requests for these services.

There are two different methods by which a program can make requests for services from the kernel:

- by making a *system call* to a function (i.e., entry point) built directly into the kernel, or
- by calling a higher-level *library routine* that makes use of this call.

Do not confuse either of these with a *system program*. The term "system program" refers to a separate program that is bundled with the kernel, that interfaces to it to achieve its functionality, and that provides higher level services to users. We can browse through the `/bin` or `/usr/bin` directories of a UNIX installation to find many different system programs. Many UNIX commands are implemented by system programs.

1.4.2 System Calls

An ordinary function call is a jump to and return from a subroutine that is part of the code linked into the program making the call, regardless of whether the subroutine is *statically* or *dynamically* linked into the code. A system call is like a conventional function call in that it causes a jump to a subroutine followed by a return to the caller. But it is significantly different because it is a call to a function that is a part of the UNIX kernel.

The code that is executed during the call is actually kernel code. Since the kernel code accesses hardware and contains privileged instructions, kernel code must be run in privileged mode. Because the kernel alone runs in privileged mode, it is also commonly called *kernel mode* or *superuser mode*. Therefore, during a system call, the process that made the call is run in kernel mode. Unlike an ordinary function call, a system call requires a change in the execution mode of the processor; this is usually implemented by a *trap instruction*. The trap is typically invoked with special parameters that specify which system call to run. The method of implementing this is system dependent. In all cases, the point at which a user process begins to execute kernel code is a perilous point, and great care must be taken by operating system designers and the programmers who code the system call interfaces to make sure that it cannot be booby-trapped by malicious programmers trying to get their programs to run in kernel mode.

Programs do not usually invoke system calls directly. The C library provides *wrappers* for almost all system calls, and these usually have the same name as the call itself. A wrapper for a function `f` is a function that does little more than invoking `f`, usually rearranging or pre-processing its arguments, checking error conditions, and collecting its return value and possibly supplying it in a different form to the caller. Wrappers for system calls also have to trap into kernel mode before the call and restore user mode after the call. A wrapper is *thin* if it does almost nothing but pass through the arguments and the return values. Often, for example, the GNU C library wrapper function is very thin, doing little work other than copying arguments to the right registers before invoking the system call, and then setting the value of a global error variable[16] appropriately after the system call has returned.

Sometimes a wrapper is not so thin, as when the library function has to decide which of several alternative functions to invoke, depending upon what is available in the kernel. For example. there is a system call named `truncate()`, which can "crop" a file to a specified length, discarding the data after that length. The original `truncate()` function could only handle lengths that could fit into a 32-bit integer, and when file systems were able to support very large files, a newer version named `truncate64()` was developed. The latter function can handle lengths representable in 64 bits in size. The wrapper for `truncate()` decides which one is provided by the kernel and calls it appropriately.

There may be system calls that do not have wrappers in the library, and for these, the programmer has no other choice but to invoke the system call with a special function named `syscall()`, passing the system call's identification number and arguments. Every system call has a unique number associated to it. Generally speaking, for a system call named `foo`, its number is defined by a macro named either `__NR_foo` or `SYS_foo`. The macro definitions are included by including the header file `<sys/syscall.h>` in the code. They may not be in that file itself; they may be another file, such as `<asm/unistd_32.h>` or `<asm/unistd_64.h>`. An example of a system call without a wrapper is

[16] To be precise, the variable is named `errno` and it has *thread local storage*, which means each thread has its own unique copy and the lifetime of this variable is the entire lifetime of the thread.

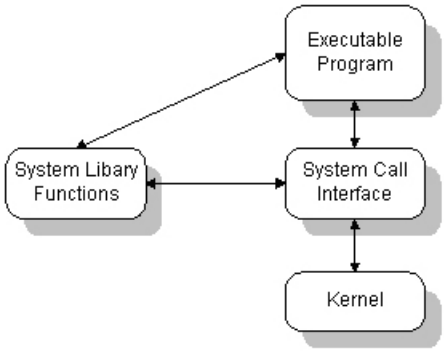


Figure 1.3: System calls versus system libraries.

`gettid()`, which returns the calling thread's thread id. It is the same as `getpid()` for a process with a single thread. The following program calls `gettid()` and prints the returned id on the screen:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Thread id %ld\n", syscall(SYS_gettid));
    /* could also pass __NR_gettid */
    return 0;
}
```

Because the function has no arguments, it is not necessary to pass anything other than the system call number to `syscall()`.

The complete set of system calls is usually available in the `syscalls` manpage in Section 2.

1.4.3 System Libraries

Many system calls are very low-level primitives; they do very simple tasks. This is because the UNIX operating system was designed to keep the kernel as small as possible. Also for this reason, the kernel typically does not provide many different kinds of routines to do similar things. For example, there is only a single kernel function to perform a read operation, and it reads large blocks of data from a specified device to specified system buffers. There is no system call to read a character at a time, which is a useful function to have. In short, there is a single kernel function that performs input operations!

To compensate for the austerity of the kernel, UNIX designers augmented the programming interface with an extensive set of higher-level routines that are kept in system libraries. These routines provide a much richer set of primitives for programming the kernel. Of course, they ultimately make calls to

the kernel, as depicted in Figure 1.3. UNIX also contains libraries for various specialized tasks, such as asynchronous input and output, shared memory, terminal control, login and logout management, and so on. Using any of these libraries requires that the library's header file be included in the code with the appropriate `#include` directive (e.g. `#include <termios.h>`), and sometimes, that the library be linked explicitly because it is not in a standard place. Manpages for functions that are part of system libraries are contained in Volume 3 of the UNIX Manual Pages.

1.5 UNIX and Related Standards

1.5.1 The Problem

The very first version of UNIX was written by Ken Thompson and Dennis Ritchie in 1969 while they were working for AT&T Bell Labs. Their fledgling operating system turned out to be full of very novel ideas, and they presented these ideas in a seminal paper at the ACM Symposium on Operating Systems at IBM Yorktown Heights in 1973. In January 1974, the University of California at Berkeley (UCB) acquired Version 4 from Bell Labs and embarked on a mission to add modern features to UNIX. Later that year, AT&T began licensing UNIX to universities. From 1974 to 1979, UCB and AT&T worked on independent copies of UNIX. By 1978, the various versions of UNIX had most of the features that are found in it today, but not all in one system. But in 1979, AT&T did something that changed the playing field; they staked proprietary rights to their own brand of UNIX, selling it commercially. In essence, they trademarked UNIX and made it expensive to own it.

BSD code was released under a much more generous license than AT&T's source and did not require a license fee or a requirement to be distributed with source unlike the GPL that the GNU Project and Linux use today. The result was that much BSD source code was incorporated into various commercial UNIX variants. By the time that 4.3BSD was written, almost none of the original AT&T source code was left in it. FreeBSD/NetBSD/OpenBSD were all forks of 4.3BSD having none of the original AT&T source code, and no right to the UNIX trademark, but much of their code found its way into commercial UNIX operating systems. In short, two major versions of UNIX came into existence – those based on BSD and those based on the AT&T version.

In 1991, the picture was further complicated by the creation of Linux. Linux was developed from scratch unlike BSD and it used the existing GNU Project which was a clean-room implementation of much of the UNIX user-space. It is a lot less like the AT&T UNIX than BSD is. In 1993, AT&T divested itself of UNIX, selling it to Novell, which one year later sold the trademark to an industry consortium known as X/Open.

There are now dozens of different UNIX distributions, each with its own different behavior. There are systems such as Solaris and UnixWare that are based on SVR4, the AT&T version released in 1989, and FreeBSD and OpenBSD based on the UC Berkeley distributions. Systems such as Linux are hybrids, as are AIX, IRIX, and HP-UX. It is natural to ask what makes a system UNIX. The answer is that over the course of the past thirty years or so, standards have been developed in order to define UNIX. Operating systems can be branded as conforming to one standard or another.

1.5.2 The Solution: Standards

One widely accepted UNIX standard is the *POSIX* standard. Technically, POSIX does not define UNIX in particular; it is more general than that. POSIX, which stands for *Portable Operating*

System Interface, is a family of standards known formally as *IEEE 1003*. It is also published by the *International Standards Organization* (*ISO*) as *ISO/IEC 9945:2003*; these are one and the same document. The most recent version of POSIX is *IEEE Std 1003.1-2008*, also known as *POSIX.1-2008*. The POSIX.1-2008 standard consolidates the major standards preceding it, including POSIX.1, and the *Single UNIX Specification* (*SUS*). The spirit of POSIX is to define a UNIX system, as is stated in the Introduction to the specification (http://pubs.opengroup.org/onlinepubs/9699919799/):

> The intended audience for POSIX.1-2008 is all persons concerned with an industry-wide standard operating system based on the UNIX system. This includes at least four groups of people:
>
> Persons buying hardware and software systems
>
> Persons managing companies that are deciding on future corporate computing directions
>
> Persons implementing operating systems, and especially
>
> Persons developing applications where portability is an objective

The Single UNIX Specification was derived from an earlier standard written in 1994 known as the *X/Open System Interface* which itself was developed around a UNIX portability guide called the *Spec 1170 Initiative*, so called because it contained a description of exactly 1,170 distinct system calls, headers, commands, and utilities covered in the spec. The number of standardized elements has grown since then, as UNIX has grown.

The Single UNIX Specification was revised in 1997, 2001, and 2003 by *The Open Group*, which was formed in 1996 as a merger of *X/Open* and the *Open Software Foundation* (*OSF*), both industry consortia. The Open Group owns the UNIX trademark. It uses the Single UNIX Specification to define the interfaces an implementation must support to call itself a UNIX system.

What, then, does this standard standardize? It standardizes a number of things, including the collection of all system calls, the system libraries, and those utility programs such as `grep`, `awk`, and `sed` that make UNIX feel like UNIX. The collection of system calls is what defines the UNIX kernel. The system calls and system libraries together constitute the UNIX application programming interface. They are the programmer's view of the kernel. The utility programs are the part of the interface that the UNIX user sees.

From the Introduction again:

> POSIX.1-2008 is simultaneously IEEE Std 1003.1™-2008 and The Open Group Technical Standard Base Specifications, Issue 7.
>
> POSIX.1-2008 defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. POSIX.1-2008 is intended to be used by both application developers and system implementers [sic] and comprises four major components (each in an associated volume):
>
> - General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions, are included in the Base Definitions volume.

- Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume.
- Definitions for a standard source code-level interface to command interpretation services (a "shell") and common utility programs for application programs are included in the Shell and Utilities volume.
- Extended rationale that did not fit well into the rest of the document structure, which contains historical information concerning the contents of POSIX.1-2008 and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume.

POSIX.1-2008 specifically defines certain areas as being outside of its scope:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

In summary, the Single UNIX Specification, Version 4, known as SUSv4, also known as The Open Group Specification Issue 7, consists of four parts: a base definition, detailed system interfaces, shell and utilities, and rationale, which describes reasons for everything else.

The fact that there are standards does not imply that all UNIX implementations adhere to them. Although there are systems such as AIX, Solaris, and Mac OS X that are fully POSIX-compliant, most are "mostly" compliant. Systems such as FreeBSD and various versions of Linux fall into this category.

Any single UNIX system may have features and interfaces that do not comply with a standard. The challenge in system programming is being able to write programs that will run across a broad range of systems in spite of this. Later we will see how the use of *feature test macros* in programs provides a means to compile a single program on a variety of different UNIX systems.

1.6 The C Library and C Standards

The interfaces described in the POSIX standard are written in C[17], mostly because C is the major language in which most systems programs are written, and because much of UNIX was originally developed in C. Because of this, POSIX depends upon a standard definition of C, and it uses the ISO standard, the most recent version of which is officially known as ISO/IEC 9899:2011, and informally known as C11. C11 incorporated the earlier ANSI C and augmented it. This version of C is known as ISO C, but people also continue to call it ANSI C, even though it is not the same thing. You can download the last free draft of this standard from C11 Standard (pdf)

In short, POSIX specifies not just what UNIX must do, but what the various parts of the C Standard Library must do as well. It specifies, in effect, a superset of the C language, including additional functions to those introduced in standard C. Therefore, a UNIX system that is POSIX compliant

[17] There are language bindings of the kernel API in *Fortran, Java, Python, Pascal, C++*, and other languages.

contains all of the library functions of the ISO C language. For example, every UNIX distribution includes libraries such as the C Standard I/O Library, the C math library, and the C string library.

The C Standard Library provided for Linux as well as several other UNIX distributions is the GNU C library, called GNU `libc`, or `glibc`. GNU often extends the C library, and not everything in it conforms to the ISO standard, nor to POSIX. What all of this amounts to is that the version of the C library on one system is not necessarily the same as that found on another system.

This is one reason why it is important to know the standard and know what it defines and what it does not define. In general, the C standard describes what is required, what is prohibited, and what is allowed within certain limits. Specifically, it describes

- the representation of C programs
- the syntax and constraints of the C language
- the semantic rules for interpreting C programs
- the representation of input data to be processed by C programs
- the representation of output data produced by C programs
- the restrictions and limits imposed by a conforming implementation of C

Not all compilers and C runtime libraries comply with the standard, and this complicates programming in C. The GNU compiler has command line options that let you compile according to various standards. For example, if you want your program to be compiled against the ANSI standard, you would use the command

```
$ gcc -ansi
```

or

```
$ gcc -std=c90
```

To use the current ISO C11 standard, either of these works:

```
$ gcc -std=c11
$ gcc -std=iso9899:2011
```

Understanding how to write programs for UNIX requires knowing which features are part of C and which are there because they are part of UNIX. In other words, you will need to understand what the C libraries do and what the underlying UNIX system defines. Having a good grasp of the C standard will make this easier.

1.7 Learning System Programming by Example

The number of system calls and library functions is so large that mere mortals cannot remember them all. Trying to learn all of the intricacies and details of the UNIX API by reading through reference manuals and user documentation would be a painstaking task. Fortunately, people often learn well by example. Rather than studying the reference manuals and documentation, we can learn the API little by little by writing programs that use it. One starts out simple, and over time adds complexity.

Bruce Molay [1] uses an excellent strategy for learning how to write system programs and discover the UNIX API:

1. Pick an existing program that uses the API, such as a shell command;

2. Using the system man pages and other information available online, investigate the system calls and kernel data structures that this program most likely uses in its implementation; and

3. Write a new version of the program, iteratively improving it until it behaves just like the actual command.

By repeating this procedure over and over, one can familiarize oneself with the relevant portions of the API as well as the resources needed to learn about it. When it is time to write a full-fledged application, the portions of it that must communicate with the kernel should be relatively easy to write. We will partly follow this same paradigm in this sequence of notes on UNIX.

1.8 The UNIX File Hierarchy

The typical UNIX file hierarchy has several directories just under the root. Figure 1.4 shows part of a typical, but hypothetical, file hierarchy. The following directories should be present in most UNIX systems, but they are not all required. The only required directories are `/dev` and `/tmp`.

Directory	**Purpose**
`bin`	The repository for all essential binary executables including those shell commands that must be available when the computer is running in "single-user mode" (something like safe mode in Windows.)
`boot`	Static files of the boot loader
`dev`	The directory containing essential device files, which will be explained later.
`etc`	Where almost all host configuration files are stored. It is something like the registry file of Windows.
`home`	The directory where all user home directories are located, but not always.
`lib`	Essential shared libraries and kernel modules
`media`	Mount point for removable media
`mnt`	Mount point for mounting a file system temporarily
`opt`	Add-on application software packages
`sbin`	Essential system binaries
`srv`	Data for services provided by this system
`tmp`	Temporary files

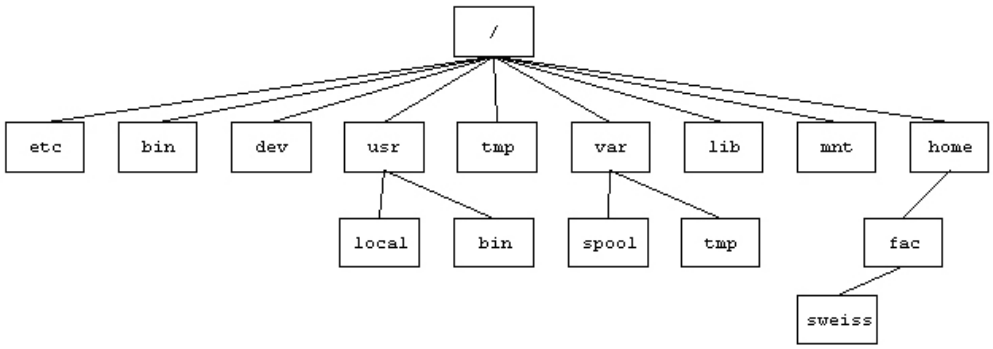


Figure 1.4: The top of a typical UNIX directory hierarchy.

Directory	Purpose
`usr`	Originally, `/usr` was the top of the hierarchy of user "data" files, but now it serves as the top of a hierarchy in which non-essential binaries, libraries, and sources are stored. Below `/usr` are directories such as `/usr/bin` and `/usr/sbin`, containing binaries, `/usr/lib`, containing library files, and `/usr/local`, the top of a third level of "local" programs and data.
`var`	Variable files (files containing data that can change)

Files have two independent binary properties: *shareable* vs. *unshareable* and *variable* vs. *static*. In general, modern UNIX systems are encouraged to put files that differ in either of these respects into different directories. This makes it easy to store files with different usage characteristics on different file systems.

Shareable files are those that can be stored on one host and used on others. Unshareable files are those that are not shareable. For example, the files in user home directories are shareable whereas boot loader files are not.

Static files include binaries, libraries, documentation files and other files that do not change without system administrator intervention. "Variable" files are files that are not static. The `/etc` directory should be unshareable and static. The `/var` directory is variable but parts of it, such as `/var/mail` may be shareable while others such as `/var/log` may be unshareable. `/usr` is shareable and static.

1.8.1 About Pathnames and Directories

A *pathname* is a character string that is used to identify a file. POSIX.1-2008, puts a system-dependent limit on the number of bytes in a pathname, including the terminating null byte[18].

There are two types of pathnames: *absolute* and *relative*. An absolute pathname is a pathname that starts at the root. It begins with a "/" and is followed by zero or more filenames separated by "/" characters. All filenames except the last must be directory names[19]. For exam-

[18]The variable `PATH_MAX` contains the maximum length of a string representing a pathname. The library function `pathconf()` can be used to obtain its value. On many Linux systems it is 4096 bytes.

[19]This is not exactly true. Filenames that are not the last in the pathname may be symbolic links to directories.

ple, `/home/fac/sweiss` is the absolute pathname to the directory `sweiss` in Figure 1.4, as is `/home//fac///sweiss`. The extra slashes are ignored. Observe that UNIX (actually POSIX) uses slashes, not backslashes, in pathnames: `/usr/local`, not `\usr\local`.

If a pathname does not begin with "/" it is called a *relative pathname*. When a process must resolve a relative pathname so that it can access the file, the pathname is assumed to start in the *current working directory*. In fact, the definition of the current working directory, also called the *present working directory*, is that it is the directory that a process uses to resolve pathnames that do not begin with a "/". For example, if the current working directory is `/home/fac`, then the pathname `sweiss/testdata` refers to a file whose absolute pathname is `/home/fac/sweiss/testdata`. The convention is to use *pwd* as a shorthand for the current working directory.

The environment variable `PWD` contains the absolute pathname of the current working directory. The command `pwd` prints the value of the `PWD`.

1.8.1.1 Working with Directories

This section may be skipped if you have experience working with directories at the user level. You do not need to know many commands in order to do most directory-related tasks in UNIX. This is a list of the basic commands. The principal tasks are navigation and displaying and altering their contents. The tables that follow give the command name and the simplest usage of it. They do not describe the various options or details of the command's usage.

Command	**Explanation**
`pwd`	print the path of the current working directory (pwd)
`ls [<dir1>] [<dir2>] ...`	list the contents of the *pwd*, or `<dir1>, <dir2> ...` if supplied
`cd [<dir>]`	change *pwd* to `HOME` directory, or `<dir>` if it is supplied
`mkdir <dir> [<dir2>] ...`	create new directories `<dir>` (and `<dir2>` ...) in the pwd;
`rmdir <dir> [<dir2>] ...`	remove the EMPTY directory `<dir>` (and `<dir2>` ...)
`rm -r <dir>`	remove all contents of `<dir>` and `<dir>` itself. Dangerous!!
`mv`	see the explanation in Section 1.8.2

Notes

1. The "`p`" in "`pwd`" stands for print, but it does not print on a printer. In UNIX, "printing" means displaying on the screen[20].
2. `mkdir` is the only way to create a directory
3. You cannot use `rmdir` to delete a directory if it is not empty.
4. You can delete multiple directories and their contents with `rm -r`, but this is not reversible, so be careful.
5. Commands that create and delete files are technically modifying directories, but these will be covered separately.

[20]That is why the C instruction `printf` sends output to the display device, not the printer. In FORTRAN, by contrast, the `print` instruction sent output to the printer.

"Changing directories" and "being in a directory" are imprecise phrases. When you `cd` to a directory named `dir`, you may think of yourself as being "in `dir`", but this is not true. What is true is that the `dir` directory is now your current working directory and that every process that you run from the shell process in which you changed directory, including the shell process, will use this `dir` directory by default when it is trying to resolve relative pathnames.

There are two special entries that are defined in every directory

`.`	The directory itself
`..`	The parent directory of the directory, or itself if it is /

Thus, "`cd ..`" changes the *pwd* to the parent of the current *pwd*, "`cd ../..`" changes it to the grandparent and "`cd .`" has no effect. Before reading further, you should experiment with these commands and make sure that you understand how they work.

1.8.2 Files and Filenames

Unlike other operating systems, UNIX distinguishes between only five types of non-directory files:

- regular files
- device files (character or block)
- FIFOs
- sockets
- symbolic links

Device files and FIFOs will be described in depth in Chapter 4, sockets in Chapter 9, and symbolic links below. That leaves regular files, which are defined quite simply: a regular file is a sequence of bytes with no particular structure.

1.8.2.1 Filenames

Files and filenames, as noted earlier, are different things. A filename, which is technically called a *file link*, or just a *link*, is just a string that names a file[21]. A file may have many filenames. Filenames are practically unlimited in size, unless you think 255 characters is not big enough. The maximum number of bytes in a filename is contained in the system-dependent constant `NAME_MAX`. Filenames can contain any characters except "`/`" and the null character. They can have spaces and new lines, but if they do, you will usually need to put quotes around the name to use it as an argument to commands. UNIX is *case-sensitive*, so "References" and "references" are two different filenames.

Unlike DOS, Windows, and Apple operating systems, filename extensions are not used by the operating system for any purpose, although application-level software such as compilers and word

[21] A filename is sometimes referred to as a "pathname component".

processors use them as guides, and desktop environments such as Gnome and KDE create associations based on filename extensions in much the same way that Windows and Apple do. But again, UNIX itself does not have a notion of file type based on content, and it provides the same set of operations for all files, regardless of their type.

Remember that a directory entry consists of two parts. One part is a filename and the other part is a means by which a file is associated with this name. You may think of this second part as an index into a table of pointers to the actual files[22].

One file can have many links, like a spy traveling with several forged passports. In one directory, the file may be known by one link, and in a different directory, it may have another link. It is still the same file though. In Figure 1.5, the file is known by three names, each a link in a different directory. There is two restrictions concerning multiple links. One is that directories cannot have multiple names. Another is that two names for the same file cannot exist on different file systems. The simplest way to understand this last point, without going into a discussion of mounting and mount points, is that different parts of the file hierarchy can reside on different physical devices, e.g., different partitions of a disk or different disks, and that a file on one physical device cannot have a name on a different physical device. This will be clarified in a later chapter.

1.8.2.2 What is a File?

All files, regardless of their type, have *attributes*. Almost all have *data*, or *content*.

- Attributes include properties such as the time the file was created, the time it was last modified, the file size expressed as a number of bytes, the number of disk blocks allocated to the file, and so on. The attributes that describe restrictions on access to the file are called the *file mode*. The attributes of a file collectively are called the *file status* in UNIX. The word "status" may sound misleading, but it is the word that was used by the original designers of UNIX.
- The data are the actual file contents. In fact, UNIX uses the term *content* to identify this part of a file. Some files do not have any content because they are merely interfaces that the operating system uses. That will be discussed further in Chapter 4.

These two items, the status and the content, are not stored together[23].

1.8.2.3 Working with Files

This section may be skipped if you have user level experience in UNIX. Commands for working with files can be classified as those that view file contents but do not modify them, those that view file attributes, and editors – those that modify the files in one way or another. Listed below are the basic commands for viewing the contents of files, not editing or modifying them. Editors are a separate topic. In all cases, if the file arguments are omitted, the command reads from standard input. There is also a separate class of programs called filters that provide sophisticated filtering of ordinary text files.

[22] Traditional UNIX system used an index number, called an *inumber*, to associate the file to the filename. This number was used to access a pointer to a structure called an *inode*, to be discussed later. The inode contains members that point to the file contents. Conceptually, the inode is like a proxy for the file.

[23] The status is stored in the inode.

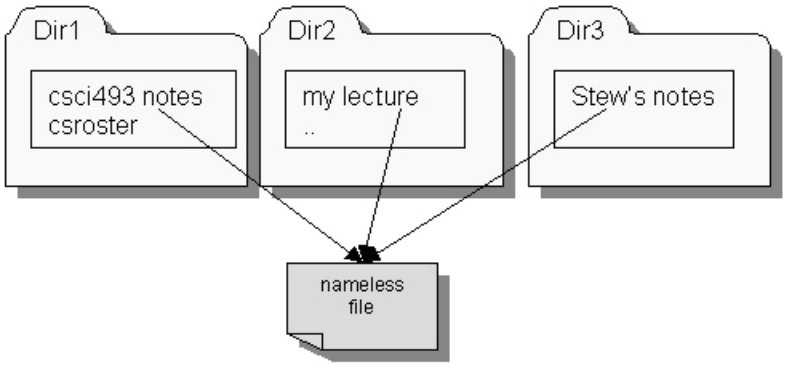


Figure 1.5: Example of multiple links to a file.

Viewing File Contents

Command	Explanation
`cat [<files>]`	display file contents
`more [<files>]`	display file contents a screen at a time
`less [<files>]`	display file contents a screen at a time with more options
`pg [<files>]`	display file contents a screen at a time.
`head [-<n>] [<file>]`	display the first `<n>` lines of a file, default n = 10
`tail [-<n>] [<file>]`	display the last `<n>` lines of a file, default n = 10

Note that the `pg` command is not POSIX and may not be available on all UNIX systems.

A note about `more` and `less`: the `more` command existed long before there was a `less` command. `more` was created to allow you to read a page at a time, and also to jump ahead in the file with regular expression searches, using the same "/" operator from `vi`. However, it was not easy to go backwards in the file with `more`. You could use the single quote operator (') to go back to the beginning of the file, but that was about it. `less` was created to do more than `more`, naturally, which is why it is called `less`. Maybe the person who wrote it knew the famous quotation from Ludwig Mies Van der Rohe, the German-born architect, whose famous adage, "Less is more," is immortalized. Maybe not. But for whatever reason that they named `less`, "`less`", `less` does more than `more`. And this is not an Abbott and Costello skit. The `less` command is the most versatile means of viewing files, and my recommendation is that you learn how to use it first.

Creating, Removing, and Copying Files and Links Remember that a link is just a name pointing to a file.

Command	Explanation
`ln <f1> <f2>`	create a new link for file `<f1>` named `<f2>`
`rm <files>`	delete a link or name for a file
`mv <file1> <file2>`	rename `<file1>` with the new name `<file2>`
`mv <files> <dir>`	move all files into the destination directory `<dir>`
`cp <file1> <file2>`	copy `<file1>` to the new name `<file2>`
`cp <files> <dir>`	copy all files into the destination directory `<dir>`

Notes.

- The `ln` command can only be used on files, not directories. There is a simplified version of this command named `link`.
- The `rm` command is irreversible; once a link is removed, it cannot be recovered.
- The `mv` command has two forms. If there are more than two arguments, the last must be an existing directory name, and all files named before the last are moved into that directory. Conversely, if the last argument is an existing directory name, then all arguments preceding it will be moved into it. The preceding arguments can be existing directories, as long as it does not create a circularity, such as trying to make a child into a parent. If the last argument is not an existing directory, the meaning of `mv` is simply to rename the first file with the new name. If the new name is an existing file, `mv` will silently overwrite it. For this reason, you should use "`mv -i`", which prompts you before overwriting files.
- The way you use the `cp` command is almost the same as how you use the `mv` command except that it replicates files instead of moving them. The main difference is that `cp` does not accept directories as arguments, except as the last argument, unless you give it the option "`-r`", i.e., `cp -r`, in which case it recursively copies the directories. Note that `cp` makes copies of files, not just their names, so changes to one file are not reflected in the other.

Examples

```
$ ls
hwk1_6.c
$ mv hwk1_6.c hwk1_finalversion.c
$ ls
hwk1_finalversion.c
```

This command changed the link `hwk1_6.c` to `hwk1_finalversion.c`.

```
$ rm hwk1.o hwk1.old.o main.o
```

This removes these three file links from the current working directory. If these files have no other links in other directories, the attributes and contents are also removed from the disk. I will explain more about this in a later chapter.

```
$ ln hwk1.1.c ../mysourcefiles/proj1.c
```

creates a new link `proj1.c` in the directory `../mysourcefiles` for the file whose name is `hwk1.1.c` in the current working directory.

```
$ cp -r main.c utils.c utils.h images ../version2
```

copies the three files `main.c`, `utils.c`, and `utils.h`, and the directory `images`, into the directory `../version2`.

1.8.2.4 File Attributes

In this section, the word "file" refers to all file types, including directories. UNIX has a very simple, but useful, file protection method. To provide a way for users to control access to their files, the inventors of UNIX devised a rather elegant and simple access control system. Every file has an owner, called its *user*. The file is also associated with one of the groups to which the owner belongs, called its *group*. The owner of a file can make the file's group any of the groups to which the owner belongs. Lastly, everyone who is neither the user nor a member of the file's group is in the class known as *others*. Thus, the set of all users is partitioned into three disjoint subsets: user, group, others, which you can remember with the acronym *ugo*.

There are three modes of access to any file: read, write, and execute. *Read access* is the ability to view file contents. For directories, this is the ability to view the contents using the ls command. *Write access* is the ability to change file contents or certain file attributes. For directories, this implies the ability to create new links in the directory, to rename files in the directory, or remove links in the directory. This will be counterintuitive – the ability to delete a file from a directory does not depend on whether one has write privileges for the file, but on whether one has write privileges for the directory. *Execute access* is the ability to run the file. For a directory, execute access is the ability to `cd` into the directory and as a result, the ability to run programs contained in the the directory and run programs that need to access the attributes or contents of files within that directory. In short, without execute access on a directory, there is little you can do with it.

For each of the three classes of users, there are three protection bits that define the read, write, and execute privileges afforded to members of the class. For each class, if a specific protection bit is set, then for anyone in that class, the particular type of access is permitted. Thus, there are three bits called the **r**ead, **w**rite, and e**x**ecute bits for the user (`u`), for the group (`g`), and for others (`o`), or nine in total. These bits, which are called *mode* or *permission bits*, are usually expressed in one of two forms: as an octal number, or as a string of nine characters.

The nine-character permission bit string is:

```
r w x       r w x       r w x
user bits   group bits  others bits
```

Various commands use a dash is used to indicate that the bit is turned off.

Examples

- The string `rwxrw-r--`, gives the user (owner) read, write, and execute permission, the group, read and write but no execute, and others, only read permission.
- The string `r-xr-xr-x` gives everyone only read and execute permission.
- The string `rwxr-xr--` gives the user read, write, and execute permission, the group, read and execute permission, and others, only read access.

The mode string can be represented as a 3-digit octal number, by treating each group of three bits as a single octal digit. Using the C ternary operator, `?:`, I would write this as

```
value = r?4:0 + w?2:0 + x?1:0
```

which results in the following table of values for each group of three bits.

rwx	rw-	r-x	r--	-wx	-w-	--x	---
7	6	5	4	3	2	1	0

For example, the octal number for `rwxrw-r--` is 764 because the user digit is 7, the group is 6 and the others is 4.

In addition to the mode bits, a file's permission string is usually displayed with a single character file attribute that characterizes the file type. The character, which appears to the left of the mode bits, can be one of `-` (regular file), `d` (directory), `b` (buffered special file), `c` (character special file), `l` (symbolic link), `p` (pipe), or `s` (socket).

1.8.2.5 Viewing and Modifying File Attributes

To see the attributes of a file, use the `-l` option to the `ls` command: The "`-l`" means "long listing" and it prints the permission string, the number of links to the file, the user-name of the owner, the group name of the group, the number of bytes in the file, the last modification time, and the file link. For example, to look at the attributes of the file named .bashrc in the current working directory, I would type

```
$ ls -l ~/.bashrc
-rw-r--r-- 1 sweiss faculty 3304 Sep 22 13:05 .bashrc
```

This file is a regular file and can be read and modified by me, its owner (`-rw-`). It can be read by anyone in the faculty group (`r--`), and it can be read by anyone else (`r--`). `ls` has many other options, for displaying additional information. Read its man page for details. You can also use the `stat` command, which will provide additional information:

```
$ stat .bashrc
  File: ‘.bashrc’
  Size: 3304      Blocks: 8   IO Block: 4096   regular file
Device: 18h/24d Inode: 1318        Links: 1
Access:(0644/-rw-r--r--)Uid:(1220/ sweiss)Gid:(400/ faculty)
Access: 2010-12-20 13:20:04.582733000 -0500
Modify: 2010-09-22 13:05:11.271251000 -0400
Change: 2010-09-22 13:05:11.278893000 -0400
```

You can get the count of bytes, words, and lines with `wc`:

```
$ wc .bashrc
156 387 3304 .bashrc
```

There are 156 lines, 387 words, and 3304 bytes in the file. Other options provide different information.

Commands that alter the attributes of a file are:

Command	Explanation
`chmod <mode> <files>`	change the file permissions
`chown <owner> <files>`	change the file ownership
`chgrp <group> <files>`	change the group ownership
`touch <files>`	update timestamps of files; create empty files (a file of size 0)

1.8.2.6 Symbolic Links

A *symbolic link*, also called a *soft link*, is a file that stores a string containing the pathname of another file. The stored string has a length of `SYMLINK_MAX` bytes or fewer. This string is not part of the content of the file, but can be used by programs to access the file whose pathname it contains. Symbolic links are like shortcuts in the Windows operating system; they have no data in themselves, but instead point to files. They are useful for overcoming the limitations of hard links that they cannot link to directories and cannot cross file systems. For example, suppose that one frequently accesses a directory whose absolute path is

```
/data/research/biochem/proteins
```

He or she could create a link in his or her home directory for easier access to this directory, as follows:

```
$ ln -s /data/research/biochem/proteins ~/proteins
```

The `ln` command, with the `-s` option, creates symbolic links. The pathname `/data/research/biochem/proteins` would be stored in the file `~/proteins` in such a way that the commands that use the filename proteins would replace it by the pathname stored there.

Symbolic links pose hazards for the operating system because of the possibility of circular references and infinite loops. More will be said about them in Chapter 3.

1.9 Under the Hood: How Logging In Works

When you log in to a UNIX system, what actually happens? What does it mean to be logged in?

There is no single answer to this question, as it depends upon (1) which version of UNIX is running and (2) whether the login is at the console, or within a terminal window, or across a network using a protocol such as SSH. Fortunately, the way that logging in works at the console or in a terminal window in the predominant UNIX systems – Linux, Mac OS, Solaris, and BSD[24] variants – is pretty much the same, with minor variations, and the way that it works over a network, while very different from how it works at a console or terminal window, is similar across the major UNIX systems.

Terminal or console logins in the predominant UNIX systems are usually based upon the method used by the early BSD UNIX system. We will examine how the BSD method worked. Modern UNIX systems have the option to use a very different method known as PAM, discussed below.

When a UNIX system is started, after the kernel initializes the data structures that it needs and enables interrupts, it creates a new process with process-id 1, named `init`. `init` is the first process

[24]BSD stands for *Berkeley Software Distribution*. See the historical notes at the end of this chapter.

created at start-up, and it is also the ancestor of all user-level processes in a UNIX system, even though it runs with root's privileges. `init` monitors the activities of all processes in the outer layers of the operating system, and also manages what takes place when the computer is shutdown.

The `init` process does a number of interesting things, but of interest to us now is that `init` uses information about available terminal devices on the system (i.e., consoles, modems, etc.) to create, for each available terminal, a process to listen for activity on that terminal. These processes are the `getty` processes[25]. The name `getty` stands for "`get tty`". The `getty` process configures the terminal device, displays a prompt such as "`login:`" in the terminal, and waits for the user to enter a user-name.

The "`tty`" in "`getty`" is short for *Teletype*. For those who do not know the history, a *Teletype* is the precursor to the modern computer terminal. Teletype machines came into existence as early as 1906, but it was not until around 1930 that their design stabilized. Teletype machines were essentially typewriters that converted the typed characters into electronic codes that could be transmitted across electrical wires. Modern computer terminals inherit many of their characteristics from Teletype machines.

When the user enters a user-name on the terminal, the `getty` process runs the `login` program[26], passing it the entered user-name. `login` performs a number of tasks and prompts the user for the password and tries to validate it. If it is valid, `login` sets the current working directory (`PWD`[27]) to the user's home directory, sets the process's user-id to that of the user, initializes the user's environment, adjusts permissions and ownership of various files, and then starts up the user's login shell. If the password is invalid, the `login` program will exit, and `init` will notice this and start up a new `getty` for that terminal, which will repeat the above procedure.

Systems that use PAM do not work this way. PAM, which is short for *Pluggable Authentication Modules*, is a library of dynamically configurable authentication routines that can be selected at runtime to do various authentication tasks, not just logins. We will not cover PAM here.

Network logins, which are usually based upon the BSD network login mechanism, must work differently. For one, there are no physical terminals, and so there is no way to know in advance how many terminals must be initialized. For another, the connection between the terminal and the computer is not point-to-point, but is a network service, such as SSH or SFTP.

BSD took the approach of trying to make the login code independent of the source of the login. The result is that it uses the idea of a *pseudo-terminal*. These will be covered in depth later. With network logins, rather than creating a `getty` process for each terminal, `init` creates the process that will listen for the incoming network requests for logins. For example, if the system supports logging in through SSH, then `init` will create a process named `sshd`, the SSH daemon, which will in turn create a new process for each remote login. These new processes will, in turn, create what is called a *pseudo-terminal driver* (pts driver), which will then spawn the `login` program, which does everything described above. The picture is quite different and much more complicated than

[25] This is not quite accurate but it is good enough for now. If you want to know the actual steps, it is best to wait until you understand how processes are created and what the difference is between creating a process and running a program.

[26] Actually, the `getty` process replaces itself with the `login` program using a system call named `execve()`. This topic is covered in a later chapter.

[27] In `bash`, the environment variable `PWD` stores the absolute pathname of the current working directory. `Bash` inherits this name from the Bourne shell, in which it stood for "present working directory." In the C-shell, it is still `cwd`. Even though it is `PWD` in `bash`, no one calls it the present working directory anymore; it is the current working directory.

a simple terminal login, because the management of the pseudo-terminal is complex. This will be covered in depth later.

1.10 UNIX From The System Programmer's Perspective

The way that a systems programmer sees UNIX is very different from the way that a user sees it. Whereas the user perceives UNIX by the functionality of the shell, the systems programmer looks "beneath" the shell at the functionality of the kernel inside, as suggested by Figure 1.6.

However, not only does the systems programmer need to understand the kernel API, he or she also has to understand how programs interact with users, how they interact with the operating system, and how they interact with other programs. Interaction encompasses three distinct actions:

- Acquiring data
- Delivering data
- Coordinating execution in time

Simple programs acquire data from an external source such as a file or the keyboard, and deliver data to external sources such as files or the console. But real applications may also have to acquire data from other programs, and the operating system in particular, or acquire it from a system resource or a shared resource such as a file already opened by a different process. They may also have to write to a shared resource, such as a window into which other processes are writing, or a file that may be shared by other processes.

Figure 1.6: The system programmer's view of UNIX.

Even more complex is the possibility that data may be delivered to the process asynchronously, meaning, not when the process asked for it, but at some later, unpredictable time. And if that is not enough, consider the possibility that the process should not continue execution until some other process has performed some other task. For example, consider a program that is playing chess against another program. Neither program is allowed to make a move until the other has finished its turn. Similarly, imagine multiple processes that are cooperating to compute the structure of a complex protein. Certain parts of the structure cannot be constructed until other parts have been calculated, so the processes must coordinate their activities with respect to the work accomplished by the others. Both of these situations require the use of process synchronization. The challenge is to master these concepts.

1.11 A First System Program

We will begin by writing a stripped down version of the `more` program, which displays a file one "screen[28]" at a time. When `more` runs, it displays one screen's many lines of a file and then displays information and a prompt on the very bottom line of the screen:

–More–(0%)

It then waits for the user to press a key such as the space-bar to advance to the next screen, or the *Enter* key to advance one line, or the "`q`" key to exit. The prompt is in *reverse video*, meaning the foreground and background terminal colors are reversed. While there are other `more` advanced options, we will limit our version to these alone. Run `more` and observe that when you press the space-bar, it responds immediately; you do not have to press the *Enter* key after space-bar.

1.11.1 A First Attempt at the `more` Program

The `more` command can be run in several ways:

```
$ more file1 file2 ... fileN
$ ls -l | more
$ more < myfile
```

The first line causes `more` to display the files named `file1`, `file2`, and so on until `fileN`, one after the other. This proves that the `more` program has to look at the command line arguments and use them as its input if they exist. In the second example, the output of the command "`ls -l`" becomes the input of the `more` program, and the result is that `more` displays the directory listing a screen at a time. This implies that `more` also has to work when its input comes from standard input through a pipe. In the last line, the file `myfile` is used as input to the `more` command, but the `more` command is simply getting input from the standard input stream.

These three examples show that `more` has to be able to read from a file specified on the command line as well as read from standard input, and in either case it has to output a screen at a time. Suppose that we let `P` represent the number of lines in a terminal window (`P` stands for "page", which is what a screenful of data is called.) A crude outline of the `more` program, ignoring the issue of where it gets its input, is:

```
1 Show P-1 lines from standard input (save the last line for the prompt)
2 Show the [more?] message after the lines.
3 Wait for an input of Enter, Space, or 'q'
4 If input is Enter, advance one line; go to 2
5 If input is Space, go to 1
6 If input is 'q', exit.
```

[28] The standard screen has 24 lines, but this is user-adjustable and also dependent on both hardware and system settings. Until we know how to find the actual number, we will assume a screen has 24 lines, and we write 23 new lines at a time.

We have to add to this the ability to extract the filenames, if they exist, from the command line. Listing 1.4 contains a C main program for a version of `more` using the above logic, that also checks if there are one or more command line arguments, and if there are, uses the arguments as the names of files to open instead of standard input. If there are no command line arguments, it uses standard input. We will assume for now that `P`=24.

Listing 1.4: A first version of the main program for `more`.

```
#include <stdio.h>
#define SCREEN_ROWS     23 /* assume 24 lines per screen */
#define LINELEN        512
#define SPACEBAR         1
#define RETURN           2
#define QUIT             3
#define INVALID          4

/** do_more_of()
 *  Given a FILE* argument fp, display up to a page of the
 *  file fp, and then display a prompt and wait for user input.
 *  If user inputs SPACEBAR, display next page.
 *  If user inputs RETURN, display one more line.
 *  If user inputs QUIT, terminate program.
 */
void do_more_of (FILE * filep);

/** get_user_input()
 *  Displays more's status and prompt and waits for user response,
 *  Requires that user press return key to receive input
 *  Returns one of SPACEBAR, RETURN, or QUIT on valid keypresses
 *  and INVALID for invalid keypresses.
 */
int  get_user_input();

int main( int argc , char *argv[] )
{
    FILE    *fp;
    int     i = 0;
    if ( 1 == argc )
        do_more_of( stdin );    // no args, read from standard input
    else
        while ( ++i < argc ) {
            fp = fopen( argv[i] , "r" );
            if ( NULL != fp ) {
                do_more_of( fp ) ;
                fclose( fp );
            }
            else
                printf ( "Skipping %s\n", argv[i] );
        }
```

```
    return 0;
}
```

If you are not familiar with some of the C functions (also in C++) used in this program, then read about them, either in the man pages, or in any introductory C/C++ textbook. In particular you need to know about the following functions and types, all part of ANSI standard C[29] and defined in `<stdio.h>`:

`FILE`	a file stream
`fopen`	opens a file and returns a `FILE*`
`fclose`	closes a `FILE` stream
`fgets`	reads a string from a `FILE` stream
`fputs`	writes a string to a `FILE` stream

One thing to observe in the code above is that it checks whether or not the return value of the `fopen()` function is `NULL`. Every time a program makes a call to a library function or a system function, it should check the possible error conditions. A program that fails to do this is a program that will have frustrated users.

If the argument list in the main program signature is also new to you, here is a brief explanation. In the signature

```
int main( int argc , char *argv[] )
```

`argc` is an integer parameter that specifies the number of words on the command line. Since the program name itself is one word, `argc` is always at least 1. If it is exactly 1, there are no command line arguments. The second parameter, `argv`, is an array of `char` pointers. In C, a `char` pointer is a pointer to a `NULL`-terminated string, i.e., a string whose last character is `'\0'`. The array `argv` is an array whose strings are the words on the command line. `argv[0]` is the program name itself, `argv[1]`, the first command argument, and so on. The name of the parameter is arbitrary. It will work whether `argc` is named *rosebud*, *numargs*, or *ac*. The shell is responsible for putting the command arguments into the memory locations where the first and second parameters are expected (which it does by making a system call and passing these arguments to the call).

Listing 1.4 does not include the definitions of the two functions used by `main()`: `do_more_of()` and `get_user_input()`, which are contained in Listing 1.5. The first of these is easy to figure out, except that you might not have used `fgets()` and `fputs()` before. The function keeps track of how many lines can be written to the screen before the screen is full in the variable `num_of_lines`, which is initially 23. The second function is also pretty straightforward. The only part that might require explanation is the `printf()` instruction.

[29] These are in all versions of C.

1.11.2 A Bit About Terminals

Our program requires that we display the prompt in reverse video. The question is how we can display text in reverse video on a terminal. Terminals are a complex subject, about which we devote almost an entire chapter later on (Chapter 4). Now we offer just a brief introduction to them.

A terminal normally performs two roles: it is an input device and an output device. As an output device, there are special codes that can be delivered to it that it will treat, not as actual text to be displayed, but as control sequences, i.e., sequences of bytes that tell it where to position the cursor, how to display text, how to scroll, how to wrap or not, what colors to use, and so on. When terminals were first developed, there were many different types of them and many vendors. Each different type had a different set of control sequences. In 1976, the set of sequences that can be delivered to a terminal was standardized by the *European Computer Manufacturers Association (ECMA)*. The standard was updated several times and ultimately adopted by the *International Organization for Standardization (ISO)* and the *International Electrotechnical Commission (IEC)* and was named *ISO/IEC 6429*. It was also adopted by the *American National Standards Institute* (*ANSI*) and known as *ANSI X3.64*. The set of these sequences are now commonly called the *ANSI escape sequences* even though ANSI withdrew the standard in 1997.

An ANSI escape sequence is a sequence of ASCII characters, the first two of which are normally the ASCII *Escape* character, whose decimal code is 27, and the left-bracket character " [". The Escape character can be written as '`\033`' using octal notation. The string "`\033[`" is known as the *Control Sequence Introducer*, or *CSI*. The character or characters following the CSI specify an alphanumeric code that controls a keyboard or display function. For example, the sequence "`\033[7m`" is the CSI followed by the control code "`7m`". The code "`7m`" is a code that reverses the video display. The escape sequence "`\033[m`" turns off all preceding character codes.

If we want to display a portion of text, such as a prompt, in reverse video, we need to send the escape sequences and text to the terminal device. Any output function that can write to the terminal will suffice. We use the `printf()` function because it is the easiest. To write the prompt " `more?` " in reverse video, we send the first escape sequence, then the prompt, then the sequence to restore the terminal:

```
printf("\033[7m more? \033[m");
```

Although the preceding discussion centered on terminals, it is not likely that you are using an actual terminal when you are working in a shell. Most likely, you are using a *terminal emulation package*, such as *Gnome Terminal* or *Konsole* on Linux, or if connecting remotely, a package such as *PuTTY*. Almost all terminal emulators running on Unix systems interpret some subset of the ANSI escape sequences. They use software to emulate the behavior of hardware terminals. One of the first terminals to support ANSI escape sequences was the VT100. To this day, most terminal emulators support the VT100. Some also support more advanced terminals such as the VT102 or the VT220. In principle, if the terminal emulator supports the full set of ANSI escape sequences, a program that uses these sequences should work regardless of which terminal is being emulated.

This preceding code to reverse the video is just one of many escape sequences that can control the terminal. We will explore a few more of them a little later.

Listing 1.5: The supporting functions for Version 1 of more.

```
void do_more_of( FILE *fp )
```

```
{
    char   line[LINELEN];       // buffer to store line of input
    int    num_of_lines = SCREEN_ROWS;  // # of lines left on screen
    int    getmore      = 1;   // boolean to signal when to stop
    int    reply;              // input from user

    while ( getmore && fgets( line, LINELEN, fp ) ){
        // fgets() returns pointer to string read or NULL
        if ( num_of_lines == 0 ) {
            // reached screen capacity so display prompt
            reply = get_user_input();
            switch ( reply ) {
                case SPACEBAR:
                    // allow full screen
                    num_of_lines = SCREEN_ROWS;
                    break;
                case RETURN:
                    // allow one more line
                    num_of_lines++;
                    break;
                case QUIT:
                    getmore = 0;
                    break;
                default:    // in case of invalid input
                    break;
            }
        }
        if ( fputs( line, stdout ) == EOF )
            exit(1);
        num_of_lines--;
    }
}

int get_user_input()
/*
 *    display message, wait for response, return key entered as int
 *    Returns SPACEBAR, RETURN, QUIT, or INVALID
 */
{
    int    c;

    printf("\033[7m more? \033[m");    /* reverse on a VT100    */
    while( (c = getchar()) != EOF )    /* wait for response     */
        switch ( c ) {
            case 'q' :                 /* 'q' pressed */
                return QUIT;
            case ' ' :                 /* ' ' pressed */
                return SPACEBAR;
```

```
            case '\n' :                 /* Enter key pressed */
                return RETURN;
            default :                   /* invalid if anything else */
                return INVALID;
        }
}
```

Compile and run this program. To compile and run, if all of the code is in the single file named `more_v1.c`, use the commands

```
$ gcc more_v1.c -o more_v1
$ more_v1 more_v1.c
```

My code is in three files, `more_v1.c`, `more_utils_v1.c`, and `more_utils_v1.h`. I compile using

```
$ gcc -o more_v1 more_utils_v1.c more_v1.c
```

When you run it you will find that

- `more_v1` does display the first 23 lines, and
- It does produce reverse video over the `more?` prompt.
- But pressing space-bar or 'q' has no effect until you press the *Enter* key.
- If you press *Enter*, the `more?` prompt is replicated and the old one scrolls up with the displayed text. In other words, the `more?` prompt is not erased.

Next, run the command

```
$ ls /bin | more_v1
```

and observe what happens. It is not what you expected. Why not? The function `get_user_input()` calls `getchar()` to get the user's response to determine what `more_v1` should do next. If you recall the discussion in the beginning of this chapter, `getchar()` reads from the standard input stream. Since standard input is the output end of the pipeline from the `ls -l` command, `getchar()` gets characters from the `ls` listing instead of from the user's typing at the keyboard. As soon as the output of `ls -l` contains a space character or a `q` or a newline, the program will treat that as what the user typed. This first version of `more` fails to work correctly because it uses `getchar()` to get the user's input in the `get_user_input()` function. Somehow, we have to get the user's input from the keyboard regardless of the source of the standard input stream. In other words, `get_user_input()` has to read from the actual keyboard device. Well, not exactly. It has to read from the terminal that the user was given when the user logged into the UNIX system.

A process can read directly from a terminal in UNIX by reading from the file `/dev/tty`. `/dev/tty` is an example of a *device special file*.

1.11.3 Device Special Files

The UNIX system deviated from the design of all other operating systems of the time by simplifying the way in which programs handled I/O. Every I/O device (disk, printer, modem, etc.) is associated with a *device special file*, which is one kind of *special file*. Special files can be accessed using the same system calls as regular files, but the way the kernel handles the system call is different when the file argument is a device special file; the system call activates the device driver for that device rather than causing the direct transfer of data. This frees the programmer from having to write different code for different types of devices: he or she can write a program that performs output without having to know whether the output will go to a disk file, a display device, a printer, or any other device. The program just connects to a file variable, which may be associated at run time with any of these files or devices. This is the essence of *device-independent I/O*.

To illustrate, suppose a portion of C++ code writes to an output stream as follows:

```
ofstream  outfile;
cout << "Where should output be sent? ";
cin >> filename;
outfile.open(filename);
```

On a UNIX system, the user can enter the filename "`/dev/console`" at the prompt, and the output will go to the display device. If the user enters "`myfile`" the output will go to `myfile`. The kernel will take care of the details of transferring the data from the program to the device or the file.

There is another type of special file called a *named pipe*, also called a *FIFO special file*, or a *FIFO* for short. Named pipes are used for inter-process communication on a single host; they will be covered in Chapter 8.

The last type[30] of special file is a *socket*. Sockets were introduced into UNIX in 4.2BSD, and provide inter-process communication across networks. Sockets are special files in the sense that they are part of the special file system, but they are different than other special files because there are different system calls needed to manipulate them. Sockets will be covered in Chapter 9.

An entry for each device file resides in the directory `/dev`, although system administrators often create soft links to them in other parts of the file system. The advantages of device files are that

- Device I/O is treated uniformly, making it easier to write programs that are device independent;
- Changes to the hardware result only in changes to the drivers and not to the programs that access them;
- Programs can treat files and devices the same, using the same naming conventions, so that a change to a program to write to a device instead of a file is trivial;
- Devices are accorded the same protections as files.

Device files are described in greater depth in Chapter 4 of these notes.

[30] This is not completely true, since certain versions of UNIX have other special files, such as the door in Sun Solaris. The ones listed here are part of most standards.

1.11.3.1 Examples

The names of device files vary from one version of UNIX to another. The following are examples of device files that with very high certainty will be found on any system you use.

- `/dev/tty` is the name of the terminal that the process is using. Any characters sent to it are written to the screen.
- `/dev/mem` is a special file that is a character interface to memory, allowing a process to write individual characters to portions of memory to which it has access.
- `/dev/null` is a special file that acts like a black hole. All data sent to it is discarded. On Linux, `/dev/zero` is also used for this purpose, and will return null characters (`'\0'`) when read.
- The names of hard disk drive partitions vary from one system to another. Sometimes they have names like `/dev/rd0a` or `/dev/hda`. The cd-rom drive is almost always mapped to `/dev/cdrom`. You should browse the `/dev` directory on your machine to see which files exist.

Now try an experiment. Login to the UNIX system and enter the command below and observe the system response.

```
$ echo "hello" > /dev/tty
```

Now do the following. Type "`tty`" and use the output in the `echo` command as below.

```
$ tty
/dev/pts/4
$ echo "hello" > /dev/pts/4
hello
```

Now open a second window. If you are connected remotely, you can either login a second time, or use the SSH client to open a second terminal window. If you are logged in directly, just create a second terminal window. Type the `tty` command in the new window and record the name of the device file. Suppose it is `/dev/pts/2`. Now in the first window, type

```
$ echo "hello" > /dev/pts/2
```

and observe what happens. You have just discovered that you can write to a terminal by writing to a device file. You will only be able to write to terminals that you own, not those of other people. Later we will see how you can write to other terminals, provided that the owner of that terminal has granted this type of access.

1.11.4 A Second Attempt at the more Program

We can use the device file idea to modify the more program to overcome the problem of redirected standard input. All that is necessary is to supply the `get_user_input()` function with a parameter that specifies the source of user input. This requires

1. That `do_more_of()` declare a `FILE*` variable and assign to it the result of a call to `fopen(/dev/tty)`.
2. That `do_more_of()` call `get_user_input()` with the `FILE*` variable assigned by `fopen()`.
3. That `get_user_input()` have a parameter `fp` of type `FILE*`.
4. That we replace the call in the `while` loop condition `(c = getchar())` by `(c=fgetc(fp))`.

The `getchar()` function always reads from the standard input stream. It does not take an argument. It may be, in fact, a macro. It is better to use `fgetc()`, which takes a `FILE*` argument. The modified functions are in Listing 1.6.

Listing 1.6: Corrected versions of `do_more_of()` and `get_user_input()`.

```
void do_more_of( FILE *fp )
{
    char line[LINELEN];              // buffer to store line of input
    int  num_of_lines = SCREEN_ROWS; // number of lines left on screen
    int  getmore      = 1;           // boolean to signal when to stop
    int  reply;                      // input from user
    FILE *fp_tty;

    fp_tty = fopen( "/dev/tty", "r" ); // NEW: FILE stream argument
    if ( fp_tty == NULL )              // if open fails
        exit(1);                       // exit

    while ( getmore && fgets( line, LINELEN, fp ) ){
        // fgets() returns pointer to string read
        if ( num_of_lines == 0 ) {     // reached screen capacity
            reply = get_user_input( fp_tty );  // NEW
            switch ( reply ) {
                case SPACEBAR:
                    // allow full screen
                    num_of_lines = SCREEN_ROWS;
                    break;
                case RETURN:
                    // allow one more line
                    num_of_lines++;
                    break;
                case QUIT:
                    getmore = 0;
                    break;
                default:    // in case of invalid input
                    break;
            }
        }
```

```
        if ( fputs( line , stdout )  == EOF )
            exit(1);
        num_of_lines--;
    }
}

int get_user_input( FILE *fp )
//  Display message, wait for response, return key entered as int
//  Read user input from stream fp
//  Returns SPACEBAR, RETURN, QUIT, or INVALID
{
    int     c;

    printf("\033[7m more? \033[m");    // reverse on a VT100
    // Now we use getc instead of getchar. It is the same except
    // that it requires a FILE* argument

    while( (c = getc( fp )) != EOF )  // wait for response
        switch ( c ) {
            case 'q' :                // 'q' pressed
                return QUIT;
            case ' ' :                // ' ' pressed
                return SPACEBAR;
            case '\n' :               // Enter key pressed
                return RETURN;
            default :                 // invalid if anything else
                return INVALID;
         }
}
```

1.11.5 What Is Still Wrong?

The second version does not display the percentage of the file displayed. It still requires the user to press the *Enter* key after the space-bar and the '`q`', and the space characters and '`q`' are echoed on the screen, which the real `more` prevents. It still keeps displaying the old `more?` prompt instead of erasing it. It has hard-coded the number of lines in the terminal, so it will not work if the terminal has a different number of lines. So how do we get our version of `more` to behave like the real thing? There is something we have yet to understand, which is how the kernel communicates with the terminal.

Displaying the percentage is mostly a matter of logic. The real `more` prints the percentage of bytes of the file displayed so far, not the number of lines displayed. If you know the size of the file, you can calculate the percentage of bytes printed. That problem is easy to solve once we know how to obtain the size of a file programmatically.

Remember that `/dev/tty` is not really a file; it is an interface that allows the device driver for the particular terminal to run. That device driver allows us to configure the terminal, to control how

it behaves. We will see that we can use this interface to do things such as suppressing echoing of characters and transmitting characters without waiting for the *Enter* key to be pressed.

The problem of the `more?` prompt not disappearing is harder to solve, and also requires understanding how to control the terminal. One alternative is to learn more of the `VT100` escape sequences and use them to erase the `more?` prompt. Another alternative is to write copies of all lines to an off-screen buffer and clear the screen and then replace the screen contents whenever the user presses the *Enter* key. This is an easier solution than the first, but it is not how the real `more` command behaves.

1.11.6 A Third Attempt at the more Program

Our final version, not a correct version, will address some of the above problems.

The first problem is how we can determine the actual size of the terminal window. There are two solutions to this problem. If the user's environment contains the environment variables `COLUMNS` and `LINES`, then the program can use their values to determine the number of columns and lines in the terminal window, as suggested in the following code snippet that stores the `LINES` value in the variable `num_lines`.

```
int    num_lines;
char  *endptr;
char  *linestr = getenv("LINES");
if ( NULL != linestr ) {
    num_lines = strtol(linestr, &endptr, 0);
    if ( errno != 0) {
        /* handle error and exit */
    }
    if ( endptr == linestr ) {
        /* not a number so handle error and exit */
    }
}
```

The code is incomplete – one should fill in the error handling portions with the appropriate code. We will discuss error handling later. But the preceding code will not work if the `LINES` variable has not been put into the user's environment, and it will not work in the case that the user resizes the window after the program starts.

The better solution is to use a more advanced function, which will be covered in more depth in Chapter 4, named `ioctl()`. The function `ioctl()` is short for I/O Control. It is designed to do a great many things with peripheral devices, not just terminals. Its prototype is

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

Notice the "`...`" in the parameter list. This is C's notation for a variable number of parameters. The first parameter is a file descriptor. File descriptors are covered in Chapter 2. The second parameter, of integer type, is a command. The following parameters are arguments to the command, and they depend on the given command. The `ioctl()` call to get the number of rows and columns of the current terminal window has three arguments:

```
ioctl(tty_file_descriptor, TIOCGWINSZ, &window_structure))
```

where `tty_file_descriptor` is a file descriptor for our terminal, `TIOCGWINSZ` is a command that means "get the terminal window size structure" and `window_size_structure` is a variable of type `struct winsize`. The structure winsize is defined in a header file that is automatically included when you include the `<sys/ioctl.h>` header file in the code. The code that would use this `ioctl` is

```
struct winsize window_arg;
int    num_rows, num_cols;

fp_tty = fopen( "/dev/tty", "r" );
if ( fp_tty == NULL )
    exit(1);
if (-1 == ioctl(fileno(fp_tty), TIOCGWINSZ, &window_arg))
    exit(1);
num_rows =  window_arg.ws_row;
num_cols =  window_arg.ws_col;
```

The function `fileno()` used inside the `ioctl` is covered in Chapter 2; it converts a `FILE*` to a *file descriptor*. File descriptors are also explained in Chapter 2. We will use the following `get_tty_size()` function in our last version of the `do_more_of()` and `get_user_input()` functions:

```
void get_tty_size(FILE *tty_fp, int* numrows, int* numcols)
{
    struct winsize window_arg;
    if (-1 == ioctl(fileno(tty_fp), TIOCGWINSZ, &window_arg))
exit(1);
    *numrows =  window_arg.ws_row;
    *numcols =  window_arg.ws_col;
}
```

Although the `ioctl()` function is standard and should be available on any UNIX system, the particular commands that it implements may not be. In particular, it is possible that the `TIOCGWINSZ` command is not available. When you write code that depends on features that are not guaranteed to be present, it is best to conditionally compile it by enclosing it within preprocessor directives to conditionally compile. The preceding function is better written using

```
void get_tty_size(FILE *tty_fp, int* numrows, int* numcols)
{
#ifdef TIOCGWINSZ
    struct winsize window_arg;
    if (-1 == ioctl(fileno(tty_fp), TIOCGWINSZ, &window_arg))
exit(1);
    *numrows =  window_arg.ws_row;
    *numcols =  window_arg.ws_col;
```

```
#else
    /* some fallback code here */
#endif
}
```

We will omit the fallback code for now, since it requires knowing more about terminals.

The second problem is how to remove the old `more?` prompt so that it does not scroll up the screen. In short we need a way to clear portions of the screen. We also need a way to move the cursor to various places on the screen. There are various ANSI escape sequences that do these things. The table below lists a small fraction of the set of ANSI escape sequences. The ones in the table are the easiest to learn and use. In the table, the symbol "`ESC`" is short for '`\033`'. In almost all cases, the default value of n in the table is 1.

Key Sequence	**Meaning**
`ESC[M`	Move the cursor up in scrolling region.
`ESC[`n`A`	Move the cursor up n lines.
`ESC[`n`B`	Move the cursor down n lines
`ESC[`n`C`	Move the cursor right n columns
`ESC[`n`D`	Move the cursor left n columns
`ESC[`n`G`	Move the cursor to column n
`ESCE`	Move cursor to start of next line
`ESC[`r`;`c`H`	Move the cursor to row r, column c.
`ESC[0K`	Erase from the cursor to the end of the line
`ESC[1K`	Erase from the beginning of the line to the cursor.
`ESC[2K`	Erase the line
`ESC[0J`	Erase from the cursor to the end of the screen.
`ESC[1J`	Erase from the bottom of the screen to the cursor
`ESC[2J`	Erase the screen
`ESC[0m`	Normal characters
`ESC[1m`	Bold characters
`ESC[4m`	Underline characters.
`ESC[5m`	Blinking characters
`ESC[7m`	Reverse video characters
`ESC[g`	Clear tab stop at current column.
`ESC[3g`	Clear all tab stops.

Examples

Following are a few examples of sequences of commands. The last two are compound control sequences, and they are of signficance because we can use them in our program.

Sequence	**Meaning**
`\033[2A`	Move the cursor up two lines.
`\033[2J`	Clear the entire screen.
`\033[24;80H`	Move the cursor to row 24, column 80.
`\033[1A\033[2K\033[1G`	Move up one line, erase that line, and move the cursor to the leftmost column.
`\033[1A\033[2K\033[1B\033[7D`	Move the cursor up one line; erase that line; move the cursor back down one line, and move it to the left 7 positions.

We can combine the formatting features of the `printf()` function with our `get_tty_size()` function and these escape sequences to park the cursor in the lower left hand corner of the screen, regardless of how large the screen is, and display the prompt in reverse video:

```
int   tty_rows;
int   tty_cols;
get_tty_size(fp, &tty_rows, &tty_cols);
printf("\033[%d;1H", tty_rows);
printf("\033[7m more? \033[m");
```

The first `printf()` moves the cursor to the last row of the screen in the first column. The second displays the prompt in reverse video and then reverts the terminal back to normal character display. The third version of `more`, in Listing 1.7 below, includes these two improvements. The main program does not change, so it is not listed.

Listing 1.7: A third version of the more program.

```
#include  "more_utils.h"
#include  <sys/ioctl.h>

#define   LINELEN 512

void get_tty_size(FILE *tty_fp, int* numrows, int* numcols)
{
    struct winsize window_arg;

    if (-1 == ioctl(fileno(tty_fp), TIOCGWINSZ, &window_arg))
        exit(1);
    *numrows =  window_arg.ws_row;
    *numcols =  window_arg.ws_col;
}

/** get_user_input(FILE *fp )
 *  Displays more's status and prompt and waits for user response,
 *  Requires that user press return key to receive input
 *  Returns one of SPACEBAR, RETURN, or QUIT on valid keypresses
 *  and INVALID for invalid keypresses.
 *  Reads from fp instead of from standard input.
 */
int get_user_input( FILE *fp )
{
    int   c;
```

```
    int    tty_rows;
    int    tty_cols;

    /*
     * Get the size of the terminal window dynamically, in case it changed.
     * Then use it to put the cursor in the bottom row, leftmost column
     * and print the prompt in "standout mode" i.e. reverse video.
     */
    get_tty_size(fp, &tty_rows, &tty_cols);
    printf("\033[%d;1H", tty_rows);
    printf("\033[7m more? \033[m");


    /* Use fgetc() instead of getc().  It is the same except
     * that it is always a function call, not a macro, and it is in general
     * safer to use.
     */
    while ( (c = fgetc( fp )) != EOF )  {
        /* There is no need to use a loop here, since all possible paths
         * lead to a return statement. It remains since there is no downside
         * to using it.
         */
        switch ( c ) {
            case 'q' :                  /* 'q' pressed */
                return QUIT;
            case ' ' :                  /* ' ' pressed */
                return SPACEBAR;
            case '\n' :                 /* Enter key pressed */
                return RETURN;
            default :                   /* invalid if anything else */
                return INVALID;
         }
    }
    return INVALID;
}

/** do_more_of ( FILE * fp )
 *  Given a FILE* argument fp, display up to a page of the
 *  file fp, and then display a prompt and wait for user input.
 *  If user inputs SPACEBAR, display next page.
 *  If user inputs RETURN, display one more line.
 *  If user inputs QUIT, terminate program.
 */
void do_more_of( FILE *fp )
{
    char    line[LINELEN];              // buffer to store line of input
    int     num_of_lines;               // number of lines left on screen
    int     reply;                      // input from user
    int     tty_rows;                   // number of rows in terminal
    int     tty_cols;                   // number of columns in terminal
    FILE    *fp_tty;                    // device file pointer

    fp_tty = fopen( "/dev/tty", "r" );  // NEW: FILE stream argument
    if ( fp_tty == NULL )               // if open fails
```

```
        exit(1);                            // exit

    /* Get the size of the terminal window */
    get_tty_size(fp_tty, &tty_rows, &tty_cols);
    num_of_lines = tty_rows;

    while ( fgets( line, LINELEN, fp ) ){
        if ( num_of_lines == 0 ) {
            // reached screen capacity so display prompt
            reply = get_user_input( fp_tty );  // note call here
            switch ( reply ) {
                case SPACEBAR:
                    // allow full screen
                    num_of_lines = tty_rows;
                    printf("\033[1A\033[2K\033[1G");
                    break;
                case RETURN:
                    // allow one more line
                    printf("\033[1A\033[2K\033[1G");
                    num_of_lines++;
                    break;
                case QUIT:
                    printf("\033[1A\033[2K\033[1B\033[7D");
                    return;
                default:    // in case of invalid input
                    break;
            }
        }
        if ( fputs( line, stdout ) == EOF )
            exit(1);
        num_of_lines--;
    }
}
```

This version still has a few deficiencies. One is that it checks whether the terminal has been resized in each iteration of the loop, whenever it retrieves the user input. Although this works, it wastes cycles. Later we will learn how to do this asynchronously, finding the size of the terminal only when it has actually been resized.

Another deficiency is that it still requires the user to press the Enter key. Remedying this requires further understanding of terminals, again in Chapter 4. One feature we can easily integrate into the program is calculating the percentage of the file displayed so far. This is left as an exercise.

1.12 Where We Go from Here

The real purpose of trying to write the `more` program is to show that, using only the usual high-level I/O libraries, we cannot write a program that does the kind of things that `more` does, such as ignoring keyboard input and suppressing display of typed characters. The objective of these notes is to give you the tools for solving this kind of problem, and to expose you to the major components of the kernel's API, while also explaining how the kernel looks "under the hood." We are going to look at each important component of the kernel. You will learn how to rummage around the file system and man pages for the resources that you need.

Appendix A Brief History of UNIX

In 1957, Bill Norris started Control Data Corporation (CDC). In 1959 Ken Olsen started DEC with $70,000 in venture capital money. The first PDP-1 (manufactured by DEC) was shipped in 1960. In 1963, Project MAC (Multiple Access Computers) was organized at MIT to do research on interactive computing and time-sharing systems. In 1965, AT&T, GE, and Project MAC at IBM joined together to develop the time-sharing system MULTICS (Multiplexed Information and Computing Service).

In 1966, Ken Thompson finished studies at University of California at Berkeley (UCB) and joined the technical staff at AT&T Bell Telephone Laboratories to work on MULTICS. Two years later, Dennis Ritchie completed work on his doctorate at Harvard and joined Bell Labs to work on MULTICS project.

Thompson developed the interpretive language B based upon BCPL. Ritchie improved on "B" and called it "C".

In 1969 while working for AT&T Bell Labs, Ken Thompson and Dennis Ritchie wrote the first version of UNICS for a PDP-7, manufactured by DEC. It ran on a machine with 4K of 18-bit words. UNICS is a pun on MULTICS and stood for Uniplexed Information and Computing Services. Three years later, it was rewritten in C so that they could port it to other architectures. By 1973, the first UNIX support group was formed within Bell Labs, and some fundamental UNIX design philosophies emerged. Ritchie and Thompson gave a presentation about the new operating system at the *ACM Symposium on Operating Systems* at IBM that year. This was the catalyst that sparked the spread of the UNIX system. The University of California at Berkeley (UC Berkeley) soon acquired Version 4 from Bell Labs and embarked on a mission to add modern features to UNIX.

Over the next four years, a number of events shaped the future of UNIX. AT&T started licensing it to universities. Boggs and Metcalfe invented the Ethernet at Xerox in Palo Alto. Bill Joy joined UC Berkeley and developed BSD Version 1. UNIX was ported to a non-DEC machine. (DEC had been unwilling to support UNIX.) P.J. Plauger wrote the first commercial C compiler, and Doug and Larry Michel formed a company called Santa Cruz Operations (SCO) to start selling a commercial version of UNIX for the PC.

In 1974, "The UNIX Time-Sharing System" was published in CACM by Ken Thompson and Dennis Ritchie. That same year, the University of California at Berkeley (UCB) got Version 4 of UNIX, and Keith Standiford converted UNIX to a PDP 11/45. *The Elements of Programming Style* by Kernighan and Plauger was published, and AT&T officially began licensing UNIX to universities.

In 1976, *Software Tools* by Kernighan and Plauger was published, and Boggs and Metcalfe invented the Ethernet at Xerox in Palo Alto.

By 1978, UNIX had essentially all of the major features that are found in it today. One year later, Microsoft licensed UNIX from AT&T and announced XENIX (even before producing MS-DOS).

UNIX spread rapidly because:

- It was easily ported from one architecture to another because it was written in a high level language (C), unlike other operating systems.

- It was distributed to universities and research laboratories for free by AT&T, and early commercial versions were sold at very low cost to academic institutions.
- The source code was freely available, making it easy for people to add new features and programs to their systems.
- It had several features (e.g., pipes) that had not been part of other operating systems.

In 1979, Bell Labs released UNIX™ Version 7, attempting to stake proprietary rights to UNIX, and Microsoft licensed UNIX from AT&T and announced XENIX (even before producing MS-DOS). In the early 1980s, a number of developments occurred that shaped the future of the UNIX evolutionary tree:

- AT&T and Berkeley (the Computer Systems Research Group, or CSRG) each started to develop their own independent versions of UNIX.
- Bill Joy left Berkeley to co-found a new company, SUN Microsystems, to produce UNIX workstations. Sun got its name from the Stanford University Network (SUN) board. The workstation was based on the Motorola 68000 chip running SunOS based on 4.2BSD. It included an optional local area network based on Ethernet. The commercial UNIX industry was in full gear.
- Microsoft shifted its focus on the development of MS-DOS, shelving marketing of XENIX.
- The X/Open standards group was formed.

Over the next fifteen years, many different versions of UNIX evolved. Although the differences between these versions are not major, care must be taken when trying to treat "UNIX" as a single operating system. Through the 1980s and early 1990s, the two major UNIX distributions were the Berkeley series known by names such as 4.2BSD and 4.3BSD and the AT&T series known by names such as System III and System V. SUN's UNIX was called SunOS, and later Solaris, and was a hybrid of the two strains.

Since its creation in 1969, UNIX has grown from a single, small operating system to a collection of distinct operating systems of varying complexity, produced by vendors all around the world. The essential features of most UNIX systems have remained the same, partly because of standardization efforts that began in the late 1980's and continued through the 1990's. Modern UNIX is a slippery concept. Several standards have been written, and they have a large common core. What is generally called UNIX is the common core of these standards, which include POSIX, and UNIX 98 from the Open Group. UNIX 98 is also called the Single UNIX Specification and it alone is UNIX as far as property rights are concerned, since the Open Group owns the UNIX trademark now. The Open Group is an international consortium of more than 200 members from government, academia, worldwide finance, health care, commerce and telecommunications. Of course, there is also Linux, which has overtaken BSD and System V on the desktop.

Appendix B UNIX at the User Level

This section is designed for people with little, if any, experience with UNIX. It covers:

- Logging in and out
- Shells
- The environment

B.1 Notation

In the description of a command, square brackets `[ ]` enclose optional arguments to the command, and angle brackets `< >` enclose placeholders. The brackets are not part of the command. A vertical bar `"|"` is a logical-or. An ellipsis `...` means more than one copy of the preceding token. For example

```
ls [<option>] ... [<directory_name>] ...
```

indicates that both the option specifiers and the argument to the `ls` command are optional, but that any options should precede any directory names, and that both option specifiers and directory names can occur multiple times, as in

```
ls -l -t mydir yourdir
```

Commands will be preceded by a dollar sign to indicate that they are typed after the shell prompt. When the input and output of a command are shown, the user's typing will be in **bold**, and output will be in regular text.

```
$ ls -l -t mydir yourdir
```

B.2 Logging In

A user uses a UNIX system by logging in, running programs, and logging out. There are two different ways to login to a UNIX system:

- When sitting at the console of the computer, i.e., the computer keyboard and monitor are physically in front of you, and no one is presently logged in to the machine, or
- Remotely, by using a remote login utility like SSH.

At the Console

If you login to the computer at its own console, the appearance of the screen will depend upon which version of UNIX you use. In all cases, there will be an option to enter a user or login name followed by a password. For RedHat 9 with the Gnome 2 desktop environment, the screen will appear as in Figure B.1. After logging in, you will see a desktop with a graphical user interface, and a menu somewhere on the screen with an option somewhere to open a terminal window. It is within that terminal window that the rest of the notes pertain.

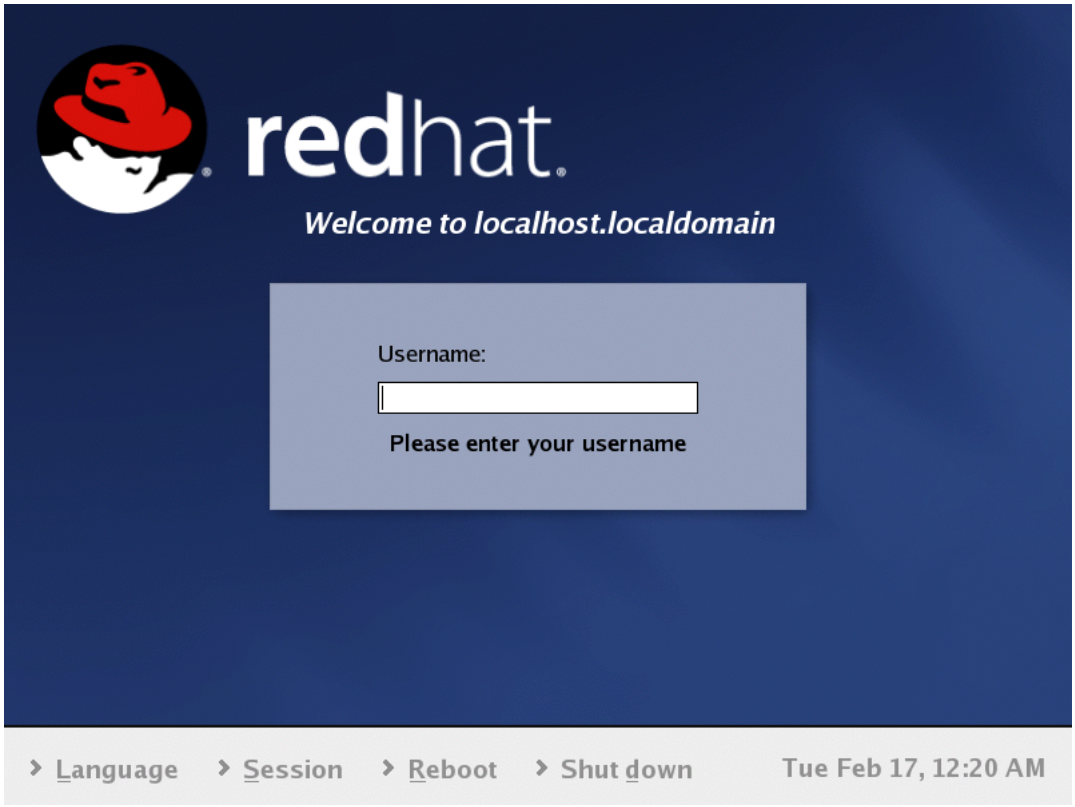


Figure B.1: RedHat 9 login screen.

Remotely via SSH

SSH is a protocol that provides encrypted communication to prevent passwords and other secure information from being captured in transit over insecure networks. SSH stands for *Secure SHell*. SSH is a client/server application. The server runs on the remote host, the one into which you want to login, and the client is on your local host. Once you have established a connection to the UNIX system from your local machine using SSH, everything that you type is encrypted by the SSH client on your local machine and decrypted by the remote host. The encryption algorithms are very strong and the keys are very secure.

These notes do not describe how to use any particular SSH client. All such clients will need from you the IP address of the remote host, the login or user name, and your password. Assuming that you have provided these, the SSH client will log you in and display a terminal window on your local machine.

You will see some messages, followed by the prompt displayed by your *login shell*. A login shell is the shell that runs when you log-in. It depends on the particular version of UNIX you are using and how the initial messages were configured by the system administrator. On the Hunter College Computer Science Department's network gateway machine, named `eniac`, which was running Solaris 9, a version of UNIX from SunSoft, it looked like:

```
Last login: Mon Aug 30 2004 19:51:19 -0500 from pool-68-161-100-
Sun Microsystems Inc.   SunOS 5.9       Generic October 1998
This workstation is running SunOS 5.x
eniac{sweiss} [42] $
```

When eniac was later converted to a Linux host, the login messages became:

```
Last login: Fri Jan 20 17:21:53 2006
from pool-68-161-21-160.ny325.east.verizon.net
eniac{sweiss} [339] $
```

In both cases, the last line begins with a *prompt string*, which is a sequence of characters followed by a waiting cursor. In each of these the prompt string looks like

```
eniac{sweiss} [n] $
```

where `n` is some number. The prompt is sometimes "`$`" alone. It depends on how it has been configured, which partly depends upon the shell. Some shells let you do more configuring than others. This prompt displays the name of the host computer, the user-name on that host, and the *history number* (42 or 339) of the current command. The history number is the ordinal number that indicates how many commands you have entered so far. It gets reset to 1 when it reaches the maximum number of saved commands. One can retrieve old commands using their numbers.

Your *login shell* is the special shell that is started up each time you login. On UNIX systems with a windowing environment, each window will have a separate instance of a running shell. Even UNIX systems with a GUI follow the shell paradigm – instead of typing commands, you use a pointing device and click on icons and controls, but the effect is still to construct calls to system programs, to a shell, or to shell scripts.

When you have logged-in, you will be "in" your *home directory*. You might feel comfortable with the idea of "being in a directory" but it makes no sense to say this. You are in your seat. You are not in the computer. Nonetheless, this language has taken hold. You probably know that "being in the home directory" means that when you view the contents of the current directory, you will see the files in your home directory. But this is an inadequate explanation. I will have more to say about the concept of "being in a directory" shortly. What you do need to understand now is that the home directory is the directory that you "are in" each time you log-in. It is the starting place for your current session. The `HOME` environment variable stores the "name" of your home directory. This will also be clarified shortly.

B.3 Logging Out

When you are finished using the system, you log out. There are usually a few ways to log out. POSIX specifies that a shell provide a built-in `exit` command, which can be used to exit any shell. There is also the `logout` command, which may or may not be built-into the shell:

```
eniac{sweiss} [342]$ logout
```

The difference is that logout can only be used to log out, not to terminate a shell that is not a login shell, such as a sub-shell of some other shell, whereas exit terminates all shells and logs you out if the shell is a login shell.

B.4 Online Help: The man Pages

Before going any further, you should understand how to get help online. Traditional UNIX provides only one source of online help – the manual pages. The single most important command to remember is the man command. The word "man" is short for "manual"; if you type man following by the name of a command, UNIX will display the manual page, called the man page for short, for that command. For example, to learn more about a command named echo, type

```
$ man echo
```

and you will see several screens of output, beginning with:

```
echo(1)                   User Commands                        echo(1)
NAME
       echo - display a line of text

SYNOPSIS
       echo [SHORT-OPTION]... [STRING]...
       echo LONG-OPTION

DESCRIPTION
       Echo the STRING(s) to standard output.
       -n     do not output the trailing newline
       -e     enable interpretation of backslash escapes
       -E     disable interpretation of backslash escapes (default)
       (remaining lines omitted )
:
```

Every man page is in the same format, which may vary from one UNIX system to another. The first line shows the name of the section of the manual in which the page was found (*User Commands*) and the name of the man page followed by the section number (*echo, section 1*). Sometimes the name of the man page will be different from the name of the command. The sections of the man page, which may not all be present, are:

NAME	name of the command
SYNOPSIS	syntax for using the command
DESCRIPTION	brief textual summary of what the command does
OPTIONS	precise descriptions of command-line options
OPERANDS	precise descriptions of command-line arguments
USAGE	a more thorough description of the use of the command
ENVIRONMENT VARIABLES	list of environment variables that affect the command execution
EXIT STATUS	list of exit values returned by the command
FILES	list of files that affect the command execution
ATTRIBUTES	architectures on which it runs, availability, code independence, etc.
SEE ALSO	list of commands related to this command
NOTES	general comments that do not fit elsewhere
BUGS	known bugs

In man pages, the square bracket notation `[ ]` as in `[SHORT-OPTION]` above defines an optional command-line option or argument. The ":" is followed by your cursor because many UNIX systems now pipe the output of the man command through the `less` command for viewing, and the ":" is the `less` prompt to the user to type something on the keyboard. More accurately, the `less` command is the viewer for the man command itself[1].

For learning how to use a command, the man page by itself is usually sufficient. For learning how a command interacts with the operating system or how it might be implemented, one must do more research. In the next chapter, I will explain how to use the man pages in more detail for the latter purpose. In the meanwhile, you should look at the man pages of the various commands mentioned in this chapter as you read on.

It should also be mentioned that modern UNIX systems also provide other sources of help, such as the help and info commands. If the system you are using has these commands, typing help or info followed by a command or a topic will display information about that topic.

B.5 Shells

Your view and appreciation of UNIX is pretty much determined by the interface that the shell creates for you, as suggested in Figure B.2. The shell hides the inner workings of the kernel, presenting a set of high level functions that can make the system easy to use. Although you may have used a UNIX system with a graphical user interface, you must be aware that this GUI is an application separate and distinct from UNIX. The GUI provides an alternative to a shell for interacting with UNIX, but experienced users usually rely on using a shell for many tasks because it is much faster to type than it is to move a mouse around pointing and clicking.

There are many different shells, including the *Bourne* shell (`sh`), the *C* shell (`csh`), the *Korn* shell (`ksh`), the *Bourne-again* shell (`bash`), the *Z* shell (`zsh`), and the *TC* shell (`tcsh`). Although a few shells may be bundled with the operating system, they are not actually a part of it.

Unlike most operating systems, UNIX allows the user to replace the original login shell with one of his or her own choosing. The particular shell that will be invoked for a given user upon login is

[1] You can change which viewer the man command uses by changing the value of the PAGER environment variable.

Figure B.2: The shell: A user's view of UNIX.

specified in the user's entry in a file called the *password file*. The system administrator can change this entry, but very often the `chsh` command available on some systems can also be used by the user to change shells. In some versions of UNIX, the user can also use the command `passwd -s` to change the login shell.

B.5.0.1 Shell Features

In all shells, a simple command is of the form

```
$ commandname command-options arg1 arg2 ... argn
```

The shell waits for a newline character to be typed before it attempts to interpret (parse) a command. A newline signals the end of the command. Once it receives the entered line, it checks to see if `commandname` is a *built-in shell command* (A built-in command is one that is hard-coded into the shell itself.) If it is, it executes the command. If not, it searches for a file whose name, either relative or absolute, is `commandname`. (How that search takes place is explained in a later chapter.) If it finds one, this file is loaded into memory and the shell creates a child process to execute the command, using the arguments from the command line. When the command is finished, the child process terminates and the shell resumes its execution.

In addition to command interpretation, all shells provide

- redirection of the input and output of commands
- pipes – a method of channeling the output of one command to the input of another
- scripting – a method of writing shell programs that can be executed as files
- file name substitution using metacharacters
- control flow constructs such as loops and conditional execution

Shells written after the Bourne shell also provide

- history mechanism – a method of saving and reissuing commands in whole or in part

- backgrounding and job control – a method of controlling the sequencing and timing of commands
- aliases for frequently used commands

The `tcsh` shell added a number of features to the C shell, such as interactive editing of the command line and interactive file and command name completion, but the Korn shell introduced many significant additions, including:

- general interactive command-line editing
- coprocesses
- more general redirection of input and output of commands
- array variables within the shell
- autoloading of function definitions from files
- restricted shells
- menu selection

`bash` borrowed many of the features of its predecessors and has almost all of the above capabilities.

B.5.0.2 Standard I/O and Redirection

UNIX uses a clever method of handling I/O. Every program is automatically given three open files when it begins execution, called *standard input*, *standard output*, and *standard error*. (POSIX does not require standard error, but it is present in all systems that I know.) Standard input is by default the keyboard and standard output and error are to the terminal window.

Commands usually read from standard input and write to standard output. The shell, however, can "trick" a command into reading from a different source or writing to a different source. This is called *I/O redirection*. For example, the command

```
$ ls mydir
```

ordinarily will list the files in the given directory on the terminal. The command

```
$ ls mydir > myfiles
```

creates a file called `myfiles` and redirects the output of the `ls` command to `myfiles` provided `myfiles` did not already exist, in which case it will display a message such as

```
bash: myfiles: cannot overwrite existing file
```

The notation "`> outfile`" means "put the output of the command in a file named `outfile` instead of on the terminal." The *output redirection operator*, "`>`", replaces the standard output of a program by the the file whose name follows the operator.

Analogously, the notation "`< infile`" means read the input from the file `infile` instead of from the keyboard. Technically, the "`<`", known as the *input redirection operator*, replaces the standard input of a program by the file whose name follows the operator.

A command can have both input and output redirected:

```
$ command < infile > outfile
```

which causes the command to read its input from `infile` and send its output to `outfile`. Th order of the command, and the input and output redirection does not matter. One can also write any of the following semantically equivalent lines:

```
$ command > outfile < infile
$ > outfile command < infile
$ < infile > outfile command
```

The concept of redirection is carried one step further to allow the output of one command to be the input of another. This is known as a *pipe* or *pipeline*, and the operator is a vertical bar "`|`" :

```
$ ls mydir | sort | lpr
```

means list the contents of the given directory, and instead of writing them on the terminal or in a file, pass them as input to the `sort` command, which then sorts them, and sends the sorted list to the `lpr` command, which is a command to send files to the default printer attached to the system. It could have been done using temporary files as follows:

```
$ ls mydir > temp1
$ sort < temp1 > temp2
$ lpr < temp2
$ rm temp1 temp2
```

but in fact this is not semantically equivalent because when a pipeline is established, the commands run simultaneously. In the above example, the `ls`, `sort`, and `lpr` commands will be started up together. The shell uses a kernel mechanism called *pipes* to implement this communication. There are other kinds of I/O redirection, including appending to a file and redirecting the system error messages; for details consult the shell's man page.

There are two other I/O redirection operators: `<<` and `>>`. If you are curious and unfamiliar with these, read about them in the `bash` man page.

B.5.0.3 Command Separators and Multitasking

Commands can be combined on single lines by using command separators. The semicolon ";" acts like a newline character to the shell – it terminates the preceding command, as in:

```
$ ls mydir ; date
```

which lists the contents of the given directory and then displays the current time and date. The `date` command displays the date and time in various formats, but by default its output is in the form

```
Mon Aug 15 22:53:54 EDT 2011
```

The semicolon ";" is used to sequentially execute the commands. In contrast,

```
$ ls mydir > outfile &
```

causes the `ls` command to work "in the background." The ampersand "`&`" at the end of the line tells the shell not to wait for the command to terminate but to instead resume immediately; it is a *backgrounding* operator. The `ls` command runs in the background, sending its output to the file `outfile` while the shell does other things. This is a form of *multitasking*. The "`&`" can be used to run separate commands as well:

```
$ ls mydir > outfile1 & date > outfile2 &
```

tells the shell to run the `ls` and `date` commands *concurrently* and in the background, putting their respective outputs in files `outfile1` and `outfile2` respectively. Note that the ">" binds more closely, i.e., has higher priority, than "`&`". Lastly, parentheses can be used for grouping:

```
$ (ls mydir; date) > outfile &
```

means "execute `ls mydir`, then `date`, and direct their combined output to `outfile`," all in the background.

B.5.0.4 The Shell as a Command

Shells can be run as commands. You can type the name of a shell, e.g., `bash`, `sh`, `csh`, etc., at the command prompt within any other shell, to start another shell:

```
$ sh
```

in this case the Bourne shell. If you do this, you will have two instances of the `bash` shell running, but the first will be dormant, waiting for the second to exit. When one shell is created as a result of a command given in another shell, the created shell is called the sub-shell of the original shell, which is called the parent shell.

If you need to know what shell you are currently running (because you forgot, for example), there are two simple commands that can tell you:

```
$ echo $0
```

and

```
$ ps -p $$
```

The first uses the `echo` command. The `echo` command may sound at first rather useless, but it is very convenient. All it does is evaluate its arguments and display them. So writing

```
$ echo hello out there
hello out there
```

But the key is that `echo` evaluates its arguments. If you give it the name of a variable preceded by a dollar sign "`$`", it displays the value of the variable, not the name. As an example, you can use it to see the value of a particular environment variable, such as

```
$ echo $SHELL
/bin/bash
```

This does not print your current shell however. It prints the name of your login shell, which may not be the current shell. The value of `$0` is the name of the currently running program, which is the shell you are using, so `echo $0` is one way to see its name. The value of `$$` is the process-id of the currently running process, and `ps -p` is a command that will display information about the process whose process-id is given to it.

B.5.0.5 Scripts

Suppose the file named `myscript` contains the following lines:

```
ls mydir
date | lpr
```

Then the command

```
$ bash < myscript
```

causes the commands in the file to be executed as if they were typed directly to the `bash` shell, and when they have terminated, the `bash` shell exits. The file `myscript` is an example of a *shell script*. A script is nothing more than a program written in an interpreted language. It is not compiled, but executed line by line. Shell scripts can have command line arguments just like ordinary shell commands, making them very general. For example, the above script can be rewritten as

```
#!/bin/bash
ls $1
date | lpr
```

and executed by typing a command such as

```
$ myscript ~/bin
```

whereupon the "$1" will be replaced by ~/bin before the ls command is executed. The very first line indicates that the bash shell must be run to execute the remaining lines of the file. For more details of specific shell languages, consult the shell's man page (or read any of a multitude of on-line tutorials.)

Each shell has properties that determine how it behaves, which are stored in the shell's environment. As noted in Section 1.3.5, the operating system passes a pointer to a copy of the user's environment when it starts up a new process. Since a running shell is a process, each time a new shell is started from within another, it is given a copy of the previous shell's environment.

More generally, when any process is created from within a shell, that process inherits a copy of the values of the environment variables of the shell. When you run a program from the command line, the program inherits the environment of the shell. This is how programs "know" what their current working directories are, for example, or which users are their owners. The environment is a key element in making things work.

You are free to customize the environment of a shell by defining new variables, or redefining the values of existing variables. The syntax is shell-dependent. In the Bourne shell and bash, variables that you define, called *locals*, are not automatically placed into the environment; to place then in the environment, you have to export them. For example,

```
$ export MYVAR=12
```

will put MYVAR into the environment with the value 12, so that it can be inherited by future commands and processes, or equivalently:

```
$ MYVAR=12; export MYVAR
```

The convention is to use uppercase names for environment variables and lowercase names for locals.

B.5.0.6 Some Historical Remarks About Shells

The C shell was written by Bill Joy at UC Berkeley and made its appearance in Sixth Edition UNIX (1975), the first widely distributed release from UC Berkeley. It was an enhancement of the original Thompson shell with C-like syntax. It is part of all BSD distributions.

The Bourne shell, written by Stephen Bourne, was introduced into UNIX in System 7 (1979), which was the last release of UNIX by AT&T Bell Labs prior to the commercialization of UNIX by AT&T. Its syntax derives in part from the programming language Algol 68 and is a part of all UNIX releases.

The Korn shell was developed by David Korn at Bell Labs and introduced into the SVR4 commercial release of UNIX by AT&T. Its syntax is based on the Bourne shell but it had many more features.

The TENEX C shell, or TC shell, extended the C shell with command line editing and completion, as well as other features which were found in the TENEX operating system. It was written by Ken Greer and others in 1983.

The Bourne-again shell (Bash) is an extension of the Bourne shell with many of the sophisticated features of the Korn shell and the TC shell. It is the default user shell in Linux and has become popular because of this.

Bibliography

[1] Bruce Molay. *Understanding UNIX/LINUX Programming: A Guide to Theory and Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[2] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

Chapter 2 Login Records, File I/O, and Performance

Concepts Covered

Man pages and Texinfo pages
The UNIX file I/O API
Reading, creating, and writing files
File descriptors
Kernel buffering
Kernel versus user mode and the cost of system calls
Timing programs
Time representation in UNIX
The utmp file
Detecting and reporting errors in system calls
Memory-mapped I/O,
Feature test macros
open, creat, close, read, write, lseek, perror, ctime, localtime, utmpname, getutent, setutent, endutent, malloc, calloc, mmap, munmap, memcpy
Filters and regular expressions

2.1 Introduction

This chapter introduces the two primary methods of I/O possible in a UNIX: *buffered* and *unbuffered*. By trying to write the `who` and `cp` commands, we will learn explore how to create, open, read, write, and close arbitrary files. "Arbitrary" in this context means that they are not necessarily text files. We will write several different versions of the `who` command, simply to illustrate different approaches to the problem of reading from a file. They will differ in their performance characteristics and their portability. The chapter uses this exercise to introduce the UNIX concept of time, and the first of several important databases provided by the kernel, as well as the kernel's interface to those databases. We also write two different versions of a simplified `cp` command, one using `read()` and `write()`, and the other using memory-mapped I/O.

2.2 Commands Are (Usually) Programs

In UNIX, most commands are programs, almost always written in C. Some commands are not programs; they are built into the shell and therefore are called *shell builtins*. Exactly which commands are builtins varies from one shell to another[1], but there are some that are common to almost all shells, such as `cd` and `exit`. When you type `cd`, for example, the shell does not run the `cd` program; it jumps to the internal code that implements the `cd` command itself. You can think of the shell as containing a C `switch` statement inside a loop. When it sees that the command is a built in, it jumps to the code to execute it. Some commands, such as `pwd`, are both shell builtins and programs. By default the shell built in will be executed if the user types `pwd`; to get the program version, one can either precede the command with a backslash "\", as in `\pwd`, or type the full path name, `/bin/pwd`.

[1]The list of built-in commands is usually provided in the shell's man page. For example, the command `man builtins` will display the `bash_builtins` man page, and at the very top of that page is the complete list of bash builtins.

Command programs are located in one of several directories, the most common being `/bin`, `/usr/bin`, and `/usr/local/bin`. The `/usr/local/bin` directory is traditionally used as a repository for commands that do not come with the UNIX distribution and have been added as local extras. Many packages that are installed after the operating system installation are placed in subdirectories of `/usr/local`. Administrative commands, such as those for creating and modifying user accounts, are found in `/usr/sbin`. Many UNIX systems still retain the old `/usr/ucb` directory. (The "ucb" in `/usr/ucb` stands for the University of California at Berkeley. The `/usr/ucb` directory, if it exists, contains commands that are part of the BSD distributions. Some of the commands in `/usr/ucb` are also in `/usr/bin` and have different semantics. If the same command exists in both `/usr/bin` and in some other directory such as `/usr/ucb`, the `PATH` environment variable just like the one used in Windows and DOS, determines which command will be run. The `PATH` variable contains a list of the directories to search when the command is typed without a leading path. Whichever directory is earliest in the list is the one whose version of the command is used. Thus, if `more` exists in both `/usr/ucb` and `/usr/bin`, as well as in your working directory, and `/usr/bin` precedes `/usr/ucb` which precedes "`.`" in your `PATH` variable, and if you type

```
$ more myfile
```

then `/usr/bin/more` will run. If instead you type

```
$ ./more myfile
```

then your `PATH` is not searched and your private `more` program will run. If you type

```
$ /usr/ucb/more myfile
```

then your `PATH` is not searched and `/usr/ucb/more` will run.

2.3 The `who` Command

There are a few different commands for checking which users are currently using the system. The simplest of these is conveniently named `who`[2]. Other commands that perform similar tasks are `w`, `users`, and `whodo`[3]. The `who` and `w` commands are required by the POSIX standard, so they are more likely to be on a UNIX installation.

The `who` command displays information about who is currently using the system. Running `who` without command-line options produces a listing such as

```
dsutton    pts/1    Jul 23 20:22   (66-108-62-189.nyc.rr.com)
ioannis    pts/2    Jul 24 16:53   (freshwin.geo.hunter.cuny.edu)
dplumer    pts/3    Jul 26 11:34   (66-65-53-41.nyc.rr.com)
rnoorzad   pts/4    Jul 23 09:25   (death-valley.geo.hunter.cuny.edu)
rnoorzad   pts/5    Jul 23 09:25   (death-valley.geo.hunter.cuny.edu)
sweiss     pts/6    Jul 26 13:08   (70.ny325.east.verizon.net)
```

[2]This is unusual. Most UNIX commands have names that are so cryptic that you have to be a wizard to guess their names. Would you have guessed, for example, that to view the contents of a directory, you have to type "`ls`" or that to view the contents of a file you can type "`cat`"?

[3]`whodo` is not available in Linux. It is found in Solaris, AIX, and other UNIX variants.

Each line represents a single login session. The `-H` option will print column headings, in case the data is not obvious. The first column is the username, the second is the terminal line on which the user is logged in, the third is the time of the login on that terminal, and the last is the source of the login, either the host name or an X display. For example, `sweiss` was logged in on terminal line `pts/6`, the session started at 13:08 on July 26th of the then current year, and the login was initiated from a computer identified as `70.ny325.east.verizon.net`. Notice that there may be multiple logins with the same username.

The output of `who` may vary from one system to another. Some of the reasons have to do with how systems treat users who have multiple terminal windows open in a single login or are running terminal multiplexers such as Gnu's `screen` program. The `w` command, by the way, is approximately equivalent to the command sequence "`uptime; who`"; it shows more information than `who` does.

2.4 Researching Commands In UNIX

UNIX is a self-documented operating system. You can use UNIX itself to learn how it works if you do a thorough exploration of the online documentation. In particular, the man pages can be a source of information about how a command might be implemented. This information is not explicit, but can be obtained by using clues within the page. The man page for a command may not have enough content, and will instead have a message such as the following in the `SEE ALSO` section at the bottom:

```
The full documentation for who is maintained as a Texinfo manual.
If the info and who programs are properly installed at your site,
the command
    info coreutils 'who invocation'
should give you access to the complete manual.
```

In this case, one should use the `info` command instead. The `info` command brings up the *Texinfo* pages. The *Texinfo* system is an alternative system for providing on-line documentation. To learn how to use the Texinfo viewer, type

```
info info
```

which will bring up a tutorial on using the Texinfo documentation system. The general idea is that the information is stored in a tree-like structure, in which an internal node represents a topic area, and its child nodes are specific to that topic. The space bar will advance within the entire tree using breadth-first search. To descend into a node's children, `d` (for `down`) works. To go back up, `u` (for `up`) works. To traverse the siblings from left to right, `n` (for `next`) does the trick, and to go back, `p` (for `previous`) works. Just picture the tree.

Note. On some systems, when you type "`info coreutils who`" , you will see the page for the `whoami` command. If you move ahead a few pages, you will find the page for `who`. On other systems you may have to type "`info who`" or "`info coreutils 'who invocation'`" to bring up the proper pages.

The man page for `who` tells us that the command may be called with zero or more of the command-line options `abdHlmpqrstTu`. It can also be called as follows:

```
$ who am i
sweiss pts/6 Jul 26 13:08 (70.ny325.east.verizon.net)
```

and, in Linux, if you supply any two words after "`who`", it behaves the same way:

```
$ who you think
sweiss pts/6 Jul 26 13:08 (70.ny325.east.verizon.net)
```

In general, the way to research a UNIX command is to use a combination of these methods:

1. Read the relevant man page.
2. Follow the `SEE ALSO` links on the page.
3. Read the Texinfo page if the man page refers to it.
4. Search the manual.
5. Find and read the header (.h) files relevant to the command.

2.4.1 Reading Man Pages

There is no standard that defines what must be contained in most man pages; it is implementation-dependent. However, most systems follow a time-honored convention for man pages in general, which is what we describe in these notes. For the purpose of understanding how a command works, the relevant sections of the man page for that command are the `DESCRIPTION`, `SEE ALSO`, and `FILES` sections.

The `DESCRIPTION` section gives the details of how the command is used. For example, reading about `who` in the man page reveals that `who` has an optional file name argument, and that if it is not supplied, `who` reads the file `/var/run/utmp` to get the information about current logins. The optional argument can be `/var/log/wtmp`. We can infer that the file `/var/run/utmp` contains information about who is currently logged in. What about `/var/log/wtmp`? If you were to try typing

```
$ man wtmp
```

you would be pleasantly surprised to discover that, although `wtmp` is not a command, there is a man page that describes it. This is because there is a section of the man pages strictly devoted to the description of system file formats. `/var/log/wtmp` is a system file, as is `/var/run/utmp`, and they are both described on the same man page in section 5 of the manual. There we can learn that `/var/log/wtmp` contains information about who has logged in previously[4].

Before we dig deeper into the man page for the `utmp` and `wtmp` files, you should also know that it is required of all POSIX-compliant UNIX systems that they also contain man pages for all of the header files that might be included by a function in the kernel's API. To put it more precisely, each function in the System Interfaces volume of POSIX.1-2008 specifies the headers that an application must include to use that function, and a POSIX-compliant system must have a man page for each of those headers. They may not be installed on the system you are using, but they are available. They will only be installed if the system administrator installed the application development files.

The man pages for the header files have a fixed format. From the POSIX.1-2008 standard:

[4] If we consult the `who` Texinfo page, we could learn that as well.

`NAME`

This section gives the name or names of the entry and briefly states its purpose.

`SYNOPSIS`

This section summarizes the use of the entry being described.

`DESCRIPTION`

This section describes the functionality of the header.

`APPLICATION USAGE`

This section is informative. This section gives warnings and advice to application developers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of POSIX.1-2008, the normative material is to be taken as correct.

`RATIONALE`

This section is informative. This section contains historical information concerning the contents of this volume of POSIX.1-2008 and why features were included or discarded by the standard developers.

`FUTURE DIRECTIONS`

This section is informative. This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

`SEE ALSO`

This section is informative. This section gives references to related information.

The important sections are `NAME`, `SYNOPSIS`, `DESCRIPTION`, and `SEE ALSO`.

For example

```
$ man stdlib.h
```

will display the man page for the header file `<stdlib.h>`. This is a useful feature. But if you do not know the name of the command that you need, nor the names of any files that might be useful or relevant, then you do not know which man page to read. UNIX systems provide various methods of overcoming this problem.

2.4.2 Man Page Searching

The most basic solution, guaranteed to work on all systems, is to use the search feature of the `man` command. To search for all man pages that contain a particular keyword in their one-line summaries in the `NAME` Section, you can type

```
$ man -k keyword
```

This will only work if the `whatis` database has been built when the man pages were installed however, so you are at the mercy of the system administrator[5]. For example, typing

[5] If you are the administrator, issue the command `/usr/sbin/makewhatis` to build the database.

```
$ man -k utmp
```

will list all man pages that contain the string `utmp` in their summaries. The command

```
$ apropos utmp
```

has the exact same meaning: `apropos` is equivalent to "`man -k`". Unfortunately, the implementation of `apropos` varies from system to system. On some systems, such as Fedora 15, the most current stable version, `apropos` has features that allow multiple keyword searches as well as regular expression searches. To search for man pages whose page names and/or `NAME` sections contain all keywords provided, one can use the `-a` option, as in

```
$ apropos -a convert case
toupper      (3)     - convert letter to upper or lower case
FcToLower    (3)     - convert upper case ASCII to lower case
tolower      (3)     - convert letter to upper or lower case
towlower     (3)     - convert a wide character to lowercase
towupper     (3)     - convert a wide character to uppercase
XConvertCase (3)     - convert keysyms
```

The number in parentheses is the section number. Section 3 contains man pages for library functions. Notice that we have output in which the string "case" is a substring of other words. If we wanted to limit it to those descriptions in which "case" is a word on its own, we could use the regular expression matching feature of `apropos`:

```
$ apropos -ar convert '\<case\>'
toupper     (3)      - convert letter to upper or lower case
FcToLower   (3)      - convert upper case ASCII to lower case
tolower     (3)      - convert letter to upper or lower case
```

Unfortunately, this powerful `apropos` is not available on all systems. In particular, it is absent on the RHEL 6 system installed on our server. This version has no options, so one cannot do such searches. In this case, to get the same effect, one can use a simple search and pipe the output through a `grep` filter. If you are not familiar with `grep` or regular expressions, see the Appendix. The equivalent command would be

```
$ apropos convert | grep '\<case\>'
FcToLower   (3)      - convert upper case ASCII to lower case
tolower     (3)      - convert letter to upper or lower case
toupper     (3)      - convert letter to upper or lower case
```

If the output list is still too long to be useful, you can filter it further with another instance of grep:

```
$ apropos convert | grep '\<case\>' | grep '\<ASCII\>'
FcToLower   (3)      - convert upper case ASCII to lower case
```

2.5 Digging Deeper into the who Command

The output of the manual search on the `utmp` file will look something like:

```
endutent [getutent]  (3)  - access utmp file entries
getutent             (3)  - access utmp file entries
getutid [getutent]   (3)  - access utmp file entries
getutline [getutent] (3)  - access utmp file entries
login                (3)  - write utmp and wtmp entries
logout [login]       (3)  - write utmp and wtmp entries
pututline [getutent] (3)  - access utmp file entries
sessreg              (1x) - manage utmp/wtmp entries for non-init clients
setutent [getutent]  (3)  - access utmp file entries
utmp                 (5)  - login records
utmpname [getutent]  (3)  - access utmp file entries
utmpx.h [utmpx]      (0p) - user accounting database definitions
wtmp [utmp]          (5)  - login records
```

The first word is the topic of the man page, the next, the man page title, the third is the section number of the manual, and the last is a brief description of the topic.

Every UNIX system has a manual volume that deals with the files used by the commands. The number may vary. From the above output, it appears that the `utmp` file is described in Section 5 of the man pages:

```
utmp [utmp] (5) - login records
```

Also, the line

```
wtmp [utmp] (5) - login records
```

shows that the man page describing the `wtmp` file is the same page as the one describing `utmp`. Obviously, there is a man page for `utmp` in Section 5 of the manual. To specify the specific section to display, you need to specify it as an option. The syntax varies; in RedHat Linux either of these will work:

```
$ man 5 utmp
$ man -S5 utmp
```

There was also a line of output

```
utmpx.h [utmpx] (0p) - user accounting database definitions
```

The `<utmpx.h>` header file describes a POSIX-compliant interface to the `utmp` file. This interface is different from that of the `<utmp.h>` file. We will use the (outdated) `<utmp.h>` interface for our initial attempts, exploring the `utmp` file in greater depth, starting with the man page that our system delivers when we type either of the above man commands. After that we will consider using two other interfaces, the POSIX `utmpx` interface and a GNU extension, the thread-safe functions `getutent_r()` and its cousins.

The beginning of the man page for `utmp` from RedHat Enterprise Linux Release 4 is displayed below.

```
NAME
       utmp, wtmp - login records
SYNOPSIS
       #include <utmp.h>
DESCRIPTION
       The utmp file allows one to discover information about who is currently
       using the system. There may be more users currently using the system,
       because not all programs use utmp logging.

       Warning: utmp must not be writable, because many system programs
       (foolishly) depend on its integrity. You risk faked system logfiles and
       modifications of system files if you leave utmp writable to any user.
       The file is a sequence of entries with the following structure declared
       in the include file (note that this is only one of several definitions
       around;  details depend on the version of libc):
```

(lines omitted here)

First note that it tells us which header file is relevant: `<utmp.h>` This is the header file that the compiler will use when the include directive `#include <utmp.h>` is in your program[6]. Next, it issues a warning to system administrators not to leave this file writable by anyone other than its owner, the superuser. Then it warns the rest of us, before showing us the contents of the include file, that the contents may differ from one installation to another.

Since UNIX is a free, community supported operating system, it has been evolving over time. You may find that what is described in a book, or in these notes, is different from what you observe on your system. It is not that anything is correct or incorrect, but that UNIX is a moving target, and that systems can differ in minor ways. For example, the man page for `utmp` in an older version of Linux will be very different from the one shown here. Even the location of the `utmp` file itself is different. Later versions of UNIX added system functions to provide a data abstraction layer so that the programmer would not need to know the actual structure of the file. The problem was that different versions of UNIX had different definitions of the utmp structure, and programs that accessed the structure directly were failing on different systems.

[6] There may be many files named `utmp.h` in the file system. Each compiler will have its own method of deciding which one to use. The GNU compiler collection (`gcc`) installs its own header files in specific places, and it uses these by default. The default search path used by `gcc` is typically

`/usr/local/include`
target-installdir`/include`
`/usr/include`

where *target-installdir* is the directory in which `gcc` was installed on the machine. This is explained in more detailed shortly.

The structures displayed in the man page may not be the same as those found on our machine. If you write code that depends critically on the structure definition, it may work on one machine but not another. In spite of this, it is valuable to study these structures. Afterward we will write more portable code. The key to that is to use preprocessor directives to conditionally compile the code based on the values of macros. The man page continues:

```
#define UT_UNKNOWN            0
#define RUN_LVL               1
#define BOOT_TIME             2
#define NEW_TIME              3
#define OLD_TIME              4
#define INIT_PROCESS          5
#define LOGIN_PROCESS         6
#define USER_PROCESS          7
#define DEAD_PROCESS          8
#define ACCOUNTING            9
#define UT_LINESIZE           12
#define UT_NAMESIZE           32
#define UT_HOSTSIZE           256
struct exit_status {
    short int e_termination;    /* process termination status.  */
    short int e_exit;           /* process exit status.  */
    };
struct utmp {
    short ut_type;              /* type of login */
    pid_t ut_pid;               /* pid of login process */
    char ut_line[UT_LINESIZE];  /* device name of tty - "/dev/" */
    char ut_id[4];              /* init id or abbrev. ttyname */
    char ut_user[UT_NAMESIZE];  /* user name */
    char ut_host[UT_HOSTSIZE];  /* hostname for remote login */
    struct exit_status ut_exit; /* The exit status of a process
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
    int32_t ut_session;         /* Session ID (getsid(2)),
                                   used for windowing */
    struct {
        int32_t tv_sec;         /* Seconds */
        int32_t tv_usec;        /* Microseconds */
    } ut_tv;                    /* Time entry was made */
#else
    long   ut_session;          /* Session ID */
    struct timeval ut_tv;       /* Time entry was made */
#endif
    int32_t ut_addr_v6[4];      /* IP address of remote host.  */
    char __unused[20];          /* Reserved for future use.  */
};
```

The page then contains a brief description of the purpose of the structure:

```
This structure gives the name of the special file associated with the user's
terminal, the user's login name, and the time of login in the form of time(2).
String fields are terminated by '\0' if they are shorter than the size of the
field.
```

More information about the specific members of the structure is contained in the comments in the `struct` definition. The man page does not describe the members in detail beyond that. The rest of the man page, which is not included here, goes on to describe how the various entries in the `utmp` file are created and modified by the different processes involved in logging in and out. We will return to that topic shortly. It reiterates the warning:

```
The file format is machine dependent, so it is recommended that it
be processed only on the machine architecture where it was created.
```

You should have noticed the following line in the man page:

```
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
```

This causes conditional compilation of the code. It means, if the machine's word size is 64 bits and it is in 32-bit compatibility mode, then use one definition of the `ut_session` and `ut_tv` members, otherwise use a different one. The macros `__WORDSIZE` and `__WORDSIZE_COMPAT32` are defined in the header file `/usr/include/bit/wordsize.h`[7]. We will ignore this subtlety for now, and rather than relying on the man page, we will examine the `<utmp.h>` header file itself.

2.5.1 Reading the Correct Header Files

Which header file to read depends upon the particular installation. For example, on my home office workstation, which is running Fedora 14, `gcc` will use `/usr/include/utmp.h`, whereas on the `cs82010` server in the Graduate Center, which is running RedHat Enterprise Linux Release 6, `gcc` will first look for `/usr/lib/gcc/x86_64-redhat-linux/4.4.5/include/utmp.h`. One method of determining which file `gcc` will actually use in a particular installation is the following:

1. Create a trivial C program such as

   ```
   int main() { return 0; }
   ```

   and suppose it is named `empty.c`.

   ```
   echo "int main() {return 0;}" > empty.c
   ```

   is an easy way to do this.

2. Run the command

[7] The macro `__WORDSIZE_COMPAT32` is only defined on 64 bit machines. One can discover this file by doing a recursive grep on the `/usr/include` directory hierarchy of the form "`grep -R WORDSIZE /usr/include/* | grep define`", which will list the files in which these macros are defined.

```
$ gcc -v empty.c
```

3. In the output produced by `gcc`, look for lines of the form

```
#include "..." search starts here:
#include <...> search starts here:
your_current_working_dir/include
/usr/local/include
/usr/lib/gcc/x86_64-redhat-linux/4.4.5/include
/usr/include
End of search list.
```

These lines will show you which directories and in which order `gcc` searches for included header files. The above output shows that `gcc` will search first in `/usr/include/local`, then in the install directory, and then in `/usr/include`. Since there is no `<utmp.h>` file in the first two directories, it will use `/usr/include/utmp.h`.

Returning to the task at hand, if you look at either of the `<utmp.h>` files mentioned above, you will see that they are mostly wrappers for a file which is in the corresponding bits subdirectory:

`/usr/include/bits/utmp.h`,

or

`/usr/lib/i386-redhat-linux3E/include/bits/utmp.h`.

Taking the liberty of eliminating the 64-bit conditional macros, and the macro names, the important elements of the header file are as follows:

```
/* The structure describing an entry in the database of
   previous logins.  */
struct lastlog
{
     __time_t ll_time;
    char ll_line[UT_LINESIZE];
    char ll_host[UT_HOSTSIZE];
};
/* The structure describing the status of a terminated
   process. This type is used in 'struct utmp' below.  */
struct exit_status
{
    short int e_termination; /* Process termination status.*/
    short int e_exit;        /* Process exit status.       */
};
/* The structure describing an entry in the user accounting
   database.  */
struct utmp
```

```
{
    short int ut_type;        // Type of login.
    pid_t ut_pid;             // Process ID of login process.
    char ut_line[UT_LINESIZE];   // Devicename.
    char ut_id[4];               // Inittab ID.
    char ut_user[UT_NAMESIZE];   // Username.
    char ut_host[UT_HOSTSIZE];   // Hostname for remote login.
    struct exit_status ut_exit;  /* Exit status of a process
                                    marked  as DEAD_PROCESS.*/
    long int ut_session;     // Session ID, used for windowing.
    struct timeval ut_tv;    // Time entry was made.
    int32_t ut_addr_v6[4];   // Internet address of remote host.
    char __unused[20];       // Reserved for future use.
};
```

The point is that login records have ten significant members, and we can write code to extract their data in order to mimic the `who` command. In particular, the `ut_user` char array stores the username, the `ut_line` char array stores the name of the terminal device of the login, `ut_time` stores the login time, and `ut_host` stores the name of the remote host from which the connection was made. Unfortunately, we will not be able to ignore indefinitely the way that time is defined on different architectures, but for the moment, we will continue to ignore it.

2.5.2 What Next?

It seems likely that `who` opens the `utmp` file and reads the `utmp` structures from that file in sequence, displaying the appropriate data for each login. We will write use this as the basis for our own implementation of the command.

2.6 Writing `who`

The program that implements the `who` command has two key tasks:

- to read the `utmp` structures from a file, and
- to display the information from a single `utmp` structure on the display device in a user-friendly format.

We begin by discussing solutions to the first task.

2.6.1 Reading Structures From a File

A *binary file* consists of a sequence of bytes, not to be interpreted as characters. It is the most general form of a file. A file consisting of a sequence of structures, such as the `utmp` file, is a binary file and cannot be read using the C I/O functions with which most programmers are familiar, such as `get()`, `getc()`, `fgets()`, and `scanf()`, nor the `istream` methods in C++, because all of these read

textual input. They are specifically designed for that purpose. Although you could read structures by reading one char at a time and then reconstructing the structure from the sequence of chars with a lot of type casts, that would be grossly inefficient and error-prone. Clearly there must be a better way.

Let us suppose that you do not know the methods of reading from a binary file. You could use a man page search such as

```
$ man -k binary file | grep read
```

Remember though that when you use multiple words with the `-k` option, they are OR-ed together, so the output includes lines with either word (or both). If you do this search, you will see a list of perhaps several dozen man pages. If you get a long list you can filter it further by limiting the output to only sections 2 or 3 of the man pages with a third stage in the pipeline:

```
$ man -k binary file | grep read | grep '([23])'
```

In this list will be the page for two prospective functions to use:

```
fread (3)         - binary stream input/output
read (2)          - read from file descriptor
```

The first, `fread()`, in Section 3, is part of the C Standard I/O Library; it is C's function for reading binary files. The second, `read()`, in Section 2, is the prototype of a system call. As we are primarily interested in what Unix in particular has to offer us, we will look at the system call. In Chapters 5 and 7, we will revisit the C Standard I/O Library.

We want to see what the man page for `read()` has to say. If you do not specify the section number when you type "`man read`", you will get the man page from the first section, and you will discover that there is also a UNIX command, `/usr/bin/read`:

```
$ man read
```

which will output the man page for the `read` command in Section 1. You must type

```
$ man 2 read
```

to get the man page for the `read()` system call. I have included the important parts of the man page below.

```
NAME
    read - read from a file descriptor
SYNOPSIS
    #include <unistd.h>
    ssize_t read(int fildes, void *buf, size_t nbyte);
DESCRIPTION
```

```
read() attempts to read up to count bytes from file descriptor fd into
the buffer starting at buf.
If count is zero, read() returns zero and has no  other  results.
If count is greater than SSIZE_MAX, the result is unspecified.
RETURN VALUE
On  success, the number of bytes read is returned (zero indicates end
of file),  and the file position is advanced by this number.  It is not
an error if this number is smaller than the number of bytes requested;
this may happen for example because fewer bytes are actually available
right  now  (maybe because we were close to end-of-file, or because we
are reading from a pipe, or from a terminal), or because  read()  was
interrupted  by  a signal.  On error, -1 is returned, and errno is set
appropriately. In this case it is left unspecified  whether  the
file position (if any) changes.
```

To use the `read()` function, the program must include the header file `<unistd.h>`. This header file serves various purposes, the most relevant for our purposes being that it contains the prototypes of the (POSIX compliant) system calls.

The difference between <stdio.h> and <unistd.h>.

> The functions that begin with "`f`": `fopen()`, `fread()`, `fwrite()`, `fclose()`, and so on, which operate on file stream pointers (`FILE` pointers) are all part of the ANSI Standard C I/O Library, whose header file is `<stdio.h>`. They are C functions that you can use on any operating system. We used `fopen()` and `fclose()` in Chapter 1 to implement our version of the `more` command.
>
> The functions `open()`, `read()`, `write()`, and `close()` are UNIX system calls and their prototypes are defined in `<unistd.h>`, which is a POSIX header file. The `<unistd.h>` header defines miscellaneous symbolic constants and types, and declares miscellaneous functions, among which are these calls. These functions exist only in UNIX systems and they exist no matter what language you use, as long as the system you are using is POSIX-compliant. POSIX does not specify whether they should be system calls or library functions, but only that they exist as one or the other. These system calls operate on file descriptors, not file streams. The UNIX system calls operate on the kernel directly; the ANSI Standard C I/O Library calls are at a higher level.

The `read()` function has three arguments. The man page says that the `read()` function reads from a file associated with a *file descriptor*. A file descriptor is a small, non-negative integer. We will study file descriptors in greater detail in a later chapter. The second parameter is a pointer to a place in memory into which the bytes that are read are to be stored. The third parameter is the number of bytes to read. The return value is the number of bytes actually read, which can never be larger, but might be smaller, or is –1, if something went wrong.

To illustrate, suppose that `filedesc` is a valid file descriptor that we can use for reading, `buffer` is a char array of size 100, and `num_bytes_read` is an integer variable. The following code fragment shows how to read 100 bytes of data at a time from this file stream until the end of data is found

```
while ( !done ) {
    num_bytes_read = read(filedesc, buffer, 100);
    if ( 0   > num_bytes_read )
        // an error code was returned during reading - bail out
    if ( 0  == num_bytes_read )
        // the end of file was reached - stop reading
        done = 1;
    else
        // do whatever has to be done to the data
}
```

This is a typical read-loop structure. The `read()` call does not fail when there is no data; it just returns 0. This is how to detect the end of the input data.

How can a program associate a file descriptor with a file? Look in the `SEE ALSO` section of the man page and you will find references to `fnctl()`, `creat()`, `open()`[8] and many other system calls. Most of these work with file descriptors. The `open()` system call is the one we need now, because the `open()` call opens a file and assigns a file descriptor to it.

2.6.2 The open() and close() System Calls

To read from a binary file, a process must

- open the file for reading,
- read the bytes, and
- close the file.

The `open()` system call creates a connection between the process and the file. Think of a connection as an object that manages the I/O operations on the file from the process. This object contains things such as the offset in the file for the next operation, various status flags, and pointers to kernel functions that the process can invoke. It is represented by a file descriptor. A process can open several files and each will have its own file descriptor. In fact, it can open the same file twice and each connection will have a different file descriptor[9]. UNIX does not prevent you or anyone else from opening the same file many times. It is up to the users and their programs to coordinate accesses to files.

If you look at the man page you will see the following synopsis of the `open()` call.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag, /* mode_t mode */...);
```

[8] All of these are in Section 2 of the man pages.

[9] You might have guessed. The file descriptor is the index into an array of structs. Each of these structs contains, among other things, a pointer to the next character in the file to be read. A process can read from two different parts of the same file at the same time in this way.

The first argument is a character string containing the path to the file to be opened. The second argument is an integer specifying how the file is to be opened: for reading, for writing, for reading and writing, for appending, and so on. If the call is successful, it returns a file descriptor. More accurately, it returns the lowest numbered file descriptor not already in use by the process. If the call is not successful, it returns –1. There are methods of detecting the type of error; these will be examined later.

The value of `oflag` is one of the following constants defined in <`fcntl.h`>:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing.

It is more complex than this, but this is enough for now. Other values can be bit-wise-OR-ed to these values.

Example. Consider the following code:

```
int fd;
if (fd = open("/var/adm/messages.0", O_RDONLY) < 0 )
    exit (-1);
```

This attempts to open the file `/var/adm/messages.0` for reading. If it fails, it exits. If it is successful, the file is ready for reading. The file descriptor stored in `fd` is the one the program must use in the `read()` call. Notice that the call is made within a conditional expression and that the return value of the call is compared to 0 in that condition. This is a common method of error handling in C programs.

Unlike other operating systems, UNIX does not prevent a file that is already open by one process from being opened by another. This is a very important feature to remember about UNIX. It is why it is possible for multiple users to run the same command or change their passwords at the same time[10].

After your process is finished reading a file, it should close the connection to the file. The `close()` system call

```
int close( int filedes)
```

has a single argument which is the file descriptor of the connection to be closed. If a file has been opened by a process via multiple calls to `open()`, then the other connections will remain open and only the one corresponding to `filedes` will be closed. If the kernel cannot close the connection, it will return –1.

Now you might wonder what could possibly go wrong when closing a file, especially when it has been opened for reading. Well, first of all, it is possible you passed it a bad file descriptor when

[10]Of course UNIX does provide the means for a process to open a file and lock it so that no other process can read or write it while it is in use, but this requires actions on the part of the process to make it happen. UNIX does not do this automatically.

you closed it. Secondly, the kernel, in the middle of the system call, may be given an urgent task to complete, so urgent that it has to drop the `close()` call in the middle to deal with it. In this case it will also return a –1. Also, the file may not have been on the local machine or the local drive, and a network connection might have gone down, in which case the file cannot be closed. Furthermore, if this file had been opened for writing, there are more reasons why `close()` might fail, the most important of which is that it is only when `close()` is called that the actual write takes place and at which point the kernel will discover it cannot complete the write for any number of reasons.

2.6.3 A First Attempt at Writing `who`

The main program must open the file and then enter a loop in which it repeatedly reads a single `utmp` record and displays it on the screen, until all records have been read. A rough sketch of this is in the listing below, which we call `who1.c`.

```
1  Listing 1. who1.c
2  #include        <stdio.h>
3  #include    <stdlib.h>
4  #include        <fcntl.h>
5  #include        <utmp.h>
6
7  int main()
8  {
9          int          fd;
10         struct utmp current_record;
11         int          reclen = sizeof(struct utmp);
12
13         fd = open(UTMP_FILE, O_RDONLY);
14         if ( fd == -1 ) {
15             perror( UTMP_FILE );
16             exit(1);
17         }
18
19         while ( read(fd, &current_record, reclen) == reclen )
20             show_info( &current_record );
21
22         close (fd);
23         return 0;
24 }
```

First observe that the first argument to the `open()` call is `UTMP_FILE`. This is a macro whose definition is included in the `<utmp.h>` header file. Its value is system-dependent; it is the path to the actual `utmp` file. It is usually `"/var/run/utmp"`. We would not know about it if we did not read the header file.

Notice which header files are included, notice that `reclen` contains the number of bytes in a `utmp` struct. The `sizeof()` function returns the number of bytes in its argument type. `reclen` will be used in the `read()` call to read exactly one `utmp` structure at a time. The call to `read()` is given the

file descriptor returned by `open()`, a pointer to a memory location large enough to hold one `utmp` record, and `reclen`, the number of bytes to be read. If the return value equals `reclen` then a full record was read. If it does not, then an incomplete record was read or the end-of-file was reached. In either case we stop reading. The `show_info()` function remains to be written. It should display the contents of the current record. The `perror()` function is described below.

2.6.4 What to Do with System Call Errors

In UNIX, most system calls simply return the value –1 when something goes wrong. This would be rather useless if that is all it did because the calling program would not know what actually went wrong. In addition to returning a –1, the kernel stores an error code in the global variable `errno` that all processes can access if they include `<errno.h>`. When you build a program in UNIX, the variable `errno` is in the namespace of the program if the header file is included.

The `<errno.h>` file defines a number of mnemonic constants for error values, such as

```
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
```

Your program can use these symbols directly with code such as

```
if ( fd = open("myfile", O_RDONLY) == -1 ) {
    printf(Cannot open file: ");
    if ( errno == ENOENT )
        printf("No such file or directory\n");
    else if
        ...
}
```

This would be very tedious, since every program you write would have long `switch` statements or cascading if-statements. It is much easier to use the UNIX library function `perror()` to do this for you. The `perror()` function, which conforms to POSIX-1.2001, has a single string as a parameter, and looks up the value of `errno` and displays the string followed by an appropriate message based on the value of `errno`. It is declared in `<stdio.h>`, so you do not need to include `<errno.h>` if you use it. The code snippet above is simplified by using `perror()`:

```
if ( fd = open("myfile", O_RDONLY) == -1 ) {
    perror("Cannot open file: ");
    return;
}
```

and it would print

```
Cannot open file: No such file or directory
```

In short, the `perror()` function prints the string you pass it followed by the message from the `<errno.h>` file. It is a good idea to create a function to handle errors, so that you do not have to type these lines all of the time. Very often, the error is a fatal one, meaning that the program cannot proceed if the error occurred. In this case, you would want to exit the program, calling `exit()` to do so, as in

```
if ( fd = open("myfile", O_RDONLY) == -1 ) {
    perror("Cannot open file: ");
    exit(1);
}
```

The `exit()` function is declared in `<stdlib.h>`; its man page is in Section 3. A simple function for handling fatal errors would be

```
#include <stdio.h>
#include <stdlib.h>

void fatal_error(char *string1, char *string2)
{
    fprintf(stderr,"Error: %s ", string1);
    perror(string2);
    exit(1);
}
```

You might also benefit from writing a second function to call when you do not want to terminate the program, or you could combine the two into a single, general-purpose function that does either, by passing a parameter to indicate the error's severity.

2.6.5 Displaying login Records

This is the first attempt at `show_info()`:

```
1  void show_info( struct utmp *utbufp )
2  {
3      printf("%-8.8s", utbufp->ut_name); /* the logname */
4      printf(" ");
5      printf("%-8.8s", utbufp->ut_line); /* the tty     */
6      printf(" ");
7      printf("%10ld", utbufp->ut_time); /* login time  */
8      printf(" ");
9      printf("(%s)", utbufp->ut_host); /* the host    */
10     printf("\n");                    /* newline     */
11 }
```

If this were compiled and run on a system that supported this API, the output would look something like

```
$ who1
             system b    952601411    ()
                         952601423    ()
LOGIN        console     952601566    ()
acotton      ttyp3       964319088    (math-guest04.williams.edu)
             ttypc       964319645    ()
```

This output differs from the output of `who` in two significant ways. First, there are records in the output of `who1` that do not correspond to user logins, and second, the login times are in some strange format. Both of these problems are easily fixed.

2.6.6 A Second Attempt at Writing who

2.6.6.1 Suppressing Records That Are Not Active Logins

The file `/usr/include/utmp.h` contains definitions of integer constants used for the `ut_type` member. They are

```
#define EMPTY         0
#define RUN_LVL       1
#define BOOT_TIME     2
#define OLD_TIME      3
#define NEW_TIME      4
#define INIT_PROCESS  5   /* Process spawned by "init" */
#define LOGIN_PROCESS 6   /* A "getty" process waiting for login */
#define USER_PROCESS  7   /* A user process */
#define DEAD_PROCESS  8
```

New entries in the `utmp` file are created by the `init` process and are initialized with a `ut_type` of `INIT_PROCESS`. Recall from Chapter 1 that what happens when a user logs in depends upon whether it is a console login, a login on an xterm window, or a login over a network using a protocol such as SSH. In all cases, the `ut_type` of the entry is changed from `INIT_PROCESS` to `LOGIN_PROCESS`, either by a `getty` process or a similar process, depending on the source of the login. The `getty` (or similar) process prints the login prompt, collects the user's input to the prompt (which should be a username) and creates a login process, handing the user's username to the login process. The login process prompts for the password and authenticates it. If it is valid, it changes the `ut_type` to `USER_PROCESS`. When a user logs out, the `ut_type` is changed to `DEAD_PROCESS`.

This implies that the `ut_type` member of a currently logged-in user record will have the value `USER_PROCESS`. No other `utmp` record will be of type `USER_PROCESS` and so all we need to do to suppress non-user records is to print only those records whose `ut_type` member is `USER_PROCESS`. The show_info() function will be modified by the inclusion of this check:

```
show_info( struct utmp *utbufp)
{
    if ( utbufp->ut_type != USER_PROCESS )
        return;
    ...
}
```

This solves the first problem.

2.6.6.2 Displaying Login Time in Human-Readable Form

Solving the second problem requires an understanding of how calendar, or universal, time is represented in UNIX systems and what functions are provided in the API for manipulating time values.

UNIX represents time as the number of seconds elapsed since 12:00 A.M., January 1, 1970, *Coordinated Universal Time* (*UTC*)[11], known as the "Epoch". UTC is essentially like Greenwich Meridian Time except that it includes occasional "leap seconds" to synchronize with the earth's rotation[12]. UNIX stores time in objects of type `time_t`, the implementation of which is not standardized. On many systems `time_t` is a typedef for a 32-bit integer. Such implementations will fail in the year 2038, when it overflows. Representing time as an integer number of seconds since the Epoch makes it easy for the kernel to update times, but not very easy for a human to determine the time.

How can we learn more about UNIX time and the various parts of the API related to it? The answer again is to do a man page search. If you search on the keyword "time" you will find too many man pages that refer to time. A second keyword will be needed to refine the search. Perhaps "convert" or "transform" or something similar, to capture functions that transform time from one form to another. Trying

```
$ man -k time | grep transform
```

we will see several functions related to time, including `ctime()` and `localtime()`. The man page will also include reference to the header file, `<time.h>`, which must be included for most of these functions. These functions share a single man page. Reading this page reveals that `ctime()` converts a `time_t` time into a human readable string of the form

```
"Mon Aug 11 23:12:06 2003\n"
```

To be precise, the `ctime()` function is declared as

```
char *ctime(const time_t *clock);
```

Observe that the argument is the address of a `time_t` value, not the value itself. The return value is a pointer to a string consisting of a 3-letter day abbreviation, a 3-letter month abbreviation, the day of the month, the 24-hour time in hours, minutes, and seconds, and the 4-digit year. The string is allocated statically by `ctime()`, so it might be overwritten by other calls, so it is best to copy it into a local variable if it needs to be available at a later time.

Note 1. `ctime()` is one of many functions that return a pointer to a string that is allocated statically. Make sure that you understand what this means. The string itself is allocated by `ctime()` and a pointer to that memory is returned to the caller. Subsequent calls to `ctime()` will overwrite the previously allocated memory. The caller will be unable to retrieve the old value unless it was copied

[11] The abbreviation UTC is a compromise between the English and French abbreviations. In English, it would be CUT and in French, TUC.

[12] The earth's rotation can vary due to astronomical conditions. UNIX systems are not required by POSIX to represent exact UTC; they are allowed to ignore the leap seconds.

to a local. Also, the caller is not responsible for freeing the memory allocated to the string; that is handled by the library. This is just one of many functions that are not *thread-safe*, a topic we discuss below.

The `localtime()` function takes a `time_t` argument but returns a pointer to a `struct tm`, which is a structure whose members are the various components of time, such as the day-of-week, the month, day, and year, and so on.

If you read through the man page carefully, which you should, you will find near the end the conformance section. It states:

```
CONFORMING TO

POSIX.1-2001.  C89 and C99 specify asctime(), ctime(), gmtime(), localtime(),
and mktime().  POSIX.1-2008 marks asctime(), asctime_r(), ctime(), and ctime_r()
as obsolete, recommending the use of strftime(3) instead.
```

The `ctime()` function is disparaged at this point. One should instead use `strftime()`, whose prototype is

```
#include <time.h>
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

This function, unlike `ctime()`, allows the calling program to specify the format of the character string to be created. It is also safer to use in that the string is passed as an argument to the function, allocated by the caller, instead of allocated statically and returned as the function value. The first argument is a pointer to the string to be filled, the second, the size of the array of chars to fill, the third is a format for the string, and the last is the `tm` structure containing the broken down time representation.

The format specification is described in great detail in the man page for the function. It is similar to the format for the `printf()` function in that it is a string literal enclosed in double-quotes, with conversion specifications of the form `%`x, where x is a character to be replaced. For example, `%M` represents minutes as a decimal number in the range 00 to 59. and `%b` is the abbreviation of the month name *in the current locale*. This phrase, "in the current locale" means that the locale settings of the user are used in deciding the exact string that `%b` will produce. Every user has a locale in UNIX. The topic of locales will be covered in a later section. The important point now is that `strftime()`, unlike `ctime()`, can use locale information in determining the format of the output string. In chapter 3 we will use this function to display time with more control. For our implementation of the who command, we will use `ctime()`.

The `who` program only displays the date, hours and minutes. For the above example, it would display only `"Aug 11 23:12"`. Our implementation of `who` must extract this substring from the larger string. In other words, given

```
"Mon Aug 11 23:12:06 2003\n"
```

it needs to print

```
"Aug 11 23:12"
```

A simple way to achieve this, perhaps not obvious, is to use pointer arithmetic to print only those characters of the source string in which we are interested. The first character is 4 characters after the start of the string, and the length of the string is exactly 12 characters. Assuming that `t` is a `time_t` variable containing the required time to be printed, the following `printf()`[13] call will do the trick:

```
printf("%12.12s", ctime(&t) + 4 );
```

which prints the 12 chars starting at position 4 in the full string. The format "`%12.12s`" forces the string to use 12 characters on the output. The complete program is shown below. You should study it carefully.

```
1  Listing who2.c
2  //    This solves the time display problem and it filters records
3
4  #include        <stdio.h>
5  #include        <stdlib.h>
6  #include        <unistd.h>
7  #include        <utmp.h>
8  #include        <fcntl.h>
9  #include        <time.h>
10
11 void show_time(long);
12 void show_info(struct utmp *);
13
14 int main(int argc, char* argv[])
15 {
16     struct utmp     utbuf;      // read info into here
17     int             utmpfd;     // read from this descriptor
18     int             reclen = sizeof(utbuf);
19
20     if ( (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1 ){
21         perror(UTMP_FILE);
22         exit(1);
23     }
24
25     while( read(utmpfd, &utbuf, reclen) == reclen )
26         show_info( &utbuf );
27     close(utmpfd);
28     return 0;
29 }
30
```

[13] If you are not familiar with the following C functions, you should take the time to familiarize yourself with them: `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, and `sscanf`. These are all part of C and hence C++ and any C or C++ book should contain adequate descriptions of them. You can also look at the manpages for them. Once you know `printf` and `scanf`, the others are trivial to understand. The best way to learn them is to write a few very simple programs of course.

```
31 void show_info( struct utmp *utbufp )
32 //   displays the contents of the utmp struct only if a user
33 //   login, with time in human readable form, and host if
34 //   not null
35 {
36     if ( utbufp->ut_type != USER_PROCESS )
37          return;
38
39     printf("%-8.8s", utbufp->ut_name); /* the logname */
40     printf(" ");
41     printf("%-8.8s", utbufp->ut_line); /* the tty     */
42     printf(" ");
43     show_time( utbufp->ut_time );        /* login time  */
44     printf(" ");
45     if ( utbufp->ut_host[0] != '\0' )  /* the host    */
46         printf(" (%s)", utbufp->ut_host);
47     printf("\n");
48
49 }
50
51 void show_time( long timeval )
52 //   displays time in a format fit for human consumption
53 //   uses ctime to build a string then picks parts out of it
54 //   Note: %12.12s prints a string 12 chars wide and LIMITS
55 //   it to 12 chars.
56 {
57     char* timestr = ctime(&timeval);
58     // string looks like "Sat Sep  3 16:43:29 EDT 2011"
59
60     // print 12 chars starting at char 4
61     printf("%12.12s", timestr + 4 );
62 }
```

2.6.7 A Third Version of who

The preceding versions of `who` read the data from the `utmp` file using the `read()` system call, reading one `utmp` struct at a time. An alternative method of accessing the data in the file is to take advantage of a data abstraction layer that the API makes available. When we did the man page search for man pages related to the `utmp` file, we ignored the functions found on the page named `getutent`:

```
endutent [getutent]  (3)  - access utmp file entries
getutent             (3)  - access utmp file entries
getutid [getutent]   (3)  - access utmp file entries
getutline [getutent] (3)  - access utmp file entries
pututline [getutent] (3)  - access utmp file entries
setutent [getutent]  (3)  - access utmp file entries
utmpname [getutent]  (3)  - access utmp file entries
```

We now take a look at what that page has to offer. The beginning of the page contains the following (depending on what system you have):

```
SYNOPSIS
       #include <utmp.h>
       struct utmp *getutent(void);
       struct utmp *getutid(struct utmp *ut);
       struct utmp *getutline(struct utmp *ut);
       struct utmp *pututline(struct utmp *ut);
       void setutent(void);
       void endutent(void);
       int utmpname(const char *file);
DESCRIPTION
       New applications should use the POSIX.1-specified "utmpx"
       versions of these functions; see CONFORMING TO.
```

The very first sentence in this man page tells us that these functions are not POSIX.1-compliant, and that there are `utmpx` versions of these functions. We will ignore this warning for the moment and see how to use these non-POSIX functions, simply because there is something that needs to be explained about the POSIX.-1-compliant interface, to which we will return afterward.

The man page basically tells us that there is a simple way of reading the records in a `utmp` file, requiring just four steps:

1. Use `utmpname()` to select the file that should be accessed by the other functions.
2. Call `setutent()` to rewind the file pointer to the beginning of the file.
3. Repeatedly call `getutent()` to get the next `utmp` record from the file; `getutent()` will return a `NULL` pointer after it has read the last record from the file.
4. Call `endutent()` when we have read all of the records.

In other words, this interface provides a hidden iterator to the `utmp` file: `setutent()` initializes it, `getutent()` advances it successively, and `endutent()` sends a signal that it is no longer needed. In addition, the `utmpname()` function simply needs to be told the pathname to the file, and it will take care of opening it.

The man page also mentions that `_PATH_UTMP` is a macro whose value is the path to the `utmp` file. We already knew that `UTMP_FILE` contained that path, but if we dig a little deeper by actually reading the header files, we will discover that the `<paths.h>` header file defines `_PATH_UTMP` and `_PATH_WTMP` and that `<utmp.h>` defines `UTMP_FILE` as another name for `_PATH_UTMP`.

We can put all of this together to create a simpler version of `who`, named `who3`. In this version we add the extra feature that the user can optionally supply the word "`wtmp`" on the command line if she wants to see records in the `wtmp` file instead. The `show_info()` and `show_time()` functions are the same, so we just display the main program in the listing.

```
1  Listing who3.c
2  #include  <stdio.h>
3  #include  <stdlib.h>
4  #include  <unistd.h>
5  #include  <utmp.h>
6  #include  <fcntl.h>
7  #include  <time.h>
8
9  int main(int argc, char* argv[])
10 {
11     struct utmp *utbufp;
12
13     if ( (argc > 1) && (strcmp(argv[1],"wtmp") == 0) )
14         utmpname(_PATH_WTMP);
15     else
16         utmpname(_PATH_UTMP);
17
18     setutent();
19     while( (utbufp = getutent()) != NULL )
20         show_info( utbufp );
21     endutent();
22     return 0;
23 }
```

This program is not *thread-safe*. Many functions in the various UNIX libraries use static variables to store their results. These variables act like global variables within the programs that call these functions. If a program is multi-threaded, these threads can corrupt each others data if they use the unsafe function calls in an overlapping way. Thread-safe functions do not have this problem. A thread-safe version of the `who3` program can use `getutent_r()`, which is a GNU thread-safe version of `getutent()`.

The man page tells us that to use the `getutent_r()` function, we have to set a macro, the `_GNU_SOURCE` macro, before including the header file `<utmp.h>`. That is the purpose of the following lines from that man page:

```
The above functions are not thread-safe. Glibc adds reentrant versions
#define _GNU_SOURCE     /* or _SVID_SOURCE or _BSD_SOURCE */
#include <utmp.h>
int getutent_r(struct utmp *ubuf, struct utmp **ubufp);
```

The macro definition of `_GNU_SOURCE` is required because the `<utmp.h>` header file contains *feature test macros*. Feature test macros can be used to control which definitions are exposed in the system header files when a program is compiled. This is important for creating portable applications, because it prevents nonstandard definitions from being exposed in the program. If you remove the definition of `_GNU_SOURCE` from your program and try to use `getutent_r()` you will get a compile time error because the declaration of this function in the header file is guarded by a conditional preprocessor directive that is true only if `_GNU_SOURCE` is defined. It is essentially of the form

```
#ifdef  _GNU_SOURCE
    extern int getutent_r (struct utmp *__buffer, struct utmp **__result) __THROW;
    /* more stuff here
#endif
```

If you put the definition of `_GNU_SOURCE` after the include directive, it will be useless because it will not be defined when the header file is preprocessed by `gcc`, and so in this case too you will get an error message.

The `feature_test_macros` man page describes everything you need to know to use these macros.

The main program of this thread-safe `who`, which we call `who4.c`, is almost the same as that of `who3.c`:

```
Listing  who4.c
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>

#define  _GNU_SOURCE
#include  <utmp.h>
#include  <fcntl.h>
#include  <time.h>

int main(int argc, char* argv[])
{
    struct utmp  utbuf, *utbufp;
    int          utmpfd;

    if ( (argc > 1) && (strcmp(argv[1],"wtmp") == 0) )
        utmpname(_PATH_WTMP);
    else
        utmpname(_PATH_UTMP);

    setutent();
    while( getutent_r(&utbuf, &utbufp) == 0 )
        show_info( &utbuf );
    endutent();
    return 0;
}
```

2.6.8 A POSIX-compliant Version

There is yet another version of the `who` program, named `who_p.c`, in the demos directory for this chapter on the server. This version is distinguished by the fact that it uses the POSIX-compliant `utmpx` interface. The `utmp` structure is not standard across all versions of UNIX. The one we described above is the GNU implementation, which is what is found on Linux systems. This GNU

version includes members that may not be present on other systems. In an effort to standardize the `utmp` interface, the POSIX standards since 2001 have replaced the definition of the `utmp` structure with a `utmpx` structure. This structure is only guaranteed to have the following members:

```
char            ut_user[]  User login name.
char            ut_id[]    Unspecified initialization process identifier.
char            ut_line[]  Device name.
pid_t           ut_pid     Process ID.
short           ut_type    Type of entry.
struct timeval  ut_tv      Time entry was made.
```

In addition, the functions `setutent()`, `getutent()`, and `endutent()` are replaced by the corresponding functions `setutxent()`, `getutxent()`, and `endutxent()`. In general, the `utmpx` structure may define a different set of members than those found in a `utmp` structure. Linux systems actually define the `utmpx` structure to be the same as the `utmp` structure, unless the `_GNU_SOURCE` macro is defined. In addition, Linux systems define a larger set of allowed values of the `ut_type` member than does POSIX. Programs that are meant to be portable can use conditional compilation with feature test macros to detect which structure is actually on the system at compile time. The `who_p.c` program demonstrates how this is done, but is not included in these notes.

2.6.9 Summary

The preceding set of implementations of the `who` command demonstrates that the man pages and header files can be used to learn enough about a command to implement it. The `utmp` interface may not be the same on every UNIX system, and as a result there are several different methods of approaching the problem. One can use the GNU, non-POSIX, thread-safe version of the interface, for example, or the POSIX-compliant `utmpx` interface. One can also use the lower-level system calls, e.g. `read()`, to access either the `utmpx` or the `utmp` structure directly. A truly portable solution would use feature test macros to conditionally compile the code depending on what system it is to be run on. The exercise introduced various concepts along the way, but we are still not finished with it. Later we will return to the problem with a more efficient solution.

2.7 Using a File in Read/Write Mode

Many applications need to have a file open for both reading and writing. A good example of this is the logout command. The logout command has to update the utmp file, finding within it the record to be updated (i.e., reading it) and then modifying that record (writing it). Most I/O libraries allow a file to be opened for both reading and writing.

2.7.1 Opening a File in Read/Write Mode

Recall that the `open()` system call's second parameter is a set of flags stored in an integer, and that the flags must include one of the access mode flags: `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. If the access mode is set to `O_RDWR`, then the file is opened in read/write mode. In read/write mode, the process can read from and write to the file. The file is not truncated as it would be if opened with

the `O_CREAT` flag. Instead it is opened with the current position pointer pointing to the start of the file. The current position pointer is a member of the *open file structure*, the data structure that is created by the kernel when a file is opened. It points to the position of the next byte to read or write in the file.

For example, to open the file whose path is stored in the C-string `file_to_open`, one could write

```
if ( ( fd = open(file_to_open, O_RDWR)) == -1 ) {
    perror(file_to_open);
    // handle error here
}
```

2.7.2 Logout Records

When a user logs out of a UNIX system, the kernel does some bookkeeping tasks. One of the tasks is to update the `utmp` file to indicate that the user logged out. In particular, it has to change the `utmp` record for the login session by changing the `ut_type` member from `USER_PROCESS` to `DEAD_PROCESS`. It also has to change the `ut_time` member to the current time and zero out the `ut_user` and `ut_host` members.

In short, the logout process has to do the following:

1. Open the `utmp` file for reading and writing
2. Read the `utmp` file until it finds the record for the terminal from which the logout took place.
3. Modify a copy of the `utmp` record in the process's memory, and replace the `utmp` record in the file with the modified one, i.e., modify the `utmp` file.
4. Close the `utmp` file.

The first and last steps need no discussion. The second step requires being able to identify which `utmp` record in the file corresponds to the one logout is trying to modify. It cannot use the `ut_user` member because a single user might have several lines open at a time. The piece of information that is unique is stored in the `ut_line`. The `ut_line` member stores the name of the pseudo-terminal as a string such as `"pts/4"`. Only one person can be using a given terminal at the same time, so it is sufficient to match the line.

The more interesting part of this task is how to replace the `utmp` record in the file. The record may be in the middle of the file, so this operation involves replacing a fixed-size sequence of bytes starting at some specific position in a file with a sequence of the exact same size.

2.7.3 Using `lseek` to Move the File Pointer

As noted above, when a file is opened and a file descriptor is returned for it, a data structure is created by the kernel. This data structure represents the connection to the file. The current position pointer of the data structure is the position of the next byte to read or write in the file. If the file is open for reading, a read of N bytes starts at this position, and then the current position pointer is advanced N bytes. If it is open for writing and writes N bytes, it writes starting at the current

position and then advances it N bytes. Usually when a file is open for writing the current position pointer is at the end of the file.

The `lseek()` system call changes the current position pointer in an open file.

```
#include <sys/types.h>
#include <unistd.h>
off_t  lseek( int fd, off_t dist, int base)
```

`lseek()` is given a file descriptor, `fd`, a distance in bytes, `dist`, and an integer flag, `base`. `base` can be one of three values. The distance, `dist`, is used by `lseek()` to move the current position pointer. If `dist` is positive, it moves forward; if it is negative, it moves backwards. The value of `base` determines the starting position of the current position pointer from which it is to be moved. The three values are

`SEEK_SET` the distance dist is forwards relative to the start of the file,

`SEEK_CUR` the distance, dist, is relative to the current position pointer and may be positive or negative

`SEEK_END` the distance, dist, is relative to the end of the file and may be positive or negative.

If `lseek()` is successful, its return value is the resulting offset location as measured in bytes from the beginning of the file, otherwise it returns –1.

When the value of the offset is positive and the base is `SEEK_END`, the file pointer is moved beyond the end of the file. Data can be written to this position, and this in effect creates a "hole" in the file. For example, if a file is currently open and has the contents "`123456789`", and a seek is performed that moves the file pointer 5000 bytes past the end, after which the characters "`abcde`" are written to the file, then the file size will be 5014 bytes, even though there is a hole of 5000 bytes within it. More will be said about this in Chapter 3.

The `lseek()` call can be used to code the third step of the logout procedure.

2.7.4 Updating the `utmp` File on Logout

The problem with updating the `utmp` file is the following. We have to find the record that corresponds to the login record on the line on which the logout occurred. Therefore we need to repeatedly read a `utmp` record and check whether the `ut_line` member matches the line. When we find the record, which has been read into a local variable in our function, we modify it and then have to write it back. But at this point, the current position pointer has already been advanced to point to the record immediately following the one we just read. Figure 2.1 illustrates this.

In the figure, the matching record is numbered k. After it is found, the pointer has been advanced to the start of record k+1. In order to write the modified record where the original was, we need to move the current position pointer back with `lseek()`. The following program demonstrates the key ideas.

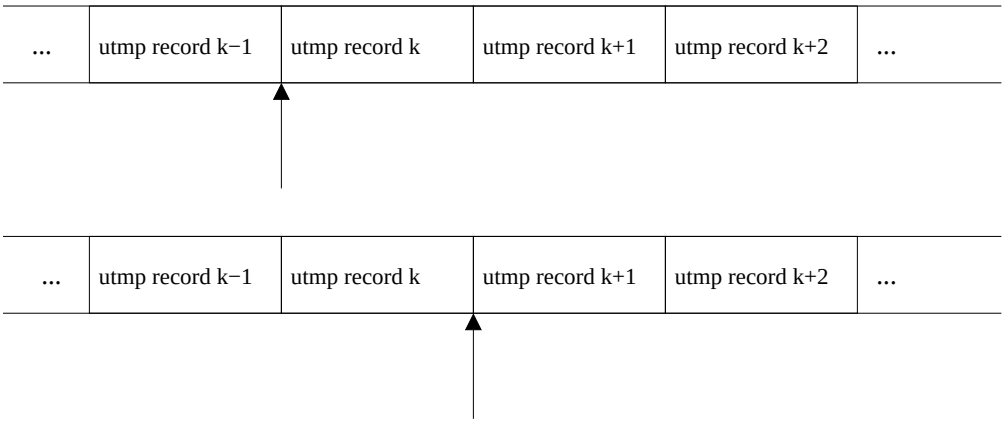


Figure 2.1: Updating a utmp record in read/write mode

```
Listing who5.c
#include ....

int main(int argc, char* argv[])
{
    struct utmp  utbuf;         // stores a single utmp record
    int          fd;            // file descriptor for utmp file
    int          utsize  = sizeof(utbuf);
    int          utlinesize = sizeof(utbuf.ut_line);

    if ( argc < 3 ){        // check usage
        fprintf(stderr,
                "usage: %s <utmp-file> <line>\n", argv[0]);
        exit(1);
    }

    // try to open utmp file
    if ( (fd = open(argv[1], O_RDWR)) == -1 ) {
        fprintf(stderr, "Cannot open %s\n", argv[1]);
        exit(1);
    }

    // If the line is longer than a ut_line permits do not
    // continue
    if ( strlen(argv[2]) >= UT_LINESIZE ) {
        fprintf(stderr, "Improper argument:%s\n", argv[1]);
        exit(1);
    }

    while ( read(fd, &utbuf, utsize ) == utsize )
        if ((strncmp(utbuf.ut_line, argv[2],utlinesize) == 0)
            && ( utbuf.ut_user[0] == '\0' ) ) {
            utbuf.ut_type     = DEAD_PROCESS;
```

```
            utbuf.ut_user[0] = '\0';
            utbuf.ut_host[0] = '\0';

            if ( gettimeofday(&utbuf.ut_tv, NULL) == 0 ) {
                if ( lseek(fd, -utsize, SEEK_CUR) != -1 )
                    if ( write(fd, &utbuf, utsize) != utsize)
                        exit(1);
            }
            else {
                fprintf(stderr, "Error getting time of day\n");
                exit(1);
            }
            break;
    }
    close(fd);
    return 0;
}
```

Notice that every system call is tested for failure before its result is used (except for the call to `write()`). Here, the calls are embedded within the conditional expressions of the `if` and `while` statements above. The first `if` checks whether the record read in the `while` condition has the same terminal line as the one we are looking for (stored in the variable `line`) and the user member is not null. If this is successful, the type member `ut_type` of the record is set to the `DEAD_PROCESS` type, the user and host members are set to null strings, and the time member, `ut_tv`, is updated to the current time. If this is successful, the `lseek()` call moves the current pointer back to the start of the last matched record, so that the write operation that follows will replace the old record. If the write operation is reached and executes without error (determined by checking that the number of bytes written is equal to the number requested to be written), then the program returns 0 for success.

2.7.5 Another Use of lseek

One other use of `lseek()` is determining an open file's size without having to look at its properties. Recall that the return value of `lseek()` is the location of the file pointer after it has been moved, relative to the beginning of the file, and expressed in bytes. If we move the file pointer to the end of the file using `lseek()`, then we get its size as the return value. If `fd` is a file descriptor for the given file, then

```
size_t filesize = lseek(fd, 0, SEEK_END);
```

stores the size of the file into the variable `filesize`. We will make use of this soon.

2.8 Performance and Efficiency : Writing the cp Command

The `who` program was an exercise in reading a system data file and extracting information from it. It was a naive start, in that we did not pay much attention to its efficiency, which is of utmost

concern with most software. To demonstrate the problem a bit more clearly, we will implement a different command, one whose efficiency or lack thereof will be much more obvious. Then we will take what we learned from that exercise and apply it to the `who` program in our final version. The command of interest is the `cp` command, which copies one or more files or directories.

2.8.1 What `cp` Does

If you are familiar with the `cp` command you can skip this section. There are several different ways in which the `cp` command can be used. The simplest is to make a copy of a single file:

```
$ cp source_file target_file
```

Whether or not `target_file` already exists, `cp` makes a copy of `source_file` named `target_file`. If it does exist, it will be overwritten, an act known as *clobbering*. This is dangerous, as you cannot recover the file once you have clobbered it. To prevent accidental overwrites, the interactive option `-i` should always be used, as in

```
$ cp -i source_file target_file
cp: overwrite ‘target_file’? n
```

It is a good idea to define an alias in the `.bashrc` file,

```
alias cp='cp -i'
```

so that you never forget to use the interactive mode.

If a new file is created, it will have the permissions and ownership of the source file. If an existing file is overwritten, it retains the permissions and ownership it had before the copy. No other attributes are preserved in a copy. To preserve the time-stamps and other attributes, you must use the `-p` (`p` for *preserve*) option.

Another form of the `cp` command is

```
$ cp source_file ... target_dir
```

in which the very last word on the command line, `target_dir`, is a directory and all preceding words are non-directory files. In this case, if the directory does not exist, it is an error. Otherwise all of the source files are copied into the directory with their existing permissions and names. If any names already exist in the target directory, the rules described above apply.

In the last form,

```
$ cp -r |-R source source ... target_dir
```

the sources can include directory names. All of the files and directories specified on the command-line, up to but not including `target_dir`, are copied into `target_dir`, which must already exist. The `-r` or `-R` option must be specified otherwise it is a syntax error. The `-r` specifies that the directories will be copied recursively. The `-R` is essentially the same; the difference has to do with how they handle pipes, which is unimportant now.

For the remainder of the chapter, we try to understand the implementation of the simple form of the command, without any options.

2.8.2 Opening/Creating Files For Writing

The `cp` command has to create a file if it does not exist and open it for writing, or overwrite it if it does exist. To overwrite a file, it is first truncated, i.e., its length is set to 0, and then the new data is written to the empty file.

2.8.2.1 Creating/Truncating Files

The first task is to learn how to create files and truncate them. In fact, one call accomplishes both. The `creat()` system call is used to open a file for writing, if it exists, setting its length to 0 first, or if it does not exist, to create it. Notice that there is no "e" at the end of `creat`. If you type "`man creat`" you will get the man page for the `open()` system call:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *path, mode_t mode);
```

The `creat()` system call has two arguments, a C string and a `mode_t`. The string should contain the path name of the file to be created and the `mode_t` specifies the file's mode, i.e., its permission string, as an octal number. For example,

```
fd = creat("prototype", 0751)
```

creates a file named `prototype` in the current working directory, if it does not exist, with permission 0751 (owner can read, write, and execute, group can read and execute, others can execute only) provided that the process's *umask* does not modify the permission. Umasks are covered in the next chapter. If the file exists, the mode argument is ignored and the file is truncated[14]. In either case, upon termination of the call, `fd` is a file descriptor associated with the write-only connection to the file.

2.8.2.2 Writing to Files

Having opened a file for writing, the next step is to write data into it. The `write()` system call is a symmetric counterpart to the `read()` call. It is used for writing sequences of bytes to the file specified by a given file descriptor:

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

[14]It is possible to prevent the file from being overwritten in case it exists, but not if you use the creat() call to try to create it. Instead the open() call must be used. Chapter 4 covers the various methods of opening a file for writing.

The `size_t` type stores the sizes of things in bytes. It is usually a typedef of an unsigned long integer, which may be 32 or 64 bits. The `ssize_t` type is almost the same as the `size_t` type. It differs only in that it is signed and that it can also store a –1. If successful, the `write()` call transfers `nbyte` bytes from the memory location pointed to by `buf` in the process's address space to the position of the file-pointer in the file associated with `fd`, and returns the number of bytes transferred. If the kernel cannot copy any of the data, `write()` returns –1.

The word "buffer" is used to describe the second parameter in the `read()` and `write()` system calls. It is declared as a void pointer. It is called a buffer because it is a storage location in the memory space of the calling process that is used to hold the data to be transferred to or from the file.

The code fragment

```
if (write(fd, buffer, num_bytes) != num_bytes ) {
        fprintf(stderr, "Problem writing to file.\n");
    }
```

attempts to transfer `num_bytes` bytes from the memory location pointed to by `buffer` to the position of the file pointer in the file opened for writing via the file descriptor `fd`. (By default, the file pointer is at "the end" of the file, unless it has been moved elsewhere.) The reason for the condition

```
if (write(fd,buffer,num_bytes) != num_bytes)
```

is that the return value of `write()` is the number of bytes actually written and it may not be equal to the number of bytes that were supposed to be written. The number of bytes successfully written may be less than `num_bytes` for any number of reasons. The file might have reached a predefined maximum size, the disk might be full, or the user's disk quota might be reached. This is why it is necessary to compare the return value of the `write()` call with the value of its third parameter.

2.8.3 A First Attempt at `cp`

The structure of the `cp` command is

open the sourcefile for reading
open the copyfile for writing
while a read of data from the sourcefile to a buffer is not an empty read
write the data from the buffer to the copyfile
close the sourcefile
close the copyfile

We know how to open and close files and we know how to read and write them, so this is a relatively easy program for us at this point. The only points that need explanation are how we create and use buffers. For example, how big should the buffer be? How do we declare it and pass it to the calls?

```
Listing cp1.c
// First attempt at cp command, based on a program
// by Bruce Molay in Understanding Uunix/Linux Programming, p.53

#include   <stdio.h>
#include   <unistd.h>
#include   <fcntl.h>

#define BUFFERSIZE      4096
#define COPYMODE        0644

void die(char* string1, char* string2);  // print error and quit

int main(int argc, char *argv[])
{
    int     source_fd, target_fd, n_chars;
    char    buf[BUFFERSIZE];

    if ( argc != 3 ){
        fprintf( stderr, "usage: %s source destination\n",
                 *argv);
        exit(1);
    }

    // try to open files
    if ( ( source_fd = open(argv[1], O_RDONLY))   == -1 )
        die("Cannot open ", argv[1]);
    if ( ( target_fd = creat( argv[2], COPYMODE)) == -1 )
        die( "Cannot creat", argv[2]);

    // copy from source to target
    while ( (n_chars = read( source_fd , buf, BUFFERSIZE) )
             > 0 ) {
        if ( n_chars != write( target_fd, buf, n_chars ) )
            die("Write error to ", argv[2]);
    }
    if ( -1 == n_chars )
        die("Read error from ", argv[1]);

    // close both files
    if ( close(source_fd) == -1 || close(target_fd) == -1 )
        die("Error closing files", "");

    return 0;
}

void die(char *string1, char *string2)
{
```

```
49          fprintf(stderr,"Error: %s ", string1);
50          perror(string2);
51          exit(1);
52  }
```

Comments

- The buffer is declared as an array of `BUFFERSIZE` chars, which is equal to the maximum number read in a `read()` call.
- The `die()` function encapsulates the error handling logic and calls the `perror()` function.
- Every call to a function in the API is checked for a possible error.
- The main work is in the while loop (lines 32-36). The entry condition is that the `read()` call transferred one or more bytes. The body is the call to write the bytes just read to the output file. The return value of `write()` is checked to see if the number of bytes transferred equals the number requested by the call.

If you compile and run this program you will see that it works correctly. But does it run fast? How long will it take to copy a very large file? How does one time programs in UNIX?

2.8.4 Timing Programs

The `time` command is a means of measuring the amount of time (and other resources) that a command uses. The `time` command has many options, but its simplest form is

```
$ time -p command
```

where `command` is the command that you wish to know about. The '`-p`' option tells `time` to display the traditional POSIX output, which consists of three values, each measured in seconds to two decimal places:

- Elapsed clock time, denoted "real"
- User time, denoted "user"
- System time, denoted "sys"

Elapsed time is the number of seconds from when the command was invoked until it completed. User time is the total amount of time that the process, and any children executing on its behalf, spent running in user mode. System time is the total amount of time spent on the process's behalf running within the kernel, i.e., in privileged mode, including such time spent by its children as well. Non-POSIX output may be more voluminous; you can read the man page for further details. Also, shells such as `bash` typically define their own version of the `time` command, so it is best to type the full path name when using it, if you want the non-`bash` version.

I created a file named `bigfile` containing about 30 MB of data. When I ran

```
$ time -p cp1 bigfile copy_of_bigfile
```

I got the following output on one of the UNIX systems at Hunter College:

```
real   4.05
user   0.01
sys    0.02
```

What accounts for the difference between the sum of user and system times and the elapsed time? It is the time that the process spent waiting for I/O to complete. When a process issues a request for I/O, it is blocked until the I/O is complete. The time that it spends in this blocked, or waiting, state is part of the elapsed (real) time. `cp1` spent about 4 seconds waiting for I/O. Although the amount of time that a process spends waiting for I/O depends heavily on what else the system is doing, the more calls it makes, the longer it will take, on average. The reason for this will be explained below.

As we use `cp1` on larger and larger files, we will see worse performance. To create a spreadsheet with the results of the time command I used a different option to it:

```
/usr/bin/time -f "\t%e\t%U\t%S"
```

The `-f` option expects a format string, which I supplied as a tab-separated string of real-time, user-time, and system-time format symbols. This allowed me to open the output with a spreadsheet program for analysis:

File Size (bytes)	Real	User	Sys
19,004,256	17.28	0.00	0.05
38,008,512	39.17	0.01	0.11
76,017,024	73.69	0.00	0.21

Notice that the real and system times increase roughly in proportion to the size of the file over this small sample.

2.8.5 Buffering and its Impact on Performance

Consider the `cp1` program above. Suppose that N is the size of the file to be copied, measured in bytes. Then the while loop in lines 32 through 36 iterates $\lceil N/BUFFERSIZE \rceil$ times, since each iteration copies `BUFFERSIZE` bytes. It follows that as `BUFFERSIZE` is increased the number of iterations decreases inversely, i.e., if we double the buffer size, we halve the number of calls to both `read()` and `write()`. The question is, how is the total running time affected as the buffer size increases, in general? Is the amount of time to make a single call to `read()` proportional to the number of bytes to be read, or are their other components of its running time that are not related to the size of the read?

To answer this question, we will first perform a little experiment. We will revise the `cp` program so that the buffer size is an input parameter, and run the program on a very large input file with successively larger buffer sizes, recording the three components of running time reported by the time command for each run, and tabulating results. The revised program, called `cp2.c`, is in the listing below.

```
1  Listing cp2.c: a version of cp with buffer size given on the
2                 command line
3  // includes and defines omitted here
4
5  int main(int argc, char *argv[])
6  {
7      int     BUFFERSIZE;
8      char    endptr[255];
9      int     source_fd, target_fd, n_chars;
10     char    *buf;
11
12     // need to check for 3 arguments instead of 2
13     if ( argc != 4 ){
14         fprintf(stderr,
15                 "usage: %s buffersize source destination\n",
16                 argv[0]);
17         exit(1);
18         }
19     //  extract number from string
20     BUFFERSIZE = strtol(argv[1], (char**) &endptr, 0);
21     if (BUFFERSIZE == 0) {
22         fprintf(stderr,
23                 "usage: buffersize must be a number\n");
24         exit(1);
25     }
26
27     //  SNIP: code cut out here, including error handling
28
29     /* allocate buffer of size BUFFERSIZE */
30     buf = (char*) calloc(BUFFERSIZE, sizeof(char));
31     if ( NULL == buf ) {
32         fprintf(stderr,
33                 "Could not allocate memory for buffer.\n");
34         exit(1);
35     }
36
37     // Everything else is the same from this point forward,
38     // and omitted from the listing
```

For those who have not seen it before, `calloc()` (in line 30) and its companion, `malloc()` are dynamic memory allocation functions in C. The prototype for `calloc()` is

```
void *calloc(size_t nelem, size_t elsize);
```

Unlike `malloc()`, `calloc()` takes two arguments: the number of elements, and the size in bytes of each element, and it attempts to allocate space for an array of `nelem` elements, each of size `elsize`.

If it is successful, it returns a `void*` pointer to the start of the array and fills the allocated memory with zeros. This pointer should be cast to the appropriate type before using it.

The table below shows the effect of buffer size on the elapsed, user, and system times when copying a file of size 19MB on a particular host in the Computer Science Department network at Hunter College running RHEL 4. As you can see, the user and system times roughly decrease in inverse proportion to the buffer size for most of the sampled range of values. The user time decreases because the process spends less time in its own code, since there are fewer iterations of the loop and hence fewer instructions to execute. The system time decreases for the same reason – the `read()` and `write()` system calls are executed fewer times and therefore less time is spent in the kernel. The elapsed time tends to reach a steady value after the buffer size reaches 16. Since the total of the user and system time continues to decrease for buffer sizes greater than 16, this suggests that the limiting factor is the time that the process spends waiting for the I/O operations to complete.

Buffer Size(bytes)	**Real**	**User**	**Sys**
2	50.19	3.11	28.27
4	33.27	1.59	13.09
8	24.28	0.76	6.08
16	22.56	0.39	3.08
32	20.53	0.21	1.57
64	21.66	0.10	0.78
128	20.12	0.04	0.43
256	18.27	0.02	0.24
512	19.70	0.00	0.15
1024	18.86	0.00	0.09

As the buffer gets larger, the kernel is called fewer times to transfer the data: as we stated above, if N is file size and B is buffer size, the number of calls is $c = \lceil N/B \rceil$. Another way to say this is that cB is constant. The table shows that, if s is total system time, sB is also approximately constant, except for $B > 256$. In other words, the total system time is roughly proportional to the number of calls made for small values of B. For larger values of B, the total system time is not in proportion to the number of calls, but is larger than it. Why is this?

There are two components to the running time of an I/O operation: the *transfer time* and the *overhead*. The overhead is largely independent of the number of bytes to be read or written; each read or write request to the disk has overhead that does not depend much on how much data is to be transferred. This includes various components of time required by the device to set up and initiate the transfer. It also includes the cost of the system call itself, which is not always negligible.

The transfer time is the time that it actually takes to copy data between the device and memory and is a function of the amount of data. The kernel's involvement in this transfer in modern machines with DMA is minor; it mostly just starts it and does more work when it is finished. Nonetheless, the kernel's involvement is a function of the amount of data to be transferred. Therefore, if B is buffer size, O is the overhead of a I/O operation, and t is a constant such that tB is the amount of time the kernel spends in a single transfer operation, a single `read()` or `write()` system call uses $O + tB$ time units, and the program takes $(\frac{N}{B}) \cdot (O + tB) = \frac{ON}{B} + tN = N \cdot (\frac{O}{B} + t)$ time. Since N is the size of our data and does not change, you can see that the system time is proportional to $(\frac{O}{B} + t)$. This explains why the system time does not keep diminishing by half. Eventually the t

term is large in proportion to the $\frac{O}{B}$ term. When O is very large in comparison to t, doubling B halves the expression, but otherwise it does not.

As we shall see shortly, in UNIX in particular, the design of a buffering system within the I/O system makes the transfer time on average even smaller.

2.8.6 System Call Overhead

System calls have overhead. When a user process makes a call to the kernel for some kind of service, the user process stops executing instructions in its own user space and starts executing instructions that are physically located in kernel space. Prior to making the call, the process executes the user program in a non-privileged mode, also known as *user mode*, and this phase of the process is called the *user phase*. During the system call, the process executes system code with the privileges accorded the kernel, and is said to be in *supervisor* or *kernel mode*; this is called the *kernel phase* of the process[15]. When the call terminates, this kernel phase terminates and the user phase resumes. This is a form of context-switch. A context-switch occurs when the kernel changes the currently executed memory image (the context). This can happen because a new process is run or because the kernel runs on behalf of a process, requiring that the memory image be switched. In some versions of UNIX such as Linux 2.6, a full context switch is not performed when a process changes from user phase to kernel phase or vice-versa.

The kernel needs to execute in kernel mode because it has to have access to all hardware instructions. In contrast, user processes must be prevented from executing special instructions. Therefore, when the system call is made, the machine must change mode twice, at the start and at the end of the call. It must also change the CPU state, because when the kernel runs, it has a different address space, different sets of resources, and so on. All of this changing means that a system call adds overhead to the running time of the program.

2.8.7 System Buffering

In addition to the overhead of the system call itself, there is overhead involved with `read()` and `write()` system calls. When a user process issues a read request from a disk, for example, the kernel does not transfer the data directly from the disk to the address space of the user process. Instead, it transfers the data from the disk to a buffer in kernel memory, and when all of the data has been transferred, it copies it into the user process's address space. This copying of data from kernel memory to user memory takes additional time. The symmetric situation occurs on writes: the kernel copies the data from the user address space into kernel memory, and from there it transfers it to disk[16].

UNIX uses this buffering scheme only for certain types of input and output[17], particularly for read

[15] On some UNIX systems, such as Linux 2.6, the user phase and kernel phase are called user mode and kernel mode respectively.

[16] There is a way to avoid this copying of data back and forth. Memory mapping is a method of I/O in which disk files are mapped directly into user memory. This topic will be discussed in a later chapter. If you are curious, read about the `mmap()` and `munmap()` system calls.

[17] There are two types of I/O in UNIX: block I/O and character I/O. The block I/O system in UNIX is used for block devices such as magnetic and optical disks and tapes. Character I/O is used for devices that are inherently one-character-at-a-time devices, such as the keyboard and terminals in general. Character I/O does not use kernel buffers for I/O. All block I/O uses the kernel's buffering system.

and write operations to and from disks. While it may seem at first that it just adds overhead, in fact it is a powerful and efficient method of reducing overall time spent performing I/O.

The buffering scheme for both reading and writing makes it seem as if read operations read directly from the device and write operations take place immediately. In fact, the kernel hides from the user an important layer of complexity. To understand this complexity, one needs to know a bit about how the disk is organized.

The disk is organized as a collection of fixed-size disk blocks. Disk blocks are numbered so that they can be identified. Each logical disk or disk partition has a unique name in UNIX, such as `sd0a` or `rsd2b`.

The kernel maintains a pool of buffers in kernel memory that can be assigned to each device. Each buffer is given a name, corresponding to the device to which it is assigned and the particular block whose contents it holds. For example, a buffer might be assigned block 511 from disk rsd2b.

On a read request by a process, the buffer pool is searched for a buffer whose name matches the block being sought on the disk. If a buffer is found, the data is read directly from memory without any physical I/O. If the buffer is not found, the data must be read from disk. A buffer will most likely have to be reused for this data. A least recently used (LRU) algorithm is used to decide which buffer to replace. After the buffer is selected, if it is "dirty" its contents are written to disk. Buffers are dirty if they were modified since the last time they were written to disk. The buffer is renamed to match the block being read and the read is performed.

Write requests are handled similarly. When a process requests a write to a specific block on a disk, the buffer pool is searched and if a buffer is not found whose name matches the disk address to be written, a new buffer is allocated for this write operation. If no buffer is available, a block is chosen using the LRU algorithm and relabeled. The data is stored in the buffer without any physical I/O (i.e, disk accesses) and the buffer is marked dirty. The write will be performed only when the block is renamed.

Note that this scheme can greatly reduce the need to perform disk I/O, because reads and writes can take place in memory, which is much faster, and it is completely transparent to the user. But what happens if the system suddenly comes to an unexpected halt? Unless the system has time to "flush" its buffers, the updates are lost. This is why one should never halt a system in the wrong way.

The advantages of buffering are a reduction in physical I/O and therefore a decrease in the overall effective disk access time. The disadvantages include that

- I/O error reporting can lag behind the logical I/O and therefore can become meaningless,
- delayed disk writes can cause loss of data and file system inconsistencies in the event of unexpected system halts, and
- the order in which buffers are written to the external device may not be the same as the order in which the logical I/O occurs, and unless programs are designed with this in mind, disk-based data structures can become inconsistent.

Writes to sequential devices such as tape drives generally do not exhibit this problem because the drivers are only allowed one outstanding write request per drive. In other words, if a logical write operation is requested for a particular drive, but there is a request that has not yet been satisfied

by a physical write, the second request cannot be satisfied until the first physical write takes place. A device like a tape drive will reject requests for service until it finishes what it is doing. It is a one-job-at-a-time device.

In Linux 2.6 and later, the kernel offers a service named direct I/O for processes that wish to bypass the kernel buffering system for block I/O. Certain types of programs such as database servers need to implement their own caching schemes for efficiency. Forcing them to also use the kernel buffering system would slow them down significantly and make the system inefficient, because then there would be duplicate copies of blocks: those in the database server's cache and those in the kernel's cache. With direct I/O transfers, the kernel transfers data directly between the disk and user space. Unfortunately, there are many problems associated with direct I/O, which you can read about in the man page for the `open()` system call. An apt conclusion is reached at the bottom of that page, with a quote from Linus Torvalds:

> In summary, `O_DIRECT` is a potentially powerful tool that should be used with caution. It is recommended that applications treat use of `O_DIRECT` as a performance option which is disabled by default.
>
> "The thing that has always disturbed me about `O_DIRECT` is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances."
>
> — Linus

2.8.8 Memory Mapped I/O

Memory mapping is a way to perform I/O without kernel buffering, and it is fully supported on almost all systems. The concept has been around for a long time. The idea in its simplest form is easy to understand: a process can request that a file be mapped to a set of virtual memory addresses. Changes to those addresses are, in effect, writes to the file. Reads of those addresses are reads of the file.

The actual use of the memory mapping system calls, `mmap()` and `munmap()`, is a bit more complex than this. The purpose of `munmap()`, as its name suggests, is to undo a mapping. The `mmap()` call has several parameters. We introduce memory mapping by writing the `cp` program a third way, using memory mapped I/O instead of reading and writing.

The basic idea is to follow the sequence of steps outlined below:

1. Map the entire input file to a region of memory. Assume it starts at address `source_addr`.
2. Determine the size of the input file in bytes. Call it `filesize`.
3. Create an output file with the given name and make it the same size as the input file.
4. Map the output file to a region of memory the exact same size as the file. Assume it starts at address `dest_addr`.
5. Do a single memory-to-memory copy of `filesize` bytes from `source_addr` to `dest_addr` using `memcpy()`.
6. Undo the mappings and close the files.

This causes the input to be copied to the output without any reads or writes. In order to implement these steps we need to know the prototypes of the mapping functions and `memcpy()`. The prototypes are

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *addr, size_t length);
```

The `mmap()` call creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in the first argument, `addr`. The second argument, `length`, specifies the length in bytes of the mapping.

If `addr` is `NULL`, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If `addr` is not `NULL`, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call. It is best to always use `NULL` as the first argument.

The third argument describes the memory protection of the mapping; it must not conflict with the open mode of the file. The possible values are

`PROT_EXEC` Pages may be executed.

`PROT_READ` Pages may be read.

`PROT_WRITE` Pages may be written.

`PROT_NONE` Pages may not be accessed.

They can be or-ed together. In other words, if the file was opened read-only (`O_RDONLY`), then the value should be `PROT_READ`. If it was opened read-write, then it should be set to `PROT_READ|PROT_WRITE`. A warning about this follows below.

The fourth argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in flags:

`MAP_SHARED` Share this mapping. Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file. The file may not actually be updated until `msync`() or `munmap()` is called.

`MAP_PRIVATE` Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the `mmap()` call are visible in the mapped region.

Because we want to do I/O we need to set the flag to `MAP_SHARED`, otherwise no changes will appear in the output file. There are other values that can be or-ed to this flag, but we will not discuss them at this point.

The next two arguments are the file descriptor of the file to be mapped and the offset in bytes relative to the start of the file at which to map the file. In other words, if you want to map only the portion of the file after the first N bytes, you would pass N as the last argument.

What you need to know is that the memory region is always a multiple of the page size of the machine and must be allocated as such. If the length is not a multiple of page size, the last page will be partly filled. The starting address must always be a multiple of page size. For now this is not our concern. After we learn how to get the page size of the machine, we will return to this issue.

A caveat – the documentation on my Linux system states that `mmap()` has been deprecated in favor of `mmap2()`, but `mmap2()` does not exist on it. In fact, `glibc` (GNU's C Standard Library) implements `mmap()` as a wrapper for the kernel's `mmap2()` call, so `mmap()` is actually `mmap2()`.

Our third copy program is in the listing below. It does not include all of the error-checking and handling that it should, but most is included. It makes use of `memcpy()` to do the actual transfer of bytes from the source to the destination, but it does so within memory. The prototype for `memcpy()` is

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

where `src` is a pointer to the start of the memory to be copied, `dest` is the starting address where the bytes should be written, and `n` is the number of bytes to copy. The memory areas cannot overlap. In other words the absolute value of (`dest - src`) must be greater than `n`.

Listing cp3.c — a copy program using memory–mapped I/O

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include "../utilities/die.h"

#define COPYMODE        0666

int main(int argc, char *argv[])
{
    int     in_fd, out_fd;
    size_t  filesize;
    char    nullbyte;
    void    *source_addr;
    void    *dest_addr;

    /* check args       */
    if ( argc != 3 ){
        fprintf( stderr, "usage: %s source destination\n", *argv);
```

```
        exit(1);
    }

          /* open files */
    if ( (in_fd = open(argv[1], O_RDONLY)) == -1 )
        die("Cannot open ", argv[1]);

    /* The file to be created must be opened in read/write mode
       because of how mmap()'s PROT_WRITE works on i386 architectures.
       According to the man page, on some hardware architectures (e.g.,
       i386), PROT_WRITE implies PROT_READ.  Therefore, setting the
       protection flag to PROT_WRITE is equivalent to setting it to
       PROT_WRITE|PROT_READ if the machine architecture is i386 or the
       like. Since this flag has to match the flags by which the mapped
       file was opened, I set the opening flags differently for the
       i386 architecture than for others.
    */
#if defined (i386) || defined (__x86_64) || defined (__x86_64__) \
     || defined (i686)
    if ( (out_fd = open( argv[2], O_RDWR | O_CREAT |
          O_TRUNC, COPYMODE )) == -1 )
        die( "Cannot create ", argv[2]);
#else
    if ( (out_fd = open( argv[2], O_WRONLY | O_CREAT |
          O_TRUNC, COPYMODE )) == -1 )
        die( "Cannot create ", argv[2]);
#endif
    /* get the size of the source file by seeking to the end of it:
       lseek() returns the offset location of the file pointer after
       the seek relative to the beginning of the file, so this is a
       good way to get an opened file's size.
    */
    if ( (filesize = lseek(in_fd, 0, SEEK_END) ) == -1 )
        die( "Could not seek to end of file", argv[1]);

    /* By seeking to filesize in the new file, the file can be grown
       to that size. Its size does not change until a write occurs
       there.
    */
    lseek(out_fd, filesize-1, SEEK_SET);

    /* So we write the NULL byte and file size is now set to filesize.
    */
    write(out_fd, &nullbyte, 1);

    /* Time to set up the memory maps */
    if ( ( source_addr = mmap(NULL, filesize, PROT_READ,
                MAP_SHARED, in_fd, 0) ) == (void *) -1 )
```

```
        die( "Error mapping file ", argv[1]);

    if ( ( dest_addr = mmap(NULL, filesize , PROT_WRITE,
                MAP_SHARED, out_fd, 0) ) == (void *) -1 )
        die( "Error mapping file ", argv[2]);

    /* copy the input to output by doing a memcpy */
    memcpy( dest_addr, source_addr, filesize );

    /* unmap the files */
    munmap(source_addr, filesize);
    munmap(dest_addr, filesize);

    /* close the files */
    close(in_fd);
    close(out_fd);

    return 0;

}
```

2.9 Returning to `who`

Our previous implementations of `who` read one `utmp` record at a time. Each read requires a system call, even though a single `utmp` record is quite small and there are many of them. We now know that this is inefficient. Just as the `cp` command can benefit by increasing buffer size, so too can `who`. We will modify it so that it reads several `utmp` records at a time and stores them in an internal array. We are now up to version 5, and this version will be called `who5.c`[18].

2.9.1 User-Defined Buffering

A process is said to perform *input buffering* when it requests more data than it can process in an input operation and stores the extra data in its own memory space until it is ready to use it. Input buffering is a way to reduce the cost of input operations because it decreases the amount of time that the process spends in system calls.

In order for `who` to perform input buffering, it needs a place to store the extra records until it is ready to use them. The logical place is in an array of records. If it reads 20 records at a time, for example, then these 20 records will be placed into its internal array. It can maintain a pointer to a *current record*. Each time it needs to examine a new record, it checks whether the current record pointer has exceeded the array bounds. If it has, it attempts to fetch the next 20 records from the `utmp` file and fill the array with them. If no records are left in the file, it cannot obtain a new record, and it is finished. Otherwise, it fetches as many as it can, up to 20, and then gets the current record from the array and advances the current record pointer.

[18] This idea is borrowed from Bruce Molay, *Understanding Unix/Linux Programming*, Prentice Hall.2003.

The logic for input buffering is encapsulated into a separate library of routines for interacting with the `utmp` records, called `utmp_utils.c`. The interface to this library consists of three functions: `open_utmp()`, `next_utmp()`, and `close_utmp()`. The `open_utmp()` function opens the given `utmp` file, the `next_utmp()` function delivers the next record, reading a new chunk from the file if the buffer is empty, and the `close_utmp()` closes the file. The interface follows.

```
Listing utmp_utils.h
typedef struct utmp utmp_record;

int open_utmp( char * utmp_file );
// opens the given utmp_file for buffered reading
// returns: a valid file descriptor on success
//          -1 on error

utmp_record *next_utmp();
// returns: a pointer to the next utmp record from the
//          opened file and advances to the next  record
//          NULL if no more records are in the file

void close_utmp();
// closes the utmp file and frees the file descriptor
```

The implementation of the library is next. It uses global variables (static variables) so that the functions can communicate. We do not want to pass these as parameters, because then client code would have to do that as well, breaking the abstraction. If this were written in C++, this library would be a class instead, and the globals would be member variables.

```
1  Listing utmp_utils.c
2  #include  <stdio.h>
3  #include  <fcntl.h>
4  #include  <sys/types.h>
5  #include  <utmp.h>
6
7  #define NUM_RECORDS           20
8  #define NULL_UTMP_RECORD_PTR ((utmp_record *) NULL)
9  #define SIZE_OF_UTMP_RECORD  (sizeof(utmp_record))
10 #define BUFSIZE              (NUM_RECORDS * SIZE_OF_UTMP_RECORD)
11
12 static char utmpbuf[BUFSIZE];   // buffer of records
13 static int  number_of_recs_in_buffer; // num records in buffer
14 static int  current_record;     // next rec to read
15 static int  fd_utmp = -1;       // file descriptor for utmp file
16
17 int open_utmp( char * utmp_file )
18 {
19     fd_utmp = open( utmp_file, O_RDONLY );
```

```
20      current_record    = 0;
21      number_of_recs_in_buffer = 0;
22      return fd_utmp;     // either a valid file descriptor or -1
23  }
24
25  int fill_utmp()
26  {
27      int    bytes_read;
28
29      // read NUM_RECORDS records from the utmp file into buffer
30      // bytes_read is the actual number of bytes read
31      bytes_read = read( fd_utmp , utmpbuf, BUFSIZE );
32      if ( bytes_read < 0 ) {
33          die("Failed to read from utmp file","");
34      }
35
36      // If we reach here, the read was successful
37      // Convert the bytecount into a number of records
38      number_of_recs_in_buffer = bytes_read/SIZE_OF_UTMP_RECORD;
39
40      // reset current_record to start at the buffer start
41      current_record  = 0;
42      return number_of_recs_in_buffer;
43  }
44
45  utmp_record * next_utmp()
46  {
47      utmp_record * recordptr;
48      int           byte_position;
49
50      if ( fd_utmp == -1 )
51          // file was not opened correctly
52          return NULL_UTMP_RECORD_PTR;
53
54      if ( current_record == number_of_recs_in_buffer )
55          // there are no unread records in the buffer
56          // need to refill the buffer
57          if ( utmp_fill() == 0 )
58              // no utmp records left in the file
59              return NULL_UTMP_RECORD_PTR;
60
61      // There is at least one record in the buffer,
62      // so we can read it
63      byte_position = current_record * SIZE_OF_UTMP_RECORD;
64      recordptr = ( utmp_record *) &utmpbuf[byte_position];
65
66      // advance current_record pointer and return record pointer
67      current_record++;
```

```
68      return recordptr;
69  }
70
71  void close_utmp()
72  {
73      // if file descriptor is a valid one, close the connection
74      if ( fd_utmp != -1 )
75          close( fd_utmp );
76  }
```

Comments

1. In `next_utmp()`, if

   `( current_record == number_of_recs_in_buffer )`

   is true, it means that the number of records read so far is equal to the number of records in the buffer, which implies that it is time to read from the file again.

2. In `next_utmp()`, the line

   ```
   recordptr = ( utmp_record *) &utmpbuf[byte_position];
   ```

   sets `recordptr` to point to the address of the array entry at the given byte position. We have to cast the address of the linear array of bytes to a `utmp_record` pointer type.

The main program must be revised to use these functions, as follows.

Listing who4.c

```
#include "utmp_utils.h"

int main(int argc, char* argv[])
{
    utmp_record *utbufp;        // pointer to a utmp record

    if ( open_utmp( UTMP_FILE ) == -1 ){
        perror(UTMP_FILE);
        exit(1);
    }
    while ( ( utbufp = next_utmp() ) != NULL_UTMP_RECORD_PTR )
        show_info( utbufp );

    close_utmp( );
    return 0;
}
```

2.9.2 Final Comments

This last version of the `who` command improved performance by reading larger amounts of the file at a time, thereby reducing the overhead of disk reads. It follows that if we could read the entire file all at once with a single `read()` call, then we would reduce the amount of overhead to the least it could be. In fact, some versions of the `who` command do precisely this. At this point we cannot write this implementation because it depends upon our knowing how to use the `stat()` system call and some knowledge of the structure of the file system, which will come later. However, this method has a pitfall: the file may be larger than the available memory for the process. In this case, the program must be able to identify this and adjust how it reads the file. The GNU implementation of `who` does exactly this.

Appendix A

A.1 Filters: An Introduction

A *filter* is a program that gets its input from the standard input (`stdin`), transforms it, and sends the transformed input to the standard output (`stdout`). The data passes "through" the filter, which typically has command-line options that control its behavior. A filter may also perform a "null" transformation, making no change at all to its input (which is what `cat` does.) Filters process text only, either from input files or from the output end of another Unix command (i.e., through a *pipe*.) All filters can be given optional filename arguments, in which case they take their input from the named files rather than from standard input. For example, in the command

```
$ cat first second third > combinedfile
```

`cat` reads files `first`, `second`, and `third` in that order and concatenates their contents, sending them to the standard output, which has been redirected to a file named `combinedfile`.

The most useful filters are

`cut`	(usually System V only)1 simple text cutting
`grep`	simple regular expressions as filtering pattern
`egrep`	extended (more powerful) regular expressions as filtering patterns
`fgrep`	fast, string matching expressions with alternation as patterns
`sed`	line-oriented text editing filter
`awk`	pattern-matching, field-oriented filter and full-fledged Turing computable programming language
`cat`	primitive filter with little transformation
`sort`	very general sorting filter
`head,tail`	lets only the top or bottom of a stream pass through
`fold`	wraps each input line to fit in a specified width

If your time is limited and you could learn but one of these, the most important would be `grep` – the return on your investment will be greatest. Coming in second would be `sed`, and then `awk`. The remaining filters are easy to learn and use and are described briefly first.

A.1.1 `sort`

`sort` is easy to use:

```
$ sort file
```

will sort the text file named `file` and print it on standard output. By default is uses collating order, the order of the characters in the character code of the terminal, which is usually ASCII or UTF-8. In this case uppercase letters precede lowercase letters. There are versions of `sort` that ignore case by default, but if your does not, you can turn off case-sensitivity with the `-i` option.

If you want to sort numerically, use the `-n` option, as in

```
$sort -n numeric_data
```

which will sort numbers correctly. Without the `-n`, 9 will precede 10 because 1 precedes 9 in the collating sequence. Read the man page for details.

A.1.2 `head` and `tail`

Simply put, `head` displays the first N lines of its input and `tail`, the last N lines. By default $N = 10$. To print a different number of lines, use

```
$ head -N
```

or

```
$ tail -N
```

respectively.

A.1.3 `cut`

`cut` is a lesser filter. You will rarely use it. It does simple tasks well. It cuts out selected pieces of lines of the input.

```
$ cut -c1-10
```

copies the first 10 characters from every line, removing the rest.

```
$ cut -f2,4
```

copies only fields 2 and 4 of every line to the output stream. Fields are delimited by the TAB character unless the delimiter character is changed using the `-d` option. Fields are 1-based, so the first field is field 1. The delimiter must be a single character:

```
$ cut -f1,5 -d: /etc/passwd
```

will display fields 1 and 5 of the `/etc/passwd` file, which are the username and gcos fields.

A.1.4 Regular Expressions and `grep`

We focus on `grep` and regular expressions. The regular expressions used by `grep` are the same as those used by `sed` and the visual text editor, `vi`. The simplest form of the `grep` command is

```
$ grep <regularexpression> files
```

where *<regular expression>* is an expression that represents a set of zero or more strings to be matched. The syntax and interpretation of regular expressions is found in the `regex` man page in Volume 7, as well as the man page for `grep`, so typing

```
$ man 7 regex
```

or

```
$ man grep
```

will give you everything you need to know on how to use them. The simplest patterns are strings that do not contain regular expression operators of any kind; those match themselves. For example,

```
$ grep  print file1 file2 file3
```

prints each line in files `file1`, `file2`, and `file3` that contains the word "`print`". It will print these in the order in which the files are listed, first lines in `file1`, then `file2`, then `file3`. If you want just a count of those lines, use the `-c` option; if you want the non-matching lines, use the `-v` option. If you want the line numbers, use the `-n` option. There are many more useful options described in its man page.

If you want to match a string that contains characters that have special meaning to the shell, such as white-space, asterisks, slashes, dollar-signs, and so on, it should be enclosed in single-quotes:

```
$ grep  'atomic energy' file1 file2 file3
```

will match all lines in the given files that have the exact string '`atomic energy`' somewhere in the line. Note that the lines merely have to contain the string as a substring; they do not have to match the the string exactly. If you want the pattern to match an entire line, you have to bracket it with operators called *anchors*. The start of line anchor is the caret `^` and the end of line anchor is the dollar sign `$`:

```
$ grep  '^atomic energy$' file1 file2 file3
```

matches lines in the given files that are exactly the string atomic energy.

Regular expressions can be formed with various operators such as the asterisk `*`, which multiplies the expression to its left 0 or more times, as in

```
a*
```

which matches strings with zero or more a's: a, aa, aaa, and the null string. To match a string like ababab, you have to enclose it in \(...\), as in

```
\(ab\)*
```

which matches 0 or more sequences of ab. Note that

```
(ab)*
```

will match strings like (ab)(ab)(ab), not ababab because in regular expressions, the parentheses by themselves are not special characters.

The period matches any character. There are character classes, which are formed by enclosing a list (or a range) in square brackets []. A character class represents a single character from that class. Because the special characters in regular expressions typically have special meaning in the shell as well, it is a good idea to always enclose the pattern in single quotes. In particular, if you give it a regular expression using an asterisk you must enclose the string in quotes[1].

A.1.4.1 Examples

In the following examples, the file argument is omitted for simplicity. In this case grep would apply the pattern against standard input, which means if you actually type this, it will wait for you to enter text followed by an end-of-file signal, Cntrl-D.

```
$ grep 'while *(.*)'
```

matches lines containing the word 'while' followed by zero or more space characters, followed by a parenthesized expression.

```
$ grep '^[a-zA-Z][a-zA-Z0-9_]*'
```

matches lines that begin with a word that starts with a letter, upper or lowercase, following by zero or more letters or digits or underscores.

```
$ grep '[0-9][0-9]*\.[0-9][0-9]\>'
```

The pattern selects strings that have 1 or more digits followed by a single period, followed by exactly two digits. The period must be preceded by a backslash so that grep does not treat the period as the special character meaning "match any character". The "\>" tells grep to anchor the pattern to the end of the word. A word is a sequence of letters and/or digits. This forces grep to select only those words that end in two digits. If I omitted the "\>" grep would have matched strings such as 1.234 or 1.23ab. There is a matching operator, \<, that anchors to the beginning of the word.

Now take a look at this one.

[1] Single quotes are better than double quotes. Single quotes prevent the shell from doing any interpretation of the enclosed characters, whereas when the shell sees a double-quoted string, it does a certain amount of interpretation. Until you understand what the shell will attempt to interpret inside double-quoted strings, use single quotes for enclosing grep patterns.

```
$ grep '\/\*.*\*\/'
```

Since / is a special character, if I want to match it I have to escape it with a \ like this: \/. Similarly, since * is a special character in regular expressions, * is how you have to match a single asterisk *. So to match the two-character sequence /* I have to write \/* and to match /* followed by any number of characters and then followed by */, I have to write

```
\/\*.*\*\/
```

in which .* matches zero or more characters of any kind (including the period itself). This finds lines with C-style comments in them.

Regular expressions also provide a means of "remembering" matched expressions, for re-use in the expression. This is very handy in `vi` and `sed`, which have substitution operators. The same operator used for grouping is also used for remembering matching strings. The remembered string is then referenced using the *back-reference* \1 (or \2, \3... if there are multiple strings remembered):

```
$ grep '\([a-z]\)\1\1\1\1'
```

matches any line that contains a sequence of 5 copies of a letter, such as `xxxxx` or `bbbbb`.

```
$ grep '\([1-9][0-9]\).*\1'
```

matches any line that has a two digit number that is repeated later in the line. The command

```
$ grep '\([a-z]\)\([a-z]\)\([a-z]\)\3\2\1'
```

has three remembered matches in the back-references \1, \2, and \3, but in reverse order. Each will have a copy of the single lower-case letter that it matched, so this pattern matches palindromes of length 6 such as `xyzzyx`.

You are encouraged to read the man page for grep. There is a lot more to regular expressions than is covered here. The best way to learn them is to experiment. You can open a terminal window and type grep followed by a pattern. It will then wait for you to type lines on the keyboard. Lines that match will be repeated. Lines that don't will not. Try it.

A.1.5 The Rest of the `grep` Family

A.1.5.1 `egrep`

`egrep` (*extended grep or expression grep*) has a larger set of regular expressions meta-symbols than `grep`, including '|', '?', '+', and parentheses. It is not a strict superset of `grep` because it does not allow \(\), \{ \}, \< \>. These are equivalent to (), {}, and <>, in egrep.

For example, you can write

```
$ egrep 'March|April|May'
```

and

```
$ egrep 'M(iss)+ippi'
```

which matches `Mississippi` as well as `Mississississippi`. Another extension in egrep is the + operator. A "+" after a regular expression indicates to search for one or more occurrences of the regular expression, as in

```
$ egrep '[a-z]+'
```

which matches 1 or more letters.

A.1.5.2 `fgrep`

The `fgrep` variant of `grep` does not support regular expressions but does support multiple strings. It is used to search quickly for many different fixed strings. For example, you can put a list of frequently misspelled words into a file and then call `fgrep` to search for them:

```
$ fgrep -f errors document
```

will print all lines in `document` that contain one of the strings in the file named `errors`.

A.2 File Globs

All UNIX shells have the ability to parse patterns that represent sets of files. These patterns are called *file globs*, or simply *globs*, or *wildcard* expressions. In essence, the shell will replace a file-glob by the list of files that it represents. For example,

```
$ ls *.c
```

is a command to list all files in the current working directory that have zero or more characters followed by a "`.c`".

The regular expressions that the shell uses for file-globbing have a different syntax from those used by `vi`, `grep`, and the other filters and commands. They are not really regular expressions. File-globs are more limited, and the asterisk `*` does not multiply the character that precedes it. It, by itself, represents zero or more characters of any kind. Thus,

```
$ rm *.o
```

removes all files ending in "`.o`" and

```
$ for i in hwk2_*.gz ; do unzip $i ; done
```

will run `unzip` on every file in the current working directory whose name starts with hwk2_ and ends in ".gz" (in `bash` and `sh` and other Bourne-shell-like shells). You must be very careful when using file globs, especially with dangerous commands such as rm that are not reversible, because they may represent files that you did not think they did. One disastrous example would be

```
$ rm -r .*
```

which a naive user might think removes the "hidden" files in the given directory and their descendants. But the pattern `.*` matches `..`, which implies that the command will recursively remove everything in `..`, the parent directory. There are many other things to know about file globs; the complete description can be found in the man page in Volume 7:

```
$ man 7 glob
```

will display it.

Chapter 3 File Systems and the File Hierarchy

Concepts Covered

UNIX file systems and file hierarchies
Internal structure of a file system
Mounting
i-nodes and file attributes
The dirent structure
Manipulating directories and i-nodes
Creation of files by the kernel
Implementing ls, pwd, and du,
Traversing file hierarchies,
opendir, readdir, closedir, seekdir,
telldir, rewinddir, stat, lstat, fstat,
chmod, chown, creat, link, unlink,
unlinkat, readlink, umask, fnmatch
chgrp, chown, utime,getpwuid,
getgrgid, getpwnam, getgrnam,
rename, ntfw, fts_ open, fts_ read,
fts_ children, fts_ close.

3.1 Introduction

This chapter looks at UNIX file systems from the programmer's perspective. The primary objective is to be able to write programs that use the part of the UNIX API concerning the file system and its components. Of necessity, we will begin with an overview of what the file system is, and to a limited extent, how it is implemented. Although it is not necessary to understand how it is implemented to write programs that use it, a basic understanding of the typical implementation can help in understanding performance considerations and limitations.

3.2 File System Abstraction

A *file system* is an abstraction that supports the creation, deletion, and modification of files, and organization of files into directories. It also supports control of access to files and directories and manages the disk space accorded to it. We tend to use the phrase "file system" to refer to a hierarchical, tree-like structure whose internal nodes are directories and whose external nodes are non-directory-files (or perhaps empty directories), but a file system is actually the flat structure sitting on a linear storage device such as a disk partition. The layout of Linux, for example, is shown in Figure 3.1. This flat structure is completely hidden from the user, but not entirely from the programmer. The user sees this as a hierarchically organized collection of files and directories, which is more properly called the *directory hierarchy* or *file hierarchy*.

3.3 File System Mounting

Multiple storage devices are usually attached to a modern computer. Some operating systems treat the file systems on these devices as independent entities. In Microsoft's DOS, and systems derived from it, for example, each separate disk partition has a drive letter, and the file hierarchy on each separate drive or partition is separate from all others attached to the computer. In effect, DOS has multiple trees whose roots are drive letters. For example, a typical Windows machine may

have a directory `E:\users` on the "`E:`" drive and a directory "`C:\Temp`" on the "`C:`" drive but these directories are in two separate trees, not a single tree.

In UNIX there is a single file hierarchy. It is a tree if you think of the leaf nodes as filenames, but it is not a tree if you think of the leaf nodes as actual files, since a single file can have more than one name, existing as a directory entry in multiple directories, making the topology a directed acyclic graph[1]. We will take the liberty of referring to it as a tree, knowing that this is inaccurate.

In UNIX, every accessible file is in this single file hierarchy, no matter how many disks are attached. There is no such thing as the "C" drive" or "E" drive" in UNIX. This is because of the concept of *mounting*, which will be described in more detail later. In short, in UNIX, a file system may be mounted onto the single file hierarchy by attaching that file system's root to some directory in the hierarchy. It is like grafting a branch onto a tree. By mounting a file system onto the file hierarchy, the file system becomes a subtree of the hierarchy, making it possible to navigate into the file system from the rest of the file hierarchy. The `mount` command without arguments displays a list showing all of the file systems currently mounted on the file hierarchy. As an example, the output of the `mount` command could be:

```
/dev/mapper/root.vg-root.lv on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw,rootcontext="system_u:object_r:tmpfs_t:s0")
/dev/sda1 on /boot type ext3 (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

Each line is of the form,

file_system_name `on` *place_where_it_is_mounted* `type` *file_system_type other_options*

The sixth line states that there is a file system named `/dev/sda1` of type `ext3` that is mounted on the directory `/boot`. You may wonder about the meaning of the line

```
proc on /proc type proc (rw)
```

In a subsequent chapter we will explore the `/proc` file system, which is not a file system that manages disk space, but an interface to the kernel's memory. Further interpretation of this output will be delayed until after a discussion of file systems and mounting in greater depth. At this point, the significance of mounting is that different file systems can be, and usually are, part of a single conceptual file hierarchy, making it possible to partition a disk into separate file systems that all become part of a single file hierarchy.

3.4 Disk Partitions

In the early versions of UNIX, the disk was configured as a single partition with a single file system. As disks grew in size it became advantageous in operating system design to partition them into multiple logical devices that were actually distinct physical portions of the same disk. Partitioning a disk allowed for

[1] If you include symbolic links. it may even be cyclic

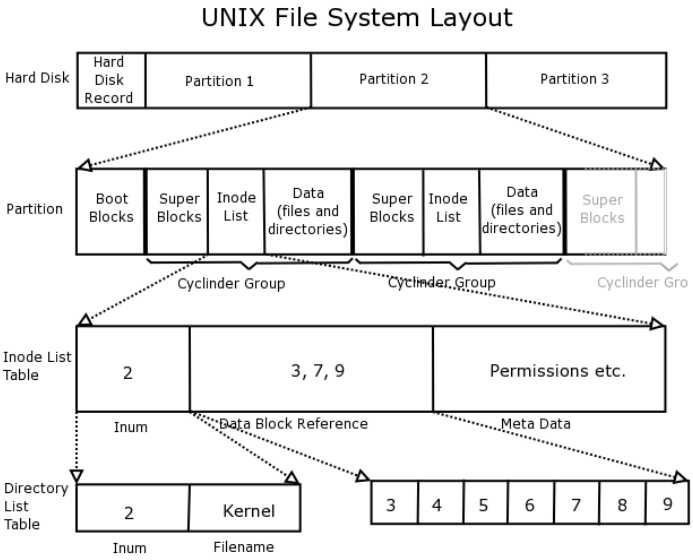


Figure 3.1: Linux file system layout, reproduced from "Linux Internals", by Simone Demblon and Sebastian Spitzner, Courtesy of The Shuttleworth Foundation.

- more control of security – different user groups could be placed into different partitions, and different mounting options could be used on separate partitions, so that some might be read only, and others might have different security options
- more efficient use of the disk – different partitions could use different block sizes and file size limits
- more efficient operation – shorter seek distances would improve disk access times
- improved back-up procedures – backups could be done on partitions, not disks, thereby making it possible to back-up different file systems at different intervals
- improved reliability – damage could be restricted to a single partition rather than the entire disk, and redundancy could be built in

The biggest disadvantage of partitioning a disk is that partitions can not be increased in size, so when they are created, if they are too small and fill up, the entire disk would need to be reorganized. There are other disadvantages of partitioning a disk, but they tend to be outweighed by the advantages.

Modern UNIX systems allow disks to be partitioned into two or more separate physical entities, each containing a distinct file system. The file systems in separate physical partitions are connected to each other by virtue of their being mounted on directories of the one and only directory hierarchy rooted at "/", but they are otherwise unrelated. Each separate file system has its own pool of free blocks, its own set of files, and its own i-node table.

3.5 UNIX File Systems

The different UNIX-like operating systems provide different file systems, each of which may be implemented in its own way. The implementation of the file system is not part of any UNIX

standard: there is no single implementation prescribed or proscribed in any standards document. Therefore, when reading about an implementation of the UNIX file system, you be aware that it is not the only way it is done.

The legacy UNIX file system is not used in many modern systems; modern implementations are more complex because they incorporate many enhancements to the original design. One reason for this is the fact that modern machines must be able to mount file systems of different types. For example, many UNIX systems allow users to mount *FAT*[2] and *NTFS* disk-based file systems, which do not follow the UNIX model. In Chapter 1 it was noted that a directory in UNIX is a file that consists of a list of directory entries, each of which contains the name of a file and its *i-number*, which serves as a pointer to the file's *i-node*. In a FAT system (persisting since early Windows operating systems), directories do not have this structure. UNIX kernels, if they are designed to mount such systems, must create a kernel object in memory to simulate the UNIX directory. Still more importantly, the UNIX kernel cannot hard-code system calls such as `read()` because the implementation of `read()` will depend on the file system. As a result, the actual machine code that is executed when these calls are invoked cannot be bound to the function name when the kernel is compiled.

This situation is analogous to the problem that is solved by pointers in a programming language or virtual functions in C++. When it is not known how much storage a data structure needs at compile time, instead of declaring it statically, the binding of the name to the storage is delayed until run-time by using a pointer instead. In C++, when classes are created with virtual functions, the function code that is executed as a result of a function call is not determined until run-time, another form of delayed binding. In this case, the solution is achieved opaquely through the use of pointers in the underlying implementation[3].

In modern UNIX systems, such as Linux, the implementation of the file system is achieved by dynamically binding the implementations of file system calls to functions that are hard-coded in the particular file system that is mounted, a form of delayed binding as well. How this is accomplished is not important right now; what matters is that modern UNIX file systems are *virtual file systems*, designed to handle many different types of underlying physical file systems. In fact, in Sun's variants of UNIX, from SunOS through Solaris, and in BSD (and FreeBSD), the concepts of i-node and i-number have been replaced by those of v-node and v-number, with the "v" standing for "virtual". Linux continues to use the term "i-node", and in these notes we will do the same.The *Linux Second Extended File System* (*Ext2*) is depicted in Figure 3.2.

The original Linux Virtual File System was developed by Chris Provenzano, and later rewritten by Linus Torvalds. The Linux Ext2 file system was developed in the mid 1990's by Rémy Card, Theodore Ts'o, and Stephen Tweedie. The next Linux file system was Ext3, which was developed by Stephen Tweedie and which differs from Ext2 only in that it contains *journaling*. Journaling is a way to maintain file system consistency in the event of hardware failures. A special journal file is used to record all of the actions that are supposed to be taken on the file system, such as creating and deleting files, changing their contents or attributes, and so on. In a journaling file system, this record can be used to recover the state of the file system without the lengthy task of examining every block and i-node. Ext2 and Ext3 are interchangeable – one can be converted to the other while the file system is mounted because the difference is the journaling. The Fourth Extended File System, Ext4, was released in 2008, mostly to improve performance. While Linux supports many types of file systems, the Ext2, Ext3, and Ext4 file systems are native to it and found on almost all

[2] *FAT* stands for *File Allocation Table*, and is the file system found on many Microsoft operating systems as well as other external storage devices such as memory cards.

[3] A special table called a *virtual dispatch table* is usually how virtual functions are implemented.

Linux systems.

Although there are many different UNIX file systems, most current implementations are built upon ideas from Dennis Ritchie's original implementation. Therefore, the implementations described in these notes should be thought of as generic implementations, meaning they do not actually exist in any one system but approximate many actual implementations. I will, as much as possible, describe how some Linux variant does things, since that is the most prevalent UNIX system that students use, and that is the one running on our host computer.

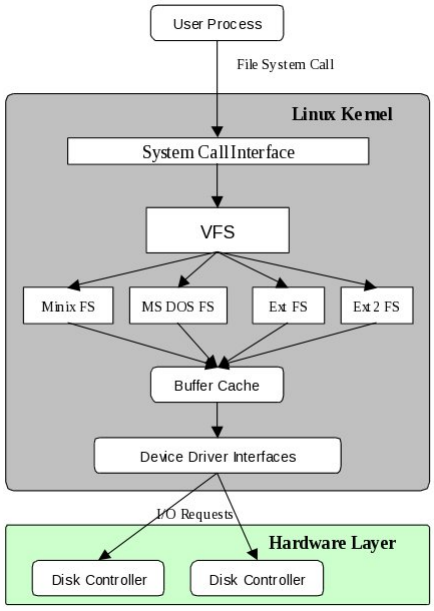


Figure 3.2: The Linux Second Extended File System (Ext2)

3.6 The Principal Components of a UNIX File System

Every file system has (at least) one *superblock* located at the beginning of its allocated storage. The superblock contains all of the information about how the file system is configured, such as block size, block address range, and mount status[4]. Copies of the superblock are almost always stored in several other places within a partition[5].

Each file system in UNIX has at least one table that identifies the files in it. The entries in this table are called *i-nodes* (pronounced "eye-nodes"), and the indices of the i-nodes in this i-node table are called *i-numbers*. The i-nodes contain the file attributes and a map indicating where the blocks of the file are located on the disk. Attributes include properties such as

- the owner,

[4]The superblock actually contains (on most UNIX systems) the block size, a pointer to the i-node table and free i-node list, a pointer to a structure specifying the type of file system, a device identifier for the block device, a structure with the allowed operations, the mount status, and other information as well.

[5]Originally there was a single superblock. In later versions of UNIX, a copy of the superblock was placed in every cylinder group in case of a disk crash.

- the group,
- the permissions allowed on the file and the file type,
- the number of links to the file
- the time of last modification,
- the time of last access,
- the time the attributes were last changed,
- the size in bytes of the file,
- the number of blocks used by the file, and
- the id of the device on which the file resides.

I-nodes and the tables that use them are important components of the UNIX file system. Modern file systems usually have multiple i-node tables, as described below.

Every file system separates the i-node tables from the data blocks. The data blocks are where file contents are stored.

Figure 3.1 depicts the structure and layout of a modern UNIX disk device with several file systems on it. Each successively lower layer of the figure is an enlargement of a structure in the preceding layer. You can see that in a modern file system, not only are there multiple superblocks in a single file system, but multiple i-node tables as well. This is an enhancement added for performance reasons. As disks grew in size, files whose blocks were on the outer edge of the disk became further away from the i-nodes that contained the block addresses and file status. Disk accesses to read or write the file required ever increasing latency caused by the disk seeks. By making several smaller tables, each in its own cylinder group, no file became too far away from the i-node table. The figure shows that the i-node in position 2 of the table usually points to the entry for the root directory file in the file system. We will return to this issue below.

3.6.1 Defining and Creating File Systems

Partitioning a disk divides the disk into logically distinct regions, often named with letters from the beginning of the alphabet, i.e., a, b, c, and so on. In UNIX, partitions are not necessarily disjoint. The "c" partition is almost always the entire disk[6], and typically does not have a file system. It is used by utilities that access the disk block by block. The "b" partition traditionally was reserved as the swapping store, i.e., the partition used for swapping; it did not have a file system written onto it. The innermost partition, "a", is where the kernel is installed and it is typically very small, since little else should be put in it. If a disk has a 100 GB storage capacity, you might make the first 1 GB partition a, the next 10 GB, b, the next 50GB d, the remainder, e, and the whole disk, c.

In order to create files in a partition, a file system must be created in that partition. Creating a file system includes doing the following:

[6]On Solaris hosts, it referred to the entire disk. In FreeBSD, it refers to entire "slices", which can be thought of as collections of partitions.

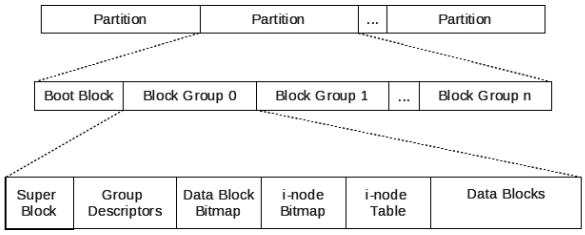


Figure 3.3: Partitions, block groups, and their structures in Ext2

- Dividing the partition into equal size logical blocks, typically anywhere from 1024 to 4096 bytes each, depending upon expected file size. The block size is fixed at file system creation time; it cannot be changed after that without rebuilding the file system. Block size is always a power of two. Larger block sizes result in more wasted disk space in internal fragmentation, whereas smaller sizes result in more disk activity and more disk waits. Larger blocks are appropriate for file systems expecting large files. In the file systems on my personal Linux host, the root file system uses 1024-byte blocks and the second partition, used for user data, uses 4096-byte blocks.

- Deciding how many alternate blocks are needed in each cylinder. (A cylinder is the set of all tracks that are accessible from one position of the disk head assembly. In other words, a cylinder is the set of tracks that are vertically aligned one on top of the other.) When a block becomes bad, it has to be removed from the file system. Alternate blocks are reserved to replace bad blocks.

  – In Linux Ext2, block groups take the place of cylinder groups. Whereas a cylinder group is a physical concept, tied to the geometry of the disk, a block group is a logical concept, independent of the disk geometry, because modern hard disk drives hide the geometry from the operating system. Figure 3.3 illustrates this.

- Deciding how many cylinders are in each *cylinder group*. Grouping cylinders is done for performance and reliability. A cylinder group is a collection of adjacent cylinders that are grouped together to localize information. The file system tries to allocate all data blocks for a given file from the same cylinder group if the file is small enough. Each cylinder group also keeps a copy of the superblock, as described below.

  – In Linux Ext2, each block group contains the following (see Figure 3.3):

    * a redundant copy of the file system's superblock,
    * a redundant copy of a table of block group descriptors; each block group descriptor contains information about the structure of a specific block group and a map of where everything in the group is located; the table is identical in all groups,
    * the block bitmap, which has a bit for each block in the group indicating whether it is free or in use,
    * an i-node bitmap, which has a bit for each i-node in the group indicating whether it is free or in use,
    * an i-node table for the i-nodes in this block group,

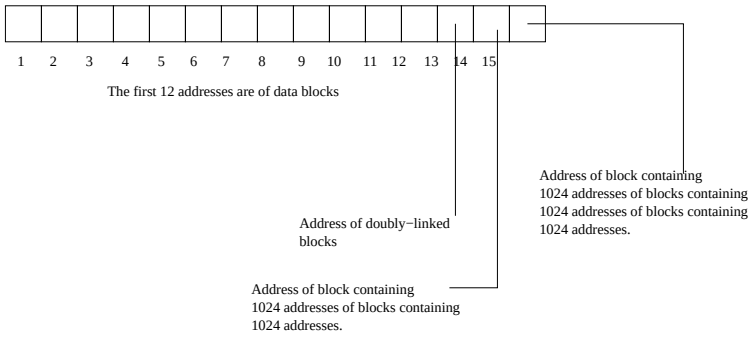


Figure 3.4: Addresses stored in an i-node

  * the actual data blocks.

- Deciding how many bytes to allocate to each i-node. They can be as small as 64 bytes. The i-nodes on the Linux Ext file systems are usually 128 bytes. Larger i-nodes means fewer i-nodes, which means fewer files. Smaller i-nodes allow more files.
- Dividing the cylinder group or partition, depending on the system, into three physical regions:
  - *The superblock.* This stores the map of how the disk is used as well as the file system parameters. In the Linux file system, the superblock contains information such as the block size in bits and bytes, the identifier of the physical device on which the superblock resides, various flags indicating whether it is read-only or locked, or how it is mounted, and queues of mounts waiting to be performed.
  - *The i-node area.* This is where used and free i-nodes are stored. The used and free i-nodes were traditionally arranged into two lists, the i-list and the free-list, with the start of each list stored in the superblock. That method of storage management is obsolete. Later versions of UNIX used a more efficient method[7], in which, when the file system is created, a fixed number of i-nodes was allocated within each cylinder group. This puts i-nodes closer to their data blocks, reducing the overall number of seeks[8].
  - *The data area.* This is where the data blocks are stored.

Many other things need to be done to the partition to make a file system. For example, because the disk rotates while it is reading data, there need to be gaps between blocks. How big should the gaps be? Because the disk head has to advance to a new cylinder to read the next block sometimes, and the disk is rotating while it advances, the next block should not be in the same sector. Which sector should be read next? Also, the superblock is usually replicated on the disk for reliability. How many times? Where should it be placed?

3.6.2 File Storage

The method of storing files in UNIX is flexible and reasonably efficient. Remember that each file has an i-node in an i-node table containing information such as the user-id of the owner, the group-id of

[7] In BSD they called this the Fast FileSystem,

[8] If the i-node is at the beginning of the disk and the data blocks are in the middle, then the head has to go back and forth, reading the address from the i-node, getting the data, going back to the i-node, etc. With the i-nodes near the data blocks, it takes a little more time to reach the i-node, but it is more than made up in the savings in data block access.

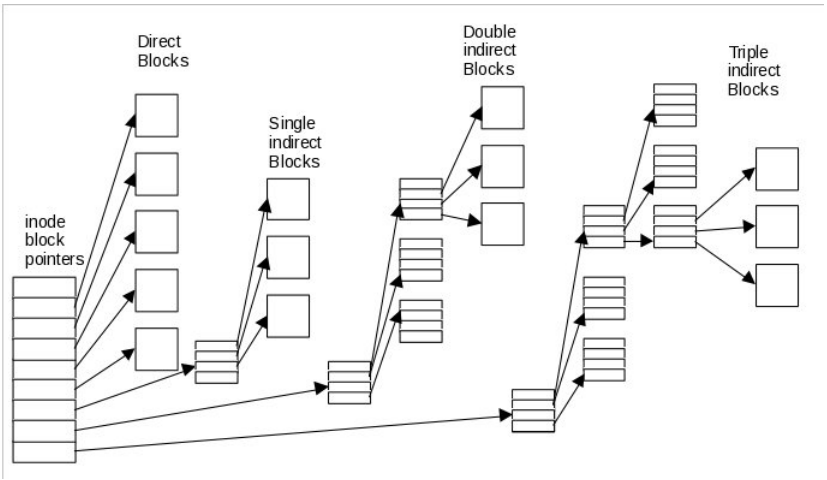


Figure 3.5: Allocation of blocks to a file in a UNIX File System

the file, the file size, permissions and so on. The i-node also contains an array 15 block addresses[9]. A block address is usually 4 bytes long.

In BSD, i-nodes contains an array of 15 block addresses, which are used as follows:

- For regular files, the first 12 block addresses in this array are the addresses of the first 12 blocks of the file. If the block size is 4096 bytes (4 KB), then a file with at most $12 \cdot 4 = 48$ KB can be accessed by one level of indirection.
- If a file is larger than 48 KB, then the 13^{th} address is the address of a *single indirect block*, which is a 4096-byte block used to store block addresses. Since a block address is 4 bytes, there are $4096/4 = 1024$ block addresses in this block. Since each of these 1024 blocks is 4096 bytes, using the first 12 addresses plus the 13^{th} address allows for addressing files whose size is at most 48 KB + $1024 \cdot 4096$ bytes, or 48 KB + 4 MB.
- For still larger files, the 14^{th} address is the address of a *double indirect block* that contains 1024 addresses of single indirect blocks, each of which contains 1024 device addresses. This accommodates files whose size does not exceed 48 KB + 4 MB + $1024 \cdot 4096$ KB , roughly 4 GB.
- The 15^{th} address is that of a *triple indirect block*, which, needless to say points to 1024 double indirect blocks and so on. It allows for $1024 \cdot 1024 \cdot 4096$ additional KB, or a maximum file size of roughly 4 terabytes.

Figure 3.4 shows how the block pointers are laid out in the i-node and Figure 3.5 illustrates the distribution of blocks that results. With this design, there can be a lot of disk seeking, because the different parts of a file can be very far away from each other. For most files though, the blocks tend to be physically contiguous and so actual I/O overhead is acceptable. For very large files requiring several addressing steps before accessing the disk blocks, overhead can be large. This is offset by the use of the kernel I/O buffering described in Chapter 2.

[9]The number of addresses varies. On early UNIX systems, it was 13. In 4.4BSD it is 15. In some implementations of Linux, it is 13 and in others, it is 15.

3.6.3 How the Kernel Creates Files

Suppose that the current working directory is `/home/sweiss/scratch`, and the command

```
$ gcc -o testprog testprog.c
```

is executed. Assuming that the program is compiled and linked, and that I have write and execute permission for the directory `/home/sweiss/scratch`, `gcc` will create the file named `testprog` in this directory. This section answers the question, "What steps are taken by the kernel to create the file `testprog`" on behalf of `gcc`? These steps can be outlined as follows:

1. The kernel creates an i-node for the file, if possible.
2. The kernel fills in the i-node with the file status.
3. The kernel allocates data blocks for the file and stores the file data in these blocks.
4. The kernel records the addresses of the data blocks in the i-node.
5. The kernel creates a directory entry in the `scratch` directory with the i-node number and file name `testprog`.

Each of these steps is explained below.

Creating the i-node

The very first step is to create an i-node for the file. To do this, the kernel must get a free i-node in the i-node table. If there isn't a free one, it must report the error and stop here. If this happens, the user will get a message that the file system is full. This is one reason to enable disk quotas on the system. A table of active i-nodes is kept in memory with a copy on disk as well.

Updating the i-node

Assume that the kernel obtained an i-node, and that it has index 47 in the i-node table. The kernel fills the i-node in with the owner, permissions, time of last modification, and so on. It then saves the i-number, 47, of this i-node. The i-node update takes place in the memory copy of the table, not the disk resident copy. This is updated periodically by the kernel.

Allocating Data Blocks

The next step is to store the file's data. To do this the kernel must acquire the right number of blocks. As `gcc` runs, it is generating the file, writing it in pieces at a time. Because the kernel does output buffering, the file is being stored in kernel buffers, which are not written to disk until the buffers are flushed. If the file is small, all of it will fit in memory and the kernel will know how many disk blocks are needed for it. If the file is very large, the kernel may start allocating blocks before it knows the file's actual size. Assuming that the file system has storage for it, it will first allocate direct blocks. If the file is larger than the number of bytes that can fill all of the direct blocks, the kernel allocates single-indirect blocks as needed. If it is larger than the amount of storage they can provide, it starts allocating double indirect blocks. It continues this procedure, using a triple indirect block if not even all of the double indirect blocks will suffice.

Recording Locations of Data Blocks

The kernel records the order and location of the data blocks in the i-node. Some file systems also have file maps for faster access to the files. Whatever data structures are used to record the block locations, these must be updated at this point.

Recording the File Name in the Directory

If all of the preceding steps were successful, then the kernel will create a new entry in the current working directory consisting of the pair (47, `testprog`), because 47 is the i-number and `testprog` is the name.

3.6.4 How the Kernel Accesses Files

When a user enters a command that has to open a file for reading, what actually happens? For example, when the command

```
$ cat myfile
```

is executed by the shell, what steps are taken and by which programs? The first step is that the shell parses the command and determines that the `cat` command is the executable and that it has a single file argument named `myfile`. It will arrange for the name `myfile` to be placed into the `argv` array that `cat` will see in its `main()` parameter list:

```
int main(int argc, char *argv[])
```

`cat` will ask the kernel to open the file for reading via the `open()` system call. The `open()` system call tries to find the i-number of the file whose name is passed to it. To do so, it has to resolve the pathname, which is a fact that we overlook for now. But just to give you an idea of the complexity, what if the filename were

```
../../shared/temp1/coursework/workspace/myfile
```

This would be a little more challenging than if it were in the working directory. For now suppose that `open()` figures out what directory to open and then searches through that directory for the name "`myfile`". When it finds it, it obtains the i-number for it. Once it has the i-number, it can retrieve the i-node information from its i-node table. If the file is already open, the i-node will be in the active i-node table in kernel memory. If not, it is retrieved from the i-node table on disk, and the i-node is copied into the active i-node table.

Once the kernel has a copy of the i-node in its memory space, it can access the i-node's information. One of the first things it has to do is check that the permission bits allow the `cat` program to access the file. The `cat` program runs with an effective user-id of the user who invoked it. If this user does not have appropriate permission to access this file in the given directory, the `open()` call will return –1 and set the static variable `errno` to `EPERM`. which is the error code that system calls assign to `errno` when they fail because of a permission problem.

If the access is allowed, the `open()` call will have returned a file descriptor for the file, and that descriptor will be pointing to a data structure that, among other things, has a pointer to the i-node for the file. The `cat` command, as it makes successive calls to read the file, is essentially asking the kernel to access the i-node and get the data from the file's data blocks. The kernel uses the i-node to determine which blocks to access. From the file size, the kernel can determine whether the data is entirely in direct blocks (e.g., with a block size of 4096 bytes and 12 direct blocks, at most 48 KB can be stored in them) or whether to read the first single indirect block. If the file is stored in indirect blocks, it will be slower to access, because most of the data blocks will require three disk accesses or even four. If the block size were 8192 bytes, then files of up to 96 KB would be accessed more quickly. The kernel also knows from the active i-node, whether the file's blocks are in a system buffer and can be accessed without disk I/O.

3.7 The `ls` Command

Equipped with a bit more understanding of what a file system does and how it does it, we can try a small exercise by writing a program that uses the file system's programming interface. The `ls` command is a simple enough start.

The `ls` command can be given a list of filenames of all kinds (e.g., regular files, special files, and directories) as arguments:

```
ls [ options ] FILE FILE ...
```

where `FILE`, `FILE`, ... are filenames, whether they are regular files, special files, symbolic links, or directories. Of course, as with almost all commands, the filename arguments can be arbitrary pathnames. If an argument is a directory, it displays the file links contained in that directory, and if the appropriate option is given, the file attributes as well. If it is not a directory, `ls` displays the filename, and if the `-l` option is specified, the file attributes as well. There are many options that change the default behavior of `ls`. You can read the man page for the full list of them.

Summarizing, `ls` does two different things, depending on whether the argument is a directory or a non-directory file.

- When the argument is a directory, `ls` displays its contents.
- When the argument is not a directory, `ls` displays its name.

In either case, the options might require that some subset of its attributes are displayed as well, in a specified order.

Somehow `ls` can determine whether an argument is a directory or not. To implement `ls`, we have to figure out

1. how to determine if a file name is that of a directory;
2. how to list the contents of a directory; and
3. how to list the attributes of a file.

When the argument to `ls` is a non-directory file, it may have to obtain the attributes of the file and display them. When the argument is a directory, say `dir`, `ls` has to do something like

```
open the directory dir
while not all directory entries in dir have been read
    read the entry
    optionally display information about the entry
close the directory
```

The questions are, how can we open, read, and close a directory, and how can we obtain information about file attributes. Fortunately none of these tasks are particularly difficult, owing to the simplicity of the file system interface.

3.8 The Directory Interface

Recall that, as far as the kernel is concerned, regular files are just sequences of bytes. Directories are like regular files except that

1. They are never empty:

   Directories are not empty because every directory has two unique entries, "." and ".." that refer respectively to the directory itself and to the parent directory. These are created when the directory is created.

2. They cannot be written to by unprivileged programs:

   They can only be modified by very specific system calls, unlike regular files. In contrast, anyone with appropriate permission may read the contents of a directory. Some commands that operate on regular files can be applied to directories, but with unexpected results.

3. They have a specific structure:

   A directory is a file that contains a collection of (name, i-number) pairs. In other words, a directory is a table of records. You can think of it as a hash table, since it is accessed with a name in order to retrieve the i-number.

Some commands that read files also read directories, but the results are not useful. For example, on some systems, the `cat` command and the `od` command may display the contents of a directory as a stream of bytes, but because a directory is not a text file, the output of `cat` will look garbled. The output of `od` may look normal. Usually, these commands, like the `more` command, check whether their argument is a directory, and if so, refuse to execute, displaying instead an error message:

```
$ more projects
*** projects: Is a directory ***
```

The `more` command refused to display `projects` because it is a directory and a directory displayed as a sequence of characters would not be meaningful. The output of `od` may reveal the structure of the directory to you if you really examine it carefully. The `-c` option to `od` displays bytes that can

be converted to characters as characters. On Linux, neither `od`, nor `cat`, nor `more` will display the contents of a directory.

If you are wondering what system calls might work with directories, the `open()`, `read()`, and `close()` system calls will act on directories because they do not modify the directory, but because a directory has a specific structure, there is not much point to using them. They are not intended to be used with directories. It is better to use the specific system calls or library functions designed to open, read, modify, and close directories.

If you use a man page search to look for man pages that refer to the word "directory" you will find a long list of them. It is best to filter the output of the search, using something like

```
$ man -k directory | grep read
```

which will display just a few entries. You might see, among other things

```
readdir (2) - read directory entry
readdir (3) - read a directory
readdir (3p)- read a directory
readdir_r [readdir] (3p)- read a directory
seekdir (3) - set the position of the next
readdir()     call in the directory stream
```

Different systems will deliver slightly different sets of man pages, depending on which version of UNIX they are running. All should list the `readdir()` system call in Section 2, and most should list a call named `readdir()` in Section 3 or 3P or both. On almost all systems, if you read the man page for `readdir()` system call in Section 2, it will advise you not to use this system call and instead to use the POSIX library function of the same name. It will refer you to the `readdir()` C library function whose man page is in Section 3 or 3P. You might need to use a lowercase p in your man command, as in

```
$man 3p readdir
```

Because of the wide variation among the different systems, we cannot know exactly what you will see, but all pages should have, at a minimum the following:

```
SYNOPSIS
     #include <sys/types.h>
     #include <dirent.h>

     struct dirent *readdir(DIR *dir);
DESCRIPTION
     The  readdir() function returns a pointer to a dirent
     structure representing the next directory entry in the
     directory stream pointed to by dir. It returns NULL on
     reaching the end-of-file or if an error occurred.
```

This tells us that `readdir()` requires an argument of type `DIR*` and returns a pointer to a `dirent` structure. The rest of the man page may contain a definition of this `dirent` structure, but it may not, depending on the system. If it does not, then you need to use the strategy described earlier for finding it. It will not explain what a `DIR*` is or where it is defined. That will almost certainly require further research. To be clear, in case the page does not contain the `dirent` definition, you should look in the `SEE ALSO` section for man pages that might give us more information; the list might look like the following:

```
SEE ALSO
       read(2), closedir(3), dirfd(3), ftw(3), opendir(3),
       rewinddir(3), scandir(3), seekdir(3), telldir(3),
       feature_test_macros(7)
```

Among the pages listed above, the most likely candidate would be the man page for `opendir()`; this is the function that most likely is the first one to call to do any directory processing. If that page does not have the definition, then we would have to read the man page for `<dirent.h>` or read the header file itself.

One way or another, you will discover that the `dirent` structure is defined as follows:

```
struct dirent {
       ino_t          d_ino;       /* inode number */
       off_t          d_off;       /* offset to the next dirent */
       unsigned short d_reclen;    /* length of this record */
       unsigned char  d_type;      /* type of file; not supported
                                      by all system types */
       char           d_name[256]; /* filename */
};
```

But you will find, whether in the Section 3 or 3P man page for `readdir()` or in the page for the `<dirent.h>` header file, a *caveat* such as the following:

> According to POSIX, the `dirent` structure contains a field `char d_name[]` of unspecified size, with at most `NAME_MAX` characters preceding the terminating null byte. POSIX.1-2001 also documents the field `ino_t d_ino` as an XSI extension. Use of other fields will harm the portability of your programs.

Translation: the only two members that are guaranteed to be present in a `dirent` structure are the `d_ino` and the `d_name` members, and the `d_name` member is not necessarily a fixed length string; POSIX only guarantees that it is a `NULL`-terminated string whose length is at most `NAME_MAX` characters. Therefore, whatever we do, we should not use those other members, and we should make sure that any variables that store a name are large enough to store `NAME_MAX` characters. We will return to the question of how to use `NAME_MAX` in a program.

We need to figure out how to use `readdir()`, what a `DIR` is, and how we get a `DIR*` to use in `readdir()`. The man page for `readdir()` basically says that successive calls to `readdir()` with the same *directory stream* pointer return successive entries in the directory pointed to, and that

when all entries have been accessed, a `NULL` pointer is returned. This is very much like how the `getutent()` function worked with `utmp` structures, and that function required that the library be initialized by a call to `setutent()`. It would be a good guess that the `opendir()` function listed in the `SEE ALSO` section performs this initialization for `readdir()`.

The man page for `opendir()` begins as follows:

```
SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>

       DIR *opendir(const char *name);

DESCRIPTION
      The opendir() function opens a directory stream corres-
      ponding to the directory name, and returns a pointer to the
      directory stream. The stream is positioned at the first
      entry in the directory.

RETURN VALUE
      The opendir() function returns a pointer to the directory
      stream. On error, NULL is returned, and errno is set
      appropriately.
```

In other words, given the pathname of a directory as a string, `opendir()` returns a pointer to something called a `DIR` that the documentation refers to as a *directory stream*. Names in uppercase letters are usually macros, defined in a header file with a "`#define`" directive, and the `DIR` type may be a macro. It could be a `typedef` as well. We do not need to know what a `DIR` is to open a directory and read from it, so for now we will move on.

In the same way that correct use of `getutent()` requires that we call `endutent()` to clean up after accessing all `utmp` records, correct use of `readdir()` requires that we clean up afterwards. The `SEE ALSO` section contains a reference to a `closedir()` function. If we look at its man page we will see that `closedir()`, given a pointer to a `DIR`, closes the connection to the directory, which is exactly what we must do when `readdir()` has returned a `NULL` pointer.

From the man pages for `readdir()` (in 3 and 3P), `opendir()`, and `dirent.h`, we can piece together how the directory reading interface works. When a directory is opened using `openddir()`, a static variable points to the first entry in the directory. The `readdir()` call reads the entry pointed to by this hidden variable, and increments the pointer so that it points to the next entry. Thus, successive calls iterate through the directory. If at any time, the program needs to know which entry it is about to read, `telldir()` :

```
#include <dirent.h>
long telldir(DIR *dirp);
```

returns its index, which is an integer offset from the beginning of the directory stream. If the program needs to start all over again without closing the directory, `rewinddir()` will do that, and `seekdir()` will move the pointer to a specified index:

```
#include <dirent.h>
void seekdir(DIR *dirp, long offset);
```

The offset returned by `telldir()` can be passed to `seekdir()`.

Although we do not need to know the structure of a `DIR` to use these calls, curiosity beckons us. What you will discover is that `DIR` is not a macro, but a typedef:

```
typedef struct __dirstream DIR;
```

and that there is no definition of the `struct __dirstream` in any exposed header files in the system. This is because `__dirstream` is an incomplete type. An incomplete type is a type that describes an object but lacks the information needed to determine its size. Each implementation of UNIX must define it, and is free to define it as it chooses, but it need not expose that implementation[10] in any header files. The `<dirent.h>` header file declares the `struct __dirstream` and makes `DIR` equivalent to it, but does not define its members. This gives programmers the ability to declare objects of type `DIR*`, but not the ability to access the members of a `DIR` object.

Our curiosity satisfied, we set this aside and tackle the `NAME_MAX` problem that arose earlier. There is no man page search that can tell you where a constant such as `NAME_MAX` might be defined, i.e. which header file needs to be included in a program to access its definition. One can do a search for that string in all of the header files on the system, but then one has to know all of the places containing header files. Such a search will probably fail in `/usr/include` for example, because `NAME_MAX` is a system-dependent value contained someplace where implementation-dependent header files are stored. One can do an exhaustive recursive `grep` for " `NAME_MAX` " (the spaces are important) and one will find it:

```
$ grep -R " NAME_MAX " /usr/* 2>/dev/null
```

In `bash`, the redirect `2>/dev/null` throws away all messages sent to standard error. The output of this command may look something like

```
/usr/include/linux/limits.h:#define NAME_MAX         255
/* # chars in a file name */
/usr/include/glib-1.2/glib.h:#    define NAME_MAX 255
```

followed by lines from other files that may or may not define the macro.

`NAME_MAX` is a system limit in UNIX. It specifies the maximum number of characters in a file name. As such it is one of many such constants defined in the header file `<limits.h>`. This fact is not as easily discovered as some of the others so far. One must include this header file to use the constant. In the next chapter, we will cover how to discover and use system limits in programs. For the moment, we use just a few.

We have enough knowledge now to make an attempt at writing `ls`.

[10] If you write a program that references a `__dirstream` object, the compiler will give an error that its size is unknown. If you instead declare a `__dirstream*` pointer, the program will compile, because the pointer's size is known. You will not be able to dereference this pointer and use what it points to because your program does not have an implementation of it. But it is implemented in the libraries that use it. This is an example of information hiding in C.

3.9 Implementing `ls`

The first attempt at writing `ls` will handle multiple command line arguments and not much more. For each argument, if it is a directory, it will display its contents, in no particular order. One problem is solved for us by the `opendir()` call; the call will fail, returning a -1 if the argument is not a directory. This will be our test for whether or not a filename is that of a directory.

```
1  Listing: ls.c Version 1
2  #include  <stdio.h>
3  #include  <sys/types.h>
4  #include  <dirent.h>
5
6  #define   HERE      "."
7  void ls(char []);
8
9  int main(int argc, char *argv[])
10 {
11     int i = 1;
12
13     if ( 1 == argc )     // no arguments; use .
14         ls( HERE );
15     else
16        // for each command line argument, display contents
17         while ( i < argc ){
18             printf("%s:\n", argv[i] );
19             ls( argv[i] );
20             i++;
21         }
22 }
23
24 // List the contents of the directory named dirname
25 // Uses opendir to check whether argument is a directory
26 // Doesn't check if argument is or "." or ".."
27 void ls( char dirname[] )
28 {
29     DIR             *dir_ptr;     // directory stream
30     struct dirent   *direntp;     // hold one entry
31
32     if ( ( dir_ptr = opendir( dirname ) ) == NULL )
33         // Could not open -- maybe it was not a directory
34         fprintf(stderr,"Cannot open %s\n", dirname);
35     else
36     {
37         while ( ( direntp = readdir( dir_ptr ) ) != NULL )
38             printf("%s\n", direntp->d_name );
39         closedir(dir_ptr);
40     }
```

```
41  }
```

Comments

- The real work is done by the `ls()` function, which will print an error if the file is not a directory. It discovers this when it tries to open the directory using the `opendir()` call. This call will return a `NULL` pointer if it cannot open the directory or if it is not a directory. It sets `errno` to a specific value to indicate the cause of the error, but we ignore `errno` in this version.
- This version lists all files, including dot-files.
- It does not sort the files, which the real `ls` does.
- It does not put the output into columns, which the real `ls` does.
- It does not handle any command-line options.
- It does not display information about the three special bits of the file.

This was just a warm-up exercise. Now we know some of the problems that we have to address. We write a second version, trying to solve some, but not all, of the problems. We also add the ability to display long listings.

3.10 A Second Version of ls

3.10.1 Adding the -l Option to ls

The `-l` option to `ls` displays file attributes. The set of displayed attributes for files that are not device files is defined in the table below.

Field Name	Description
mode	A ten character field in which the first character is a single letter that denotes the file type. A "-" means it is a regular file; a "d" means it is a directory. There are other types as well. This was described in Chapter 1. The remaining nine characters define the file mode, also described in Chapter 1.
number of links	The number of names this file has in all directories combined. A single file may have entries in multiple directories, possibly with the same name or with different names.
owner name	The user-name of the user who owns this file.
group name	The name of the group to which this file belongs. If the group does not have a name its group-id might appear instead.
size in bytes	The actual number of data bytes in the file, not the number of bytes in the blocks allocated to it, except that if the file is a directory, it is the number of bytes in the blocks allocated to the directory and is therefore a multiple of the block size.

Field Name	Description
date of last modification	The date and time that the file was modified last. If the file is relatively new, it shows month, day, time, but if it is older it shows month, day , year. The definition of new varies, depending on which standard the implementation adheres to. In POSIX compliant ls, a file is recent if the date is within the past 6 months. Also, the exact date format can be changed with various options.
file name	The name of the file in the directory.

If a file is a symbolic link, then the default behavior of `ls` is to display this information about the link itself, not the file that it references. To display the referenced file's attributes, `ls` must be given the `-L` option (on most systems) in addition to the `-l` option.

The question is, how does the `ls` command access this information, which we know is in the i-nodes associated to the file? We need a function that accesses the information in an i-node. Once again, the way to find out is by searching the man pages. This time it is likely to be difficult to find the information. Searching for man pages referring to i-nodes will come up empty. We can try searching for a piece of information from the i-node, such as mode, size, or date of last modification, but these terms are too common and might produce long lists. The key is that the information in an i-node is called the file *status*. Searching for status will succeed:

```
$ man -k status | grep file
fileno [ferror]      (3)  - check and reset stream status
fstat                (3p) - get file status
fstat [stat]         (2)  - get file status
ifcfg-ppp0 [pppoe]   (5)  - Configuration file used by adsl-start(8), adsl-stop(8),
                            adsl-status(8) and adsl-connect(8)
lam_rfstate          (2)  - Report status of remote LAM file descriptors
lstat [stat]         (2)  - get file status
stat                 (1)  - display file or filesystem status
stat                 (2)  - get file status
stat                 (3p) - get file status
```

The returned list may be different than this, but it will contain the `stat` family of calls: `stat()`, `lstat()`, and `fstat()`, probably with two pages for `stat()`, one in Section (2) and another in (3) or (3P). The `stat` (3P) man page is the POSIX version. The man page for `stat` (2) begins with:

```
NAME
     stat, lstat, fstat - get file status

SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>

     int stat(const char *path, struct stat *buf);
     int lstat(const char *path, struct stat *buf);
```

```
     int fstat(int fildes, struct stat *buf);

DESCRIPTION

     These functions return information about the specified
     file. You do not need any access rights to the file to get
     this information but you need search rights to all
     directories named in the path leading to the file.

     stat stats the file pointed to by file_name and fills in
     buf.

     lstat is identical to stat, except in the case of a
     symbolic link, where the link itself is stat-ed, not the
     file that it refers to.

     fstat is identical to stat, only the open file pointed
     to by filedes (as returned by open(2)) is stat-ed in place
     of file_name.

     They all return a stat structure which has the following
     fields:

    mode_t   st_mode;   /* File mode (see mknod(2)) */
    ino_t    st_ino;    /* Inode number */
    dev_t    st_dev;    /* ID of device containing a */
                        directory entry for this file */
    dev_t    st_rdev;   /* ID of device */
                        /* This entry is defined only for */
                        /* char special or block special files */
    nlink_t  st_nlink;  /* Number of links */
    uid_t    st_uid;    /* User ID of the file's owner */
    gid_t    st_gid;    /* Group ID of the file's group */
    off_t    st_size;   /* File size in bytes */
    time_t   st_atime;  /* Time of last access */
    time_t   st_mtime;  /* Time of last data modification */
    time_t   st_ctime;  /* Time of last file status change */
                        /* Times measured in seconds since */
                        /* 00:00:00 UTC, Jan. 1, 1970 */
    long     st_blksize;/* Preferred I/O block size */
    blkcnt_t st_blocks; /* Number of 512 byte blocks allocated*/
```

In short, we pass the `stat()` or `lstat()` function the pathname to the file and it fills in the `stat` structure pointed to by the second argument. If the pathname is not absolute, `stat()` and `lstat()` treat it as relative to the working directory. This is important, because if we pass it just a filename, the working directory of the process must be the one containing that filename. If a file is a symbolic link, `lstat()` displays information about the link itself, rather than the file to which it points, so if we want our program to behave like the real `ls`, we should use `lstat()` instead of `stat()`.

There is more information returned by a call to one of the `stat()` functions than is displayed by `ls -l`, so we have to select which members of the `stat` structure we need to display. If you were wondering, the POSIX description of the structure in Section (3P) is similar[11]. It will point out

[11] In fact, the POSIX page now contains example source code that solves the ls problem.

that some of the members of the `stat` structure are required and others are not. All of the members that we need are in the POSIX definition of this structure.

Much of the work is writing a function that, when given a filename, displays a line of output in the form of `ls -l`; therefore we concentrate on that function as a start. Initially we will create a function that displays the individual pieces of information one per line. Once we have that working, we can format it to fit on a line. The "one-per-line" version will be named `print_file_status()`. We can create a driver program that simply calls this function, using its command line argument as the file that we want to "stat". Since we already learned how to display time in Chapter 2, we can reuse that function here. A first pass at this function would be something like the following:

```
void print_file_status(char *fname, struct stat *buf)
{
    printf(" mode: %o\n", buf->st_mode);      // type + mode
    printf("links: %d\n", buf->st_nlink);     // # links
    printf(" user: %d\n", buf->st_uid);       // user id
    printf("group: %d\n", buf->st_gid);       // group id
    printf(" size: %d\n", buf->st_size);      // file size
    printf("mtime: %s\n",
            time2str(buf->st_mtime));         // last modified
    printf(" name: %s\n", fname );            // filename
}
```

Here, `time2str()` is a simple function that converts the `time_t` into a string without the "day of week" name at the start. It will be replaced by something else later. In this first attempt, the mode is numeric, the user and group are displayed as user and group-ids instead of with actual names. We will take each line one at a time and refine it so that it displays the information in the proper form. First we will fix up the mode.

3.10.2 Converting File Mode to String Format

The file mode is stored in the `st_mode` member of the `stat` structure as a 16-bit quantity. The 16 bits are used for different purposes.

The first four bits are reserved for the file type, such as regular file or directory. The next three bits are the special bits: the set-user-id bit, the set-group-id bit, and the sticky bit. We will get to these afterwards. The next nine bits are three sets of three bits each. Each set has a read, write, and execute bit. The three sets of bits are the user, group, and others sets of bits. A 1-bit means the permission or property is on, and a 0-bit, that it is off. With this in mind, it is easy to write code to convert this 16-bit quantity into a ten-character mode string using bit masks.

Other than the file type, which is stored in bits 12 to 15, all other flags are single bits in `st_mode`. In fact, UNIX provides all of the masks we need, in the file `<sys/stat.h>`. These masks are standardized in POSIX and are described in one of the `stat` man pages. The header file `<sys/stat.h>` has the definitions, which are replicated here. The single-bit masks are

```
S_ISUID     0004000    set UID bit
S_ISGID     0002000    set-group-ID bit (see below)
S_ISVTX     0001000    sticky bit (see below)
S_IRWXU     00700      mask for file owner permissions
S_IRUSR     00400      owner has read permission
S_IWUSR     00200      owner has write permission
S_IXUSR     00100      owner has execute permission
S_IRWXG     00070      mask for group permissions
S_IRGRP     00040      group has read permission
S_IWGRP     00020      group has write permission
S_IXGRP     00010      group has execute permission
S_IRWXO     00007      mask for permissions for others
S_IROTH     00004      others have read permission
S_IWOTH     00002      others have write permission
S_IXOTH     00001      others have execute permission
```

and the masks for extracting file type are

```
#define    S_IFMT      0170000   /* type of file */
#define    S_IFREG     0100000   /* regular */
#define    S_IFDIR     0040000   /* directory */
#define    S_IFBLK     0060000   /* block special */
#define    S_IFCHR     0020000   /* character special */
#define    S_IFIFO     0010000   /* FIFO */
```

POSIX defines macros for testing the file type, which are much easier than using the masks:

```
S_ISREG(m)     is it a regular file?
S_ISDIR(m)     directory?
S_ISCHR(m)     character device?
S_ISBLK(m)     block device?
S_ISFIFO(m)    FIFO (named pipe)?
S_ISLNK(m)     symbolic link?
S_ISSOCK(m)    socket?
```

We can use these masks to write a function to convert the numeric mode to the string that is displayed by ls. For now it does not handle the setuid, setgid, and sticky bits. Before incorporating them into the string, we should understand what they do and how they can be displayed in the mode string. After all, the string has ten characters and we need all ten – 1 for type, and 3 sets of 3 for permissions – so where could we display the values of the setuid, setgid, and sticky bits anyway?

The following function, `mode2str()`, given the mode as a 16-bit integer, fills the character string `str` with a permission string in the standard format.

```
void mode2str( int mode, char str[] )
{
    strcpy( str, "----------" );

    if ( S_ISDIR(mode) )        str[0] = 'd';    // directory?
    else if ( S_ISCHR(mode) )   str[0] = 'c';    // char devices
    else if ( S_ISBLK(mode) )   str[0] = 'b';    // block device
    else if ( S_ISLNK(mode) )   str[0] = 'l';    // symbolic link
    else if ( S_ISFIFO(mode))   str[0] = 'p';    // Named pipe (FIFO)
    else if ( S_ISSOCK(mode))   str[0] = 's';    // socket

    if ( mode & S_IRUSR ) str[1] = 'r';     // 3 bits for user
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';     // 3 bits for group
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';     // 3 bits for other
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';
}
```

3.10.3 Converting User/Group ID to Strings

The next step is to convert the user-id and the group-id to names. We need a function that is given a user-id and returns the name of the person with that user-id, and a function that is given the group-id and returns the name of the group. How do we find such a function? We can try man page searches using keywords such as username, name, user, and so on. After a few tries like this, you could hone it down to something like

```
$man -k name | grep user
```

You will come up with a not-too-long list that includes the following prospect:

```
getpwnam (3p) - search user database for a name
```

If we look at this man page, we will discover that, while `getpwnam()` is not what we want, `getpwuid()` is. Both of these functions access the password database, regardless of where the actual file is located, i.e., even if it is the network database, and return a `passwd` structure. The function `getpwuid()` accesses the password database by user-id and `getpwnam()` accesses it by username. We have a user-id, but we need the name, so we want the `getpwuid()` function. It is given a user-id as an argument and it returns a pointer to a `passwd` structure, which is defined in `/usr/include/pwd.h` as

```
struct passwd {
        char  *pw_name;       /* Username */
        char  *pw_passwd;     /* Password */
        _uid_t pw_uid;        /*  User ID */
        _gid_t pw_gid;        /*  Group ID */
        char  *pw_gecos;      /* Real name */
        char  *pw_dir;        /* Home directory */
        char  *pw_shell;      /* Shell program */
}
```

The pointer returned by `getpwuid()` can be dereferenced to access an individual member of the `passwd` structure, provided that the pointer is not `NULL`. The man page for `getpwuid()` notes that the returned pointer will be `NULL` if there was no matching entry in the password database. The application must check for this.

You may wonder why there may not be a matching name. It sometimes happens that a user account is deleted from a UNIX system, but some files created by that user are left behind. For example, the `/tmp` directory is usually world-writable, and anyone can put files there. If the deleted user account had created files there, the system administrator probably did not delete them, since searching through the entire file system for traces of a user's files is resource-consuming. If `ls` were trying to list the contents of `/tmp`, it would not be able to display that username.

It does not end there. Suppose that someone else had hard links to the deleted user's files. Even if the administrator deleted the "original" files created by that user, the actual files still exist because their link counts are not zero. The user-id of the owner of the file is stored in the i-node of the file, not the directory entry. So, for example, if George linked to a file owned by Sherry and Sherry's account was deleted, then George's links are still owned by Sherry. The administrator cannot rightly delete these links if George needs them, and a listing of George's files would fail to display a username.

It is also possible that the user-id does correspond to a legitimate username, but the username and user-id are defined in an NIS map from a different domain than the one to which the host belongs. For example, if a file system is remotely mounted from a different NIS domain, then the owners of the files will have user-ids and names from that domain, and the `getpwuid()` function will not be able to find the username. As a concrete example, suppose a host named `earth` is in the `geography` domain, but I have an account on `earth` in that domain (say `sweiss@geography`) and I, for a while, was remotely mounting `earth`'s file system on my desktop machine, which lived in the `csci` domain. All of the files that I saw belonged to me, `sweiss`, but it was not the me I knew; it was `sweiss@geography`, so the user-id was different than the one in the `csci` NIS maps. As a result, `getpwuid()` would fail to find that user-id in the `csci` maps. So you see that `getpwuid()` can fail for that reason as well.

Since there may not be an entry in the passwd file for a given user, there may not be a user-name associated with the file. There is still a user-id, but no user-name. In this case, the best that ls can do is to print the user-id.

This is not the end of it. It is possible that `ls` finds a name but it is the wrong name. Suppose that the administrator reused the deleted user-id for a new user, not knowing that there were still files from the old user. When this happens, `getpwuid()` will find the user-id in the password database, but it will be wrong. It will now be associated to a different person, an impostor, if you like, for the old owner. `ls` will display the new user-name as the owner of the files, as the old account rolls

over in its cybergrave, and as far as UNIX is concerned, the new user is now the inadvertent owner. There is nothing we can do to prevent this.

What about group names? It turns out that a similar database, similar functions, and similar problems exist for groups. The `/etc/group` file lists all of the groups in the system, together with the users who are in each group. The file has an entry for each group consisting of the group name, the group password, the group-id and a comma-separated list of user-names, such as

```
root::9:root
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
staff::4:tbw,snw
```

From the list above you can see that `root` belongs to more than one group. In general, users can belong to more than one group. The primary group of a user is the group specified in the password file entry for that user. When a user creates a file, the user's primary group is made the group of the file. The owner of a file can change the group of a file with the `chgrp` command.

The `getgrgid()` system call, given a group-id, searches the list of groups and returns a pointer to the group structure of the group whose group-id matches, or `NULL` if no such group is found. Similarly, the `getgrnam()` function searches the group database by name and returns a pointer to the group structure that matches that name, or `NULL`. The group structure is defined in `/usr/include/grp.h` as

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* group members */
};
```

It is easy to write a function to convert group-ids to group-names, provided that such groups have names. The same arguments about passwords apply to groups; it is possible that the group-id will not resolve to a correct group name.

3.10.4 Formatting Time

In Chapter 2, we used `ctime()` to format time values. Here we will use `strftime()` and `localtime()` together, so that our program can behave more closely to the real `ls` program. The real `ls` program formats times differently depending upon the user's locale. In addition, for files whose time of last modification is not "recent", the format is different. The definition of "recent" is six months or less. Of course six months is different at different times of the year, so this is an approximate definition. We will use the financial sector's definition of six months – 182 days.

For files that are not recent, ls displays the month name, the day of the month, and the year. For recent files, it displays the month, the day of the month, and the hour and minute. The exact format

depends upon the locale settings. We will put this logic into a function named `get_date_no_day()`, since it never displays the day of the week. this function is given a time_t value and produces a pointer to a statically allocated string as its return value. If that value is not copied before the function is called again, it will be overwritten. The function's logic is

1. Check if the file is recent by comparing current_time - given_time to the number of seconds in 182 days.
2. Create a struct `tm` from the given_time value.
3. If the file is not recent, format it in the format `MMM DD YYYY`.
4. If the file is recent, first try to format it using the user's locale settings.
5. If that fails, format it using the format `MMM DD HH:MM`

The function is shown in the listing below.

```
char* get_date_no_day( time_t timeval )
{
    const int  sixmonths = 15724800; /* number of secs in 182 days */
    static char outstr[200];
    struct tm *tmp;
    time_t current_time = time(NULL);
    int     recent = 1;

    if ( ( current_time - timeval ) > sixmonths )
        recent = 0;

    tmp = localtime(&timeval);
    if (tmp == NULL) {
        perror("get_date_no_day: localtime");
    }

    if ( ! recent ) {
        strftime(outstr, sizeof(outstr), "%b %e  %Y", tmp);
        return outstr;
    }
    else if (strftime(outstr, sizeof(outstr), "%c", tmp) > 0)
        return outstr+4;
    else {
        printf("error with strftime\n");
        strftime(outstr, sizeof(outstr), "%b %e %H:%M", tmp);
        return outstr;
    }
}
```

3.10.5 Getting the Name of the Reference of a Link

If the file is a symbolic link, we need to print out the pathname of the file to which it points. If we do a manpage search for functions that might work with links, i.e.

```
man -k link | grep '([23])'
```

we will see, among the set of choices,

```
readlink (2)         - read value of a symbolic link
```

This is apparently what we need. Reading its manpage we see that the `readlink()` function reads the value of a link:

```
SYNOPSIS
       #include <unistd.h>
       ssize_t readlink(const char *path, char *buf, size_t bufsiz);
   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
       readlink(): _BSD_SOURCE || _XOPEN_SOURCE >= 500 ||
       _POSIX_C_SOURCE >= 200112L
DESCRIPTION
       readlink() places the contents of the symbolic link path in the  buffer
       buf,  which has size bufsiz.  readlink() does not append a null byte to
       buf.  It will truncate the contents (to a length of bufsiz characters),
       in case the buffer is too small to hold all of the contents.
```

`readlink()` fills its second argument with the contents of the symbolic link path. This storage is allocated by the calling program, not by `readlink()`, so you need to declare a large enough string to hold the pathname. You could declare this to be of size `PATH_MAX`[12], but this will be extremely large and it is probably better to truncate the name if it is very large. Because `readlink()` does not append the null byte to the string, the proper usage of this function is roughly as follows, if `filename` is the link being read:

```
ssize_t count;
if ( -1 == (count = readlink(filename, buf, NAME_MAX-1)) )
    perror("function calling this ");
else {
    buf[count] = '\0';
    // use the buf contents here
}
```

`NAME_MAX` is the maximum length of a file name, usually around 255. This is also probably too large.

A second version of `ls` follows. This version still does not parse the command line to detect whether the `-l` option is present, but the code is revised to make it easy to incorporate that change later. The other deficiencies are noted after the listing.

[12]Recall from Chapter 1 that `PATH_MAX` is a system dependent limit specifying the maximum number of bytes in a pathname.

```
Listing ls.c Version 2
void    print_file_status( char *, struct stat *);
void    mode2str( int , char [] );
char * get_date_no_day( time_t * time );
char * uid2name( uid_t );
char * gid2name( gid_t );
void    ls(char [], int);

int main(int argc, char *argv[])
{
    int i = 1;

    if ( argc == 1 )     // no arguments; use .
        ls( HERE );
    else
        while ( i < argc ){
            printf("%s:\n", argv[i] );
            ls( argv[i], 1 );
            i++;
        }
}

// Does not check whether entries are files or directories
// or . files
void ls( char dirname[], int do_longlisting )
{
    DIR             *dir_ptr;     // directory stream
    struct dirent   *direntp;     // holds one entry
    struct stat     info;         // stores stat results

    if ( ( dir_ptr = opendir( dirname ) ) == NULL )
        // could not open -- maybe it was not a directory
        fprintf(stderr,"ls1: cannot open %s\n", dirname);
    else {
        while ( ( direntp = readdir( dir_ptr ) ) != NULL ) {
            if ( do_longlisting ) {
              if (lstat(direntp->d_name, &statbuf) == -1) {
                  perror(direntp->d_name);
                  continue; // stat call failed but we go on
              }
              print_file_status(direntp->d_name, &statbuf);
            }
            else // not long -- just print name
               printf("%s\n", direntp->d_name);
        }
        closedir(dir_ptr);
    }
}

void print_file_status( char *filename, struct stat *info_p )
{
    char    modestr[11];
    ssize_t count;
    char    buf[NAME_MAX];
```

```
    mode2str( info_p->st_mode, modestr );
    printf( "%s"    , modestr );
    printf( "%4d "  , (int) info_p->st_nlink);
    printf( "%-8s " , uid2name(info_p->st_uid) );
    printf( "%-8s " , gid2name(info_p->st_gid) );
    printf( "%8ld " , (long)info_p->st_size);
    printf( "%.12s ", get_date_no_day(info_p->st_mtime));
    printf( "%s"    , filename );
    if ( S_ISLNK(info_p->st_mode) ) {
        if ( -1 == (count = readlink(filename, buf, NAME_MAX-1)) )
            perror("print_file_status: ");
        else {
            buf[count] = '\0';
            printf("->%s", buf );
        }
    }
    printf("\n");
}

// Given user-id, return user-name if possible
char *uid2name    ( uid_t uid )
{
    struct  passwd *pw_ptr;
    static  char numstr[10];  // must be static!

    if ( ( pw_ptr = getpwuid( uid ) ) == NULL ) {
        // convert uid to a string; using sprintf is easiest
        sprintf(numstr,"%d", uid);
        return numstr;
    }
    else
        return pw_ptr->pw_name;
}

char *gid2name     ( gid_t gid )
{
        struct group *grp_ptr;
        static  char numstr[10];

        if ( ( grp_ptr = getgrgid(gid) ) == NULL ) {
        // convert gid to string
                sprintf(numstr,"%d", gid);
                return numstr;
        }
        else
                return grp_ptr->gr_name;
}
```

What Is Still Wrong?

- It still treats all arguments like directories and displays an error if given a regular file.

- It still will not correctly display the files in directories specified on the command-line unless they are within the current working directory.
- The program does not print the total lines printed.
- It does not sort by filename.
- It displays all entries, including . and ..
- It does not display information about the three special bits of the file.

The most serious of these problems is that this version will fail if the directory argument is not in the current working directory. For example, if this is called as follows:

```
$ ls2 /
```

passing it the root directory, and there are subdirectories `/etc`, `/usr`, and `/bin`, the `stat()` function will be given the filenames `"etc"`, `"usr"`, and `"bin"`. It will treat these as if they are in the current working directory since it has no way of knowing their absolute pathnames. It will fail to find them in the current working directory and will thus fail. The solution is to pass `lstat()` the absolute pathnames of these directories, or their correct relative names. This is accomplished by passing `lstat()`, for example, `"/etc"` instead. This implies that we need to concatenate the path of the directory argument to its file names. We could also change the working directory each time we need to list its contents, saving the old one to return to afterwards.

3.11 The Three Special Bits

A file's i-node and the `st_mode` member of the `stat` structure returned by the various `stat()` calls, each use 16 bits to define the file's permissions and type. We have accounted for a 4-bit type and 9 bits of permissions. That leaves three unexplained bits. The three bits are the *set-user-id* bit (setuid bit), the *set-group-id* bit (setgid bit) and the *save-text-image* bit (the sticky bit).

3.11.1 The Set-User-ID Bit

The setuid bit plays a critical role in UNIX. Consider the problem of allowing users to change their own passwords. A user should be able to change her password, but no one else should be able to do so, except the super-user. Since all passwords are stored in a single file, the user has to have permission to modify the password file. This implies that the user needs write permission on the password file. But if the user has write permission on the password file, then she can modify anyone else's password too, which is not acceptable. So users cannot have write permission on the password file; the file should be owned by root, and no user except the superuser should have write permission to the file. So we are back where we started, right?

Not exactly. To change a password, a user runs the `passwd` command. The `passwd` command accesses the password file and changes it. Ordinarily, when a user runs a command, the process executing the command has the same effective user-id as the user who invoked the command from the shell, and the permissions associated with that effective user-id. If this were true for the `passwd`

command, then the `passwd` program would not be able to modify the password file, since only `root` has write permission on it. What if somehow we could give the `passwd` program the permissions associated with `root` so that it could modify the password file. Enter the setuid bit.

Recall that, at any given time, a process has two associated user-ids called its real user-id and its effective user-id. The real user-id can never be changed. The effective user-id can vary. Ordinarily, when a program is run, its effective user-id is set to be the effective user-id of the process that created it, such as the shell. This is usually the real user-id of the user who indirectly ran the process. But if the setuid bit is set on an executable file, the effective user-id of the process that executes the program in that file is the user-id of the owner of the file. In this case, `root` owns the `/usr/bin/passwd` program file, so when `passwd` runs, its effective user-id is that of `root`.

Recapping, the `passwd` program is owned by `root`, but it has its setuid bit turned on. When a user executes this program, it runs with `root` as the effective user-id, not the user. Because the program checks what the real user-id of the caller is by calling `getuid()`, it knows that it is being run by a specific user and it can access the appropriate line of the password file. This prevents `passwd` and consequently the user from modifying anything other than the password of its own entry. Other uses of the setuid bit include protecting global game data, protecting the print spooler, protecting global databases in general.

3.11.2 The Set-Group-ID Bit

The set-group-id bit is similar to the set-user-id bit except it sets the effective-group-id of the running program. If a program belongs to a group and has the setgid bit on, then when the program runs, it runs with the privileges accorded to the group that owns it rather than the privileges of the group that runs it.

The set-group-id bit is used by the `write` command. The `write` command (`/usr/bin/write`), not the `write()` system call, is a command that lets users write to a terminal. The syntax is

```
write username [ ttyname]
```

After entering this command, all input will be displayed on the given user's terminal, until an end-of-input signal (Ctrl-D) is received. To try it, type `who` to see who is logged on, and which terminals they are using. Suppose I am logged in on terminal `/dev/pts/2`. You could type

```
$ write sweiss /dev/pts/2
Can I bother you?
Ctrl-D
```

and wait for my response. Your typing will appear on my terminal window. How is it possible that one person can write on another person's terminal?

The `write` command needs write permission on the terminal on which it wants to write. First take a look at the list of pseudo-terminal devices in `/dev/pts`. The list will look something like

```
crw-------  1 tlewis   tty 136, 1 Mar  5 17:50 1
crw--w----  1 tbw      tty 136, 3 Mar  3 16:22 3
crw-------  1 nguyen04 tty 136, 4 Mar  5 10:36 4
crw--w----  1 tbw      tty 136, 5 Mar  3 15:40 5
crw--w----  1 sweiss   tty 136, 7 Mar  5 18:00 7
```

Notice that some of these have the group write bit set and others do not. Notice that all terminals belong to the `tty` group. This means that any process that runs with an effective group-id of `tty` can write to those terminals whose write bit is set. Now take a look at the write command's status:

```
$ ls -l /usr/bin/write
-rwxr-sr-x 1 root tty 10124 Jul 27 2005 /usr/bin/write*
```

and observe that the `write` executable is in the `tty` group and its setgid bit is set. When a user runs `write`, the process that executes it runs with the effective group-id of the `write` program, which is the `tty` group. This implies that the `write` command will be able to write to any terminal whose group write bit is set. Since it can be annoying to receive messages on your terminal while you are working, UNIX provides a simple command to query, enable, or disable this bit:

```
$ mesg [y/n]
```

If you type `mesg` alone, it will display `y` or `n`, depending on whether the bit is set. Typing `mesg y` turns it on, and `mesg n` turns it off.

3.11.3 The Sticky-Bit

The *sticky bit*, also called the *save-text-image bit*, serves two different purposes when it is applied to files and directories. Originally, UNIX was a pure swapping operating system. Processes were swapped in and out of memory to maintain the multiprogramming level. The swapping store was a separate disk or a separate partition of a disk that was used exclusively for storing process images when they were swapped out. The executable code and other data were kept in contiguous bytes on the swapping store, making reads and writes faster.

A program that was used by many people might go through many memory loads and unloads each day. Putting it in the swapping store made loads and unloads easier, because the file was in one piece. Setting the sticky bit on a program file prevented it from being removed from the swapping store.

If a directory has the sticky bit on, then any file you place in the directory will be protected from being deleted by anyone except you. You put a file in the directory and no one but you can remove that file. The directory is readable and writable by everyone, but the sticky bit prevents one person from deleting another person's files in that directory. This is how UNIX can implement directories such as `/tmp`, which is used to store temporary files.

3.11.4 The Special Bits and `ls`

How does `ls -l` display these three bits? If the set-uid bit is turned on, the permission string has an `s` instead of an `x` in the owner set:

```
-rws------
```

instead of

```
-rwx------
```

If the setgid bit is set, the second `x` becomes an `s`:

```
-rwxrws---
```

If the sticky bit is set, the rightmost character is a `t` instead of an `x` or a dash:

```
-rwxrwxrwt
```

We simply have to test these bits and modify the string accordingly. The revised `mode2str()` function is included in the listings that follow.

3.12 A Final Version of `ls`

The final version of the `ls` program can use the bit masks described above to display the appropriate letter in case the set-uid, set-gid, or sticky bit is set on a file. The problem of not displaying the files when the directory argument is not within the current working directory has an easy solution. The argument to the `stat()` call must be constructed by concatenating the directory name, a slash, and the directory entry, as in

```
...
while ((dp = readdir(dir)) != NULL) {
    sprintf(fname,"%s/%s",dirname, dp->d_name);  // changed here
    if (stat(fname, &statbuf) == -1) {
...
```

The `sprintf()` function is used to concatenate the directory name and the directory entry with the slash in between. This way, the argument to `stat()` will be either an absolute path name, if the directory argument was, or it will be relative to the working directory, since if it were not, the `opendir()` call would fail, and that would be the user's error in passing a non-existent directory name to the program.

The error that occurs when a command-line argument of `ls` is not a directory is slightly more work to remove. The problem is that the `ls` function called by `main()` starts out with

```
if ( ( dir_ptr = opendir( dirname ) ) == NULL )
    // could not open -- maybe it was not a directory
    fprintf(stderr,"Cannot open %s\n", dirname);
else {
    ...
```

The `opendir()` call fails because the argument is not a directory, and the program displays an error, e.g.

```
Cannot open foo
```

Instead we need to first test whether the file is a regular file or a directory. In order to do this, we need to call `stat()` with the file name, and check whether the file is a directory by using the `S_ISDIR()` macro on the `st_mode` member. The following is how the function should start:

```
if ( lstat(dirname, &statbuf) == -1 ) {
    perror(fname);
    return;       // stat call failed so we quit this call
}
else if ( ! S_ISDIR(statbuf.st_mode) ) {
    if ( do_longlisting )
        print_file_status(dirname, statbuf);
    else
        printf("%s\n", dirname);
    return;
}

if ( ( dir = opendir( dirname ) ) == NULL )
    fprintf(stderr,"Cannot open %s\n", dirname);
else {
    ...
```

The main program needs to be modified to parse the command line, using the `getopt()` function that POSIX requires of a compliant UNIX system. We need to include the `<unistd.h>` header file to use it. For this version we have a single option letter, "`l`", but the program can easily be extended to include other options. The main program is in the following listing, and the supporting functions are in the listing that follows it.

```
Listing ls.c Final Version of main()

void ls( char dirname[], int do_longlisting );
void print_file_status  ( char *fname,
                          struct stat statbuf );
char* mode2str          ( int mode  );
char *uid2name          ( uid_t uid );
char *gid2name          ( gid_t gid );

int main(int argc, char *argv[])
{
    int longlisting = 0;
    int ch;
    char options[] = ":l";

    opterr = 0;  // turn off error messages by getopt()

    while  (1) {
        ch = getopt(argc, argv, options);
```

```
        // it returns -1 when it finds no more options
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'l':
            longlisting = 1;
            break;
        case '?' :
            printf("Illegal option ignored.\n");
            break;
        default:
           printf ("getopt returned character code 0%o ??\n",
                   ch);
           break;
        }
    }

    if ( optind == argc )     // no arguments; use .
        ls( ".", longlisting );
    else
        while ( optind < argc ){
            ls( argv[optind], longlisting );
            optind++;
        }

    return 0;
}
```

The supporting functions follow. Some functions are the same as in the previous version. Those are not listed to save space. A comment indicates this when appropriate.

```
void ls( char dirname[], int do_longlisting )
{
    DIR           *dir;         // pointer to directory struct
    struct dirent *dp;          // pointer to directory entry
    char          fname[PATH_MAX];// string to hold path name
    struct  stat  statbuf;      // to store stat results

    /* test if a regular file, and if so, just display it */
    if ( lstat(dirname, &statbuf) == -1 ) {
        perror(fname);
        return;       // stat call failed so we quit this call
    }
    else if ( ! S_ISDIR(statbuf.st_mode) ) {
        if ( do_longlisting )
            print_file_status(dirname, statbuf);
        else
            printf("%s\n", dirname);
        return;
    }

    if ( ( dir = opendir( dirname ) ) == NULL )
```

```
        fprintf(stderr,"Cannot open %s\n", dirname);
    else {
        printf("\n%s:\n", dirname);
        // Loop through directory entries
        while ((dp = readdir(dir)) != NULL) {
            if (strcmp(dp->d_name, ".") == 0 ||
                strcmp(dp->d_name,"..") == 0 )
                // skip dot and dot-dot entries
                continue;

            if ( do_longlisting ) {
                // construct a pathname for the file using the
                // directory name passed to the program and the
                // directory entry
                sprintf(fname,"%s/%s",dirname, dp->d_name);

                // fill the stat buffer
                if (lstat(fname, &statbuf) == -1) {
                    perror(fname);
                    continue;  // stat call failed but we go on
                }
                print_file_status(dp->d_name, statbuf);
            }
            else
                printf("%s\n", dp->d_name);
        }
    }
}


void print_file_status     ( char        *dname,
                            struct stat statbuf  )
{
    ssize_t count;
    char    buf[NAME_MAX];

    /* Print out type, permissions, and number of links */
    printf("%10.10s", mode2str (statbuf.st_mode));
    printf("%3d", (int) statbuf.st_nlink);

    /* Print out owner's name if it is found using getpwuid() */
    printf(" %-8.8s", uid2name(statbuf.st_uid));

    /* Print out group name if it is found using getgrgid() */
    printf(" %-8.8s", gid2name(statbuf.st_gid));

    /* Print size of file */
    printf(" %8jd", (intmax_t)statbuf.st_size);

    /* Print time of last modification */
    printf( " %.12s ", get_date_no_day(statbuf.st_mtime));

     /* print  file name  and if a link, the linked file */
    printf(" %s",  dname);
```

```
    if ( S_ISLNK(statbuf.st_mode) ) {
        if ( -1 == (count = readlink(dname, buf, NAME_MAX-1)) )
            perror("print_file_status: ");
        else {
            buf[count] = '\0';
            printf("->%s", buf );
        }
    }
    printf("\n");

}

char* mode2str              ( int mode  )
{
    static char str[11];
    strcpy( str, "----------" );           // default=no perms

    if ( S_ISDIR(mode) )        str[0] = 'd';   // directory?
    else if ( S_ISCHR(mode) )   str[0] = 'c';   // char devices
    else if ( S_ISBLK(mode) )   str[0] = 'b';   // block device
    else if ( S_ISLNK(mode) )   str[0] = 'l';   // symbolic link
    else if ( S_ISFIFO(mode))   str[0] = 'p';   // Named pipe (FIFO)
    else if ( S_ISSOCK(mode))   str[0] = 's';   // socket

    if ( mode & S_IRUSR ) str[1] = 'r';     // 3 bits for user
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';     // 3 bits for group
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';     // 3 bits for other
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';

    if ( mode & S_ISUID ) str[3] = 's';     // set-uid
    if ( mode & S_ISGID ) str[6] = 's';     // set-gid
    if ( mode & S_ISVTX ) str[9] = 't';     // sticky bit
    return str;
}

char *uid2name      ( uid_t uid   )
// same as previous version and omitted here.

char *gid2name      ( gid_t gid   )
// same as previous version and omitted here.
```

This last version of `ls` solves all of the important problems. The goal was really to understand how to interface to various system functions, and writing `ls` correctly was a way to get experience with a few important structures: the `dirent` object and the i-node.

3.13 Modifying File Attributes

The preceding exercise accessed attributes stored in the i-nodes, but did not modify any of them. This is a good time to consider which of how attributes can be modified by user level programs, and how. When appropriate, various shell commands that are related will be noted.

3.13.1 Type of a File

The type of a file is initialized when the file is first created and it cannot be changed after that. As mentioned before, a file can be a regular file, a directory, a device special file (of which there are several types), a socket, a symbolic link, or a named pipe.

The `creat()` system call creates regular files; the `mkdir()` call makes a directory. The `mkdir` command creates a new directory. In fact it is the only way to make a directory. Other calls make the other types of files. For example, `mknod()` is the system call that creates special files such as device special files. The `mknod` command makes these at the user level. There is also a special function to make FIFO special files, the `mkfifo()` call.

3.13.2 Permission Bits and Special Bits

The permission bits are initialized when the file is created by the kernel. The second argument of the `creat()` call, for example, is a number used to initialize the file mode. However, the mode assigned to the file is not exactly this argument; it is this number modified by applying the process's *umask*. The umask of the process is the umask of the effective user-id of the process.

Every user has a umask. The umask is an inverted mask; a 1 in the umask represents a bit to turn off in the masked value, i.e., the bitwise C operation `(mode & ~umask)` is applied. For example, if the umask is octal 022, then it is binary 000010010. This shows that bits 2 and 5 of the mask are set, which means bits 2 and 5 are turned off in the masked value. Since 2 is write by others and 5 is write by group, this umask sets the default file mode so that no one other than the owner can write to the file. The call,

```
fd = creat("newfile", 0766 );
```

would create the file named `newfile` with permission string `-rwxrw-rw-` if there were no umask. If the umask has value 022, then the umask is subtracted and the file permissions would be `-rwxr-r--`. Remember, the umask is an un-mask mask – it unsets the bits that are set in it. And it only does this when the file is created, not after.

A process inherits its umask value when it is started up[13], but it can change it by calling `umask()`:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

[13] All processes are created by other processes. When you run a program from the command line, the shell creates the process that executes the program and the shell is its parent process. The umask of the shell becomes the umask of the new process. The umask of the shell is initialized when the shell starts up, usually by reading a shell configuration script such as the .bash_profile file.

The `umask()` system call changes the umask to the bitwise AND of mask and octal 0777 and returns the value of the previous mask. In other words, it ignores the parts of the mode_t value passed to it that define the file type and special bit values. There is also a shell command named `umask` that can be used to inspect or change the user's umask.

A file's permission bits can be changed by a process by calling `chmod()` :

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

As with the mode in the `creat()` system call, the mode may be supplied as an integer or as the bitwise-or of one or more of the bitmasks described earlier – e.g. `S_IRUSR`.

Of course at the user level, `chmod` is the command to change the permission bits of a file. See the man page for all of the details on how to use it. There are many variations on it.

3.13.3 Number of Links to a File

The name of a file is just a name stored in a directory. The total number of names that a file has is called its *link count*. The i-node contains this link count. When the link count is decremented because a name is deleted, it is compared to zero. When it reaches zero, the file is actually removed.

The `link()` system call creates a new name for an existing file and the `unlink()` and `unlinkat()` calls remove a name for a file:

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
int unlink(const char *path);

#include <fcntl.h>
int unlinkat(int dirfd, const char *pathname, int flags);
Feature test macro on glibc for  unlinkat():
    Since  glibc 2.10: _XOPEN_SOURCE >= 700 || _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10: _ATFILE_SOURCE
```

If the new pathname exists already, `link()` will not overwrite it, instead returning -1 and setting `errno` to `EEXIST`. Most implementations require that both pathnames be on the same file system, although POSIX.1 allows an implementation to support linking across file systems. There are various other conditions that can cause it to fail as well, and you should read its man page for details.

Unlinking a file, whether with `unlink()` or `unlinkat()`, requires that the process has write permission and execute permission in the directory containing the name to be removed, since it is the directory entry that is removed. If the sticky bit is set in this directory the process must have write permission for the directory and either

- own the file,
- own the directory, or

- have superuser privileges.

If none of these are satisfied, `errno` is set to `EPERM`. The `unlink()` call may do more than just delete the name of the file. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse. However, if one or more processes still have the file open, it will remain in existence until the last file descriptor referring to it is closed, even if the link count went to zero. Some versions of Unix will set `errno` to `EBUSY` in this case (but not Linux.)

The `unlinkat()` system call does the same thing as `unlink()`, but it does so relative to a given directory that has been opened[14]. Specifically, one can use the `open()` system call to open a directory and get a file descriptor for it, and use that file descriptor as the reference point for the `pathname` second argument. Thus, a path relative to a given directory can be supplied. The last argument is a flag that can be used to make `unlinkat()` behave more like the `rmdir()` system call. If it is passed `AT_REMOVEDIR`, then it is as if `rmdir()` was called on `pathname`. The following example demonstrates its use.

```
Listing unlinkatdemo.c
/* usage:  unlinkatdemo  directory file
           where path is the path to a file relative to
           the given directory.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int   dirfd;

    /* check args       */
    if ( argc < 3 ){
        fprintf( stderr, "usage: %s directory file\n", *argv);
        exit(1);
    }

    /* open directory   */
    if ( (dirfd = open(argv[1], O_RDONLY)) == -1 ) {
        fprintf( stderr, "could not open %s\n", argv[1]);
        exit(1);
    }
    /* We could check here if argv[2] refers to a directory or a file
       to make this more robust, but we don't. */
```

[14]The reason that `unlinkat()` exists is to prevent possible race conditions which `unlink()` is susceptible to. A component of a pathname given to `unlink()` might change in parallel to the call if the pathname is not within the current working directory of the process. By opening a directory and getting its file descriptor, the race condition is eliminated.

```
    /* unlink the file relative to dirfd */
    if ( -1 == unlinkat(dirfd, argv[2], 0) ) {
        fprintf( stderr, "could not unlink %s/%s\n", argv[1], argv[2]);
        exit(1);
    }
    return 0;
}
```

A descriptor for the directory supplied as the first argument to the program is obtained and stored in `dirfd`. If this is successful then `unlinkat()` is called with this descriptor and the name of a file relative to that directory, which is supplied in the second argument to the program.

At the user level, the commands to create new links and remove old links are `link` and `unlink`. There is also `ln`:

```
/usr/sbin/link existing-file new-file
/usr/sbin/unlink file
/usr/bin/ln [ -fns ] source_file [ target ]
```

Consult their man pages for details.

3.13.4 Owner and Group of a File

The owner and group of a file are established when the file is created. Remember, you as a user do not create any files. The kernel does. Whether you use the shell to create a file, or `vi`, or any other program, the program makes a request to the kernel, sooner or later, by calling `creat()`. The kernel uses the effective-user-id and the effective group-id of the process that issued the `creat()` call as the owner and group of the file. Sometimes though, it uses the group-id of the parent directory.

The owner and group of a file are changed only by the `chown()` and `chgrp()` system calls or their command equivalents. It is very unusual for anyone other than the super-user to change ownership of a file. The `chown` command or system call changes who owns a file. You cannot change the ownership of a file that you do not already own. You should consult the man pages for the details.

3.13.5 Size of a File

A file increases in size as data is written to it using the `write()` system call. Its size is set to zero by the `creat()` call. There is no call to reduce the size of a file other than to zero it. In other words, files can grow and can be reset to zero size, but nothing else. The `lseek()` system call may be used to reposition a file pointer beyond the physical end of the file. This by itself does not change the size of the file. Only actual writes can do that. If data is written to this position, the file's stored size will be increased, with a hole in its middle. The number of disk blocks that it uses will be the number actually required to store data excluding the hole. The following listing illustrates.

```
Listing file_hole.c
#include <stdlib.h>
#include <fcntl.h>
```

```
int main(int argc, char *argv[])
{
    int     fd;

    /* create a new file named file_with_hole in the pwd */
    if ((fd = creat("file_with_hole", 0644)) < 0)
        exit(1);

    /* put a small string at the beginning */
    if (write(fd, buffer, 10) != 10)
        exit(1);

    /* seek 131072 = 2^17 bytes past the beginning of the file */
    if (lseek(fd, 131072, SEEK_SET) == -1)
        exit(1);

    /* write a small string there as well. */
    if (write(fd, enddata, 10) != 10)
        exit(1);

    /* we now have a large file with a big hole. */
    exit(0);
}
```

If you do an `ls -sl` on the file `file_with_holes` you will see that its size is 131082 bytes and uses 12 blocks. A block is usually 1KB.

```
$ ls -sl file_with_hole
12 -rw-r--r-- 1 stewart stewart 131082 Feb 21 20:09 file_with_hole
```

3.13.6 Modification and Access Time

Every i-node maintains three timestamps:

- `st_mtime`, the time the file was last modified
- `st_ctime`, the time the file attributes were last modified
- `st_atime`, the time the file was last read

These three timestamps are set by the kernel as the file is accessed and modified. You have no control over `st_ctime`, but you can change `st_atime` and `st_mtime` manually using the `utime()` system call:

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

where a `utimbuf` is defined as

```
struct utimbuf {
     time_t actime;          /* access time */
     time_t modtime;         /* modification time */
};
```

It is possible to change the modification time of a file to a future time! The system does not check that any time values make sense. The listing below can be used to change the timestamps on a file to any time at all.

There are many system calls that affect the time stamps on a file. For example, when you remove an entry from a directory, the directory itself is modified and its `st_mtime` value changes. If you list the contents of a directory, its `st_atime` value changes.

```
Listing changefiletimes.c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <utime.h>

#define CH_ACCESS 1
#define CH_MOD    2

// setclockback
// Given a time_t now, returns a new time_t by subtracting the days
// hours, minutes, and seconds specified
time_t setclockback( time_t now, int days, int hours,
                     int mins, int secs)
{
    struct tm time_tm;
    time_t    newtime;

    localtime_r(&now, &time_tm);
    time_tm.tm_sec -= secs;
    time_tm.tm_min -= mins;
    time_tm.tm_hour -= hours;
    time_tm.tm_mday -= days;
    newtime = mktime(&time_tm);

    return newtime;
}

void backdate(const char* fn, int mode, int times[], int size)
```

```
{
    struct stat buf;
    time_t temp_t;
    time_t access_t, mod_t;

    struct utimbuf utbuf;

    if ( 4 != size ) {
        fprintf(stderr, "wrong number of parameters to backdate.\n");
        exit(1);
    }

    if ( -1 == stat(fn, &buf ) ){
        perror("Could not stat file.\n");
        exit(1);
    }

    time(&access_t);
    if ( mode & CH_ACCESS ) {
        temp_t = setclockback( access_t, times[0], times[1],
                               times[2], times[3]);
        access_t = temp_t;
    }
    time(&mod_t);
    if ( mode & CH_MOD ) {
        temp_t = setclockback( mod_t, times[0], times[1],
                               times[2], times[3]);
        mod_t = temp_t;
    }
    utbuf.actime = access_t;
    utbuf.modtime = mod_t;

    if ( -1 == utime(fn, &utbuf) ) {
       perror("Error changing times");
       exit(1);
    }

}
int main(int argc, char* argv[])
{
    int i = 0;
    time_t now;
    char*  timestr;
    char   filename[255];
    int    times[4] = {0,0,0,0};

    if ( argc < 2) {
        printf("usage: %s filename days hours minutes seconds\n",
```

```
                argv[0]);
        exit(1);
    }

    strcpy(filename, argv[1]);
    while ( (--argc >= 2) && (i < 4) ) {
        times[i] = atoi(argv[i+2]);
        i++;
    }

    time(&now);
    timestr = ctime(&now);
    backdate(filename, CH_ACCESS|CH_MOD, times, 4);
    return 0;
}
```

3.13.7 Name of a File

The system call to rename a file is `rename()`:

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

It returns –1 on failure, 0 on success. `oldname` is the pathname of the file or directory to be renamed and `newname` is the new pathname. The behavior is complex, depending on whether `oldname` refers to a file, a directory, or a symbolic link, and whether `newname` already exists and whether it is on the same file system. To illustrate just some of the complexity:

- If `oldname` specifies a file that is not a directory, then if `newname` exists, it must not refer to a directory. If `newname` exists and is not a directory, it is removed, and `oldname` is renamed to `newname`. The process must have write permission for the directory containing `oldname` and for the directory containing `newname`, since both are being changed.
- If `oldname` specifies a directory, and if `newname` exists, it must refer to an empty directory. If so, it is removed, and `oldname` is renamed to `newname`. Additionally, `newname` cannot contain a path prefix that includes `oldname`. For example, you cannot rename `/class_stuff/notes` to `/class_stuff/notes/systemcalls`, because the old name `/class_stuff/notes` is a path prefix of the new name and cannot be removed.
- As a special case, if `oldname` and `newname` refer to the same file, the function returns successfully without changing anything.

The process calling `rename()` must have appropriate permissions in all cases. A simplistic description is that `rename()` renames a file, and if this involves deleting a name from one directory and creating one in another, that is what it does. For complete details see the man page.

3.14 Traversing the Tree, Up and Down

Two common things that we do with trees are ascending them and descending them. By "ascending" we mean traveling from a given node to the parent and to the grandparent and so on until we reach the root. By "descending" we mean something more extensive, as there are many nodes in the subtree rooted at a given node – we mean visiting all nodes in that subtree in some specified order. This is a descent into the tree.

The first type of traversal is what the `pwd` command does; it travels from the current working directory up the tree so that it can display the path from the root to the current working directory. With a little research you will discover that there is a system call[15], `getcwd()`, that does exactly this. The second type of traversal is what recursive versions of commands such as `ls`, `grep`, `chown`, `chmod`, and many more, do, and what the `find` command must do as well. They start in the given directory and visit all files in the subtree rooted at that directory.

Both of these problems present interesting programming challenges and require us to learn a bit more about the file system and the file hierarchy. We will first solve the problem of finding the path to the current working directory. After that we will look at two different functions provided in the API for "walking" the file tree.

3.15 The pwd Command

Recall that the `pwd` command prints the absolute pathname to the current working directory. As a warm-up, we will exercise our knowledge of directories and i-nodes to make sure we understand the task that lies ahead. Specifically we will try to reconstruct a portion of a file hierarchy from limited information about i-numbers in a set of directories.

Suppose that we are given a directory named `scratch` that contains subdirectories that also have subdirectories, and so on. Each of these subdirectories may have regular files as well. The command

```
$ ls -iaR scratch
```

will recursively display the i-numbers and filenames of all files within these directories, including entries for "`.`" and "`..`". The listing below shows the output of this command on a hypothetical directory named `scratch`, except that directory names that would ordinarily appear in the output and just displayed the files contained in them. From this listing, it is possible to reconstruct the file hierarchy rooted at `scratch`.

```
725 .                                   732 stuff
449 ..
753 temp                                727 .
727 test1                               725 ..
728 test2                               729 file1
                                        730 file2
731 .                                   733 garbage
728 ..
733 junk                                728 .
```

[15] In Linux it is a system call. On other systems it is a library function.

```
725 ..                                  729 temp
731 data
748 srcs
```

Before reconstructing the hierarchy, you should be able to answer the following questions:

- What is the i-number of `scratch`?
- Which filenames are links to the same file?

The key to reconstruction is to use the i-numbers of the parent entries to obtain their names, and use process of elimination for the rest. See Figure 3.6.

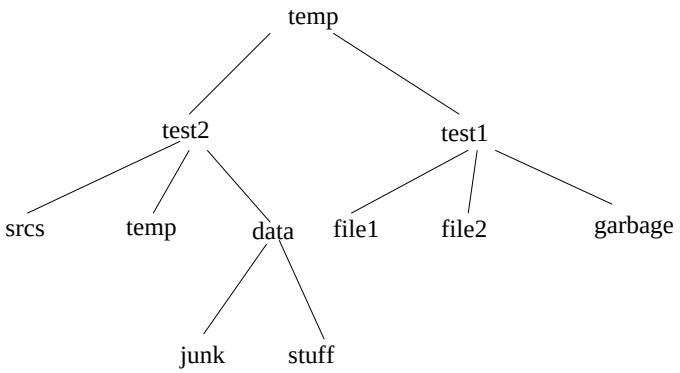


Figure 3.6: Hierarchy below `scratch` directory

This exercise shows that the parent directory entries in a directory play a vital role in the hierarchy, because they are, in essence, back links. They are a way to ascend the tree.

3.15.1 Implementing the pwd Command

Suppose my working directory is `chapter03`, which is located in the directory `unix_demos`, which is in `class_stuff`, which is in `cs82010`, which is in `home`, which is in the root directory. Then typing `pwd` prints the path

```
/home/cs82010/class_stuff/unix_demos/chapter03
```

Of course when it starts, `pwd` does not "know" where it is. It is somewhere in a node of a very large tree, but it does not know the path to it. It could use an exhaustive search to find the path, starting at the root and recursively searching through every directory until it finds one whose i-number matches the i-number of the current working directory, but that would not be a very useful command. From the preceding exercise it should be clear that `pwd` has work backwards to construct the path, using the parent entries in the directories as it goes along.

3.15.2 About pwd

Before proceeding, let us note that the actual `pwd` command is very often a shell built-in, not a separate program. The easiest way to determine if it is a built-in or a program, at least on a Linux host, is to enter

```
$ type pwd
```

If it is a shell built-in, the reply will be

```
pwd is a shell builtin
```

If not, it will be the pathname to the executable, such as `/bin/pwd`. You can then use the `file` command to identify the type of file:

```
$ file /bin/pwd
/bin/pwd: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.9, stripped
```

which shows the type of executable it is. The shell built-in most likely just retrieves the value of the environment variable `PWD`, which is updated as the working directory is changed. The shell will use the kernel's `getcwd()` system call to accomplish this. We exclude the possibility of solving this problem by calling `getcwd()`; that defeats the objective, which is in effect to write our own version of `getcwd()`.

3.15.3 How pwd Works

How does `pwd` construct the path? It is not stored anywhere. In fact, no directory has information about where it is located except for that one little item – the dot-dot entry. The `".."` in a directory always contains the i-number of the parent directory, with one exception: *the root of the file system has no parent.* If you list the i-numbers in the root directory you will see that . and .. have the same i-number:

```
$ cd /
$ ls -iaF | grep '\./'
2 ./
2 ../
```

This provides a stopping criterion for an iterative solution to printing the pathname. The idea is to do the following:

1. Record the i-number, n, of the current directory, i.e., the i-number for ".".
2. Change directory (`chdir()`) to the parent directory.

3. Compare the i-number of the parent directory, which is now the current directory, to n. If the i-number of the parent directory is n, stop. Otherwise, find the name of the link with i-number n and append that name to the left of the current pathname, and append a "/" to the left of that, and go back to step 1.

4. Print the current pathname.

This algorithm seems like it should work. However, as we have to construct the string from the right end to the left end, it is easier to use a recursive solution. Recursion is never a fast strategy and you should not think that the real `pwd` is recursive, however, it is an easy way to create a working solution.

3.15.4 A First Version of `pwd`

The first version of `pwd.c` is below, named `pwd1.c`. As we will see shortly, this program does not work correctly. The bug is only exposed in limited conditions. Some comments are omitted from the listing to save space.

Listing 1. pwd1.c

```
#include    <stdio.h>
#include    <string.h>
#include    <stdlib.h>
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>

// BUFSIZ is defined in stdio.h: it is the maximum string size
#define     MAXPATH      BUFSIZ

// print_pwd prints the pwd to the string abs_pathname and
// NULL-terminates it
void  print_pwd    (  char * abs_pathname );

// print_path prints to str the path from / to the file
// specified by  cur_inum.
void  print_path ( char* str, ino_t cur_inum );

// inum_to_name puts the filename of the file with i-node inum
// into string buf, at most len chars long and 0 terminates it.
void  inum_to_name(ino_t inum, char * buf, int len );

//  get_ino gets the i-node number of file fname, if that fname
//  is the name of a file in the current working directory
//  Returns 0 if successful, -1 if not.
int   get_ino(char * fname, ino_t * inum);

/*********************************************************/
int main(nt argc, char* argv[])
{
    char  path[MAXPATH] = "\0";  //  string to store pwd
```

```
    print_pwd( path );              //  print pwd to string path
    printf("%s\n", path);           //  print path to stdout
    return 0;
}

/*********************************************************/
void print_pwd    (  char * pathname )
{
    ino_t inum;

    get_ino(".", &inum );
    print_path(pathname, inum );
}

/*********************************************************/
void print_path( char * abs_pathname, ino_t this_inode )

{
    ino_t       parent_inode ;
    char        its_name[BUFSIZ];

    // get inumber of parent
    get_ino("..", &parent_inode );

    // At root iff parent inum == cur inum
    if ( parent_inode != this_inode ) {
        chdir( ".." );    // cd up to parent
        //  get filename of current file
        inum_to_name(this_inode, its_name, BUFSIZ);
        // recursively get path to parent directory
        print_path(abs_pathname,  parent_inode );
        strcat(abs_pathname, "/");
        strcat(abs_pathname,  its_name );
    }
    else
        strcat( abs_pathname, "/");
}

/*********************************************************/
void inum_to_name(ino_t inode_to_find , char *namebuf,
                  int buflen)
{
    DIR            *dir_ptr;
    struct dirent  *direntp;

    dir_ptr = opendir( "." );
    if ( dir_ptr == NULL ) {
        perror( "." );
        exit(1);
    }

    // search directory for a file with specified inum
    while ( ( direntp = readdir( dir_ptr ) ) != NULL )
        if ( direntp->d_ino == inode_to_find ) {
```

```
            strncpy( namebuf, direntp->d_name, buflen);
            namebuf[buflen-1] = '\0';
            closedir( dir_ptr );
            return;
        }
    fprintf(stderr, "\nError looking for i-node number %d\n",
            inode_to_find);
    exit(1);
}

/*********************************************************/
int  get_ino( char *fname, ino_t *inum )
{
    struct stat info;
    if ( stat( fname , &info ) == -1 ){
        fprintf(stderr, "Cannot stat ");
        perror(fname);
        return -1;
    }
    *inum   = info.st_ino;
    return 0;
}
```

The recursive function in the above listing is the `print_path()` function. Given an i-number, `this_inode`, of the current working directory file, it checks whether the i-number of the parent directory is equal to `this_inode`. If they are equal, it stops the recursion, because this means it is at the root. In this case, it appends a slash, "/", to the pathname under construction. If it is not, it has to get the filename of the current working directory. Following the logic we described above, it does this by going up one level to the parent directory and looking for the i-number in that directory. When it finds it, it retrieves the name matching the i-number.

Therefore, it begins by getting the i-number of the parent directory into `parent_inode`. It then compares `parent_inode` to `this_inode`, and if unequal it calls `chdir("..")` to step into the parent directory. Once in the parent directory, it calls `inum_to_name()` with `this_inode`. The `inum_to_name()` call will store into `its_name` the actual name of the directory with i-number `this_inode`. `print_path()` then makes a recursive call to itself, passing the i-number of the parent, `parent_inode`, and the string in which it is constructing the pathname. In the recursive call, it is one step closer to the root of the file system, ensuring that the problem size is diminished. After the recursive call, we assume that the absolute pathname from the root down to the parent directory is in the argument `abs_pathname`, so we append a slash followed by the file name we found for the current directory, `its_name`, to `abs_pathname.` and we return. If `parent_inode == this_inode`, then `print_path()` appends a slash to `abs_pathname`. When it does this, `abs_pathname` is the null string, so when this program is called from within the root directory, it will display "/".

The function that obtains the i-number of a filename is simple; it calls `stat()` with the given filename and returns the `st_ino` member of the `stat` structure filled by the `stat()` call. The `inum_to_name()` function is a slightly more complex. It opens the current working directory and repeatedly reads the entries in the directory until it finds an entry with the given i-number. This is a linear search of the directory. It uses the `readdir()` call to retrieve a pointer to a `dirent` structure and pulls the name member out of the `dirent` structure. If it does not find an entry with the given i-number it prints an error message on the standard error stream.

Running the program from various directories, you should discover that

- most pathnames start with a leading double-slash instead of a single slash, and
- the root directory is correctly displayed as `/`.

Now try running the program from within a directory in a file system that has been mounted on the root file system. You will discover that either the program does not print the full absolute path, that it prints an incorrect path, or that it generates the error

```
Error looking for i-node number n
```

for some n. This is a result of the program's not taking into account the way in which file systems are mounted. In order to understand this, we will revisit how mounting works.

3.15.5 Multiple File Systems: Mounting

Section 3.3 introduced the concept of file system mounting. In UNIX, unlike DOS or Windows, all files on all volumes are part of a single directory hierarchy; this is achieved by mounting one file system onto another. What exactly do we mean by mounting one file system onto another? To make it clear, suppose that we have a root file system that looks in part like the one in Figure 3.7.

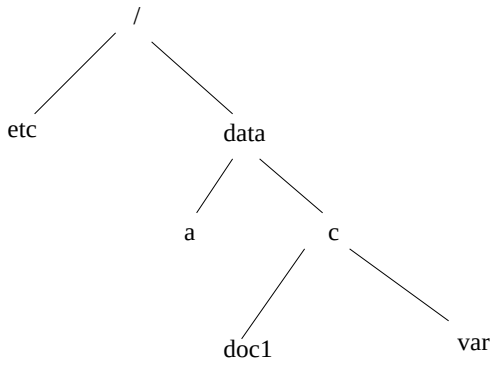


Figure 3.7: Root file system before mount

It has a subdirectory named `data` with two subdirectories named `a` and `c`. Notice that `c` is not empty. Suppose this file system is located on one internal disk of the computer, and suppose that there is a second internal disk that contains its own file system, as shown in Figure 3.8. The second disk is represented by a device special file named `/dev/hdb`. There is a file system on this second disk, whose root has two subdirectories named `staff` and `students`, and `students` has subdirectories `grad` and `undergrad`. To make the files in this second file system available to the users of the system, it has to be mounted on a *mount point*. A *mount point* is a directory in the root file system that will be replaced by the root of the mounted file system.

If the directory `/data/c` is a mount point for the `/dev/hdb` file system, then we say that `/dev/hdb` is mounted on `/data/c`. The following `mount` command will mount the `/dev/hdb` file system on `/data/c`. It does not matter that `c` contains file links already; the `mount` merely hides them while it is there. They would disappear from view until the file system was unmounted, when they would reappear.

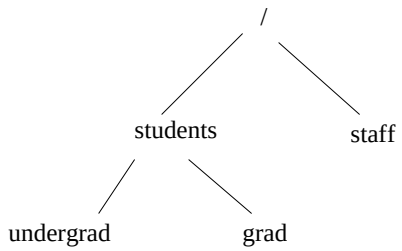


Figure 3.8: File system `/dev/hdb`

```
$ mount /dev/hdb /data/c
```

After this command, the root file system will be as in Figure 3.9.

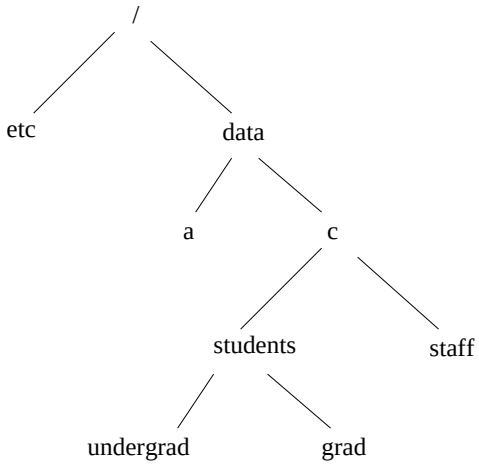


Figure 3.9: Root system after mount of `/dev/hdb`

The absolute pathname of the `grad` directory would then be `/data/c/students/grad`.

In reality, we would have to specify the type of file system written on `/dev/hdb`, unless it is the default file system, and only the superuser is allowed to run the `mount` command, with a few exceptions, such as mounting removable storage devices such as CD-ROMs, DVDs, and USB devices.

When a directory becomes a mount point, the kernel restructures the directory hierarchy. The directory contents are not lost; they are masked by the root directory of the mounted file system. The kernel records in a list of mount points that this file system is mounted on this directory. Different versions of UNIX implement mounting in different ways. This relevant part of this discussion is that operations that traverse the tree can identify mount points because they are directories that are the roots of different file systems than their parents.

3.15.6 Duplicate I-node Numbers and Cross-Device Links

The advantage of mounting is that it simplifies the user's conceptualization and navigation of the file hierarchy. It is a single hierarchy. One problem that the kernel must deal with is that there may be many files with the same i-number, since i-numbers are unique only within a single file system. In fact, the root of every file system has i-number 2, and its `".."` entry is also 2, so there will be

many i-nodes with i-number 2. The kernel is able to distinguish i-nodes because i-nodes are also known by the name of the device on which they are located. On all modern UNIX systems, the i-node contains a member that stores the name of the device on which it is located. It is like having a dog tag on a dog; if the dog gets lost, someone can look at it and know where it came from. (I'll admit i-nodes can't wander around, but a process looking at a memory copy of an i-node may need to know which device it got it from.) So i-nodes are more accurately represented by (i-number, device number) pairs.

To pursue this line of thought, suppose there is a file in the root file system with i-number 52. Suppose it has a hard link named `/data/a/doc1`. Suppose also that, referring to the figure above, `/dev/hdb` has a file `/students/undergrad/hwk1` with i-number 52. If UNIX allowed the creation of a hard link across the two file systems with the call

```
link( "/data/a/doc1", "/data/c/students/grad/doc1")
```

then, after the call, there would be two links in the `/dev/hdb` file system, each having the same i-number, but these i-numbers would refer to different i-nodes. This would break the file system, unless directories were able to store device numbers as well as i-numbers with file names, which would require rewriting almost the entire kernel. Therefore, UNIX systems generally do not allow anyone to create hard links that span across file systems. All hard links to a file must be in a single file system. For the same reason, you cannot use the `rename()` call to move a file across file systems. The `rename()` call makes a new name in the specified directory pointing to the original file. The call,

```
rename( "/data/a/doc1", "/data/c/students/grad/doc1")
```

would cause the file system to have two links to different files but with the same i-number.

3.15.7 A Second Version of `pwd`

It is now possible a correct implementation of the `pwd` command, which we call `pwd2.c`. The problem with the original program is what it does when it reaches a mount point as it works its way towards the root. Suppose that the current working directory is

```
/data/mnt/backups/backup1/current_backup/home/class_stuff
```

and that `/data/mnt/backups` is actually a mount point on which the file system `/dev/sdc1` has been mounted. The device `/dev/sdc1` is an external drive that is mounted to backup and restore files as needed, and it has a top-level directory named `/backup1`, under which one can find the directory `current_backup/home/class_stuff`.

As our program ascends the tree, it eventually reaches the directory `backup1`. Suppose this has i-number 6643. When the program has made `backup1` its current working directory, it obtains the i-number of its parent, which is the root of the mounted file system `/dev/sdc1` and therefore has i-number 2.

It then changes directory to this parent directory, the `/data/mnt/backups` directory, and calls `inum_to_name()` with i-number 6643. `inum_to_name()` searches through the directory entries in the directory `/data/mnt/backups`, finds 6643, and sets its_name to `"backup1"`.

Then the program makes the next recursive call, in which `this_inode` = 2. When it changes directory to its parent though, it has just crossed the mount point. There will not be a directory entry in this directory with the i-number 2. Although `backups` has i-number 2 as the root of the `/dev/sdc1` file system, the directory for `/data/mnt` does not have a `dirent` entry for it with the i-number 2. As a result `inum_to_name()` will fail (unless by coincidence, it has an entry with i-number 2 for some other name); it will search through the entire directory and then report that it could not find i-number 2.

Although the dirent entry for backups in /data/mnt does not have i-number 2, its i-node does have that i-number. This is why, if we were to issue the command

```
ls -i /data/mnt
```

we would see the i-number 2 displayed for `backups`, because `ls` calls `stat()` to retrieve the i-node contents. We must do the same thing. We make several changes to make the program correct.

1. We replace `get_ino()` by a function, `get_deviceid_inode()` that gets the i-number and device-id for the given file name. We call this function when the program starts so that we have the device-id of the current working directory, and pass it to the recursive function, `print_path()`.
2. We change the test within `print_path()` for when we have reached the root: we have reached the root if and only if the parent i-number matches the child i-number and they have the same device ids, otherwise we are not there yet.
3. We change the way we find the name of the current directory in `inum_to_name()`. We pass `inum_to_name()` the device-id and the i-number of the current working directory. `inum_to_name()` calls `stat()` for every entry in the current working directory, trying to match the i-number and the device-id stored in that entry's i-node against the i-number and device-d it was passed. If the i-number and device match the pair passed to `inum_to_name()`, then the `d_name` member of the `dirent` structure is the name of the current working directory, and this is what is returned to `print_path()`.
4. To fix the problem with the terminating slash, which should not be at the end of the path, we can keep a static variable in the recursive function that keeps track of the level of recursion. The trailing slash will only be printed if we are in a recursive call, not at the top-level call.

All of these changes are incorporated into `pwd2.c`, whose code is below in Listing 2.

```
Listing 2. pwd2.c
// same includes as before

#define     MAXPATH      BUFSIZ

/*********************************************************/
void  print_pwd    (  char * abs_pathname );

// print_path prints to str the path from / to the file
// specified by  cur_inum on the device with device id
// dev_num.
```

```
void  print_path( char* str, ino_t cur_inum, dev_t  dev_num);

// inum_to_name puts the filename of the file with inode inum
// on device with device number dev_num into string buf, at
// most len chars long and 0 terminates it.
void  inum_to_name ( ino_t inum, dev_t dev_num,
                     char * buf, int len );

// get_dev_ino gets the device id and the inode number of
// file fname, if that fname is the name of a file in the
// current working directory. Returns 0 if success, -1 if not.
int   get_deviceid_inode  ( const char *fname, dev_t *dev_id,
                      ino_t *inum );

/*********************************************************/

int main(int argc, char* argv[])
{
    char  path[MAXPATH] = "\0";

    print_pwd( path );
    printf("%s\n", path);
    return 0;
}

/*********************************************************/
void print_pwd    (  char * abs_pathname )
{
    ino_t inum;
    dev_t devnum;

    get_deviceid_inode(".", &devnum, &inum );
    print_path(abs_pathname, inum, devnum);
}

/*********************************************************/
void print_path  (  char *  abs_pathname,
                    ino_t   this_inode,
                    dev_t   this_dev )
// Recursively prints path leading down to file with this
// inode on this_dev  Uses static int height to determine
// which recursive level it is in
{
    ino_t       parent_inode ;
    char        its_name[BUFSIZ];
    dev_t       dev_of_node,
                dev_of_parent;
    static int  height = 0;

    // get device id and inumber of parent
    get_deviceid_inode("..", &dev_of_parent, &parent_inode );

    // At root iff parent inum == cur inum & device ids
    // are same
```

```
    if ( ( parent_inode != this_inode )
         || ( dev_of_parent != this_dev) ) {
        chdir( ".." );

        inum_to_name(this_inode, this_dev, its_name, BUFSIZ);
        height++;  // about to make recursive call
        print_path(abs_pathname, parent_inode, dev_of_parent);
        strcat(abs_pathname,  its_name );

        if ( 1 < height )
            /*  Since height is decremented whenever we
             *  leave call it can only be > 1 if we have not
             *  yet popped all calls  from the stack
             */
            strcat(abs_pathname,  "/" );
        height--;
    }
    else  // must be at root
         strcat(abs_pathname, "/");
}
/*********************************************************/
void inum_to_name(ino_t inode_to_find, dev_t devnum,
                  char *name, int buflen)
{
    DIR             *dir_ptr;
    struct dirent    *direntp;
    struct stat      statbuf;

    if ( NULL == (dir_ptr = opendir( "." ) ) ) {
        perror( "." );
        exit(1);
    }

    while ( ( direntp = readdir( dir_ptr ) ) != NULL ) {
        if ( -1 == (stat(direntp->d_name, &statbuf)) ) {
              fprintf(stderr, "could not stat");
              perror(direntp->d_name);
              exit(1);
        }
        if ( ( statbuf.st_ino == inode_to_find ) &&
             ( statbuf.st_dev == devnum) ) {
            strncpy( name, direntp->d_name, buflen);
            name[buflen-1] = '\0';   //  just in case
            closedir( dir_ptr );
            return;
        }
    }
    fprintf(stderr, "Error looking for i-node %d\n",
            inode_to_find);
    exit(1);
}

/*********************************************************/
int  get_deviceid_inode( const char *fname, dev_t *dev_id,
```

```
                    ino_t *inum )
{
    struct stat info;
    if ( stat( fname , &info ) == -1 ){
        fprintf(stderr , "Cannot stat ");
        perror(fname);
        return -1;
    }
    *inum   = info.st_ino;
    *dev_id = info.st_dev;
    return 0;
}
```

This concludes the implementation of the `pwd` command.

3.15.8 Symbolic Links

Users should be able to create links across file systems, and they should also be able to make links to directories. For example, I might have a directory,

```
/data/c/sweiss/development/sourcecode/c-sources
```

that I access frequently while working on lecture notes. If my lecture notes are in `/data/c/sweiss/classes/notes`, then it would be convenient to have a directory

```
/data/c/sweiss/classes/notes/c-sources
```

that is just a link to the `c-sources` directory under `sourcecode`. Then if my working directory were `/data/c/sweiss/classes/notes`, I could type

```
$ ls c-sources
```

and it would display the contents of `/data/c/sweiss/development/sourcecode/c-sources`.

Similarly, users should not be limited in their ability to make links to files. The fact that the file system depends on the uniqueness of i-numbers within a file system should not limit a user's ability to make links that are convenient, even if they span file systems.

For these reasons, most UNIX systems provide an alternative kind of link called a *symbolic link* (or *soft link*). A symbolic link is a file that contains a reference to the name of the file to which it links. The command `ln -s` creates a symbolic link instead of a hard link. To illustrate, suppose we have a directory named `temp`, and inside it we run the commands

```
$ who > users
$ ln users whoson          # hard link
$ ln -s users ulist        # soft link
$ ls -1i
628 lrwxrwxrwx   1 sweiss  staff    5 Aug 30 00:22 ulist -> users
614 -rw-------   2 sweiss  staff  676 Aug 30 00:22 users
614 -rw-------   2 sweiss  staff  676 Aug 30 00:22 whoson
```

You can see a new notation. First, the "`l`" in the file type specifies that the file is a symbolic link. Second, the notation "`ulist -> users`" indicates that `ulist` is a symbolic link to the file `users`. The `symlink()` system call does the same thing.

Symbolic links are actual files. They have i-nodes, as demonstrated above. But the i-node contains different information than the i-node of a hard link, and it refers to the name of the file to which it is linked. In other words, the actual filename is stored in the link.

Symbolic links can be broken easily. If I now type

```
$ rm users
rm: remove users (yes/no)? y
$ ls -li
total 4
628 lrwxrwxrwx 1 sweiss student 5 Aug 30 00:22 ulist -> users
614 -rw------- 1 sweiss student 676 Aug 30 00:22 whoson
$ more ulist
ulist: No such file or directory
```

you see that `ulist` still points to `users`, even though it does not exist. This is not an error. The error only occurs when the process tries to access the file.

3.15.9 System Calls Related to Symbolic Links

The following system calls are related to the use of symbolic links.

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
int lstat(const char *file_name, struct stat *buf);
```

The system call that creates symbolic links is `symlink()`. The `readlink()` call obtains the name of the file to which a symbolic link is pointing. The `lstat()` call obtains a `stat` structure for a file that is a link, not the file to which it is linked.

3.16 Tree Walks

There are four different ways that we can visit all of the nodes in a given subtree of the hierarchy:

- Write our own recursive function to traverse the tree;
- Write a non-recursive function to traverse the tree;
- Use the `nftw()` POSIX library function;
- Use the `fts()` function available in systems including the 4.4BSD API.

Commands such as `grep`, `chmod`, `chown`, `rm`, `cp`, and `chgrp` use the `fts()` functions to perform their recursive tree traversals. The `GNU` version of the `ls` command, written by Richard Stallman and David MacKenzie, uses internal stacks and queues to recurse the tree. The `GNU find` command, originally written by Eric Decker, uses mutually recursive functions whereas the versions on BSD systems use the `fts()` functions. In this section we describe the functions provided by the libraries, beginning with the POSIX `nftw()` function and following with `fts()`.

3.16.1 The nftw() Tree Walk Function

`nftw()` replaces the older `ftw()` function, which still exists, but is deprecated. Its prototype is

```
#include <ftw.h>
int nftw(const char *path,
         int (*fn)(const char *, const struct stat *, int, struct FTW *),
         int fd_limit, int flags);
```

The `nftw()` function recursively descends the directory hierarchy rooted in `path`. For each object in the hierarchy, it calls the function pointed to by `fn`, passing it

- a pointer to a null-terminated character string containing the pathname of the object (e.g. a file's path from the root of the walk),
- a pointer to a `stat` structure containing information about the object, filled in as if `stat()` or `lstat()` had been called to retrieve the information,
- an integer that gives more information about the object, whose value is one of the following predefined constants:

`FTW_D`	The object is a directory.
`FTW_DNR`	The object is a directory that cannot be read. The `fn` function shall not be called for any of its descendants.
`FTW_DP`	The object is a directory and subdirectories have been visited. (This condition occurs if the `FTW_DEPTH` flag is included in `flags`.)
`FTW_F`	The object is a file.
`FTW_NS`	The `stat()` function failed on the object because of lack of appropriate permission. The `stat` buffer passed to `(*fn)` is undefined. Failure of `stat()` for any other reason is considered an error and `nftw()` returns -1.
`FTW_SL`	The object is a symbolic link. (This condition occurs if the `FTW_PHYS` flag is included in `flags`.)
`FTW_SLN`	The object is a symbolic link that does not name an existing file. (This condition shall only occur if the `FTW_PHYS` flag is not included in `flags`.)

- a pointer to an `FTW` structure, which is defined as

```
struct FTW {
     int base;
     int level;
};
```

The `FTW` structure gives syntactic information about the filename and depth information about its place in the search. Specifically, the value of `base` is the offset of the object's filename in the pathname passed as the first argument to `(*fn)`. This makes it possible to extract the filename from the pathname easily. The value of `level` indicates depth relative to the root of the walk, where the root level is 0. For example, if the pathname of the file is `documents/pictures/2012/january/nyc`, and `nyc` is the file object being processed by the call to `(*fn)`, then `base` would be the length of the string "`documents/pictures/2012/january/`" and `level` would be 4, since `nyc` is at level 4 in the tree rooted at `documents`.

The `(*fn)` function is a programmer-defined function that is called for every object that `ntfw()` visits in the tree. It can have any semantics provided that has the prototype described above. The function has access to the information returned by a call to `stat()` as well as information contained in the integer flag. It can be designed to extract and utilize this information, but the problem, which is a big one, is that the function has no hooks to pass in user-supplied data or pointers, so that it is not possible to use this function to modify any variables unless they are globally scoped. For example, to do something relatively simple such as computing the total number of bytes used by all objects in the subtree rooted at a given directory, and printing that value when it is totaled, we would either have to make it a static variable within `(*fn)` and do a post-order traversal, printing only when we return to the root directory, or declare the variable outside of the function with file scope and do any printing in the function that calls `nftw()` when the call terminates. It is expected that the programmer will use global variables when using this function. This will be clear when we look at the example code below. We will have more to say about this below.

The third parameter to the `nftw()` function itself is an integer `fd_limit` that sets the maximum number of file descriptors that should be used by `nftw()` while traversing the file tree. Only one descriptor is needed for each level of the tree. If `fd_limit` is smaller than the depth of the tree, then performance will be degraded because the function will have to keep opening and closing directories.

The last parameter is a flag consisting of a bitwise-OR of zero or more of the following constants, which control how `ntfw()` handles mount points and soft links and what it uses as its current working directory, and whether it follows a pre-order or post-order traversal of the tree. Specifically,

`FTW_CHDIR` If set, `nftw()` changes the current working directory to each directory as it reports files in that directory. If clear, `nftw()` does not change the current working directory.

`FTW_DEPTH` If set, `nftw()` reports all files in a directory before reporting the directory itself. If clear, `nftw()` reports any directory before reporting the files in that directory.

`FTW_MOUNT` If set, `nftw()` reports files only in the same file system as `path`.

`FTW_PHYS` If set, `nftw()` performs a physical walk and does not follow symbolic links. If it is clear, it follows symbolic links but does not visit any file twice. If `FTW_PHYS` is clear and `FTW_DEPTH` is set, `nftw()` follows soft links but does not report on any directory that would be a descendant of itself. If both `FTW_PHYS` and `FTW_DEPTH` are clear, `nftw()` follows soft links but does not report on the contents of any directory that would be a descendant of itself.

There is one other flag that not in the above list, `FTW_ACTIONRETVAL`, only available with `_GNU_SOURCE` set, which we ignore for now.

The `nftw()` function runs until the first of the following conditions occurs:

- an invocation of `(*fn)` returns a non-zero value, in which case `nftw()` returns that value;
- it detects an error other than `EACCES`, in which case it returns -1 and sets `errno` to indicate the error; or
- the tree is exhausted, in which case it returns 0.

As usual, we will look at an example to see how it is used. The following program displays the size and name of every file in the tree rooted at its argument. It indents the name in proportion to its depth in the tree, and prints the total size in bytes of all files visited. It lets the user control whether to cross mount points with the `-m` option, to do a post-order in stead of a pre-order with the `-d` option, and does not follow symbolic links unless the `-p` option is provided.

Listing nftwdemo.c

```
#define _XOPEN_SOURCE 500
#include <ftw.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <limits.h>

#define  TRUE  1
#define  FALSE 0
#define  MAXDEPTH    20

/* For getopt() we need to use this feature test macro */
#if  ( _POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE )

static int display_info(const char *fpath, const struct stat *sb,
             int tflag, struct FTW *ftwbuf)
{
    char  blanks[PATH_MAX];
    const char *filename = fpath + ftwbuf->base;
    int    width          = 4*ftwbuf->level;

    /* fill blanks with a string of 4*level blanks */
    memset(blanks, ' ', width);
    blanks[width] = '\0';

    /* print out blanks then filename (not full path) */
    printf("%s%s", blanks, filename );

    /* Check flags and print a message if need be */
    if ( tflag == FTW_DNR )
        printf(" (unreadable directory)");
    else if ( tflag == FTW_SL )
        printf(" (symbolic link)" );
    else if ( tflag == FTW_SLN )
        printf(" (broken symbolic link)" );
    else if ( tflag == FTW_NS )
        printf("stat failed " );

    printf("\n");
```

```
    return 0;           /* To tell nftw() to continue */
}

int main(int argc, char *argv[])
{
    int flags = 0;
    int status;
    int ch;
    char options[] = ":cdpm";
    opterr = 0;  /* turn off error messages by getopt() */

    while  (TRUE) {
        /* call getopt, passing argc and argv and the options string */
        ch = getopt(argc, argv, options);

        /* it returns -1 when it finds no more options  */
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'c':
            flags |= FTW_CHDIR;
            break;
        case 'd':
            flags |= FTW_DEPTH;
            break;
        case 'p':
            flags |= FTW_PHYS;
            break;
        case 'm':
            flags |= FTW_MOUNT;
            break;
        default:
           fprintf (stderr, "Bad option found.\n");
           return 1;
        }
    }

    if (optind < argc) {
        while (optind < argc) {
            status = nftw(argv[optind], display_info,
                          MAXDEPTH, flags);
            if ( -1 == status ) {
               perror("nftw");
               exit(EXIT_FAILURE);
            }
            else
                optind++;
        }
    }
    else {
        status = nftw(".", display_info, MAXDEPTH, flags);
        if ( -1 == status ) {
           perror("nftw");
           exit(EXIT_FAILURE);
```

```
            }
        }
        exit(EXIT_SUCCESS);
}
#endif
```

Notes.

- The main program allows multiple filename arguments .
- The `display()` function uses `memset()` to set the string blanks to the correct number of blanks.
- To print the file name without the leading path, it prints the string starting at `fpath+ftwbuf->base`.
- It also displays information about whether the object is a symbolic link, broken or otherwise, or an unreadable directory, or if the `stat()` call failed on the object.

This was a warm-up exercise. In general, the `nftw()` is challenging to use if the task requires changing state information that must be preserved across calls to the `(*fn)` function. We must use variables that are either static and locally scoped or global. To see what the problem is, we will try to implement a simple version of the `du` command.

3.16.2 The `du` Command

The `du` command in its simplest form is invoked with

```
$ du file file ...
```

The `du` command estimates disk usage for each file it is given, and if any are directories, it does this recursively on each. In other words, when it is given a directory as an argument, it traverses the tree rooted at that directory, and for each directory that it visits, it prints its disk usage. The disk usage of a directory is the sum of the usage required for the directory file itself together with all files in its tree. It seems like an ideal candidate for a command to be implemented using `nftw()`, and a good exercise in using `nftw()` for a realistic application.

The `du` command has several options, but we will write a simple version of it that accepts no options. The actual `du` command by default displays the total number of blocks that each file uses. Rather than displaying block usage, ours will display the actual number of bytes. This is equivalent to the command `du -b`. Also, `du` by default does not display the usage of all files; to do that it needs the `-a` option. Therefore, we will write the equivalent of the command

```
$ du -ab file file ...
```

The du command should not follow symbolic links, otherwise it may double count files or count files that are not within the directory argument. For this first version, we will also disable crossing mount points, so that it measures usage only within a single file system. It is easy enough to provide an option to let it cross mount points.

Obviously it has to do a post-order traversal of the tree, because otherwise when it visits a directory it will not have the total usage of that directory's children. Thus, in Figure 3.10, assuming a left-to-right traversal of the children of a single node (since we can draw them in any order we wish, we can assume it is always left-to-right), the files visited will be `srcs`, `cpy`, `bin`, `pics`, `stuff`, `data`, `work`, `ideas`, `projects`, `file2`, `garbage`, `misc`, `A`. Henceforth it is convenient to assume that the children of a node are always visited in a left-to-right order.

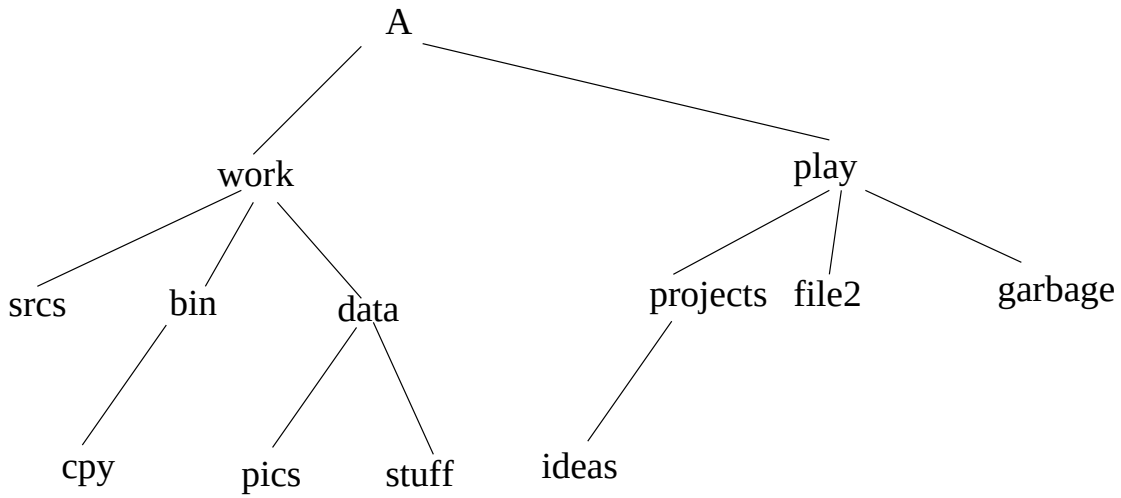


Figure 3.10: Sample tree hierarchy.

The interesting problem is how to recursively accumulate the sizes of the files that it visits. It has to be able to print out the size of each file that it visits, and when it reaches a directory, to print out that directory's total usage. Thus, in Figure 3.10, when it reaches data, it has to print out the sum of the sizes of `pics`, `stuff`, and `data`, but it also has to add the sizes of `srcs`, and the current accumulation in `bin` to a running total to pass up to `work` when it reaches it. This suggests that if we keep a set of running totals indexed by level in the tree, it should suffice to record total size, provided that we do this carefully.

Because the only way to share data between the main program and the function argument to `nftw()` is by making it global, we will declare the array

```
#define MAXDEPTH  50
static  uintmax_t totalsize[MAXDEPTH];
```

in file scope. `uintmax_t` is a type defined in `<stdint.h>`. It is the largest unsigned integer type available on the machine. It is often equivalent to `unsigned long long int`. We will assume that the depth of the tree is never greater than 50 in out implementation. This array must be made global because the main program zeroes it initially and the function that we pass to `nftw()`, which we will call `file_usage()`, must be able to modify it and access it. The prototype for this function is

```
int file_usage(const char *fpath, const struct stat *sb,
                  int tflag, struct FTW *ftwbuf)
```

At any instant of time, `file_usage()` will be visiting a specific file in the tree. Let us call this the current file, and its level, the current level, and let us use the variable `cur_level` to represent this. We call the level of the file processed immediately before the current file the previous level and we will use the variable `prev_level` to store that level. Both of these take on values up to `MAXDEPTH` and no larger. The current file has a size which we will store in the variable `cur_size`. This is the size of the actual file, not the sum of the sizes of any children it may have. Even directories have size – they are usually allocated a single block (4096 bytes these days on most machines). We can obtain this value from the `stat` structure passed into the function; it is `sb->st_size`.

The action that `file_usage()` must take depends on the values of `cur_level` and `prev_level` and nothing else. This is what we will now justify. The following invariant will be maintained by `file_usage()` after it has processed a file:

> `totalsize[cur_level]` is the sum of the sizes of all subtrees in the tree whose roots are at level `cur_level` and are siblings to the left of the currently processed node plus the size of the subtree rooted at the current node.

Suppose first that `prev_level < cur_level`. This implies that we just descended from a node closer to the root of the tree. There is only one way in which this can happen during a post-order traversal: when we reach a leaf node that is leftmost in its tree. For any other node, either the previous node will be at the same level or will be below it. Thus, we have just reached a bottom level and we must set `totalsize[cur_level]` to 0 and add the current file's size to it. Equivalently,

```
cur_size = sb->st_size;
totalsize[cur_level] = cur_size;
```

Notice that `totalsize[cur_level]` satisfies the invariant in this case.

Suppose instead that `prev_level == cur_level`. In this case we are visiting a file that is a sibling of the one previously visited and it cannot be a directory, otherwise the previous node would have been at a lower level. This implies that we have visited all siblings to the left of the current file and that it has no children. Therefore, we need to update `totalsize[cur_level]` by adding the file's size (i.e., `cur_size`) to it:

```
cur_size = sb->st_size;
totalsize[cur_level] += cur_size;
```

Assuming that the invariant was true prior to entering `file_usage()`, it remain true as a result of adding `cur_size` to it in this case since `cur_size` is the size of the subtree rooted at this file.

The last case to consider is when `prev_level > cur_level`. This can only mean one thing – we have just returned in the post-order traversal to a directory all of whose children have just been visited. It can only be the case that `prev_level == cur_level+1` but we do not need this fact to do what needs to be done. This is where the transfer of sizes between levels takes place. First, the size that we have to report for this file is not the directory size itself, but the directory size plus the sum of the sizes passed up the tree to each of its children.

We take advantage of the invariant regarding `totalsize[prev_level]`. What we know is that, since the last node processed was the rightmost node in the subtree rooted at the current directory, and its level is prev_level, `totalsize[prev_level]` must be the sum of the sizes of all subtrees of this directory. Therefore the size to display for it is its own size plus `totalsize[prev_level]`:

```
    cur_size    = totalsize[prev_level] + sb->st_size;
```

To maintain the invariant, we add to `totalsize[cur_level]` the new value of `cur_size`. You can verify that it will hold for the current level in so doing. The last step is the less obvious one, and it will need explanation. We must set `totalsize[prev_level]` to 0. All together, the actions in this case are

```
    cur_size    = totalsize[prev_level] + sb->st_size;
    totalsize[cur_level] += cur_size;
    totalsize[prev_level] = 0;
```

To see why we must zero `totalsize[prev_level]`, consider what will happen when `file_usage()` exits and is called for the next file. Using the file tree in Figure 3.10, suppose the current file is the directory `work`, and `file_usage()` just processed the directory named `data`. `cur_level` = 1 and `prev_level` = 2. The next file that `file_usage()` will process is `ideas`, and then `projects`. After it visits `ideas` and returns to `projects`, `totalsize[1]` must start with the value 0, otherwise the size of `projects` will include the sizes of `srcs`, `bin`, and `data`. In other words, every time that we finish a row of siblings in a subtree, having reached the rightmost sibling, and return to the parent, we must zero out the entry in the `totalsize[]` array for that level. The only chance to do this is when we have recorded its value in the parent and are finished with that node. Doing this preserves the invariant, as no nodes are currently being visited in that level anymore.

We put all of this together in the following listing, which is our initial version of a simple `du` command. Our command will print a message next to a file name if that file had a problem such as being a broken link or an unreadable directory. We could do instead what the real `du` does and print the message to the standard error stream. Comments are omitted to reduce the size of the listing.

Listing simpledu.c

```
#define _XOPEN_SOURCE 500
#include <ftw.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <limits.h>

#define  TRUE  1
#define  FALSE 0
#define  MAXDEPTH  200

/* For getopt() we need to use this feature test macro */
#if  ( _POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE )

static  uintmax_t totalsize[MAXDEPTH];

int file_usage(const char *fpath, const struct stat *sb,
                int tflag, struct FTW *ftwbuf)
{
    static int prev_level = -1;
    int        cur_level;
```

```
    uintmax_t  cur_size;

    cur_level = ftwbuf->level;
    if ( cur_level >= MAXDEPTH ) {
        fprintf(stderr, "Exceeded maximum depth.\n");
        return -1;
    }

    if ( prev_level == cur_level ) {
        cur_size = sb->st_size;
        totalsize[cur_level] += cur_size;
    }
    else if ( prev_level > cur_level ) {
        cur_size    = totalsize[prev_level] + sb->st_size;
        totalsize[cur_level] += cur_size;
        totalsize[prev_level] = 0;
    }
    else {
        cur_size = sb->st_size;
        totalsize[cur_level] = cur_size;
    }

    printf("%ju\t%s", cur_size, fpath);
    prev_level = cur_level;

    if ( tflag == FTW_DNR )
        printf(" (unreadable directory)");
    else if ( tflag == FTW_SL )
        printf(" (symbolic link)" );
    else if ( tflag == FTW_SLN )
        printf(" (broken symbolic link)" );
    else if ( tflag == FTW_NS )
        printf("stat failed " );
    printf("\n");

   return 0;
}

int main(int argc, char *argv[])
{
    int flags = FTW_DEPTH | FTW_PHYS | FTW_MOUNT;
    int status;
    int i = 1;

    if ( argc < 2 )  {
        memset( totalsize, 0, MAXDEPTH*sizeof(uintmax_t));
        status = nftw(".", file_usage, 20, flags);
        if ( -1 == status ) {
           fprintf(stderr, "nftw exited abnormally.\n");
           exit(EXIT_FAILURE);
        }
    }
    else
        while (i < argc) {
```

```
            memset( totalsize, 0, MAXDEPTH*sizeof(uintmax_t));
            status = nftw(argv[i], file_usage, MAXDEPTH, flags);
            if ( -1 == status ) {
               fprintf(stderr, "nftw exited abnormally.\n");
               exit(EXIT_FAILURE);
            }
            else {
                i++;
            }
        }
    exit(EXIT_SUCCESS);
}
#endif
```

If you run this and compare the output to `du -ab`, you might discover that it is not always the same. But try comparing it to the output of `du-alb` and you will see it is the same. Consult the manpage and you will see that the `-l` option tells `du` to count sizes multiple times for files that have multiple links. What is the problem?

This version of `du`, as it stands, counts files with multiple links as many times as they have links. If a file has two names in two different subdirectories of our root directory, each will be counted. What is the way to prevent this? The `stat` structure has both the i-number of the file and the number of links. We can inspect the number of links for non-directory files. If it is greater than one, then it has another name somewhere. We do not know where, but we can store the i-number in a lookup table and set its count to the number of links-1. Each time that we find a file whose link count is greater than one, we can look up its i-number in the table. If it is there with a positive link count, we decrement the link count in the table and do not add the file's size to the running total, and do not even print its name. If the link count reaches zero, we remove it from the table. If the file is not there, we add it to the table with its link count.

Exercise 1. Write an implementation of `du` that does not count files with multiple links more than once.

Exercise 2. Write an implement ion of `du` that has the option to count either 512-byte blocks or bytes.

3.16.3 The `fts` File Hierarchy Traversal Functions

Unlike `nftw()`, `fts` is a family of functions, in much the same way that `opendir()`, `readdir()`, `rewinddir()`, and `closedir()` are inter-related functions, which conform to 4.4BSD but are not POSIX functions. The `fts` set of functions includes `fts_open()`, `fts_read()`, `fts_set()`, `fts_children()`, and `fts_close()`. Just as `opendir()` creates a directory stream object and returns a pointer to it, `fts_open()` creates a *handle* that is used by the other functions. A *handle* is a pointer to an `FTS` structure. Unlike `nftw()`, which does not allow the application to control the order in which files are searched other than whether it is pre-order or post-order, `fts` allows the calling program to specify this order. We begin by looking at the manpage for `fts`. The synopsis is as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fts.h>
```

```
FTS *fts_open(char * const *path_argv, int options,
        int (*compar)(const FTSENT **, const FTSENT **));
FTSENT *fts_read(FTS *ftsp);
FTSENT *fts_children(FTS *ftsp, int options);
int     fts_set(FTS *ftsp, FTSENT *f, int options);
int     fts_close(FTS *ftsp);
```

The paradigm for using `fts` is that we begin by calling `fts_open()`, passing an array of strings representing the roots of trees that we wish to traverse, an integer encoding options, and a function to be used for determining the order in which files are visited. It returns a handle, i.e., a pointer to an `FTS` structure that is then passed to `fts_read()`. Each time that `fts_read()` is called, it visits a file. Each file in the tree is visited just once, except for directories, which are visited before and after their children. `fts_read()` returns a pointer to an `FTSENT` structure for each file that it visits. Files and `FTSENT` structures are in one-to-one correspondence. `FTSENT` structures have a member that allows them to be linked together; the function `fts_children()` returns a pointer to a linked list of these, representing all of the children in a directory, exactly which will be explained below. The fts_set() function allows a file to be reprocessed after it has been returned by a call to `fts_read()`. When one is completely finished processing the directory tree passed to `fts_open()`, it should be closed with `fts_close()`.

There is no need to know the internal structure of a `DIR` structure because the application should treat it opaquely. However, we do need to know the content of the `FTSENT` structure, because that is what characterizes each visited file. The two structures are defined in the header file `<fts.h>`, found in the standard directory, `/usr/include`. Their definitions (omitting some macros here) are

```
typedef struct {
struct _ftsent *fts_cur;    /* current node */
struct _ftsent *fts_child;  /* linked list of children */
struct _ftsent **fts_array; /* sort array */
dev_t fts_dev;              /* starting device # */
char *fts_path;             /* path for this descent */
int fts_rfd;                /* fd for root */
int fts_pathlen;            /* sizeof(path) */
int fts_nitems;             /* elements in the sort array */
int (*fts_compar) (const void *, const void *); /* compare fn */
int fts_options;            /* fts_open options, global flags */
} FTS;
typedef struct _ftsent {
    unsigned short fts_info;     /* flags for FTSENT structure */
    char          *fts_accpath;  /* access path */
    char          *fts_path;     /* root path */
    short          fts_pathlen;  /* strlen(fts_path) */
    char          *fts_name;     /* filename */
    short          fts_namelen;  /* strlen(fts_name) */
    short          fts_level;    /* depth (-1 to N) */
    int            fts_errno;    /* file errno */
    long           fts_number;   /* local numeric value */
```

```
        void          *fts_pointer;  /* local address value */
        struct ftsent *fts_parent;   /* parent directory */
        struct ftsent *fts_link;     /* next file structure */
        struct ftsent *fts_cycle;    /* cycle structure */
        struct stat   *fts_statp;    /* stat(2) information */
    } FTSENT;
```

The `FTSENT` structure has many members, and they all need to be understood to use the `fts` functions to their maximum extent, but for relatively simple applications it is not necessary to understand them all. The most important members, with brief descriptions are

`fts_info` An integer that encodes information about the type of object represented by this structure.

`fts_accpath` A path for accessing the file from the current directory.

`fts_path` The path for the file relative to the root of the traversal. This path contains the path specified to `fts_open()` as a prefix.

`fts_name` The file name.

`fts_errno` Upon return of a `FTSENT` structure from the `fts_children()` or `fts_read()` functions, with its `fts_info` field set to `FTS_DNR`, `FTS_ERR` or `FTS_NS`, the `fts_errno` field contains the value of the external variable errno specifying the cause of the error. Otherwise, the contents of the `fts_errno` field are undefined.

`fts_number` This field is provided for the use of the application program and is not modified by the `fts` functions. It is initialized to 0.

`fts_pointer` This field is provided for the use of the application program and is not modified by the `fts` functions. It is initialized to `NULL`.

`fts_parent` A pointer to the `FTSENT` structure referencing the file in the hierarchy immediately above the current file, i.e. the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the `fts_level`, `fts_number` and `fts_pointer` fields are guaranteed to be initialized.

`fts_link` Upon return from the `fts_children()` function, the `fts_link` field points to the next structure in the `NULL`-terminated linked list of directory members. Otherwise, the contents of the `fts_link` field are undefined.

`fts_statp` A pointer to a `stat` structure for the file.

The `fts_info` field can have any of the following values:

`FTS_D` A directory being visited in pre-order.

`FTS_DC` A directory that causes a cycle in the tree. (The `fts_cycle` field of the `FTSENT` structure will be filled in as well.)

`FTS_DEFAULT` Any `FTSENT` structure that represents a file type not explicitly described by one of the other `fts_info` values.

`FTS_DNR` A directory which cannot be read. This is an error return, and the fts_errno field will be set to indicate what caused the error.

`FTS_DOT` A file named "." or ".." which was not specified as a file name to `fts_open()` (see `FTS_SEEDOT`).

`FTS_DP` A directory being visited in post-order. The contents of the `FTSENT` structure will be unchanged from when it was returned in pre-order, i.e. with the `fts_info` field set to `FTS_D`.

`FTS_ERR` This is an error return, and the `fts_errno` field will be set to indicate what caused the error.

`FTS_F` A regular file.

`FTS_NS` A file for which no `stat` information was available. The contents of the `fts_statp` field are undefined. This is an error return, and the `fts_errno` field will be set to indicate what caused the error.

`FTS_NSOK` A file for which no `stat` information was requested. The contents of the `fts_statp` field are undefined.

`FTS_SL` A symbolic link.

`FTS_SLNONE` A symbolic link with a non-existent target. The contents of the `fts_statp` field reference the file characteristic information for the symbolic link itself.

Some immediate observations from the above information are that:

- The `fts_info` member has information similar to that found in the integer passed to the `(*fn)` function by `nftw()`.
- Unlike the `nftw()` function, `fts` provides hooks for the application to use. In particular, `fts_number` and `fts_pointer` are two members of the returned structure that can be used for application specific data, making it possible to change state and data among different invocations of the `fts_read()` function.
- The `fts_parent` field provides a means to access the parent node, unlike `ntfw()`.
- It has `stat` information for the returned file in the `fts_statp` member, unless `fts_info` is either `FTS_NS` or `FTS_NSOK`.
- The name of the file is in the `fts_name` member. The `fts_path` member has the pathname of the file relative to the root of the search. For all practical purposes, this is the same path as `fts_accpath`.

It is also to important to know that a single buffer is used for the various path members of all `FSTENT` structures of all files in file hierarchy. Because of this, there is no guarantee that the `fts_path` and `fts_accpath` members of a previously returned `FTSENT` structure are still properly null-terminated, because they might have been written over already. Only the `fts_path` and `fts_accpath` members of the file most recently returned by `fts_read()`.are guaranteed to be null-terminated. To use these fields to reference any files represented by other `FTSENT` structures will require that the path buffer

be modified using the information contained in that `FTSENT` structure's `fts_pathlen` field. Any such modifications should be undone before further calls to `fts_read()` are attempted. In contrast, the `fts_name` field is always null-terminated.

Our first example illustrates the basic concepts. It is derived from a program from *RosettaCode.org*. The program, named `ftsdemo`, is given the name of a directory and a shell-style regular expression, enclosed in single quotes to prevent shell expansion of it, and it displays the names of all files in the given directory's hierarchy that match the expression. For example, `'*.c'` will cause all files ending in `'.c'` to be matched. The listing and an explanation follow.

Listing ftsdemo.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <err.h>
#include <errno.h>
#include <fnmatch.h>
#include <fts.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/*****************************************************************************
                            Utility Function Prototypes
*****************************************************************************/
/** usage()
 *  Print usage message on standard output
 */
void usage(char* progname);

/** entcmp()
 * Compare files by name.
 */
int entcmp(const FTSENT **s1, const FTSENT **s2);

/** pmatch()
 * Print all files in the directory tree that match the glob pattern.
 * Example: pmatch("/usr/src", "*.c");
 */
void pmatch(char *dir, const char *pattern);

/*****************************************************************************
                                Main Program
*****************************************************************************/
int main(int argc, char* argv[])
{
    if ( argc < 3 )  {
        usage(argv[0]);
        exit(1);
    }
    pmatch(argv[1], argv[2]);
    return 0;
}
```

```
/*****************************************************************************/
int entcmp(const FTSENT **s1, const FTSENT **s2)
{
    return (strcoll((*s1)->fts_name, (*s2)->fts_name));
}


/*****************************************************************************/
void usage(char* progname)
{
    printf("usage: %s directory pattern\n", progname);
}

/*****************************************************************************/

void pmatch(char *dir, const char *pattern)
{
    FTS    *tree;   /* pointer to file tree stream returned by fts_open */
    FTSENT *f;      /* pointer to structure returned by fts_read */
    char *argv[] = { dir,  NULL };

    /* Call fts_open(0 with FTS_LOGICAL to follow symbolic links
     * including links to other directories. Since it detects cycles,
     * we do not have to worry about infinite loops.
     */
    tree = fts_open(argv, FTS_LOGICAL , entcmp);
    if (tree == NULL)
        perror("fts_open");

    /* Repeatedly get next file, skipping "." and ".." because
     * FTS_SEEDOT was not set.
     */
    while ((f = fts_read(tree))) {
        switch (f->fts_info) {
        case FTS_DNR:  /* Cannot read directory */
            fprintf(stderr, "Could not read %s\n", f->fts_path);
            continue;
        case FTS_ERR:  /* Miscellaneous error */
            fprintf(stderr, "Error on %s\n", f->fts_path);
            continue;
        case FTS_NS:  /* stat() error */
            /* Show error, then continue to next files. */
            fprintf(stderr, "Could not stat %s\n", f->fts_path);
            continue;
        case FTS_DP:
          /* Returned to directory for second time as part of
              post-order visit to directory, so skip it. */
            continue;
        }

        /*
         * Check if the name matches pattern, and if so, print out its
         * path. This check uses FNM_PERIOD, so "*.c" will not
         * match ".invisible.c".
```

```
             */
            if (fnmatch(pattern, f->fts_name, FNM_PERIOD) == 0)
                printf("%s\n", f->fts_path);

            /*
             * A cycle happens when a symbolic link (or perhaps a
             * hard link) puts a directory inside itself. Tell user
             * when this happens.
             */
            if (f->fts_info == FTS_DC)
                fprintf(stderr, "%s: cycle in directory tree",
                        f->fts_path);
        }

        /* fts_read() sets errno = 0 unless it has an error. */
        if (errno != 0)
          perror("fts_read");

        if (fts_close(tree) < 0)
          perror("fts_close");
}
```

The main program does very little; it checks usage and if the usage is correct, it calls `pmatch()`, passing the name of the directory and the expression to be matched.

`pmatch()` begins by constructing a proper first argument for the call to `fts_open()`. The first argument must be a `NULL`-terminated list of directory names. If we want to traverse just a single directory, we therefore have to create a list of the form `{dirname, NULL}`, where `dirname` is a `NULL`-terminated string.

It then calls `fts_open()`, setting the `FTS_LOGICAL` flag so that it can follow symbolic links. A nice feature of `fts` is that it detects when it is about to complete a cycle caused by symbolic links, and it sets the `fts_info` member of the returned structure to `FTS_DC` in this case, so that the application can handle it. The third argument to `fts_open()` is a pointer to the comparison function that will be used for sorting the files. If a `NULL` pointer is supplied, the directory traversal order is in the order listed in the `argv` array passed as argument one for the root paths, and in the order listed in each directory for everything else.. The function prototype must be

```
int compare(const FTSENT **, const FTSENT **));
```

and it must return a negative value, zero, or a positive value to indicate if the file referenced by its first argument comes before or after, the file referenced by its second argument. The `fts_accpath`, `fts_path` and `fts_pathlen` members of the `FTSENT` structures may never be used in this comparison. If the `fts_info` member is set to `FTS_NS` or `FTS_NSOK`, the `fts_statp` field may not be used either. If `fts_open()` fails, it returns a `NULL` pointer.

In this program, files will be sorted by the default collating order, which is accomplished by calling `strcoll()` with the two file names:

```
int entcmp(const FTSENT **s1, const FTSENT **s2)
{
    return (strcoll((*s1)->fts_name, (*s2)->fts_name));
}
```

After getting the `FTS` pointer, tree, it uses this to repeatedly call `fts_read()`. For each file that it visits, `fts_read()` returns a pointer to an `FTSENT` structure as long as there are files remaining and no error occurred. If an error occurred it returns `NULL` and sets `errno` to some error value, and if there are no more files left, it returns `NULL` and sets `errno` to zero. The program checks the `fts_info` member of the structure to determine whether an error occurred, and if it matters, whether it is a directory or a file. In this case it does not matter what type of file it is, but only whether some type of error occurred. If no error occurred, it calls the `fnmatch()` library function, whose prototype is

```
#include <fnmatch.h>
int fnmatch(const char *pattern, const char *string, int flags);
```

The `fnmatch()` function checks whether its string argument matches its pattern argument, which must be a valid shell pattern, and returns zero if it matches, `FNM_NOMATCH` if it does not match, and some other value if there is an error. The third argument can be used to pass various flags that control how it behaves. In our program, we want the shell pattern to explicitly use a period character if the file is supposed to have a leading period. For example, the pattern `'*.h'` should not match the file `.foo.h`. The flag `FNM_PERIOD` tells `fnmatch()` to match only if that period is the first character in the string. Every file that matches the pattern is printed on standard output.

When the loop terminates, it checks `errno` to see if there was an error, and it calls `fts_close()` to cleanup and release resources.

This program was relatively simple. We now show a slightly more interesting application of `fts` that uses the `fts_children()` function and two different methods of ordering the tree traversal.

As noted above, the `fts_children()` function returns a pointer to a linked list of `FTSENT` structures that represent the files of a single directory in the hierarchy. The burning question is, which directory? The answer is not simple. If `fts_read()` was never called at all after `fts_open()` was called, then `fts_children()` returns a pointer to the list of top-level directories passed to `fts_open()` in its `argv` array. If `fts_read()` was called prior to calling `fts_children()` and the last object returned by a call to `fts_read()` was a directory, then it is this directory whose children are delivered in the linked list returned by `fts_children()`. But if the last call to `fts_read()` returned a non-directory file, then `fts_children()` returns a `NULL` pointer and sets `errno` to zero. This return value is indistinguishable from what it will do if the last object returned by a call to `fts_read()` was an empty directory. In this case also, `fts_children()` will return a `NULL` pointer and set `errno` to zero. So this final state implies that either the last object was an empty directory or a non-directory file. In either case it has no children. Lastly, if an error condition arises, `fts_children()` will return a `NULL` pointer but set `errno` to some non-zero value.

To demonstrate the use of `fts_children()`, we will use it to implement our `ls` command once more, but this time we will allow the user to specify a few different orderings in which to list the files. Specifically, the user can list by collating order, by time of last modification, or by file size, in all cases ascending. To make this possible, the program will implement three different `compare()` functions; which one is used will depend on the command line option supplied by the user. The program's usage is

```
ls_fts [-l] [-m | -s ] [file file ...]
```

The `-m` option sorts by modification time, `-s`, by size, and `-l` turns on long listings. The program listing below omits the helper functions already displayed in previous versions of `ls`, as well as almost all comments.

```
Listing ls_fts.c
#include <sys/types.h>
#include <err.h>
#include <errno.h>
#include <fnmatch.h>
#include <fts.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include "utils.h"

#define  BYTIME  1
#define  BYNAME  2
#define  BYSIZE  3
#define  HERE    "."
#define  TRUE    1
#define  FALSE   0

int entcmp(const FTSENT **a, const FTSENT **b)
{
    return strcoll((*a)->fts_name, (*b)->fts_name);
}

int mtimecmp(const FTSENT **s1, const FTSENT **s2)
{
    return (int) (((*s1)->fts_statp)->st_mtime - ((*s2)->fts_statp)->st_mtime);
}

int szcmp(const FTSENT **s1, const FTSENT **s2)
{
    return (int) (((*s1)->fts_statp)->st_size - ((*s2)->fts_statp)->st_size);
}

void ls  ( char dir[], int do_longlisting, int sortflag )
{
    FTS *tree;
    FTSENT *f;
    char *argv[] = { dir, NULL };

    switch ( sortflag ) {
    case BYTIME:
        tree = fts_open(argv, FTS_LOGICAL , mtimecmp);
        break;
    case BYNAME:
        tree = fts_open(argv, FTS_LOGICAL , entcmp);
        break;
```

```
    case BYSIZE:
        tree = fts_open(argv, FTS_LOGICAL , szcmp);
        break;
    }

    if (tree == NULL)
        perror("fts_open");

    f = fts_read(tree);
    if ( NULL == f ) {
        perror("fts_read");
        return;
    }

    f = fts_children(tree, 0);
    if ( NULL == f )
        if ( errno != 0 )
            perror("fts_children");
        else
            fprintf(stderr, "empty directory\n");

    while ( f != NULL ) {
        switch (f->fts_info) {
        case FTS_DNR:  /* Cannot read directory */
            fprintf(stderr, "Could not read %s\n", f->fts_path);
            continue;
        case FTS_ERR:  /* Miscellaneous error */
            fprintf(stderr, "Error on %s\n", f->fts_path);
            continue;
        case FTS_NS:  /* stat() error */
            fprintf(stderr, "Could not stat %s\n", f->fts_path);
            continue;
        case FTS_DP:
          /* Returned to directory for second time as part of
             post-order visit to directory, so skip it. */
            continue;
        }
        if ( do_longlisting )
            print_file_status(f->fts_name, f->fts_statp);
        else
            printf("%s\n", f->fts_name);

        f = f->fts_link;
    }
    if (errno != 0)
      perror("fts_read");

    if (fts_close(tree) < 0)
      perror("fts_close");
}

/**********************  Main Program ***********************************/

int main(int argc, char* argv[])
```

```
{
    int longlisting = 0;
    int howtosort   = BYNAME;
    int ch;
    char options[] = ":lms";
    opterr = 0;

    while  (TRUE) {
        ch = getopt(argc, argv, options);
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'l':
            longlisting = 1;
            break;
        case 'm':
            if ( howtosort != BYSIZE )
                howtosort = BYTIME;
            else {
                printf("usage: %s [-l] [-m|-s] [files]\n", argv[0]);
                return 1;
            }
            break;
        case 's':
            if ( howtosort != BYTIME )
                howtosort = BYSIZE;
            else {
                printf("usage: %s [-l] [-m|-s] [files]\n", argv[0]);
                return 1;
            }
            break;

        case '?' :
            printf("Illegal option ignored.\n");
            break;
        default:
           printf ("?? getopt returned character code 0%o ??\n", ch);
           break;
        }
    }

    if ( optind == argc )     /* no arguments; use . */
        ls( HERE, longlisting, howtosort );
    else
        /* for each command line argument, display file */
        while ( optind < argc ) {
            ls( argv[optind], longlisting, howtosort );
            optind++;
        }
    return 0;
}
```

Notes.

- The main program does the option parsing and prevents the user from choosing to sort by both size and modification time, because sorting by both implies having to resort the linked list returned by `fts_children()`.
- The main program calls `ls()` for each command line argument rather than assembling them into an array of strings, because `ls()` is designed to display a single directory's contents only.
- The `mtimecmp()` and `szcmp()` functions use arithmetic rather than conditional evaluation. This makes them faster, but on a non-POSIX system, the time subtraction may not work. Both work by accessing the `stat` structure pointed to by the `FTSENT` structure's `fts_statp` member. No check is made that it is `NULL`.
- `ls()` begins by calling `fts_open()`. It then calls `fts_read()` so that the directory object will be the last object read by a call to `fts_read()`, ensuring that `fts_children()` will return a pointer to the list of children of that directory, if it is non-empty.
- Unlike the previous program, this does not repeatedly call `fts_read()`. It traverses the linked list of `FTSENT` structures until it reaches a `NULL` link and then terminates.

This is just a short overview of the `fts` functions. It does not show how to use the hooks provided to the application inside the `FTSENT` structure.

Exercise. A good exercise would be to implement the `du` command using fts, taking advantage of the members of the `FTSENT` structure available to the application, and the parent pointers.

Appendix A

A.1 Useful Command-Line Options for ls

Some of the most useful command-line options for ls are listed below, with brief descriptions.

Option	Description
`-a`	show dot-files (i.e. hidden files)
`-l`	display a long listing, with file type, permissions, number of links, owner, group, size in bytes, time of last modification
`-lu`	same as `ls -l`, except uses the last time the file was read instead of modified.
`-s`	show the file size in blocks
`-t`	sort the files by timestamp specified (`u` for access, modification by default)
`-F`	show file types
`-i`	show the i-number of the file
`-d`	if `ls` has a directory argument, show the attributes of the directory, instead of its contents
`-R`	recursively descend any directories or their subdirectories, applying any other options listed

As you can see, several of these options can be extremely useful when searching for things in the file system. A "dot-file" is not a file filled with dots. It is a file whose name begins with a dot, such as "`.bashrc`" or "`.login`" or "`.evolution`". You read the first as "dot-bash-R-C", and the second as "dot-login". Dot-files are thought of as hidden files because the plain `ls` command does not display files whose names begin with a dot, and `ls` is the only command to view the content of a directory. Even in desktop environments, the default settings of a browser like Nautilus will hide these files. In your home directory, if you type "`ls -a`" you will see all of the files and directories whose names begin with a dot. In this list you will see "`.`" and "`..`". Recall that these refer to the directory itself and its parent. Note that unlike Windows, the UNIX kernel has no concept of a hidden file.

A.1.1 Bit Masks

Suppose we want to extract the value of bit 8 of some 16-bit number named `flags`. If we perform a bitwise AND of `flags` and the binary number that has 0's everywhere except in bit 8, called `mask8` here, as illustrated:

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	`mask8`
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	`flags`
0	0	0	0	0	0	0	b_8	0	0	0	0	0	0	0	0	`mask8 & flags`

then the result, `mask8 & flags`, will have 0's everywhere and the value of bit 8 of `flags` in bit 8. Therefore `mask8 & flags == 0` if and only if bit 8 in `flags` is 0. I can represent the value of this 16-bit mask variable as the octal number `000400`. Octal numbers in C and C++ must begin with a leading 0, so this would be written `0000400`, and the bitwise AND would be written as

```
flags & 0000400
```

in a program. We can then test the bit and take action accordingly with code like

```
if (flags & 0000400 ) ...
```

A.2 The `find` Command

In addition to the commands you have seen in previous chapters that can recursively traverse the file hierarchy, there are a few UNIX commands that are designed to traverse the entire tree at a given directory, without having to specify any special recursive option. The most useful of these is the `find` command. The `du` command also does this:

`find` searches for files in a tree structure rooted at the given directory.

`du` displays the disk usage of all files in a given directory tree

The find command is a very powerful command. its basic usage is

```
find [options] path ... expression
```

where `path ...` means one or more directories, and `expression` is composed of options, tests, and actions joined by various operators. Unless the expression explicitly prunes the tree with the use of the `-prune` option, the entire tree rooted at each directory argument will be traversed. Options are expressions that always return true. Actions are expressions that can return true or false. The operators that combine expressions are boolean operators, `and`, `or`, `not`, and a list operator. If no operator is present between expressions, operator `and` is used instead. The best way to see how `find` can be used is by some examples.

```
$ find . -name '*.c' -print
```

searches for all files in the tree rooted at '.' matching the pattern `*.c`, printing out the matching pathnames. The `-print` action can be omitted because `-print` is the default action. The `-name` test will return true if a file matches the given pattern. When the test is true, the next action will be applied, which is how the file name is printed.

```
$ find . -amin -30
```

searches for all files in the tree rooted at '.' that have been accessed within the past 30 minutes and prints them.

```
$ find . -mmin -120
```

searches for all files in the tree rooted at '.' that have been modified within the past 120 minutes and prints them.

```
find / -links +1 ! -type d
```

searches through the entire file system looking for all files whose link count is greater than 1 and which are not directories (`! -type d`).

```
find . -samefile myfile
```

searches for all files that are hard links to `myfile`

```
find / -name core -exec /bin/rm -f '{}' \;
```

searches for all files named `core` in the file hierarchy and runs `/bin/rm -f` on each one, which deletes them. The notation `{}` means the currently matched file name. To protect the braces from being interpreted by the shell, they are enclosed in single quotes. The command that find runs must be terminated by a semi-colon, but again it must be escaped so that the shell does not interpret it as the end of the find command itself. We could write `\;` or `';'`.

```
find . -perm /o=w
```

finds all files in the current directory and below that others can write.

This is just a handful of search options that are possible. There are options to test file size, check ownership. permissions, group ownership, compare two files, and so on. It is well worth learning how to use this command.

Chapter 4 Control of Disk and Terminal I/O

Concepts Covered

File structure table, open file table, file status flags, auto-appending, device files, terminal devices, device drivers, line discipline, termios structure, terminal settings, canonical mode, non-canonical modes, IOCTLs, fcntl, ttyname, isatty, ctermid, getlogin, gethostname,tcgetattr, tcsetattr,tcflush, tcdrain, ioctl,

4.1 Overview

This chapter begins by examining how a program can exercise some control over the connections that it makes with disk files. It describes the data structures used by the kernel to manage open files, and explains how processes can interact with these files. It then explores device files and how they are used and structured, and device drivers and their roles and structures. From there it looks at block and character device file differences, after which it turns to terminals. The remainder of the chapter is then devoted to controlling terminals.

4.2 Open Files

The programs that we have studied so far operate on disk files, which are files that reside, of course, on disks[1]. You are by now well-acquainted with the general model of file I/O: open the file, access it, and close it. Remember that this model works for any kind of file, not just disk files. Here though, we are interested only in disk files.

In this model of file I/O, opening a file returns a reference to an object that can be used to access the file, either for reading or writing. When you use the `open()` system call, you get a *file descriptor* in return; when you use the `fopen()` C standard I/O library call, you get a `FILE` *pointer* in return. Either way, you are getting a scalar object (i.e., a small integer or a pointer) that is associated with a hidden (kernel) data structure that allows you to access the file. Here, we will explore how much control we can exercise over the way in which the program is connected to the file.

When a process opens a disk file using the `open()` system call, the kernel creates a data structure that represents the connection between the process and the file. The returned file descriptor can be used by the process to access the file. A file descriptor is simply an index into a per-process table[2] that the kernel uses to locate that specific data structure. Different versions of UNIX call this data structure different things, and the data structure may have slightly different members from one UNIX variant to another, but the basic members and purpose of the structure are invariant: its most important member is the *file pointer*, and its purpose is to store the position in the file from which the next operation will take place (whether it is a read or a write.)

[1] This is in contrast to device files, which are very different from disk files.
[2] A "per-process" table is a table of which each process has its own instance.

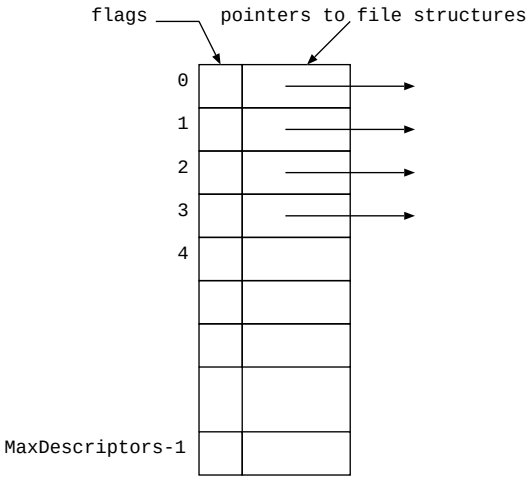


Figure 4.1: Per-process open file table

This data structure is called a *file entry* in BSD UNIX and a *file structure* in Linux. I will call it a *file structure* in these notes. The connection between a process and a file is completely characterized by the information contained in the file structure. Aside from the file pointer, the kinds of information it typically contains include:

- whether or not the file is open for reading, writing, reading and writing, or appending,
- whether or not the I/O is buffered or unbuffered, and
- whether or not the access is exclusive or whether other processes also access the file.

as well as other information that is required by the kernel. The information contained in the file structure characterizes the connection between the open file and the process; it is specific to this single connection. Other processes might have this file open with different attributes. Many of the attributes of the connection can be changed by the process; others cannot. Which can and which cannot, and how is it done? These are the questions we will answer.

Let us begin by examining all of the data structures related to open files. Each process has a table that is usually called the *open file table*. (In 4.4BSD, this table was called the *descriptor table*, and each entry was called a *filedesc* structure.) In Linux, this table is the `fd` array, which is part of a larger structure called the *files_struct*. The file descriptor returned by the `open()` system call is actually an index into the open file table of the process making the call. Recall that every process is given three file descriptors when it is created: 0, 1, and 2, respectively, for standard input, output, and error. These are the first three indices in this table, as shown in Figure 4.1.

When a process issues the `open()` system call, the kernel creates a new file structure and fills the lowest-numbered available slot in the process's open file table with a pointer to that file structure. The file structures for all open files are contained in a table called the *file structure table*. which resides in the address space of the kernel. See Figure 4.2.

The left side of the figure contains the per-process open file tables, one for each process. These tables, although in the virtual address space of the processes, are accessible only by the kernel. The right side of the table contains the file structure table, in kernel memory. The gray region at the bottom is on disk; everything above the gray area is memory-resident. You will notice in the figure

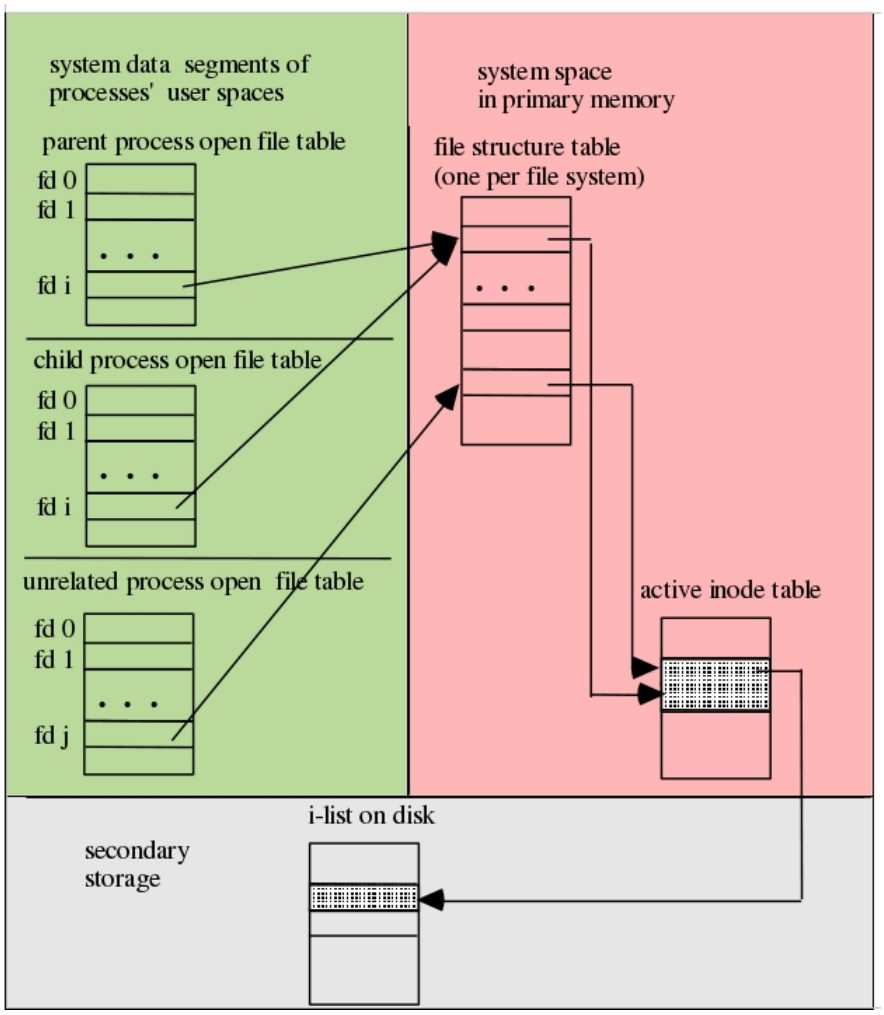


Figure 4.2: Kernel data structures related to open files.

that some processes' file descriptors point to the same file structure for the same open file, whereas others' descriptors point to separate file structures for the same file. Sometimes processes share a connection and other times, even though two or more processes may have the same file open, they access it through different connections. When different processes are connected to the same file through different file structures, the file pointers are different. When a file structure is shared by processes, they share the file pointer. You will see in the chapter on process management how these situations arise.

The file structures in the file structure table point to the active i-nodes for the open files. These i-nodes are maintained in the active i-node table in kernel memory. The active i-node table contains copies of the i-nodes on disk. If the i-nodes change in memory, the changes are written to the i-nodes on disk.

4.2.1 Using `fcntl()` to Control File Descriptor Attributes

The file structure contains a set of flags that control I/O with respect to the file. These flags are called *file status flags*, and they are shared by all processes that share that file structure. The file

descriptor is the means by which the process can modify those flags and alter the behavior of that connection to the file. The method of modifying the flags of an existing file structure is a three-step procedure:

1. The process gets a copy of the current attributes of the connection from the kernel, storing it in its own address space;
2. The process modifies the current attributes in its copy;
3. The process requests the kernel to write its copy back to the kernel's in-memory structures.

The system call that performs steps 1 and 3 is the `fcntl()` call. You can pronounce `fcntl` as "f control", short for file control. `fcntl()` is a function that operates on open files. Depending upon its arguments, `fcntl()` will either get the attributes of a file connection or set them. It has a very long man page that starts as follows:

```
NAME
       fcntl - manipulate file descriptor
SYNOPSIS
       #include <unistd.h>
       #include <fcntl.h>

       int fcntl(int fd, int cmd, ... /* arg */ );


DESCRIPTION
     fcntl performs one of various miscellaneous operations
     on fd.  The operation in question is determined by cmd.
     ...
```

Remarks

- The first parameter is the file descriptor of an already open file.
- The second parameter is an integer that `fcntl()` interprets as a command. Names for these integers are defined in `<fcntl.h>`; the names that are relevant to getting or setting file status flags are:
  - `F_GETFL` which tells `fcntl()` to return a copy of the set of flags;
  - `F_SETFL` which tells `fcntl()` to expect a third integer parameter that contains a new flag set to replace the current one.

  Some of the other commands that `fcntl()` can perform include
  - `F_DUPFD` which duplicates an existing file descriptor
  - `F_GETFD`, `F_SETFD` which get and set *file descriptor flags* (see below)
  - `F_GETOWN`, `F_SETOWN` which get and set the ownership of the `SIGIO` signal (see below)

  - `F_GETSIG, F_SETSIG` which get and set the signal that is sent when using asynchronous I/O
  - `F_GETLK`, `F_SETLK`, `F_SETLKW` which acquire, release, and test for the existence of record locks.

- Each control flag is a single bit in a long integer. To turn on an attribute, you need to set the bit. To turn it off, you need to zero it. The `<fcntl.h>` header file contains definitions of masks that can be used for this purpose. To set a bit, you can do a bitwise-or of the particular mask with the flag variable; to unset it, you can do a bitwise-and of the complement of the mask with the control flag variable. The masks are defined in `/usr/include/bits/fcntl.h`, which is included in the `<fcntl.h>` header file. They are also defined in the man page for `<fcntl.h>`.

File status flags reside in the file structure, which may be shared by multiple processes. *File descriptor flags* are part of the entry in a process's open file table and are associated with the actual file descriptor. At present there is only one such flag, `FD_CLOEXEC`, the *close-on-exec* flag. This flag is not relevant to anything that we cover in this chapter. The `F_GETOWN` and `F_SETOWN` commands will be explained when we cover asynchronous I/O, and we ignore them for now.

Similarly, commands related to record locking (`F_GETLK`, `F_SETLK`, `F_SETLKW`) will be covered when we turn to the topic of file sharing.

Not all file status flags can be changed after a file is opened. For example, if a file is opened for writing with the `O_WRONLY` flag, you cannot use `fcntl()` to change its access mode to reading. The flags that can be changed after a file has already been opened are a subset of the file status flags. The most important of them, and their mnemonic masks are:

`O_APPEND` Append mode. Before each `write()` operation, the file pointer is positioned at the end of the file, as if with `lseek()`, *atomically*. This may not work on remotely mounted file systems. Setting `O_APPEND` is done to eliminate race conditions.

`O_ASYNC` Asynchronous writes. Generate a signal when input or output becomes possible on this file descriptor. This feature is only available for terminals, pseudo-terminals, and sockets, not for disk files!

`O_NONBLOCK` or `O_NDELAY` Non-blocking mode. No subsequent operations on the file descriptor will cause the calling process to wait. This is strictly for *FIFOs* (also known as *named pipes*) and may not have any effect on files other than FIFOs. POSIX specifies the mask `O_NONBLOCK`, but some systems expect `O_NDELAY` instead. Many systems patch their header files so that they are the same. In Linux `O_NONBLOCK` may not be equivalent to `O_NDELAY`. Check the man page for your system.

`O_SYNC` Synchronous I/O. Any writes on the file descriptor will block the calling process until the data has been physically written to the underlying hardware. In Linux this attribute cannot usually be modified by `fcntl()`, and it may or may not be implemented.

From the above descriptions. you can see that there is little that we can actually do through `fcntl()` in Linux for disk files. The `O_ASYNC` flag will be important when we study terminal connections. The `O_NONBLOCK` flag often has no effect. The `O_APPEND` flag is useful, and this is one we will explore. The `O_SYNC` flag turns on synchronous writing. Synchronous writing is writing in which the process

is blocked until the data is actually written to the device, rather than to the kernel buffers. In other words, it turns off kernel buffering for this connection. There are very few reasons why a process should want to do this, as it slows down its execution significantly. If a process wants to force disk writes, it can always use `fsync()` periodically.

Remember that, in order to modify either the file descriptor flags or the file status flags, you cannot just issue an `F_SETFD` or an `F_SETFL` command through `fcntl()`, as this could turn off flag bits that were previously set. Instead you have to follow the three-step procedure outlined above. A typical code sequence to set a flag such as `O_APPEND`, given that `fd` is the file descriptor of a file that is open for writing, is

```
int flags, result;
flags  = fcntl(fd, F_GETFL);
flags |= (O_APPEND);
result = fcntl( fd, F_SETFL,flags);
if ( -1 == result )
    perror("Error setting O_APPEND")
return 0;
```

Notice that the mask is bitwise-or-ed with the flag variable. To turn off a bit, you would bitwise-and the complement of the mask, as in the sequence:

```
int flags, result;
flags  = fcntl(fd, F_GETFL);
flags &= ~(O_APPEND);
result = fcntl( fd, F_SETFL,flags);
if ( -1 == result )
    perror("Error unsetting O_APPEND");
return 0;
```

Although we are limited in which flags we can set in Linux for disk files, we can use `fcntl()` to check the values of all flags. The following function demonstrates how to check the state of various flags. The function requires inclusion of the `<fcntl.h>` header file. The macro `O_ACCMODE` is a mask that can be used to check which of `O_WRONLY`, `O_RDONLY`, or `O_RDWR` is set. These values are stored in the low-order two bits of the integer flagset returned by `fcntl()` and are defined below. These are not independent bits, which is why you need the two-bit mask.

```
#define O_ACCMODE 0003
#define O_RDONLY 00
#define O_WRONLY 01
#define O_RDWR   02
```

We put these ideas together in a function named `check_file_status()` which, when given the file descriptor of an open file, prints its access mode and which status flags are set on that descriptor.

```
int check_file_status ( int fd )
{
    int flags = fcntl ( fd, F_GETFL );
    if ( -1 == flags ) {
        perror ( "Could not get flags.");
        return ( -1 );
    }

    switch ( flags & O_ACCMODE ) {
    case O_WRONLY:
        printf("write-only\n");
        break;
    case O_RDONLY:
        printf("read-only\n");
        break;
    case O_RDWR:
        printf("read-write\n");
        break;
    }

    if ( flags &  O_CREAT )     printf("O_CREAT flag is set\n");
    if ( flags &  O_EXCL )      printf("O_EXCL flag is set\n");
    if ( flags &  O_NOCTTY )    printf("O_NOCTTY flag is set\n");
    if ( flags &  O_TRUNC )     printf("O_TRUNC flag is set\n");
    if ( flags &  O_APPEND )    printf("O_APPEND flag is set\n");
    if ( flags &  O_NONBLOCK )  printf("O_NONBLOCK flag is set\n");
    if ( flags &  O_NDELAY )    printf("O_NDELAY flag is set\n");
#if defined(O_SYNC)
    if ( flags &  O_SYNC )      printf("O_SYNC flag is set\n");
#endif
#if defined(O_FSYNC)
    if ( flags &  O_FSYNC )     printf("O_FSYNC flag is set\n");
#endif
    if ( flags &  O_ASYNC )     printf("O_ASYNC flag is set\n");
    printf("\n");
    return 0;
}
```

As `O_SYNC` and `O_FSYNC` are not necessarily defined on all UNIX systems, the tests for these flags are conditionally compiled. You can embed this function into a main program such as

```
int main(int argc, char *argv[])
{
    int  flags, fd;

    if ( argc != 2 ) {
        printf("usage: %s <descriptor#>\n", argv[0]);
        exit(1);
     }
    if ( ( fd = atoi(argv[1])) < 0 ) {
        printf("usage: %s <descriptor#>\n", argv[0]);
```

```
            exit(1);
        }
    check_file_status(fd);
    return 0;
}
```

and run the program redirecting standard input, output and error. In the process you will discover some surprising results. For example, if the above program is named `checkstatus`, try to predict the output of

```
$ ./checkstatus 1 > out1
$ cat out1
```

and of

```
$ ./checkstatus 0
$ ./checkstatus 2 2>errs
```

The point is to determine, when input or output is redirected, how the status flags for the three standard devices, standard input, output, and error, are changed.

4.2.2 Appending and Race Conditions

The `O_APPEND` flag controls append mode. Consider the following problem. A shared log file is used to record various system activities. Each of several different processes adds its own entry to the end of the log file to record its activity.

Let us make this concrete. Recall that UNIX uses the `wtmp` file to record each and every login and logout. Each login must be recorded at the end of this file when it occurs. When UNIX starts up, `init()` creates a `getty()` process for each terminal, or some other comparable process in the case of network logins, and when a user logs in at a terminal, the `getty()` or other similar process spawns a `login()` process for that terminal. As a consequence, multiple `login()` processes might exist at any time. Imagine that two users login on different terminals at the exact same time. The two `login()` processes each have to add an entry to the end of the table. To add an entry to the end of a file, the file must be opened for writing, using `O_WRONLY`, or reading and writing, using `O_RDWR`. Thus, a process that needs to add a record to the end of the file must perform the following steps:

1. Open the file in read/write mode.
2. Seek to the end of the file.
3. Write a login record at the position obtained by the seek.

In terms of system calls, this would look something like the following:

```
fd = open(_PATH_WTMP, O_RDWR);
// error check the open() and if successful then
// create the wtmp record to write ... and then
lseek(fd, 0, SEEK_END);
write(fd, &record, len);
```

The `lseek()` call obtains the file pointer for the file and sets it to the end of the file at the time the call is made. This pointer is stored in the file structure for the this process's connection to the file. When the process calls `write()`, the data will be written at the position of this file pointer.

Imagine now that two different processes, identified here as `login1` and `login2`, execute this same sequence of instructions independently and simultaneously. For simplicity, suppose that they run on a one-processor machine and they are time-sliced onto the processor. In particular, suppose the processor executes the following sequence of instructions. The leftmost integer represents the current time in some fixed time unit. `login1` executes instructions in the left column; `login2`, in the right.

time	**login1**	**login2**
0	`fd = open(_PATH_WTMP, O_WRONLY);`	
1	`...`	
5	`lseek(fd, 0 SEEK_END);`	
6		`fd = open(_PATH_WTMP, O_WRONLY);`
7		`...`
10		`lseek(fd, 0 SEEK_END);`
11		`write(fd, &record, len);`
12	`write(fd, &record, len);`	

`login1` sets the file pointer at time 5. It is removed from the CPU and `login2` sets the file pointer to the same position as `login1` did, at time 10. It writes to the file at time 11. Then `login1` writes to the same position, overwriting the data just written by `login2`; the record written by `login1` replaces the one just written by `login2`. This is a classic example of a *race condition*.

Race conditions are removed by using some type of mechanism that will allow a sequence of instructions to be executed as an atomic operation, which simply means that once the first instruction is started, no other process can execute any instruction that accesses any of the shared data until the last instruction is completed. UNIX solves this particular problem by providing write connections with an optional *auto-append* mode. When auto-append mode (`O_APPEND`) is enabled, every write operation is preceded immediately and atomically by a seek to the end of the file. This guarantees that each write occurs at the end of the file, regardless of how many other processes are trying to do the same thing simultaneously. Therefore the login and logout processes simply have to open the file in auto-append mode, or enable it after the file is opened with `fcntl()`. The preceding code would be reduced to the following:

```
fd = open(_PATH_WTMP, O_WRONLY | O_APPEND);
// create the record to write ...
write(fd, &record, len);
```

4.2.3 Controlling the Connection When Opening a File

Rather than using `fcntl()` to adjust the attributes of the connection to a previously opened file, one can open the file with the desired attributes in the first place. These attributes can be passed as parameters in the `open()` system call, by bitwise-or-ing them in the second argument to the call. For example, to open a file with name `foobar` with the write-only, auto-append, and synchronous I/O bits set, you would write

```
fd = open(foobar, O_WRONLY | O_APPEND | O_SYNC);
```

You can read the `open()` man page or the `<fcntl.h>` header file for a list of the flags that can be set in the `open()` call. A file must be opened either read-only, write-only, or read-write. Therefore, the bitwise-or must always contain *exactly one* of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. Other flags can be bitwise-or-ed to it. Which flags are appropriate depend upon the mode in which it is opened. In general, flags fall into two categories: *file creation flags* and *file status flags*.

The file creation flags are `O_NOCTTY`, `O_TRUNC`, `O_CREAT`, and `O_EXCL`. These are only relevant when the file is opened in a mode that allows writing, either write-only or read-write. Understanding the `O_NOCTTY` flag requires more knowledge of terminals and processes; this will be explained in the chapter on processes. The semantics of the remaining three are worthy of discussion.

`O_CREAT` When no flags are set, if a file is opened for writing and it does not exist, `open()` will return -1 and fail. The `O_CREAT` flag prevents the failure: when it is set, if a file is opened for writing and it does not exist, `open()` will create it. The ownership of the file is determined by the effective userid of the process, and the file's mode will be whatever mode is specified in the `open()` call, with the umask applied. If the file already exists, this flag has no effect, meaning that the file will be opened with the file pointer set to the start of the file.

`O_EXCL` This flag is intended to be used in conjunction with the `O_CREAT` flag. If a file is opened for writing with the `O_CREAT` flag by itself, and the file exists already, the contents of the file can be overwritten, depending on where the writes occur. To prevent this possibility, the `O_EXCL` can be bitwise-or-ed to the flagset. When both `O_CREAT` and `O_EXCL` are set, if the file exists, the open will fail. In contrast, if the file does not exist, it will be created just as if `O_EXCL` were not set. If this flag is used without `O_CREAT`, the results are the same as if no flags were set.

`O_TRUNC` Without this flag, if a file is opened for writing and it already exists, the contents of the file are not necessarily destroyed; it depends whether `O_APPEND` is set and whether the process seeks to specific places in the file prior to writing. If the program just opens a file for writing and starts writing without seeking and without appending, the file contents will be replaced, but opening for writing does not automatically zero the contents of the file. The purpose of the `O_TRUNC` flag is to force the file to be zeroed before any writes. When it is set, and the `open()` allows writing and the file exists, the file is truncated to zero length. This flag will have no effect unless all of these conditions are true. It will also have no effect if the file is anything but a regular file. If all of `O_TRUNC, O_EXCL,` and `O_CREAT` are set, `O_EXCL` will override this one – if the file exists, `open()` will fail and it if does not exist, it will be created.

The table below summarizes the effects of the possible combinations of these three flags when a file is opened for writing using the call

```
open( "file", O_WRONLY | flags );
```

flags	If the file exists	If the file does not exist
0	opens for writing and sets pointer to first byte	fails
O_CREAT	opens for writing and sets pointer to first byte	creates "file"
O_EXCL	opens for writing and sets pointer to first byte	fails
O_TRUNC	opens for writing and zeroes its contents	fails
O_CREAT | O_EXCL	fails	creates "file"
O_CREAT | O_TRUNC	opens for writing and zeroes its contents	creates "file"
O_TRUNC | O_EXCL	opens for writing and zeroes its contents	fails
O_CREAT | O_TRUNC | O_EXCL	fails	creates "file"

The program `wrflagtest.c` in the demos directory for Chapter 5 can be used to test the effects of all combinations of these flags.

4.3 Device Files

Not only is every physical device in a UNIX system associated with a device special file, but every logical device is as well. Logical devices are devices that exist as abstractions of real physical devices[3]. Although UNIX does not require this, it has been the convention that all of the device files are located in the `/dev` directory.

4.3.1 Naming and Organizing Device Files

Different versions of UNIX use different methods of organizing and naming device files. For example, under Solaris 9, you will find that almost all of the terminal files contained there are symbolic links to files in the directory `/devices/pseudo`, and that the targets of these symbolic links have names such as `pts@0:2`.

In contrast, in Linux 2.6, instead of a `/devices/pseudo` directory there is a directory `/dev/pts`, and all of the device files within `/dev/pts` have small numbers such as 1,2,3, ... that correspond to the terminals of active connections[4].

[3]Sometimes, for example, there may be more than one name for a given physical device, such as a port or a disk. A logical device is another name for that device.

[4]At installation time, Linux can be configured so that its support of pseudo terminal devices is the same as that of systems such as Solaris 9 by using the `CONFIG_UNIX98_PTYS` and `CONFIG_DEVPTS_FS` flags. The Unix98 standard specifies how `/dev/pts` is used for pseudo-terminal support.

Hard disks are represented by device files as well. Their names vary from one system to another. In Linux, the names may look like `/dev/hd1a` or `/dev/hd2c`. If the hard disks are attached via a SCSI or SATA bus, then their device files may have names such as `/dev/sda1`. In Solaris 9, on the other hand, `/dev/dsk/c0t3d0s5` would be a device file for a file system attached to Controller 0 ("`c0`"), at target position 3 ("`t3`") for that controller, on logical unit 0 ("`d0`") since that controller may have several identical units differing only by number, and in partition or "slice" 5 ("`s5`") on that disk. Thus, the device file name reveals much information about the device.

4.3.2 Accessing Devices Via Device Files

All device files support the pertinent system calls. For example, a device such as a magnetic tape that can be read and written will have a device file whose name might be `/dev/rst0` and which will support the `open()`, `read()`, `write()`, `lseek()`, and `close()` system calls. Input-only devices such as mice and keyboards will not support `write()` or `lseek()` since it would make no sense, and similar common sense reasoning applies to all devices in general.

In previous chapters you saw how you could access your terminal using its device file. For example, to write a message to the terminal device `/dev/pts/4` you would write:

```
$echo "Where are you?" > /dev/pts/4
```

If you had permission to write to this terminal, the message "`Where are you?`" would appear on the screen in the corresponding terminal window.

If you do not know the name of the device file for the terminal in which your shell is running, the shell command `tty` will provide it to you. `tty` displays on standard output the absolute pathname of the device file representing the terminal from which the command is issued:

```
$ tty
/dev/pts/4
```

At the programming level, the library function `ttyname()` serves the same purpose, returning the full pathname of the terminal device whose file descriptor is passed to it.

```
NAME
       ttyname  - find pathname of a terminal
SYNOPSIS
       #include <unistd.h>
       char *ttyname(int fd);
DESCRIPTION
     The function ttyname() returns a pointer to the null-
     terminated pathname of the terminal device that is open on
     the file descriptor fd, or NULL on error (for example, if
     fd is not connected to a terminal). The return value may
     point to static data whose content is overwritten by the
     next call.
...
```

To find the name of the terminal device connected to standard input, for instance, a program can call `ttyname(0)`, The following program prints the name of its *control terminal*[5]. The macro `STDIN_FILENO` is defined in `<unistd.h>` and is simply the number 0.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Terminal is %s\n", ttyname(STDIN_FILENO));
    return 0;
}
```

If you name this program `show_tty` and run it, you will see something like:

```
$ show_tty
Terminal is /dev/pts/2
```

On the other hand, try running it as follows:

```
$ ls | show_tty
```

and you will see

```
Terminal is (null)
```

The problem is that in the second example, standard input was redirected, so file descriptor 0 was attached to a pipe instead. The `ttyname()` function returns the `NULL` string when there is no terminal attached to the descriptor. The same thing will happen if you try calling it with file descriptor 1 while standard output has been redirected. You can therefore use `ttyname(0)` as a test for whether or not standard input or output is redirected , as in

```
if ( ttyname(0) )
    // not redirected
else
    // is redirected
```

There is a function specifically for testing whether a file descriptor is attached to a terminal or not :

[5]The *control terminal* for a process is the terminal device from which keyboard-related signals may be generated. For example, if the user presses a Ctrl-C or Ctrl-D on terminal `/dev/pts/2`, all processes that have `/dev/pts/2` as their control terminal will receive this signal.

```
NAME
        isatty - does this descriptor refer to a terminal

SYNOPSIS
       #include <unistd.h>
       int isatty(int desc);

DESCRIPTION
     returns  1 if desc is an open descriptor connected to a
     terminal and 0 otherwise.
```

The `isatty()` function is useful for testing descriptors in this way. We can use it together with `ttyname()` as follows.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    if (isatty(0))
            printf("%s\n", ttyname(0));
        else
            printf("not a terminal\n");
    return 0;
}
```

If you run this program you will see something like

```
$ mytty
/dev/pts/1
```

and when input is redirected, you will see this:

```
$ ls | mytty
not a terminal
```

When you are writing a program that expects the standard output device to be a terminal, you can use either `isatty()` or `ttyname()` to check whether any standard stream has been redirected. However, both of these functions can only be used with an open file, since they need a file descriptor, and file descriptors exist only for open connections. In contrast, the `ctermid()` standard I/O library function will always display the pathname of the controlling terminal.

```
NAME
     ctermid  - generate path name for controlling terminal
SYNOPSIS
     #include <stdio.h>
     char *ctermid(char *s);
DESCRIPTION
    The ctermid() function generates the path name of the
    controlling terminal for the current process and stores it
    in a string. If s is a null pointer, the string is stored
    in an internal static area whose address is returned and
    whose contents are overwritten at the next call to ctermid().
        ...
BUGS
    The path returned may not uniquely identify the controlling
    terminal; it may, for example, be /dev/tty.
```

The problem is that on Linux, it will only display `/dev/tty`, just as the man page's `BUGS` section notes, so if the program needs the actual filename, using `ctermid()` is not the solution. It does work correctly in Solaris 9.

4.3.3 Device Drivers and the /dev Directory

Device files provide an interface between processes and devices. They are like regular files in the following ways: they have a mode, they have owners and groups, times of access, modification, and status change, and they have names and links to them. But they differ in one significant way. Unlike disk files, they are not storage containers; rather than storing data, they provide access to the entry points of functions. Because they are not containers, they do not have size. Because they are really just interfaces, they are associated with executable code that manages a connection to a device.

The code that manages the connection between a process and a device is inside a device driver. A device driver is a program that provides an interface between a device and the processes that communicate with it. Device files are the means in UNIX for a process to communicate with device drivers.

If you type `ls -l` in the `/dev` directory, you will see output such as this:

```
total 0
crw-rw----  1 root root     4,   0 Feb  6 11:07 tty0
crw-------  1 root root     4,   1 Feb  6 16:09 tty1
crw-rw----  1 root tty      4,  10 Feb  6 11:07 tty10
crw-rw----  1 root tty      4,  11 Feb  6 11:07 tty11
crw-rw----  1 root tty      4,  12 Feb  6 11:07 tty12
crw-rw----  1 root tty      4,  13 Feb  6 11:07 tty13
crw-rw----  1 root tty      4,  14 Feb  6 11:07 tty14
```

If you type `ls -l` in the `/dev/pts` directory, you will see something like this:

```
total 0
crw--w---- 1 root     tty 136,  1 Oct 14 14:46 1
crw--w---- 1 lsmarque tty 136, 10 Sep 12 13:13 10
crw--w---- 1 lsmarque tty 136, 11 Sep 12 18:39 11
crw--w---- 1 chays    tty 136, 12 Sep 13 20:02 12
crw--w---- 1 chays    tty 136, 13 Sep 13 20:02 13
crw--w---- 1 lsmarque tty 136, 14 Oct  3 13:22 14
crw--w---- 1 lsmarque tty 136, 15 Sep 12 13:13 15
crw--w---- 1 shixon   tty 136, 19 Oct 14 15:19 19
crw--w---- 1 sweiss   tty 136, 20 Oct 14 15:23 20
```

Notice that the first line, which always reports the total number of blocks used by the files in the directory, shows that these files do not use any storage.

Next observe that the type designator of each file listed above is '`c`', which represents the type character special file. The `c` indicates that this is a *character device*, as opposed to a block device. Every I/O device is accessed through either a block I/O interface or a character I/O interface. One major difference between them is that block I/O uses kernel buffering whereas character I/O does not. When a character I/O interface is used, the data flows in a character stream between the device and connected processes without using system buffers. In contrast, block devices use the kernel buffering system and transfer large chunks of data at a time. Some devices, such as disk partitions, may be accessed in block or character mode. Because each device file corresponds to a single access mode, devices that have more than one access mode have more than one device file.

Recall that the `write` command lets one user write to the terminal of another user, provided that the group write bit is set on the recipient's pseudo-terminal device file. As a warm-up exercise, we will eventually write an implementation of `write`. First we will examine how device drivers work.

Notice in the listings of the `/dev` and `/dev/pts` directories that the size field consists of a pair of numbers. The first number is called the *major device number* and the second, the *minor device number*. For example, `/dev/pts/12` has major device number 136 and minor device number 12. The major device number identifies the type of device, e.g., SCSI disk, pseudo-terminal, or mouse; the minor device number specifies which particular instance of this type of device is represented by the file, or the action associated with this particular interface to the device.

Each major device number is an index into the block or character device table maintained in the kernel. This table is used by the kernel to access the device drivers. Basically the table contains the address inside the kernel of the entry point into the device driver code. When a system call such as `read()` is invoked and the file descriptor passed to the `read()` system call is the file descriptor of a device file, the `read()` code (inside the kernel) finds the i-node belonging to that descriptor. The i-node contains the type of the file, which will indicate that it is a device special file, and whether it is block or character. The kernel will look up the major and minor device numbers from inside the i-node. It will then use the major device number to locate the device driver code for the particular kind of device. For example, if the device file has major number 136 and is a character device file, it will search the character device table for index 136. An attempt to access `/dev/pts/12` with major number 136 and minor number 12 will therefore result in execution of the driver whose index is 136[6]. There is usually a file in the file system that contains the mapping of major and minor device

[6]This is true in BSD and SunOs at least. In the Ext2 file system of Linux, there is more indirection, and the code that is executed might reside in a separate module. The device drivers are not necessarily part of a large executable image, but are instead in separate executable files, more like Windows, and the kernel contains stubs that are resolved dynamically at the time of the call, depending upon the type of file system on which the i-node resides.

numbers to devices. On many Linux systems, you can often find this mapping at

```
/usr/share/doc/kernel-doc-<kernel-version>/Documentation/devices.txt
```

where <kernel-version> is to be replaced by the version number, such as 2.6.9, as in

```
/usr/share/doc/kernel-doc-2.6.9/Documentation/devices.txt
```

If it is not there, then you can read the file `/proc/devices`, which contains the major device numbers for the various groups of devices.

Every I/O device with the same major device number uses the same driver. The minor device number is passed as a parameter to the device driver. The driver may use this number to select which unit of the device to use. Notice, for instance, that each of the pseudo-terminals in use in the above listing has the same major device number, 136, and a minor device number that corresponds to the name of the device file. As another example, if there are multiple disks, the minor device number would specify which disk is being accessed.

Sometimes the minor device number is used to distinguish different actions for the device to take. It is up to the device driver to decide how it will use the minor device number. The use of device numbers as indices into an array of drivers simplifies the customization of each system for the particular hardware configuration.

The device drivers in BSD UNIX, like the kernel itself, are divided into three sections:

1. Initialization and configuration routines
2. Routines to handle I/O service requests (upper half of the driver)
3. Interrupt service routines (lower half of the driver)

The lower half runs in an uninterruptible mode. The upper half runs synchronously and can block itself. They communicate through work queues. In Linux, although the terms "upper half" and "lower half" are not used explicitly, the device drivers work in a similar way. Sometimes they are called "master" and "slave" drivers.

4.3.4 Disk Drivers

An especially important group of drivers are the disk drivers. A disk driver is a software module that manages and controls the I/O that passes between a disk file and processes that have requested I/O to or from that file. This section briefly describes the role of disk drivers in handling I/O.

The purpose of a disk driver is to process requests for disk I/O. These requests are represented by transaction records, each of which consists of

1. a flag indicating whether to read or write data,
2. a primary memory address,
3. a secondary memory address, and

4. a count of the number of bytes to transfer.

The driver maintains these records in a queue. An I/O request causes the upper half of the driver to run. When the upper half is entered, it creates a transaction record and puts it into its work queue. It usually sorts the requests to reduce the latency for the particular device. (This is device dependent code designed to minimize seek time to particular cylinders. In 4.4BSD, for example, the elevator algorithm is used to sort the requests.)

When an I/O request has been satisfied or some other event occurs that warrants intervention by the kernel, the disk sends an interrupt to the processor. The interrupt service routine handles these interrupts; after saving the appropriate portion of the processor state, if the event was an I/O completion, it searches the queue to find the transaction that was completed, de-queues it, changes the state of the requesting process to indicate that it is no longer blocked on that I/O, and selects the next record to service.

Disk drivers are general enough that they can implement block I/O devices, character I/O devices, and swapping devices. To implement block I/O, the block I/O interface passes the address of a system buffer in field (2). To implement character I/O, the character I/O interface passes in field (2) an address that is in the user area, and it ensures that the process is not swapped out during the transfer. To implement a swapping device, the driver is tailored to make requests to the swapping device's controller.

4.3.5 Pseudo-Terminals

A *terminal* is a hardware device that emulates the old Teletype machines. Up until the early 1980's, most users connected to a mainframe computer through terminals. The terminals were connected to the computer via RS-232 lines, into *terminal multiplexers*, which were special-purpose devices that multiplexed multiple terminal lines into the computer[7]. The computer had device drivers whose role was to communicate with these multiplexed terminals. The terminal driver had to control all aspects of the communication path, including modem control, hardware flow control, echoing of characters, buffering of characters, and so on. When microprocessors were invented and personal computers became affordable, personal computers replaced terminals as the front ends to mainframes. Special terminal emulation software could be installed on a personal computer to make it appear to be a "dumb terminal" to the mainframe. One of the earliest such programs was Kermit, developed at Columbia University. Terminal emulation software interposed a terminal interface between the user on his or her local computer and a remote computer on a local area network or the Internet.

Whether you are working on a Linux machine or some other UNIX system locally or remotely via some remote login facility such as Telnet or SSH, if you are using the traditional command-line interface to UNIX, you are using a *pseudo-terminal*. A pseudo-terminal is a software-emulated terminal. When you open a terminal window in a desktop environment such as Gnome or KDE, you are using a pseudo-terminal. When you connect via an SSH client you are using a pseudo-terminal. The device files in the `/dev` directory that have names of the form `pts*` or `pty*` are pseudo-terminal device files. The device drivers for these files manage pseudo-terminals. Pseudo-terminals and how they work are described in more detail later.

[7]Terminals were just one of a class of devices called serial devices. RS-232 lines were serial lines, on which characters were sent one bit at a time. Modern UNIX systems, including Linux, continue to support many different types of serial lines.

4.3.6 Character I/O Interfaces

Almost all I/O devices have a character I/O interface, even if they are block devices. For example, the hard disk has a character interface to allow reads and writes of unstructured byte streams. The character interface is needed for programs such as `fsck`, which performs checks of file system integrity. The character interface translates these requests into block requests for the hardware but presents a character stream to the client. Printers, and modems have character interfaces, and these would have entries such as `/dev/tty0a`, `/dev/lp0`, `/dev/cua0`. Even physical memory has a character interface to allow memory to be treated like a RAM file. The device file for memory is `/dev/mem`.

The character I/O interface to block devices such as disks or tapes is called the *raw interface* because it allows access to the device and ignores its structure. A disk partition such as `sda1` may have two device files in `/dev`: the block interface and the character interface. The block interface is usually `/dev/sda1` in Linux but because individual partitions do not have character interfaces and only disks do, the character interface will be the "SCSI generic" device `/dev/sg1`. The `sg` interface allows byte by byte access to the entire drive. In Solaris 9, raw device names are of the form `/dev/rdsk/c0t3d0s5`.

Character device drivers do not use system buffers, except for terminal drivers, which use a linked list of very small (typically 64 byte) buffers. Character device drivers transfer characters directly to or from the user process's virtual address space. Because the transfers are directly to user memory and use DMA, the drivers must lock the memory to prevent the physical pages from being replaced during the transfer.

4.3.7 Other Character Devices

Some devices have character interfaces even though they do not fit the byte stream model of I/O. An example is a high-speed graphics display, which has such a fast transfer rate that it requires buffers in its own address space to handle the volume of data. These devices would swamp a driver that passed one character at a time, and are handled as special cases in the kernel.

4.3.8 Writing to a Device File

As an exercise in writing to a device file, we will write our own, somewhat simplified, version of the `write` command. The `write` command writes messages to terminals. The `write` man page begins as follows:

```
NAME
        write - write to another user
SYNOPSIS
        write user [ terminal ]
DESCRIPTION
     Write allows you to communicate with other users, by copying
     lines from your terminal to theirs.
     When you run the write command, the user you are writing to gets
     a message of the form: :
             Message from yourname@yourhost on yourtty at hh:mm...
     Any further lines you enter will be copied to the specified
```

```
    user's terminal. If the other user wants to reply, they must run
    write as well.
    When you are done, type an end-of-file or interrupt character.
    The other user will see the message EOF indicating that the
    conversation is over.
...
    If the user you want to write to is logged in on more than one
    terminal, you can specify which terminal to write to by
    specifying the terminal name as the second operand to the write
    command.  Alternatively, you can let write select one of the
    terminals - it will pick the one with the shortest idle time.
    This is so that if the user is logged in at work and also dialed
    up from home, the message will go to the right place.
```

Our first version of `write` will ignore the optional line argument and will not try to pick the terminal with the smallest idle time. The main program follows. It calls two functions, `get_user_tty()` and `create_message()`, which follow thereafter.

```
int main( int argc, char *argv[] )
{
    int     fd;
    char    buf[BUFSIZ];
    char   *user_tty;
    char    eof[] = "EOF\n";

    if ( argc < 2 ){
        fprintf(stderr,"usage: write1 username\n");
        exit(1);
    }

    if ( ( user_tty = get_user_tty( argv[1] ) ) == NULL ) {
        fprintf(stderr, "User %s is not logged in.\n", argv[1]);
        return 1;
    }

    sprintf(buf, "/dev/%s", user_tty);
    fd = open( buf, O_WRONLY );
    if ( fd == -1 ){
        perror(buf); exit(1);
    }
    create_message(buf);

    if ( write(fd, buf, strlen(buf)) == -1 ) {
         perror("write");
         close(fd);
         exit(1);
    }
```

```
    while( fgets(buf, BUFSIZ, stdin) != NULL )
        if ( write(fd, buf, strlen(buf)) == -1 )
            break;
    write(fd, eof, strlen(eof));
    close( fd );
    return 0;
}
```

The most important part of the main program is the loop. The while-loop looks exactly like a loop to read from one file and write to another. The only difference is that it is using `fgets()` instead of a `read()` system call, and it "hard-wires" `stdin` into the code. The `fgets()` function reads from any `FILE` stream until it sees an end-of-line (EOL) or end-of-file (EOF) mark. Therefore, you can see that reading from the keyboard and writing to a terminal is essentially the same as reading from one file and writing to another.

The `get_user_tty()` function searches through the `utmp` file for entries that match the given login name, returning a pointer to a static string containing the line. This could have been allocated on the stack and later freed, but for demonstration purposes, this is easier.

```
char * get_user_tty( char *logname )
{
    static struct utmp utrec;
    int            utrec_size = sizeof(utrec);
    int            utmp_fd;
    int            namelen = sizeof( utrec.ut_name );
    char           *retval = NULL ;

    if ( (utmp_fd = open( UTMP_FILE, O_RDONLY )) == -1 )
        return NULL;

    while ( read( utmp_fd, &utrec, utrec_size) == utrec_size )
        if ( strncmp(logname, utrec.ut_name, namelen ) == 0 ) {
            retval = utrec.ut_line ;
            break;
        }

    close(utmp_fd);
    return retval;
}
```

The `create_message()` function constructs the message to display on the user's console. It uses the `getlogin()` function to get the real username of the owner of the calling process, the `gethostname()` function to get the hostname of the host on which the sender process is running, the `ttyname()` function to get the terminal device name of the controlling terminal of the sending process, and the `time()` function to get the current time, which it converts to a `struct tm` using `localtime()`.

```
void    create_message(char buf[] )
{
    char    *sender_tty,  *sender_name;
```

```
    char    sender_host[256];
    time_t  now;
    struct tm *timeval;

    sender_name  = getlogin();
    sender_tty   = ttyname(STDIN_FILENO);
    gethostname(sender_host, 256);
    time(&now);
    timeval      =  localtime(&now);
    sprintf(buf, "Message from %s@%s on %s at %2d:%02d:%02d ...\n",
            sender_name, sender_host, 5+sender_tty,
            timeval->tm_hour, timeval->tm_min, timeval->tm_sec);
}
```

This version is relatively easy to construct. The next version must allow the user to enter a terminal device name in the form "`pts/n`" and try to open that specific terminal for writing, if possible. If it fails it must return an error message. The changes to allow the optional terminal name are minor:

1. The main program has to check the command line arguments (not shown below).
2. The `get_user_tty()` function needs a second argument that it compares to the `ut_line` field whenever the `ut_name` field matches.

```
char * get_user_tty( char *logname, char *termname )
{
    static struct utmp utrec;
    int              utrec_size = sizeof(utrec);
    int              utmp_fd;
    int              namelen = sizeof( utrec.ut_name );
    char             *retval = NULL ;

    if ( (utmp_fd = open( UTMP_FILE, O_RDONLY )) == -1 )
        return NULL;

    /* look for a line where the user is logged in */
    while ( read( utmp_fd, &utrec, utrec_size) == utrec_size)
        if ( strncmp(logname, utrec.ut_name, namelen ) == 0 )
            if (( termname == NULL ) ||
               ( strncmp(termname, utrec.ut_line,
                           strlen(termname)) == 0 ))
            {
                retval = utrec.ut_line ;
                break;
            }
    close(utmp_fd);
    return retval;
}
```

This version of the `write` command still does not choose the terminal line with the smallest idle time, for a given username, when the terminal line is not specified. How would you go about finding

the terminal line that has been idle the least amount of time? (Hint: the least idle terminal line has been accessed most recently.)

In addition, if you experiment with the real `write` command, you will discover that this version is still lacking other functionality:

- It does not check that the sender's terminal does not have I/O redirected.
- It does not check whether the sender's messaging is turned on (which it must be for the write program to write back to it.)
- It does not check whether the receiver is actually logged in, and whether messaging is enabled on the receiver's line.
- It does not check whether the sender is trying to send to him or herself.

Although none of these checks are difficult, and this program is easy to write, in general, communicating through the terminal interface is much more complicated than this; remember that we failed to write a good version of the `more` command because we could not suppress echoing, we had trouble getting input accepted without the Enter key press, and we did not suppress the scrolling of the prompt.

4.4 Terminals and Terminal I/O

Terminal I/O is probably the messiest and most disorganized part of any operating system. This is partly due to the way that I/O interfaces have developed over the years, in an *ad hoc* fashion, responding to changes in hardware and user demands, partly due to the wide variety of I/O devices that have to be accommodated under the aegis of a single I/O system, and partly due to the general lack of standards that governed how terminal I/O should be handled.

In UNIX, part of the problem is the result of the rift between the BSD versions and System V versions of the operating system, which had very different sets of terminal I/O routines. The POSIX standard provided a unification of the two sets of interfaces, and modern systems provide POSIX compliant routines. However, there are still many parts of the I/O interface that are platform-specific.

If you understand how terminals work, then you will have a better understanding of which routines you need to use to achieve various objectives as well as how to use those routines. The kinds of questions that will be answered here include:

- Why is it that we have to press the `Enter` key in order for the typed characters to be received by a program, and is there a way to avoid this?
- Some program suppress the echoing of characters as they are typed. How can we do this?
- Some programs are able to time-out while waiting for user input. How does that happen?
- Some programs, such as `vi` and `emacs`, override the meaning of various control sequences such as Ctrl-D and Ctrl-C. How can we do that?
- Terminals have a fixed number of rows and columns. How can a program get that number dynamically and control how it wraps its output?

- Why is it that sometimes the backspace key erases characters and sometimes the delete key does, and sometimes neither does? How does the terminal erase?

Adding to the problem of understanding terminals and terminal I/O is the fact that the C Standard I/O Library adds another layer of complexity, including various forms of buffering. We have to figure out what the terminal does and what the library does.

4.4.1 An Experiment

We begin by rewriting the simple I/O program from Chapter 1 so that it does not use C `FILE` streams. This way, whatever we observe is independent of that library's semantics. Unlike the simple I/O program there, this uses the kernel's `read()` and `write()` system calls.

```
Listing copychars.c
#include  <string.h>
#include  <stdlib.h>
#include  <unistd.h>

int main( int argc, char *argv[] )
{
    char inbuf;
    char prompt[] =
           "Type any characters followed by the 'Enter' key. "
           "Use Ctrl-D to exit.\n";

    if ( -1 == write(1, prompt, strlen(prompt)) ) {
        write(1, "write failed\n", 13);
        exit(1);
    }

    while( read(0, &inbuf, 1) > 0  )
        write( 1, &inbuf, 1 );
    return 0;
}
```

Assume this has been compiled into an executable named `copychars`. The `copychars` program reads one character at a time and writes one character at a time until it receives an end-of-file indication, which is that `read()` returns 0. If the user types a `Ctrl-D`, provided that the terminal has the default settings, the `read()` on the standard input stream will return 0.

When you run this program you will notice that, even though the main loop reads a single character and immediately writes that character, nothing gets written on the screen until you press the `Enter` key. As we are not using the C Library's streams, we cannot attribute this behavior to the library's buffering. The terminal is responsible for this. Somehow the characters that we type are stored, but where, and how many can be stored before they are lost?

We could answer these questions by doing a bit of research in the manpages, but this time we will begin with an experimental answer. We can determine the maximum number of characters that the terminal buffers by supplying larger and larger sequences of characters before pressing the `Enter`

key and counting the number of characters output on the screen to make sure none were lost. We are not allowed to redirect input to `copychars` to facilitate this because redirecting input will imply that the terminal is not involved in supplying characters to the program, so we have to do this in a more tedious way. We do not have to type the characters one at a time, but can create files containing the required number of characters on a single line, and if our terminal emulator supports copy and paste, we can paste them into the terminal window and then press the `Enter` key. We are free to redirect output to the `wc` command to count the number of characters written to standard output.

We can create a file named `a_N` containing `N` consecutive 'a' characters with the command

```
$ ( for i in `seq 1 1 N` ; do echo -n 'a' ; done ) > a_N
```

where `N` is replaced by the desired number. Suppose we create such files of sizes 128, 256, 512, 1024, 2048, 4096, 8192 characters. We can open each file in a text editor, copy the sequence to the clipboard and paste it into the window after running the command

```
$ copychars | wc
```

We will see the following sequence of numbers output by `wc`

```
      2      13     198
      2      13     326
      2      13     582
      2      13    1094
      2      13    2118
      2      13    4165
      2      13    4165
```

As you can see, the maximum number of characters reported by `wc` is 4165. Subtract the length of the prompt and the following newline, 70 characters, and you see that 4095 characters seems to be the maximum size of the buffer on the system used for this experiment. We will see shortly that this number is much larger than the documentation states. Where these characters are stored cannot be answered experimentally. That question requires some research.

Try one other thing. Run `copychars`, but this time, use your backspace key to change some of the characters. When you press the `Enter` key this time, there is no trace of that backspace character. It is not part of the stream of characters that the program received. The characters seem to be stored in the buffer in such a way that we can use editing keys to remove them from the set of characters delivered to the program.

Before we go further, and as a sort of advance peek at what we are about to do, try running the program as follows. Type the commands as they appear below:

```
$ stty -icanon ; copychars
```

When you are finished observing what happens, type the command

```
$ stty icanon
```

What you should have observed was that the program behaved as the code suggests – that each time you typed a character, it was immediately echoed on the screen. The `stty` command allows us to control terminal characteristics. What we just did in part disabled buffering of input characters in the terminal. Repeat this and try deleting characters and you will see that editing seems to be disabled. We will return to this later.

Now we will look at a second example. The following program *does* use the C Standard I/O Library, and so we will need to understand how that library confounds the picture, but that will happen a bit later. In the program below, named `listchars.c`, `getchar()` is used to obtain characters typed on the keyboard and `printf()` is used to display each character as an actual character (a *glyph*) with its character code in the current encoding (ASCII).

```
#include <stdio.h>
int main(int argc  char* argv[] )
{
    int   ch;

    printf("Type any characters followed by the 'Enter' key.");
    printf(" Use Ctrl-D to exit.\n");

    while( ( ch = fgetc(stdin)) != EOF  )
        printf("char = '%c' code = %03d\n", ch, ch );
    return 0;
}
```

Notice that `ch` is declared as an `int`. This is because `fgetc()` returns an `int` rather than a `char`, because it returns `-1` to indicate `EOF`. In fact `EOF` has the value `-1`, which means that whatever variable is assigned the return value must be a `signed int`. The `char` type is `unsigned`, so we cannot use it. Notice too that `printf()` will display an integer whose value is a legal character value as a glyph if the `%c` format specification is used, and as an integer if the `%d` format specification is used, so that `printf()` can conveniently print the character code and the character itself.

When you run this program you will again see that the program does not display any characters until the `Enter` key is pressed. If we build it and name it `listchars`, and run it and type `abcde` followed by `Enter`, and `Ctrl-D` to quit, we will see the following output:

```
$ listchars
Type any characters followed by the 'Enter' key. Use Ctrl-D to exit.
abcde
char = 'a' code = 097
char = 'b' code = 098
char = 'c' code = 099
char = 'd' code = 100
char = 'e' code = 101
char = '
' code = 010
$
```

First, notice that no output took place until the `Enter` key was pressed. Once again, this means that the characters that were typed before the `Enter` key was pressed had to be stored in a buffer.

Second, unlike our `copychars` program, this lets us see the character codes of the characters that we enter. It seems that there was a character after the '`e`' but before the `^D`, and that this character caused a new line to appear on the screen and then caused "`code = 10`" to appear after it. But we pressed the `Enter` key after the '`e`'. The fact that the second quote is in the first column on the next line implies that two characters were transmitted: a *linefeed*, ASCII 10, was transmitted to the screen to advance output to the next line, and a *carriage return*, ASCII 13, was transmitted, which forced the next character to appear in column 1 of that line. The ASCII code that was printed is decimal 10, which is the linefeed character, also known as *newline*. Therefore, only one ASCII code was generated, so even though both a newline and a carriage return were sent to the screen, only a newline character was seen by the program. Therefore, our pressing the `Enter` key caused only a newline code to be sent to the program. Usually, when you type the `Enter` key, it causes both the linefeed and the carriage return to be inserted in a document, but here you can see that only the linefeed is transmitted. (If you do not remember your ASCII codes, type `man ascii` for the listing.)

Third, in the `printf()` statement, the program sent a "`\n`" (newline) character to the terminal, but this seems to cause both a linefeed and a carriage return to be placed on the terminal. The linefeed character, by definition, should cause the cursor to move down to the next line in the same "column" on the screen, whereas a carriage return should move the cursor to column one, the left margin, without moving it down one line. But somehow both things happen even though a single character is transmitted.

What we can conclude from this is that the terminal driver must be doing some character processing on the inputs it receives from the keyboard and on the outputs that the processor sends to the display device. (Remember that the terminal driver is controlling a logical input/output device.) We now explore what the terminal driver does and how we can control its behavior.

4.4.2 Terminal Devices: An Overview

The preceding experiment showed that the default input mode of a terminal includes assembling the input into lines, processing various special characters such as backspace, and delivering the input lines to the process after they have been processed. This mode of operation is called *canonical input mode*. Terminals can be operated in various non-canonical input modes as well. In a non-canonical mode, some part of this processing of input is turned off. Programs like `emacs`, `vi`, and `less` put the terminal into a non-canonical mode[8].

The behavior of a terminal device is controlled entirely by a piece of software called a *terminal driver*. A terminal driver is not the same thing as a *terminal device driver*. A terminal driver consists of two components:

1. a *terminal device driver*, and
2. a *line discipline*.

[8]System 7 and BSD systems supported three different input modes: cooked, raw, and cbreak. POSIX.1 does not support these, although many systems still provide support for them.

The terminal device driver is usually a part of the kernel and its main function is to transfer characters to and from the terminal device; it is the software that talks directly with the hardware at one end, and the line discipline at the other. The line discipline is the software that does the processing of input and output. It maintains an input queue and an output queue for the terminal. The relationship between these queues, the process using the terminal, and the terminal itself, is illustrated in Figure 4.3. The terminal driver is the sum of the parts in this figure; it is the combination of line discipline and device driver. Notice that

- When echoing of characters is on, characters are copied from the input queue to the output queue.
- The size of the input queue is `MAX_INPUT`. If characters are typed faster than they are removed for processing, and the queue fills, UNIX systems typically ring the bell character and discard any extra characters.

  In the experiment that we performed with the `copychars` program, we saw that the input queue was able to store 4095 characters, which would suggest that 4095 is the value of `MAX_INPUT`. `MAX_INPUT` is one of many system limits that are defined in `<limits.h>`. Its value can also be obtained by calling `pathconf(ttyname(0), _PC_MAX_INPUT)` within a program. If we do either, we will see that `MAX_INPUT` is 255. The documentation states that `MAX_INPUT` may be smaller than the actual value. In fact, the terminal driver is configured to allow 4095 characters to be queued, even though the system limit is 255.
- Even though the output queue is also finite, if a process tries to write to it faster than the driver can transfer characters to the device, the kernel will simply block the process until the queue has more room.

The figure does not display all of the data structures used by the line discipline. Another queue is not shown, the *canonical input queue*. This is the queue in which input characters are processed when the terminal is in canonical mode. The canonical processing center is part of the line discipline. The figure does not show the internal data structure that the line discipline uses to control the terminal. The kernel provides an interface to access this structure, and UNIX provides a command, `stty`, to access and modify the various attributes of the terminal that are stored in this structure. The name of the interface to this structure, in POSIX.1 compliant systems, is the `termios struct`[9].

Almost all of the terminal device characteristics that can be examined and changed are contained in this `termios` structure, which is defined in the header file `<termios.h>`. That structure is a collection of four flagsets, and an array of character codes:

```
struct termios {
    tcflag_t  c_iflag;    /* input flags */
    tcflag_t  c_oflag;    /* output flags */
    tcflag_t  c_cflag;    /* control flags */
    tcflag_t  c_lflag;    /* local flags */
    cc_t      c_cc[NCCS]; /* control characters */
};
```

[9] In System V, the structure used for controlling terminals was the `termio struct`, and its header file was the `termio.h` file. POSIX added an 's' to the name to distinguish the new structure from theirs. A single 's'!

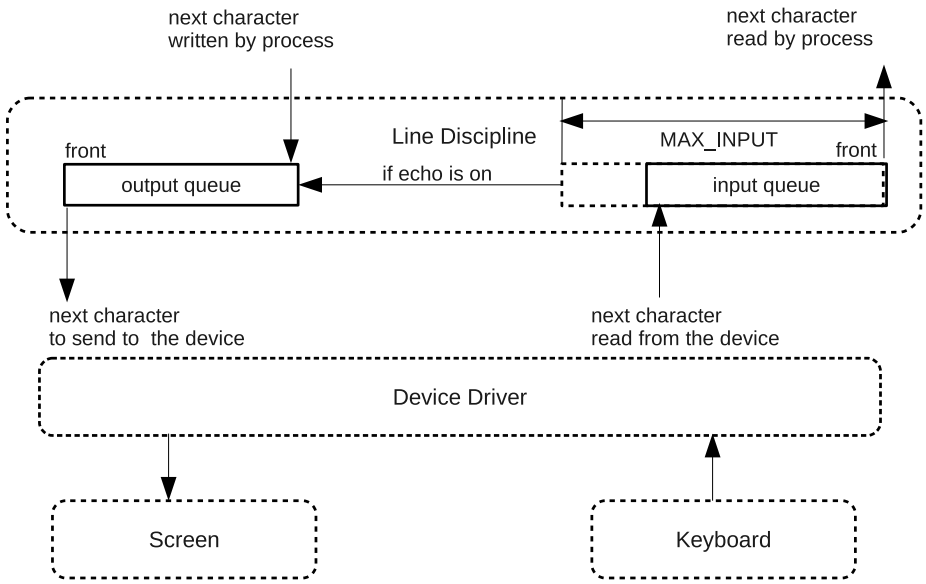


Figure 4.3: Terminal data structures

The type `tcflag_t` is an unsigned integer. The four flagsets are just four integers:

- The input flagset, `c_iflag`, controls the input of characters by the terminal device driver.
- The output flagset `c_oflag`, controls the driver output.
- The control flagset `c_cflag` controls the behavior of asynchronous serial transmission lines (such as RS-232); it may not have meaning for other kinds of terminal ports (such as pseudo-terminals).
- The local flagset `c_lflag` affects the interface between the driver and the user (echo, erase characters, etc.)

The `c_cc` array defines the special characters that can be changed. The array elements are of type `unsigned char` (`cc_t` is typedef-ed to `unsigned char`.)

Figure 4.3 shows the relationship between the terminal device and the terminal driver and its components. When you type on the keyboard, the character codes are transmitted to the device driver, which in turn passes them to the line discipline, placing them in its input queue. The line discipline, under the normal circumstances, copies the typed characters into the output queue, to be displayed on the console as you type. The line discipline is responsible for processing of input, including processing of special characters. It passes the processed characters to the kernel's `read()` routine[10], which passes them to the process that requested the read operation. When a process issues a write request, the kernel's write function passes the characters to the line discipline, which performs any processing that is supposed to be performed, and then puts the characters in the

[10]Technically, to the `sys_read()` function, which is the actual function within the kernel, as opposed to `read()`, which is a wrapper for it.

output queue. The device driver retrieves the characters from the front of the queue and displays them on the console.

Before we explore the `termios` structure at the programming level, we will look at how we can change it from the command level.

4.4.3 The `stty` Command

A user can use the `stty` command to view and alter terminal characteristics. `stty` without options or arguments displays a selection of the current settings of the terminal connected to the shell in which the command is invoked. Different systems will display different amounts of information. To see the complete list of terminal settings, use "`stty -a`". The output will look something like the following (rearranged and labeled):

```
$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
cchars:   intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
          eol = M-^?; eol2 = M-^?; start = ^Q; stop = ^S; susp = ^Z;
          rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
           min = 1; time = 0;
control flags: -parenb -parodd cs8 -hupcl -cstopb cread
               -clocal -crtscts
input flags:  -ignbrk -brkint -ignpar -parmrk -inpck -istrip
              -inlcr -igncr icrnl ixon -ixoff -iuclc ixany imaxbel
output flags:  opost -olcuc -ocrnl onlcr -onocr -onlret -ofill
              -ofdel nl0 cr0 tab0 bs0 vt0 ff0
local flags:   isig icanon iexten echo echoe echok -echonl -noflsh
              -xcase -tostop -echoprt echoctl echoke
```

It is important to understand that these settings are associated to the device file for the terminal, not to the connection between the shell or another process and the terminal. To convince yourself of this, perform the following experiment. To perform the experiment, we will play with the erase character. `stty` can be used to change the erase character used by the terminal, by typing the following, but don't do it yet:

```
$ stty erase x
```

where `x` is the character that you wish to use for erasing a single character on the command line. The default is usually either `Ctrl-H` or `Ctrl-?`, denoted `^H` and `^?` respectively. The backspace key sometimes generates `^H` and the delete key, `^?`, but not always. This experiment will not work in all shells, because some of them are designed to allow command line editing. In other words, the shell itself provides the functionality, overriding the terminal settings. If `bash` is your shell, then turn off command line editing by entering a non-editing `bash` subshell with the command line

```
$ bash --noediting
```

Then check which key is the erase character on your terminal by seeing which one actually erases, and then type

```
$ stty -a | egrep -o '\<erase = ...'
```

This will produce a line like

```
erase = ^?
```

Change the erase key to something else. For simplicity, make it the character 'X' and then verify that the change took place as follows:

```
$ stty erase X
$ stty -a | egrep -o '\<erase = ...'
erase = X;
```

Verify that it works by typing some random characters and using the `X` to erase them. Now invoke another new shell process by typing "`bash --noediting`" and then run

```
$ stty -a
```

in the new shell. You will see that the erase character is still `X`. Next, in the new shell, change the erase character back to the original, or to something else. To make it a control character, you can type the caret ('^') followed by the character.

```
$ stty erase ^H
$ stty -a | egrep -o '\<erase = ...'
erase = ^H;
```

Exit the current shell with the `exit` command and view the terminal settings again. You will see that the erase character is the one you set it to be in the shell that you just killed. This proves that you are really changing the device file settings, not those of the process's connection to it. It is not like the connection to a disk file. You can now exit the first `bash` subshell and check yet again that in your first shell the erase key is the last one you chose.

Finally note that if you have two different windows open, and one is pseudo-terminal `pts/3` and the other is `pts/4`, these may have different settings; the settings "stick" to devices (or pseudo-devices in this case), not processes.

Returning to the details of terminal settings, notice that there are two kinds of variables that are listed in the output of `stty -a`: those listed in the form

1. *var* = *value*; or *var value*; and
2. *var* or *-var*

The first type are non-Boolean variables. The second are Boolean. The Booleans are called *switches* or *flags*. I prefer the term *switch*, because they act like switches: a switch prefixed with a minus sign (-) is off, and a switch that is not prefixed with a minus sign is on. Examples of switches are `echo`, `inlcr`, `icanon`, and `icrnl`. To change the value of a switch you type

```
stty [-]switch
```

where `switch` is replaced by the name of the switch, and the minus sign is present to turn it off, absent to turn it on. Thus,

```
$ stty -echo
```

will turn off `echo` on the screen. Try it now but remember that to turn it back on you will have to type "`stty echo`" without being able to see your typing. An alternative is to open a second terminal window, and in that window, type `w` to see the which device files are associated with your two logins. Suppose that the first terminal, in which you turned off echoing, is `/dev/pts/2`. Then in the second terminal, you can type

```
$ stty --file=/dev/pts/2 echo
```

and this may turn `echo` on in the first terminal. (Remember that the `tty` command will display your terminal device filename.) In general, the `--file` option lets you specify an alternate device file for the `stty` command, provided you have permission to modify it. It may not always work, and not all systems support the `--file` option.

Non-Booleans are displayed in one of two forms

- variable = value;
- variable value;

For example, the line

```
speed 38400 baud; rows 24; columns 80; line = 0;
```

means that the baud rate is 38400, the number of rows in the terminal is 24 and the number of columns is 80. The "`line = 0`" refers to the line discipline. There are certain predefined combinations of settings that are collectively known as the line discipline. Many of these are derived from the early UNIX implementations of terminal devices; setting the line discipline to 0 means that the default behavior is the typical one wanted by most users.

The characteristics of the terminal that can be controlled via the `stty` command fall into six classes:

Class	Description
Special Characters	Characters that are used by the driver to cause specific actions to take place, such as sending signals to the process, or erasing characters or words or lines. Special characters include the `erase`, `werase`, and `kill` characters, as examples. Characters used to send signals include `Ctrl-C`, which sends the *interrupt* signal, and `Ctrl-\`, which sends the *quit* signal. Signals are covered in a later chapter, but for now it suffices to know that signals such as the interrupt and quit signal usually terminate the process that receives them.
Special Settings	Variables that control the terminal in general, such as its input and output speeds and dimensions. These include the `rows`, `cols`, `min`, and `time` values. `rows` and `cols` are the numbers of rows and columns in the window. `min` and `time` are used when the terminal is in non-canonical mode to control how characters are returned by `read()` calls; these will be discussed later. Many of these variables do not apply to pseudo-terminals.
Input Settings	Operations that process characters coming from the terminal. This includes changing their case, converting carriage returns to newlines, and ignoring various characters like breaks and carriage returns.
Output Settings	Operations that process characters sent to the terminal. Output operations include replacing tab characters by the appropriate number of spaces, converting newlines to carriage returns, carriage returns to newlines, and changing case.
Control Settings	Operations that control character representation such as parity and stop bits, hardware flow control. Several of these do not apply to pseudo-terminals.
Local Settings	Operations that control how the driver stores and processes characters internally. For example, echo is a local operation, as is processing erase and line-kill characters.
Combination Settings	Combinations of various settings that define modes such as cooked mode or raw mode.

Input settings start with 'i' and output settings start with 'o'. One input setting of interest is `icrnl`. This is the switch that, when set, causes input carriage returns to be converted to newline characters. Read it as `i` for input, `cr` for carriage return, `nl` for newline. This switch explains why the `Enter` key is converted to a newline character. Try running the following:

```
$ stty -icrnl ; listchars
```

and typing 'abc' followed by `Enter` then `Ctrl-D`. The `Enter` will appear as a `^M` on the screen. Then type `Ctrl-C` to quit the program. You will see that the code listed for the carriage return is now ASCII 13. You will also see that the word "`char`" is not displayed on the screen for that line:

```
abc^Mchar = 'a' code = 097
char = 'b' code = 098
char = 'c' code = 099
' code = 013
```

This is because the carriage return character causes the output to start in column 1 but there was no linefeed, so the output " code 13" is over-writing the "char = ". To prove this, modify the program to force a newline just before the '%c' in the `printf()` call. You will see the word "char" reappear. Restore your settings by typing

```
$ stty icrnl
```

The output setting of interest is the `onlcr` switch. This is what adds a carriage return to each newline character sent to the terminal. Try turning it off and looking at the output:

```
$ stty -onlcr ; listchars
[stewart@harpo chapter05]$ stty -onlcr; listchars
Type any characters followed by the 'Enter' key; Ctrl-D to exit.
                                                                    abc
                                                                       c
       har = 'a' code = 097
                           char = 'b' code = 098
                                                 char = 'c' code = 099
                                                                     char
       = '
           ' code = 010
```

Notice that each new line starts just below the end of the previous line; there were no carriage returns inserted into the output stream. To restore your settings, type

```
$ reset
```

or

```
$ stty onlcr
```

Local settings include whether or not canonical mode and echo are enabled. By default, the terminal is in canonical mode. The output of the `stty` command indicates this:

```
$ stty -a | grep canon
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

In canonical mode, the driver processes the line editing keys:

- *character erase* (`erase`),
- *word-erase* (`werase`),
- *line-kil*l (`kill`), and
- *line redraw* (`rprnt`)

and recognizes the following special input characters[11]: *line delimiter* (`eol` and `eol2`) and *end-of-file* (`eof`). The `eol` and `eol2` characters are equivalent. It also buffers the input characters until the `Enter` key is pressed, at which point it delivers them to the process. Shells that support line editing natively (by the shell itself) such as `bash` and `tcsh` ignore the `stty` command to turn off canonical mode because they process the characters themselves. Canonical mode is turned off, but within the shell, there will be no effect. To see the effect of disabling canonical mode, you have to run `bash` with the `--noediting` command line switch.

In canonical mode, typed characters are processed and placed into the canonical input queue. In non-canonical mode, they are delivered to the `read()` system call directly. We will demonstrate this now. We will run a simple program similar to the `copychars` program but, just to make it different, it will display not the typed character, but its transpose, defined here by *transpose(c) = islowercase(c)? 'a' + 'z' - c:c*. The program is therefore

```
Listing transpose.c
#include  <unistd.h>
#include   <ctype.h>

int main(int argc, char* argv[])
{
    char ch;
    while ( read(0, &ch, 1) > 0  ) {
        if (islower(ch))
            ch = 'a' + 'z' - ch;
        write( 1, &ch, 1 );
    }
    return 0;
}
```

We turn off canonical mode with the command, `stty -icanon`, and run the `transpose` program in non-canonical mode. In my terminal, the erase character is the Backspace key, which echos as "^?" (control-?). As soon as canonical mode is disabled, it will be hard to correct typing mistakes, so it is easier to turn off canonical mode and issue the `transpose` command as a single job. The interaction is as follows:

```
$ stty -icanon; transpose
azbycxdw^? ev

^C
```

For each character that I type, the transpose is printed immediately after, because the character is given to the `transpose` program as soon as it is typed. This shows that when canonical mode is off, there is no input buffering. Also, when I try to erase using the Backspace key, the character pair "^?" appears on the screen but nothing is erased. Following it is a non-printing character. The character code for the Backspace key is something that cannot be printed, so some type of symbol

[11] There are many more special input characters than these. These are just the ones that are processed in canonical mode.

will appear after the "`^?`", which is what is echoed for Backspace. If you try entering the line-kill character, `Ctrl-U`, it will not erase the line. Nor will `Ctrl-D` send an `EOF` signal. The program must be killed with `Ctrl-C`. Furthermore, when I typed the `Enter` key, that was echoed and then printed again, so two newlines were displayed. In a later chapter, we will explore other effects of disabling canonical mode.

Terminals are complex structures. Most of their attributes should not be changed unless you really understand how they are used. Some are communication characteristics, such as parity, start and stop bits, handshake methods, hang-up methods, and so on. These are often not relevant to your terminal settings; they are part of the `stty` command because it is also a general command for controlling serial devices and real tty's, for which they are necessary. You can read the man pages for more detailed explanations of all of these attributes.

4.4.4 Programming the Terminal Driver

The `stty` command allows you to modify terminal settings from the shell, but it cannot be used from within a program. To control most of the terminal characteristics from within a program, we can use the pair of system calls, `tcgetattr()` and `tcsetattr()`. These get and set driver attributes respectively. There is an alternative function, `ioctl()`, that can be used for controlling terminal settings, but it is not preferred, because it is not supported by the standard. We will look at `ioctl()` later. The `ioctl()` function is necessary for controlling devices other than terminals.

`tcgetattr()` and `tcsetattr()` are not the only functions that operate on the terminal settings. There are actually 13 different functions in the POSIX standard. The remaining ones are

`cfgetispeed()`	gets input speed
`cfgetospeed()`	gets output speed
`cfsetispeed()`	sets input speed
`cfsetospeed()`	sets output speed
`tcdrain()`	waits for all output to be transmitted
`tcflow()`	suspends transmission
`tcflush()`	flushes input and/or output queues
`tcsendbreak()`	sends a break character
`tcgetpgrp()`	gets foreground process groupid
`tcsetpgrp()`	sets foreground process groupid
`tcgetsid()`	gets process group ID of session leader for control of tty

Some of these act on the line discipline; others act on the device driver settings. We will only explore a few of these.

Whereas `fcntl()` allows you to control disk file connections, these allow you to program terminal driver attributes. Unlike `fcntl()`, which can be used for both getting and setting attributes, the work of getting terminal attributes is in one function, and setting, in another. The method of making changes is the same though; you have to

- retrieve the current settings into a structure in the process's address space using `tcgetattr()`,
- modify that structure locally, and
- write it back to the driver using the `tcsetattr()` call.

The attributes are passed back and forth in a `termios` structure. There is a single man page for `termios`, `tcgetattr()`, and `tcsetattr()` and all of the other functions listed above except the last three. The following excerpts the relevant information:

```
NAME
     termios,  tcgetattr, tcsetattr, tcsendbreak, tcdrain,
     tcflush,tcflow, cfmakeraw, cfgetospeed, cfgetispeed,
     cfsetispeed, cfsetospeed
     get and set terminal attributes,
     line control,get and set baud rate
SYNOPSIS
     #include <termios.h>
     #include <unistd.h>
     int tcgetattr(int fd, struct termios *termios_p);
     int tcsetattr(int fd, int optional_actions,
                    struct termios *termios_p);
          ...
DESCRIPTION
The termios functions describe a general terminal interface
that is provided to control asynchronous communications ports.

Many of the functions described here have a termios_p argument
that is a pointer to a termios structure. This structure
contains at least the following members:
               tcflag_t c_iflag;      /* input modes */
               tcflag_t c_oflag;      /* output modes */
               tcflag_t c_cflag;      /* control modes */
               tcflag_t c_lflag;      /* local modes */
               cc_t c_cc[NCCS];       /* control chars */
      ...
```

The `tcgetattr()` system call is given a file descriptor and a pointer to a `termios` structure. The file descriptor must refer to a terminal device file otherwise it is an error[12]. The `tcgetattr()` call will store the attributes of the referenced terminal device in the `termios` structure. The `tcsetattr()` system call is also given a file descriptor and a pointer to a `termios` structure, but its second parameter is a set of optional actions. These optional actions specify when to apply the changes to the terminal device. There are three possible values for this parameter. From the man page (with my corrections):

`TCSANOW` The change occurs immediately.

`TCSADRAIN` The change occurs after all output written to `fd` has been transmitted. This function should be used when changing parameters that affect output.

`TCSAFLUSH` The change occurs after all output written to the object referred to by `fd` has been transmitted; furthermore, before the change takes place, all input data that has not been read is discarded.

[12]This is yet another way to test whether a particular file descriptor points to a terminal device, along with `isatty()` and `ttyname()`.

`TCSANOW` forces the change immediately; this can cause problems if the terminal driver is writing to the terminal and the changes modify the output flags. `TCSADRAIN` forces the changes, but only after the output queue has been emptied by the driver. This is the action that should be chosen whenever the changes affect output to the terminal. `TCSAFLUSH` forces the changes only after the output queues are emptied and after it causes the input data sitting in the queue to be discarded. It is safest, when restoring the terminal to its original state, to use `TCSAFLUSH`.

Different versions of UNIX have different definitions of the `termios` structure. The definition found in Solaris 9 complies with the Single UNIX Specification, Version 3, of the Open Group[13] (also known as IEEE Std 1003.1), the most generally accepted UNIX standard. The definition found in the `/usr/include/bits/termios.h` header file on Linux 2.6 contains other fields (which is allowed by POSIX):

```
struct termios
{
    tcflag_t c_iflag;           /* input mode flags */
    tcflag_t c_oflag;           /* output mode flags */
    tcflag_t c_cflag;           /* control mode flags */
    tcflag_t c_lflag;           /* local mode flags */
    cc_t c_line;                /* line discipline */
    cc_t c_cc[NCCS];            /* control characters */
    speed_t c_ispeed;           /* input speed */
    speed_t c_ospeed;           /* output speed */
}
```

The `c_line`, `c_ispeed`, and `c_ospeed` members are not part of the standard, which allows the structure to contain additional members. Although `tcflag_t` is an integer whose individual bits are flags, you do not need to know the individual bit positions of the flags, because the header file defines masks for each one. The order of the bits may vary from one implementation to another. POSIX specifies one set of flags, XOpen another, Open Source yet another, and so on.

The flagsets are illustrated in Figure 4.4 Each box is a single integer flagset. The names inside each are the names of the individual bit masks. Each of these flags is described in the man page for `termios`. The `c_iflag` member contains flags that define input processing. The `c_oflag` member contains flags that define output processing. The `c_cflag` has flags that define control characteristics, and the `c_lflag` member has flags that define how characters are processed locally, i.e., internally in the driver. The `c_cc` array is an array that stores control character assignments. This is where the map of erase key, backspace key, and so on, is stored. POSIX requires that the following subscript names must exist:

[13] You can download version 3, the latest version, from http://www.unix.org/version3.

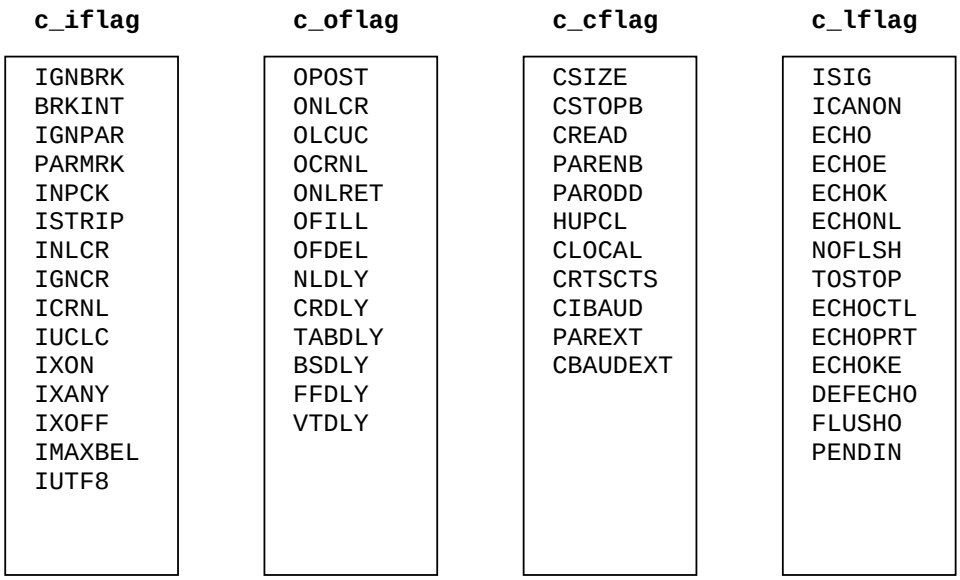


Figure 4.4: Flagset bitmasks of the `termios` struct

Canonical Mode	**Non-Canonical Mode**	**Description**
VEOF	EOF	character
VEOL	EOL	character
VERASE	ERASE	character
VINTR	VINTR	INTR character
VKILL	KILL	character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value

The size of the `c_cc` array is defined to be `NCCS`. We will cover this array in a later chapter.

You can change individual bits in the flagsets with bitwise operations. If `MASK` represents an arbitrary bit mask, then the following test, set, and clear the bits in a flagset:

```
if ( flagset & MASK ) ... //tests the masked bit
flagset |= MASK           //sets the masked bit
flagset &= ~MASK          //clears the masked bit
```

For example, to turn off terminal echo, assuming `flagset` is a local copy of the `c_lflags` flagset from the `termios` structure, we would use

```
flagset = flagset & ~ECHO;
```

Following is a program that prompts the user to enter a password and makes the typing invisible when the password is entered, as the `login` program does. The program turns off the echo switch and resets it afterward.

```
Listing login.c
#include   <stdio.h>
#include   <stdlib.h>
#include   <termios.h>

/*************************************************************************
                                Main Program
*************************************************************************/

int main(int argc, char* argv[] )
{
    struct termios info, orig;
    char username[33];
    char passwd[33];
    FILE   *fp;

    /* get a FILE* to the control terminal --   don't assume stdin  */
    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
         return(1);

    printf("login: ");                    /* display message          */
    fgets(username, 32, stdin);           /* get user's typing        */

    /* Now turn off echo */
    tcgetattr(fileno(fp), &info);         /* Get current terminal state */
    orig = info;                          /* Save a copy of it */
    info.c_lflag &= ~ECHO ;               /* Turn off echo bit   */
    tcsetattr(fileno(fp),TCSANOW,&info); /* Use this state in line discipline */

    printf("password: ");
    fgets( passwd, 32, stdin);            /* Get user's non-echoed typing
*/
    tcsetattr(fileno(fp),TCSANOW,&orig); /* Restore saved settings
*/

    printf("\n");                         /* Print a fake message
*/
    printf("Last login: Tue Apr 31 21:29:54 2088 from the twilight zone.\n");
    return 0;
}
```

Comments

- This is the standard procedure: use the `tcgetattr()` to get the old state, save a copy, make the change, set it with `tcsetattr()`, do the work, and then restore.
- We read from and write to only the control terminal and return an error if we can't open this device for reading and writing. We can get the name of the control terminal with `ctermid()`, and get the file descriptor from the name with the `fileno()` function. This is more secure.
- We don't use `"/dev/tty"` ; it is not as reliable as calling `ctermid()`.

4.4.5 Implementing `stty`: `showtty`

Implementing a user-friendly, perhaps more verbose, version of `stty` is a good exercise in using the `termios` structure. We will write a POSIX compliant version, so it should display the correct values of any flags that it does attempt to display.

The basic idea is very straightforward: `tcgetattr()` is used to retrieve the settings of the flagsets. The `show_flagset()` function prints the status of each flag in the four flagsets and the `show_cc_array()` function prints the mapping of control characters to character codes. Their designs are similar and are based on the use of an array of structures that contain a value and a textual description or name associated with that value:

```
/* Mapping of flag bit position to name */
typedef struct _flaginfo  flaginfo;
struct _flaginfo
{
    int   fl_value;   // flag value
    char  *fl_name;   // string describing flag
};

/* Mapping of index to description */
typedef struct _cc_entry  cc_entry;
struct _cc_entry {
    int  index;
    char *description;
};
```

A single flagset is then represented by an array of `flaginfo` structures, and the array of control character codes is represented in a similar way by an array of `cc_entry` structures. For example, the input flags are defined by the constant array

```
flaginfo input_flags[] = {
    {IGNBRK  , "Ignore break condition" },
    {BRKINT  , "Signal interrupt on break" },
    {IGNPAR  , "Ignore chars with parity errors" },
    {PARMRK  , "Mark parity errors" },
    {INPCK   , "Enable input parity check" },
    {ISTRIP  , "Strip character" },
    {INLCR   , "Map NL to CR on input" },
    {IGNCR   , "Ignore CR" },
    {ICRNL   , "Map CR to NL on input" },
    {IXON    , "Enable start/stop output control" },
    {IXOFF   , "Enable start/stop input control" },
    {-1      ,  NULL }
};
```

and the control characters, by the constant array

```
cc_entry  cc_values[] = {
    { VEOF,     "The EOF         character"},
    { VEOL,     "The EOL         character"},
    { VERASE,   "The ERASE character"},
#if defined VWERASE
    { VWERASE,  "The WERASE character"},
#endif
    { VINTR, "The INTR character"},
    { VKILL,    "The KILL character"},
    { VQUIT,    "The QUIT character"},
    { VSTART,   "The START character"},
    { VSTOP,    "The STOP character"},
    { VSUSP,    "The SUSP character"},
    { VMIN,     "The MIN value "},
    { VTIME,    "The TIME value"},
    { -1,      NULL}
};
```

Notice that each array is terminated by a sentinel value -1. Notice too that the `VWERASE` code is only conditionally compiled, because POSIX does not define it. The logic in processing either array is similar. The function to display the character codes for the `c_cc` array is

```
void show_cc_array( struct termios ttyinfo,
                    cc_entry       controlchars[] )
{
    int  i = 0;

    while ( -1 != controlchars[i].index ) {
        printf( "%s is ", controlchars[i].description);
        printf( "Cntl-%c\n",
                ttyinfo.c_cc[controlchars[i].index] -1 +'A');
        i++;
    }
}
```

For each character code in the array whose index is not -1, it prints the description from the structure and the value of the code as a "control-something" name. These "control-something" names, e.g., "`Ctrl-A`", "`Ctrl-B`", and so on, through "`Ctrl-_`", derived from the fact that on a keyboard, it is often possible to generate a control code using the control (Ctrl) key and a normal key. By adding the code for 'A', less one, to the index stored in the `c_cc` array, we get the keyboard key that should be typed to enter that code from the keyboard.

The `show_flagset()` function is similar. It is called by a function, `show_flags()`, that simply calls `show_flagset()` for each different flagset.

```
void show_flagset( int flag, flaginfo bitnames[] )
{
```

```
    int  i = 0;

    while (-1 != bitnames[i].fl_value ) {
        printf( "%s is ", bitnames[i].fl_name);
        if ( flag & bitnames[i].fl_value )
                printf("ON\n");
        else
                printf("OFF\n");
        i++;
    }
}
```

The main program calls a function, `show_baud_rate()`, that when given the `termios` structure `ttyinfo`, converts the symbolical values of the baud rate into numerical values. The `show_baud_rate()` function calls `cfgetospeed()` to extract the baud rate from the structure. The main program is below.

```
int main(int argc, char* argv[])
{
    struct      termios ttyinfo;
    FILE   *fp;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL )
        exit (1);
    if ( tcgetattr( fileno(fp) , &ttyinfo ) == -1 ){
        perror( "Cannot get info about this terminal.");
        exit (1);
    }

    show_baud_rate (ttyinfo);
    printf("\n");
    show_cc_array(ttyinfo, cc_values);
    printf("\n");
    show_flags(ttyinfo, input_flags, output_flags, local_flags);
    return 0;
}
```

4.4.6 Non-Canonical Terminal Modes

Recall that in canonical mode, the terminal driver assembles input into lines, interpreting various special characters such as backspace, and then delivering it to the process. Above we noted that in non-canonical mode, input buffering is turned off and typed characters are delivered immediately to the reading process. Here we will explore the features of the line discipline that allow us to control more precisely how many characters are delivered to the process and when they are delivered. We will use the `transpose` program for demonstrating this.

When canonical mode is turned off, the `min` option to `stty` controls the minimum number of characters that must be typed before they are transmitted to the program. For example

```
$ stty -icanon min 4
```

puts the terminal into non-canonical mode and sets the minimum number of characters that must be typed before anything is transmitted to the program to 4. Try typing the following:

```
$ stty -icanon min 3; transpose
abczyxdefwvu
^C
```

Notice that three characters at a time are sent to the program, and that until you type three characters, none will be sent. This shows that although there is no buffering until an `EOL` is entered, a read operation is not considered to be complete unless the minimum number of characters is present in the driver.

Within the `termios` structure, the `MIN` member of the `c_cc` array (`c_cc[VMIN]`) specifies the minimum number of bytes before a `read()` call returns. `MIN` can be zero or positive. The `TIME` member of the `c_cc` array (`c_cc[VTIME]`) specifies the number of tenths of a second that a `read()` call must wait for data to arrive. It too can be zero or positive. There are thus four combinations of values for this pair of variables, each with different semantics. Assume in the following that the call to `read()` is

```
read(STDIN_FILENO, buf, numbytes_to_read);
```

Case 1. `MIN` $>$ 0, `TIME` $>$ 0

> The `read()` call will block (and hence the calling process will block) until it receives the first byte. `TIME` specifies an *inter-byte timer* that is activated only when the first byte is received. In other words, until the user types, the timer is inactive. After each byte is received, the timer is reset to 0. Once the user starts to type, if the lesser of `numbytes_to_read` and `MIN` bytes are received before the timer expires, `read()` returns that many bytes. Remember that the timer is reset after each byte. If the timer expires before `MIN` bytes are received, `read()` returns whatever bytes it received. (At least one byte is returned if the timer expires, because the timer is not started until the first byte is received.) If data is already available when `read()` is called, it is as if the data had been received immediately after the `read()`. This means that the `read()` can return immediately if enough data was in the input queue of the line discipline (inside the terminal driver).

Case 2. `MIN` $>$ 0, `TIME` $==$ 0

> The `read()` does not return until `MIN` bytes have been received. Setting `TIME` to zero means that the timer is turned off. It is given an infinite amount of time, in effect and so this can cause a `read()` to block indefinitely, waiting for input. The lesser of `numbytes_to_read` and `MIN` bytes are returned to the user process. This means that if `numbytes_to_read` $<$ `MIN`, then `read()` is not satisfied until `MIN` bytes are received, but it only gives `numbytes_to_read` to the caller. If `numbytes_toread` $>$ `MIN`, then `read()` returns when `MIN` bytes are received and `MIN` bytes are given to the caller.

Case 3. `MIN` == 0, `TIME` > 0

> Unlike Case 1, `TIME` specifies a timer that is started when `read()` is called. *It is not an inter-byte timer.* In this case, `read()` returns when a single byte is received or when the timer expires. If the timer expires, `read()` returns 0, otherwise it returns the number of bytes read. If there are characters in the terminal's input queue before the `read()` call is processed, the minimum of the number of characters in the queue and `numbytes_to_read` bytes will be delivered to `read()` immediately, satisfying the call, and `read()` will return the number of bytes actually read.

Case 4. `MIN` == 0, `TIME` == 0

> The minimum of either the number of bytes requested, i.e. `numbytes_to_read`, or the number of bytes currently available is returned to the caller without waiting for more bytes to be input. If no characters are available, `read()` returns 0, having read no data. Otherwise the number of bytes actually read is returned. In both cases, `read()` returns immediately.

To demonstrate, first, in non-canonical mode we will set `min` = 0 and `time` = 30. What you will see is that if a character is typed within 3 seconds (30 tenths of a second), it will be delivered immediately, but if no character is typed within 3 seconds, the read will return with an `EOF` character. Type some characters with 3 seconds each and then let 3 seconds elapse. You will see the program terminate.

```
$ stty -icanon min 0 time 30 ; transpose
```

You may also discover that your shell terminates. Before you do this, make sure that you set the `IGNOREEOF` environment variable so that you do not lose your shell. If you see repeated messages of the form "Use exit to logout" then turn off the terminal settings with `reset`.

Next, test the effect of setting `min` and `time` to be non-zero. Type the following command sequence:

```
$ stty -icanon min 4 time 30 ; transpose
```

Wait at least 3 seconds and notice that nothing happens. The program is waiting for input. There is no timeout and the read has not returned. Now type a single character and wait several seconds. You will see its transpose after 3 seconds, demonstrating that the read returned with only 1 character after a delay of 30 time units. Now type 4 characters and see how quickly the program prints their transposes. The driver is waiting 3 seconds for the minimum number of characters to be entered; if they are entered before 3 seconds, it returns, otherwise it returns after 3 seconds with as many characters as were entered. Kill the process with `Ctrl-C` and turn canonical mode back on.

These experiments demonstrate that with canonical mode off, you can control input precisely. So far we have been using the `stty` command to control the terminal driver. In a program you can use `tcgetattr()` and `tcsetattr()` instead. The following listing is of a program that allows you to test the effects of various combinations of terminal settings, giving specific values to the `MIN` and `TIME` parameters in non-canonical mode.

Listing canon_mode_test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define  SLEEPTIME   2

/**
   Gets  input under the mode specified by the values of iscanon, vmin, and
   vtime. See the notes or the termios man page for the meaning of the vmin
   and vtime parameters.
 */
void get_response( int iscanon, int vmin, int vtime);

/* Sets the VMIN value to vmin, VTIME to vtime, turns off icanon */
void set_non_canonical(int vmin, int vtime);

/* if how == 0 this saves the termios state for later restoring     */
/* if how == 1 this restores the termios state from the saved state */
/* CANNOT CALL with how == 1 before first calling with how == 0     */
void tty_mode(int how);

int main(int argc, char* argv[])
{
    int  bNoCanon = 1;  /* default is non-canonical mode */
    int  vmin     = 1;  /* default is one char */
    int  vtime    = 0;  /* default is to force reads to wait for vmin chars */
    int  ch;
    char optstring[] = ":hcm:t:";

    while (1) {
        ch = getopt(argc, argv, optstring);
        if ( -1 == ch )
            break;
        switch (ch) {
        case 'h' :
            printf("Usage: %s [-hbc] [ -m MIN ] [ -t TIME ]\n",argv[0]);
            printf("       -h : help\n");
            printf("       -c : turn on canonical mode\n");
            printf("       -m : set VMIN value \n");
            printf("       -t : set VTIME value \n");
            exit(0);
        case 'c' :
            bNoCanon = 0;  /* meaning canonical mode is ON */
            break;
        case 'm' :
            vmin = strtol(optarg,'\0',0);
            break;
        case 't' :
            vtime = strtol(optarg, '\0',0);
            break;
```

```
        case '?' :
            break;
        case ':' :
            break;
        default:
           fprintf(stderr, " ");
        }
    }

    /* Report on state that we will put terminal in */
    if ( bNoCanon )
        printf("Canonical mode off\n");
    else
        printf("Canonical mode on\n");

    printf("VMIN set to %d\n",vmin);
    printf("VTIME set to %d\n",vtime);

    /* save current terminal state */
    tty_mode(0);

    /* if canonical mode flag, unset icanon */
    if ( bNoCanon )
        set_non_canonical(vmin, vtime);

    /* call function to get input */
    get_response(!bNoCanon, vmin, vtime);

    /* restore terminal state */
    tty_mode(1);

    /* flush input stream */
    tcflush(0,TCIFLUSH);
    return 0;
}

void get_response( int iscanon, int vmin, int vtime)
{
    int    num_bytes_read;
    char   input[128];

    fflush(stdout);
    sleep(SLEEPTIME);
    if ( iscanon )
        printf("About to call read() in canonical mode\n");
    else
        printf("About to call read() with MIN = %d and TIME = %d\n",
               vmin, vtime);

    printf("Enter some characters or wait to see what happens.\n");
    num_bytes_read = read(0, input, 10) ;
    if ( num_bytes_read >= 0 ) {
        input[num_bytes_read] = '\0';
        if ( num_bytes_read > 0 )
```

```
            printf("\nReturn value of read() is %d; chars read are %s\n",
                num_bytes_read, input );
        else
            printf("\nReturn value of read() is %d;"
                    " no chars were read\n",num_bytes_read);
    }
    else
        printf("read() returned -1\n");
}

void set_non_canonical(int vmin, int vtime)
{
    struct termios ttystate;

    tcgetattr( 0, &ttystate);               /*  read curr. setting    */
    ttystate.c_lflag        &= ~ICANON;     /*  no buffering          */
    ttystate.c_cc[VMIN]      =  vmin;       /*  get min of vmin chars */
    ttystate.c_cc[VTIME]     =  vtime;      /*  set timeout to vtime  */
    tcsetattr( 0 , TCSANOW, &ttystate);     /*  install settings      */
}

void tty_mode(int how)
/* if how == 0 saves the termios state  and fd flags for later restoring    */
/* if how == 1 restores the termios state and fd flags from the saved state */
/* CANNOT CALL with how == 1 before first calling with how == 0 */
{
    static struct termios original_mode;
    static int            original_flags = -1;

    if ( 0 == how ){
        tcgetattr(0, &original_mode);
        original_flags = fcntl(0, F_GETFL);
    }
    else {
        if ( -1 == original_flags ) {
            fprintf(stderr, "tty_mode(1) called without saving first.\n");
            exit(1);
        }
        tcsetattr(0, TCSANOW, &original_mode);
        fcntl( 0, F_SETFL, original_flags);
    }
}
```

Before we leave this topic, let us see what happens when we disable the ability to process signals from the terminal. When you type a `Ctrl-C` at the keyboard, you usually do so to kill the foreground process. Why does this happen? The answer is a bit deeper than we explain here, but the general idea is that the `Ctrl-C` is mapped by the terminal driver to a signal named `SIGINT`, and that signal is delivered to the foreground process. This usually results in its being terminated. A similar sequence takes place when you type other keyboard signals, such as `Ctrl-\` and `Ctrl-Z`. These do not result in the same signals, but they are delivered to the process nonetheless.

The terminal is aware of three special control characters, known as the *interrupt*, *quit*, and *suspend* characters, usually `Ctrl-C`, `Ctrl-\`, and `Ctrl-Z` respectively. The `isig` switch to `stty` enables

detection and transmission of these characters. We will experiment with this idea. Have a second terminal window open before you start this. Make sure you restore the terminal to canonical mode and type the following:

```
$ stty -isig ; transpose
```

You can now type a line of characters and transpose will correctly transpose them. But try to terminate the program with `Ctrl-C` and you will be disappointed. Nor can you cannot stop it with `Ctrl-Z` or `Ctrl-\`. The terminal will not interpret these character codes as signals. Now you will have to issue a `kill -9` to terminate the `transpose` process in the other terminal window. You can use `pkill` to do this; for example:

```
$  pkill -9 transpose
```

You can remap other characters to these special characters using the terminal driver. It is a good exercise to try.

4.4.7 Other Terminal Processing Modes

There are a large number of possible combinations of terminal settings that make the terminal behave somewhere between the extremes of canonical mode and *raw mode*, in which most, but not all processing is turned off. For example, the driver can buffer characters but not allow editing. It can buffer characters, allow editing, but not echo anything. Most of these other modes are nameless. Raw mode was the name of a mode defined in Version 7 UNIX, as was *cbreak* mode. Programs that need complete control of their windows or the screen operate in raw mode usually, since they have to turn off all processing and do it on their own.

Regardless of the input mode of the terminal, the terminal driver must manage flow control of output. A process will typically generate characters faster than the hardware can display them. The driver has to block the process when the buffers are full.

4.5 I/O Control Using `ioctl`

The `fcntl()` call accesses disk driver attributes and the `tcgetattr()` call accesses terminal driver attributes. UNIX provides a more general-purpose device-control system call, `ioctl()`, which can be used to access and control any I/O device for which the manufacturer has provided a device driver. Although most operations on devices can be achieved with the previously described system calls, most devices also have some device-specific operations that do not fit into the general model, such as

- changing the character font used on a terminal,
- telling a magnetic tape system to rewind or fast forward[14],
- ejecting a disk from a drive,

[14]Since tapes do not move in byte increments, `lseek()` cannot do this.

- playing an audio track from a CD-ROM drive, and
- maintaining routing tables for a network.

The `ioctl()` function was designed to overcome these problems. It was first introduced in Version 7 AT&T UNIX as a general purpose I/O control program, to allow user level programs to access device drivers, hidden within the depths of the kernel. As time went on, more and more devices were handled through `ioctl()` calls. However, POSIX separated out the terminal control functions into the `termios` structure and functions, and for the most part, the `ioctl()` function is used today for accessing other devices. POSIX does not define `ioctl()`, but GNU includes the `ioctl()` function in its distribution of the C Standard Library.

The various device operations, known as *IOCTLs*, are assigned code numbers and multiplexed through the `ioctl()` function, which is defined in `<sys/ioctl.h>`. The code numbers themselves are defined in many different headers.

The `ioctl()` system call is given a file descriptor of a device file, and an IOCTL, which is an integer that represents a particular request that `ioctl()` should execute. If the request requires arguments, these are included after the command number. The man page begins as follows:

```
NAME
       ioctl - control device
SYNOPSIS
       #include <sys/ioctl.h>
       int ioctl(int fildes, int request, /* arg */ ...);
DESCRIPTION
    The  ioctl() function manipulates the underlying device
    parameters of special files. In particular, many operating
    characteristics of character special files (e.g.,terminals)
    may be controlled with ioctl() requests. The argument d
    must be an open file descriptor.

    The second argument is a device-dependent request code. The
    third argument is an untyped  pointer to memory.  ...
```

There are over 400 different request types, each defined by a different mnemonic code. The `ioctl_list` man page contains the list of valid request mnemonics that can be passed to `ioctl()` together with their argument types. Each device driver can define its own set of `ioctl` commands. The system, however, provides generic `ioctl` commands for different classes of devices. Generally speaking, the commands for the different classes have a prefix that identifies the type of command. For example, the magnetic tape commands are of the form `MTIOxxx` and can be found in `<sys/mtio.h>`, and the terminal I/O commands are of the form `TIOxxx`.

A simple example of an `ioctl()` is the call to retrieve the size of a terminal window. Terminal window sizes are defined by the `winsize` structure in the `<ioctl-types.h>` header file, which is included in `<sys/ioctl.h>`. The `winsize` structure looks like:

```
struct winsize
{
    unsigned short int ws_row;
    unsigned short int ws_col;
    unsigned short int ws_xpixel;
    unsigned short int ws_ypixel;
};
```

The pixel width and height are not normally assigned values by the kernel, but the rows and columns fields are. Therefore, we can obtain the rows and columns with a simple program such as the following.

```
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

void get_winsize(int fd,
                 unsigned short *rows,
                 unsigned short *cols )
{
    struct winsize size;

    if (ioctl(fd, TIOCGWINSZ, &size) < 0) {
        perror("TIOCGWINSZ error");
        return;
    }
    *rows = size.ws_row;
    *cols = size.ws_col;
}

int main(int argc, char* argv[])
{
    unsigned short int rows, cols;

    if (isatty(STDIN_FILENO) == 0) {
        fprintf(stderr, "Not a terminal\n");
        exit(1);
    }
    get_winsize(STDIN_FILENO, &rows, &cols);
    if ( rows > 0 )
        printf("%d rows, %d columns\n", rows, cols);
```

```
    return 0;
}
```

In general, one should try to avoid the use of `ioctl()` if there is an alternative means of performing the same task, as it is not completely portable. There are many IOCTLs for which alternative functions exist. For example, there are IOCTLS for the terminal interface. The `tty_ioctl` manpage has a complete list of them. Some have equivalents and others do not:

IOCTL	**argument**	**equivalent to**
`TCGETS`	`struct termios *argp`	`tcgetattr(fd, argp)`
`TCSETS`	`const struct termios *argp`	`tcsetattr(fd, TCSANOW, argp)`
`TCSETSW`	`const struct termios *argp`	`tcsetattr(fd, TCSADRAIN, argp)`
`TCSETSF`	`const struct termios *argp`	`tcsetattr(fd, TCSAFLUSH, argp)`
`TCSBRK`	`int arg`	`tcsendbreak(fd, arg)`
`TIOCINQ`	`int *argp`	*none - counts bytes in the input buffer*
`TIOCOUTQ`	`int *argp`	*none - counts bytes in output buffer*
`TCFLSH`	`int arg`	`tcflush(fd, arg).`
`TIOCSTI`	`char *argp`	*none – inserts *argp into input queue*

We could rewrite the `setecho` program to use these IOCTLs instead of using `tcgetattr()` and `tcsetattr()`, simply as an exercise. It is displayed in the listing that follows.

```
Listing setecho_ioctl.c

int main(int argc, char *argv[])
{
    struct termios info;
    FILE   *fp;

    if ( argc < 2 ) {
        printf("usage: %s [y|n]\n", argv[0]);
        exit(1);
    }

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
         return(1);

    /* retrieve termios struct */
    if ( ioctl( fileno(fp) , TCGETS, &info ) == -1 )
        die("ioctl", "1");

    /* If second argument starts with 'y' it is yes, echo on otherwise off */
    if ( 'y' == argv[1][0] )
        info.c_lflag |= ECHO ;
    else
        info.c_lflag &= ~ECHO ;

    /* replace termios with the modified copy */
    if ( ioctl(fileno(fp),TCSETS, &info) == -1 )
        die("ioctl","2");

    return 0;
}
```

Another use of `ioctl()` is to manipulate the terminal input queue. One can use the `TIOCSTI` IOCTL to insert bytes into a terminal input queue. The following listing demonstrates how one can insert bytes into the input queue of a pseudoterminal.

It repeatedly inserts a single 'x' into the terminal input queue of the controlling terminal of the process, `BUFFERSIZE-1` times, after which it inserts a newline so that if the terminal is in canonical mode, a call to `read()` will deliver the characters. It then removes the characters from the queue by entering a loop that reads a single character at a time. To allow a second program to be run to watch the queue size, it sleeps a bit before starting to read from the terminal.

During the read loop, it sleeps enough time so that the second program can watch the queue size diminish. Without this bit of delay between each read, the watching program would not see the queue decrease by one character at a time, but would just see it go straight to zero. When it is finished reading, it writes it to the standard output.

```
Listing ioctl2.c
#include <unistd.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdio.h>

#define BUFFERSIZE   30
int main ()
{
    int  i;
    char ch      = 'x';
    char newline = '\n';
    char buf[BUFFERSIZE];

    /* Insert BUFFERSIZE-1 'x's into the terminal input queue */
    for ( i = 0; i < BUFFERSIZE-1; i++ )
        ioctl(0, TIOCSTI, &ch);

    /* Because we assume canonical mode is on, we put a newline there
       so that when the read is issued, the characters will be
       transmitted */
    ioctl(0,TIOCSTI, &newline);

    /* Delay to allow queue-reading demo to start up */
    sleep(10);

    /* Read the queue one char at a time */
    i = 0;
    while ( read(0 , &ch, 1) > 0 ) {
        buf[i] = ch;
        i++;
        if ( ch == '\n' ) /* to exit loop */
            break;
        /* Delay a bit so other program can see queue size drop */
        usleep(100000);
    }

   /* Write the buffer contents to the terminal */
```

```
    write(1, &buf, i);
    return 0;
}
```

The last demo program shows how the `TIOCINQ` IOCTL can be used to query the size of the input queue. It opens the terminal device file specified on the command line, and then enters a loop of fixed duration in which it uses the `ioctl()` function to query the size of that terminal's input queue. It writes the size of the queue into a file named `queue_log` in the current working directory.

```
Listing ioctl3.c
#include   <stdio.h>
#include   <unistd.h>
#include   <stdlib.h>
#include   <termios.h>
#include   <fcntl.h>
#include   <sys/ioctl.h>
#include  "utils.h"

int main(int argc, char* argv[] )
{
    FILE   *fp;
    int     count, i;
    int     fd;

    /* Try to open given terminal device file */
    if ((fd = open(argv[1], O_RDONLY)) == -1)
        die("open", argv[1]);

    /* Open queue log in current working directory */
    if ((fp = fopen("queue_log", "w")) == NULL) {
        printf("Could not open queue_log\n");
        exit(1);
    }

    /* For a fixed amount of time (for simplicity) count queue size */
    for ( i = 0; i < 400; i++ ) {
        if ( ioctl( fd, TIOCINQ, &count ) == -1 )
            die( "ioctl", "TIOCINQ");
        fprintf(fp, "%d chars in queue\n", count);
        /* delay to see changes */
        usleep(100000);
    }

    fclose(fp);
    return 0;
}
```

If you start up `ioctl2` and note which terminal it is in, and then start `ioctl3` in a different terminal, specifying the first terminal as its argument, when `ioctl3` terminates, look at the contents of

`queue_log` and you will see the queue size diminish with each read.

One could write a program that kept track of the input queue size of all open terminals, with root privilege, for administrative purposes, using this idea.

4.6 Summary

A process's connnection to a file is encapsulated in a data structure maintained by the kernel, invaraiably called something like a file structure or a file entry. That structure contains flags that determine how the I/O is performed. A process can set various flags when it first opens the file, using the `open()` system call, and it can modify various flags using the `fcntl()` system call.

Terminals and pseudo-terminals are represented by device files, and as such, `fcntl()` can also be used to modify a process's connection to them ,but in addition, there is an API for altering the characteristics of terminals directly. This chapter explored the ways in which a process could control the terminal characteristics, both by altering them directly and by altering the process's connection to terminals. It also showed how the non-POSIX `ioctl()` function could be used for controlling terminals and other devices.

Chapter 5 Interactive Programs and Signals

Concepts Covered

Software tools, daemons,
interactive programs
non-blocking reads, signals,
signal, sigaction, kill, fflush, raise
sigemptyset, sigfillset, sigaddset,
sigdelset,sigprocmask, sleep

5.1 The Different Types of UNIX Programs

Programs that run in a UNIX environment can be classified by their relationship to terminal devices and by their input/output streams. They generally fall into one of three categories: *software tools*, *daemons*, and *interactive user programs*.

5.1.1 Software Tools (Filters)

A *software tool* is a program that

- receives its input from either
  - one or more files given as command-line arguments, or
  - from standard input if no filename arguments are present,
- expects its input to be an unstructured stream of bytes, almost always treated as plain text, and
- puts its output, which is also a stream of bytes, usually plain text, on standard output.

Because software tools can read from standard input and write to standard output, they can be connected via shell pipes to form pipelines, like factory assembly lines. UNIX has many software tools, including `awk`, `cat`, `cut`, `du`, `fold`, `grep`, `od`, `sed`, `sort`, `tr`, and `uniq`.

5.1.2 Daemons

Another class of programs are device drivers, which are not even attached to a terminal. A program that is not attached to a terminal is called a *daemon* in UNIX. A commonly used, but inaccurate, definition of a daemon is that it is a "background" process. To be precise, it is a process that executes without an associated terminal or login shell, waiting for an event to occur. The event might be a user request for a service such as printing or connecting to the Internet, or a clock tick indicating that it is time to run. The word "daemon" is from Greek mythology, and refers to a lesser god who did helpful tasks for the people he or she protected. Daemons are like these lesser gods; they are created at boot time, and exist, in hiding, ready to provide services when called upon.

Because daemons must not be connected to a terminal, one of their first tasks is to close all open file descriptors (in particular, standard input, standard output and standard error). They usually make their working directory the root of the file system. They then take additional steps to break their association with any shell or terminal, among which are leaving their process group and registering their intent to ignore all incoming signals. The concept of process groups will be discussed in Chapter 8. Signals are covered later in this chapter.

Daemon process names typically end in 'd'. This is one way to identify a daemon process in the output of the `ps -ef` command; find the names ending in "d" as in `ftpd`, `httpd`, `lpd`, `sshd`, `syslogd`, and `telnetd`. Daemons will be covered thoroughly in Chapter 8.

5.1.3 Interactive User Programs

Another category of programs are those that are tied to the user terminal in an inextricable way, because they customize the terminal for their own use. Programs that interact with the user through the terminal, such as text editors (`vi`, `nano`, or `emacs`), pagers (`more` and `less`), terminal-based administrative tools such as `top`, games (`snake`, `worm`, and `chess`), and terminal-based mail clients (`pine` and `mailx`), are tightly coupled to the terminal and must control its settings and attributes. They cannot use the standard input and output streams for communicating with the user because these lack the types of controls that a terminal has. These types of programs usually need to control

- whether or not characters are echoed,
- the number of characters that are buffered, if any,
- the movement of the cursor on the screen,
- whether certain key presses should have their default meaning or have application-defined meaning,
- whether timeouts should occur on input,
- whether signals such as `Ctrl-C` should be ignored, queued, or handled immediately.

We already saw how to control the state of the terminal using `stty` at the command level and the `tcgetattr()` and `tcsetattr()` functions at the programming level. Here we will explore the various modes into which we can put the terminal for the benefit of creating interactive programs.

5.2 Designing Interactive User Programs

Most interactive user programs are event-driven or menu-driven, which means that they perform some short task and then wait for user input to do the next task. All window-based applications are event-driven; they are idle, often blocked on input, while they wait for mouse, keyboard, or other events to be delivered to them by the window manager.

Here we go through the steps that are necessary to design and develop an interactive, terminal-based application. We will begin by understanding the problem, and then we will go through successive stages of making a program more and more responsive to user inputs.

5.2.1 Two Different Paradigms

Consider the kind of terminal-based program in which the program repeatedly prompts the user for input and takes an action accordingly. One of the user responses such programs expect is some type of quit signal, which typically breaks the loop and ends the program. This can be modeled by a control structure such as the following:

```
while (true) {
    prompt the user to do something
    wait for the user to respond
    if user's response is to quit
        break the loop and quit
    handle the response
}
```

The input part of this loop usually results in the process's being blocked on input, but it does not have to be designed this way. It might look like

```
while (true) {
    prompt the user to do something
    if the user responded
        handle the response
    otherwise
        do other things
}
```

In this paradigm, the program checks whether there is input and if there is, it responds to it, and if not, it does something else. This approach requires the ability to check if there is input without blocking while waiting for it. In short, it requires a type of input operation known as non-blocking input, which will be discussed below.

Regardless of which input method is used, programs such as video games, ATM machines, and text editors respond immediately to user key presses, rather than waiting for the `Enter` key to be pressed. They run in non-canonical mode, so they do not buffer input characters. Usually, they do not echo the input characters when these characters behave like function keys[1]. Also, they usually ignore illegal key presses. Thus, one task in designing interactive programs is to determine how to control the state of the terminal in this way.

But this is not enough. There is a big difference between a video game and a text editor, having to do with their relationship to time. We can distinguish between two kinds of interactive programs: those whose state is independent of time, and those whose state depends upon time. Any program that animates, in any way, is time-dependent; its state changes as a function of time. Programs that terminate or advance to a different state if the user does not respond within a certain amount of time are also time-dependent, because their state changes as a function of time. Video games are time-dependent. In contrast, a text editor is usually time-independent; it has no time-outs and no animation of any kind.

Programming with time is more complex than programming in a time-independent way because it requires knowledge of several different parts of the kernel API. Before we tackle this problem, we will explore a simpler one, namely how to write a text editor.

[1] Think about `vi` for example, and how it behaves in Command mode; you type a 'j' and it moves the cursor without displaying the letter, or more, when you type a space character and it advances a screen's worth of lines.

5.2.2 A Simple Text Editor

Note. The code in this section is not yet working properly. It is in progress.

The `vi` text editor is a very complicated piece of software, but we can create an extremely stripped-down version of it and still learn quite a bit. We will call it `simplevi`. This simple editor will allow the user to insert characters anywhere in a text document, but it will not provide a means of deleting characters. This is just a minor extension to the program. Also, it will not open an existing text file, instead creating a new one each time. Adding a feature to open a file does not provide much more insight into the interactive design of the program, but it does make the program larger.

It would be much easier to write this program if we used the Curses library, but as we have not yet covered that API, we will do it the hard way, the way it was done before Curses existed. It will give you an appreciation of Curses when we get to it. Instead, we will use the ANSI esacpe sequences that we covered in Chapter 1.

The design challenge in writing this simple text editor is integrating the two major objects that the program must manage:

- A behind the scenes text buffer, and
- The visible screen.

The text that the user types must be stored in a buffer of some kind. There are many possible ways to organize this buffer, with time-space trade-offs associated to each. A reasonable solution would be to create an array of pointers to the individual lines of the buffer, each of which would be a dynamically allocated fixed-size array. The array of pointers could be replaced by a doubly-linked list of pointers with added programming complexity. The arrays holding the lines of text could be started smaller and reallocated to larger sizes as the lines grow in size. These details are not important to us now, as performance is less of a concern than understanding the principles. Therefore, we take an even simpler approach: the text buffer is just one large linear array of characters named `buffer`:

```
char buffer[BUFSIZ];   /* BUFSIZ is a system limit */
```

It would be very difficult to manage this buffer if we did not have a convenient means on knowing where each line began within the buffer. Therefore, we use a second object, an array with an entry for each text line, whose values are the lengths of those lines. We shall declare it as

```
int line_len[MAXLINES];
```

where `MAXLINES` is some predefined limit on the number of lines in any file that `simplevi` will create.

The screen is treated as nothing more than a display object, which is all that it really is. There are two types of actions that the program must perform relative to the screen:

- Moving the cursor around and keeping track of its position, and
- Changing the appearance of the screen by writing text onto it or removing text from it.

5.2.2.1 Managing the Cursor

The primary challenge in managing the cursor is that, at any given time, the program must be able to map the cursor position to a position in the text buffer, and vice versa. Therefore, we need two additional objects: a cursor, and a position in the buffer. A cursor will be a structure with a row and column index, representing a position in a two-dimensional array whose upper-left corner is position (1,1). The position in the buffer is an integer:

```
typedef struct _cursor
{
   int r;
   int c;
} Cursor;

Cursor  curs;
int     index_in_buf;
```

A text file is really a sequence of lines of text terminated by newline characters. Each newline character forces the next line to start at the left margin in the row below the last character of the current line. As characters are inserted into a line of text, that line changes its form, but not the others above or below it, although they may shift downward. Therefore, it is convenient to maintain two other pieces of information associated to the cursor's current position:

```
int     cur_line;           /* current text line, not screen line */
int     index_in_cur_line;  /* index in current line of cursor */
```

`cur_line` is the index of the text line within which the cursor is located. `index_in_cur_line` is the offset from the first character of that line to the location of the cursor. If the line is N characters long, `index_in_cur_line` can be a number from 0 to $N-1$. Given these two indices, we can compute the position in the buffer by calling

```
index_in_buf = buffer_index(index_in_cur_line, cur_line, line_len);
```

where `buffer_index()` is defined as

```
int buffer_index( int index_in_line, int cur_line, int linelen[] )
{
    int totalchars = 0;
    int i = 0;

    while (  i < cur_line ) {
        totalchars += linelen[i];
        i++;
    }
    totalchars += index_in_line;
    return totalchars;
}
```

The program will always make sure that it knows the current line and the current index in the line as the cursor moves around on the screen. In fact, the cursor movement operations will actually update these variables, and from those recalculate the cursor position. This needs explanation. Text lines may be longer than the screen width. When this happens, they wrap onto two or more lines. In `vi`, when the cursor is moved upward or downward, it "jumps" over the wrapped lines. In other words, it moves from one text line to another, not from one screen row to another. Our `simplevi` program emulates this behavior. Therefore, when the user presses an arrow key up or down, it will move to the preceding or following text line. To make this possible, it will increment or decrement the `cur_line` variable as needed, and possibly adjust the `index_in_cur_line` variable, as will be explained below. But this implies that it has to find the new position of the cursor on the screen. The function `get_screenpos()` does that. For example, calling

```
get_screenpos(index_in_cur_line, cur_line, line_len, cols, &curs );
```

will store into `curs` the screen coordinates for the text buffer position in the line whose index is `cur_line`, and whose offset in that line is `index_in_cur_line`. The `line_len[]` array and the `cols` variable are the array of line lengths and the width of the screen respectively. The function is defined as follows:

```
void get_screenpos( int index, int lineno, int linelength[], int numcols,
                    Cursor *curs )
{
    int total_lines_before = 0;
    int lines_in_current_textline = 0;
    int i;

    for ( i = 0; i < lineno; i++ ) {
        total_lines_before += (int) ceil((double)linelength[i]/numcols);
    }
    lines_in_current_textline = index/numcols;
    curs->r = total_lines_before + lines_in_current_textline;
    curs->c = index - lines_in_current_textline*numcols;
}
```

5.2.2.2 Writing to the Screen

Since all output operations must bypass the C standard I/O library, lest they fall victim to its internal buffering, the program will only use the kernel's `write()` system call for writing to the screen. A write to the terminal device will always place the bytes to be written at the current cursor position. After a write completes, the cursor is advanced to the right of the last character written, unless what was written was an ANSI escape sequence that does not actually write to the screen. This implies that sometimes we will have to save the current position of the cursor before the write operation in order to return to it.

5.2.2.3 Operations Supported by simplevi

The vi text editor is a three-state finite automaton, and so is simplevi. The three states are

- input mode

- command mode, and
- last_line mode

The program always starts in command mode. If the user enters the command 'i', `simplevi` enters input mode and remains there until the user presses the `Escape` key. If in command mode the user enters the ':' character, `simplevi` enters last_line mode. In last_line mode, only two commands are possible:

`q` which causes `simplevi` to quit, and

`w` which causes `simplevi` to write the current contents of the text buffer to a file whose name is `tempfile` and which is located in the process's current working directory.

These can be entered separately or in sequence or as the string "wq". In last_line mode, the cursor is on the last line of the screen (hence its name). The last line of the screen is reserved for last_line mode and for other messages that `simplevi` may write. It is thus off-limits to the cursor. The program must ensure that this line is treated as if it is not part of the screen when in input mode or command mode.

In command mode, the cursor can be moved using the arrow keys. As will be noted below, this is a problematic part of the program when the Curses library is not used, because it is very hardware dependent, and the program does not handle this dependence. It is designed under the assumption that the arrow keys generate a specific sequence of bytes. Nonetheless, the purpose is to show how to handle cursor movement. Command mode also allows the user to press `Ctrl-D`, `Ctrl-C`, and `Ctrl-H` and not have their default behavior be invoked. It does this by disabling the terminal's handling of keyboard generated signals.

5.2.2.4 Terminal Support Functions

Three functions are used for modifying or querying the terminal state, as shown in the following program listing.

```
void modify_termios(int fd, int echo, int canon )
{
    struct termios cur_tty;
    tcgetattr(fd, &cur_tty);

    if ( canon )
        cur_tty.c_lflag     |= ICANON;
    else
        cur_tty.c_lflag     &= ~ICANON;
    if ( echo )
        cur_tty.c_lflag     |= ECHO;
    else
        cur_tty.c_lflag     &= ~ECHO;
    cur_tty.c_lflag     &= ~ISIG;
    cur_tty.c_cc[VMIN]   = 1;
    cur_tty.c_cc[VTIME]  = 0;
    tcsetattr(fd, TCSADRAIN, &cur_tty);
}
```

```
void save_restore_tty(int fd, int action)
{
    static struct termios original_state;
    static int            retrieved = FALSE;

    if ( RETRIEVE == action ){
        retrieved = TRUE;
        tcgetattr(fd, &original_state);
    }
    else if (retrieved &&  RESTORE == action ) {
        tcsetattr(fd, TCSADRAIN, &original_state);
    }
    else
        fprintf(stderr, "Illegal action to save_restore_tty().\n");
}

void get_winsize(int fd, unsigned short *rows, unsigned short *cols )
{
    struct winsize size;

    if (ioctl(fd, TIOCGWINSZ,  &size) < 0) {
        perror("TIOCGWINSZ error");
        return;
    }
    *rows = size.ws_row;
    *cols = size.ws_col;
}
```

`modify_termios()` either enables or disables echo and canonical mode, and it disables keyboard signals. It also sets the `MIN` and `TIME` line discipline values to 1 and 0 respectively so that the program reads a single character at a time and does not time-out. `save_restore_tty()` can be used to save the current terminal state into a local static variable for later restoration. `get_winsize()` uses the `ioctl()` function to get the current window size when the program starts up. If the window is resized while the program is running, all bets are off – it does not handle resizing events.

5.2.2.5 Other Support Functions

These include one for showing the program state, specifically the cursor position, the line index and the offset within the line:

```
void show_cursor( Cursor cursor, int index_in_line, int line_number )
{
    char curs_str[80];

    sprintf(curs_str, "Cursor: [%d,%d]  line index: %d  offset in line: %d ",
                cursor.r+1, cursor.c+1, line_number, index_in_line  ) ;
    write(1,SAVE_CURSOR, strlen(SAVE_CURSOR));
    write(1,PARK, strlen(PARK));
    write(1, curs_str, strlen(curs_str));
    write(1,REST_CURSOR, strlen(REST_CURSOR));
}
```

This function uses several constant strings that contain ANSI escape sequences. The full set of these is displayed in the listing below:

```
const char CLEAR_DOWN[]   = "\033[0J";
const char CLEAR_RIGHT[]  = "\033[0K";
const char CURSOR_HOME[]  = "\033[1;1H";
const char CLEAR_SCREEN[] = "\033[2J";
const char CLEAR_LINE[]   = "\033[2K";
const char SAVE_CURSOR[]  = "\033[s";
const char REST_CURSOR[]  = "\033[u";
const char UP[]           = "\033[1A";
const char DOWN[]         = "\033[1B";
const char RIGHT[]        = "\033[1C";
const char LEFT[]         = "\033[1D";
```

The string PARK used in show_cursor() above is defined dynamically, because it depends upon the screen size. It is defined as follows:

```
get_winsize(STDIN_FILENO, &rows, &cols);
sprintf(PARK, "\033[%d;1H", rows);
```

The last two functions used by the main program are one for inserting characters into the buffer, and one for moving the cursor to an arbitrary position:

```
void insert( char buf[],
             int insertpos,
             int cur_length, char ch )
{
    int i;
    for ( i = cur_length; i > insertpos; i-- )
        buf[i] = buf[i-1];
    buf[insertpos] = ch;
}

void moveto(int line, int column )
{
    char seq_str[20];

    sprintf(seq_str, "\033[%d;%dH", line+1, column+1);
    write(1, seq_str, strlen(seq_str));
}
```

5.2.2.6 The Main Program

The main program begins by initializing all variables and setting up the terminal. It is fairly self-explanatory:

```
int     quit = 0;
int     in_input_mode    = 0;
int     in_lastline_mode = 0;
char    buffer[BUFSIZ];      /* text buffer */
char    statusstr[80];
```

```
int     line_len[MAXLINES];   /* lengths of text lines, including newline
                                 characters */
int     num_newlines = 0;     /* number of newline characters in buffer */
char    prompt = ':';         /* prompt character */
unsigned short rows, cols;    /* screen dimensions */
Cursor  curs;                 /* cursor position (0,0) is upper left */
int     index_in_buf;         /* index of cursor in text buffer */
int     buf_size;             /* total chars in buffer */
int     cur_line;             /* current text line, not screen line */
int     index_in_cur_line;    /* index in current line of cursor */
int     i;
int     fd;
char    c;

/* Check if either input or output is redirected and exit if so */
if ( !isatty(STDIN_FILENO) || !isatty(STDOUT_FILENO) ) {
    fprintf(stderr, "Not a terminal\n");
    exit(1);
}

/* Save the original tty state */
save_restore_tty(STDIN_FILENO, RETRIEVE);

/* Modify the terminal - turn off echo, keybd sigs, and canonical mode */
modify_termios( STDIN_FILENO, 0, 0);

/* Obtain terminal window size and create string to park cursor */
get_winsize(STDIN_FILENO, &rows, &cols);
sprintf(PARK, "\033[%d;1H", rows);

/* Clear the screen and put cursor in upper left corner */
write(1,CLEAR_SCREEN, strlen(CLEAR_SCREEN));
write(1,CURSOR_HOME, strlen(CURSOR_HOME));

/* Initialize the counters and other locators */
num_newlines   = 0;
cur_line       = 0;
line_len[0]    = 0;
curs.r         = 0;
curs.c         = 0;
buf_size       = 0;
index_in_cur_line = 0;
index_in_buf    = 0;
```

After this it enters a loop in which it waits for user input. the form of this loop, with the operative code omitted is as follows:

```
while ( !quit ) {
    if ( in_input_mode ) {
        if (  read(STDIN_FILENO, &c, 1) > 0  ) {
            if ( c != ESCAPE ) {
                /* insert typed char and echo it
                   updating screen as needed */
            }
            else { /* ESCAPE so exit */
```

```
                in_input_mode = 0;
            }
        }
    }
    else {
        if (  read(STDIN_FILENO, &c, 1) > 0  ) {
            switch ( c ) {
            case 'i':
                in_input_mode = 1;
                break;
            case ':':
                /* do last line mode */
                break;
            case '\003':
                /* do ctrl-c */
                break;
            case '\004':
                /* do ctrl-d */
                break;
            case '\010':
                /* do ctrl-h */
                break;
            case ESCAPE:
                read(STDIN_FILENO, &c, 1);
                if (  c  == 91 ) {
                    read(STDIN_FILENO, &c, 1);
                    switch ( c ) {
                    case KEY_UP:
                         /* do key up */
                         break;
                    case KEY_DOWN:
                        /* do key down */
                        break;
                    case KEY_RIGHT:
                        /* do key right */
                        break;
                    case KEY_LEFT:
                        /* do key left */
                        break;
                    }
                }
                break;
            }
        }
    }
}
/* cleanup code follows here */
```

Of course the real work is not displayed here yet. Handling the cursor movements and insertions is where the complexity enters the picture. Only selected parts of this are contained here. The actual program is in the demos directory.

5.2.3 Non-Blocking Input

Changing the state of the terminal itself is preferable to changing the attributes of the open file descriptor, because we can exercise more control over it. However, the downside of this is that all programs that use that terminal will be affected by the changes. Usually this is not a problem, since the program you are writing will be the only foreground process, and no other process will be reading from the terminal while it is running. However, it is worthwhile to understand how to exercise some limited control over input through the file descriptor, using the `O_NDELAY` flag (called `O_NONBLOCK` in POSIX).

The `O_NONBLOCK` flag controls whether reads and writes are blocking or non-blocking. When a read is blocking, the process that executes the read waits until input is available, and only then does it continue. This is the semantics that beginning programmers learn. This makes sense; after all, why would you ever want a program to continue past a read instruction if the read did not yield any data?

Exercise. Before reading further, try to answer the preceding question.

Non-blocking I/O is a property of open file connections, not of terminals or devices; when you open a file and get a file descriptor as the return value of the `open()` call, you can specify in the call that the file connection should be non-blocking. In other words, the property of being non-blocking is part of the process's connection to the file or the terminal, not the terminal itself. Two different processes can have the same file open for reading, one using blocking reads and the other, non-blocking.

Remember that when a connection is established between a process and an input source, a buffer is created that holds data from the source on its way to the process. Whether it is a disk file, a terminal, a pointing device, or an audio source, there is some temporary storage area used for buffering input. When a process opens a non-blocking connection to an input source, whether it is a file or a device, calls to read data from that source retrieve whatever data is in the buffer at the time of the call, up to the amount requested in the read request, and return immediately. If the buffer is empty, they return immediately with no data. To be clear, in the call

```
if ( (bytesRead = read(fd, dataToProcess, bytesToGet) ) > 0 )
    {/* statement block */}
```

if the connection is non-blocking and no data is in the buffer of the file descriptor `fd`, then the call returns immediately with `bytesRead` == 0 and the statement block will be skipped.

Similarly, a call to the C library function `getchar()` will return either the character in the front of the buffer or, if the buffer is empty, nothing at all. In either case, `getchar()` returns immediately, i.e., in

```
if ( ( c = getchar())  != EOF )  { /* statement block */ }
```

if there is no character in the buffer, `getchar()` will return `EOF`, which the program can use to decide how to proceed.

Non-blocking input should not be confused with *asynchronous input*. Asynchronous input occurs when the process makes a call to read data and returns immediately, without waiting for the data

to be ready. The `read()` call is executed by a separate process, and as soon as the data are available for it, that process performs the I/O and fills the buffer passed to it by the one calling `read()`. This is called asynchronous input because the caller does not synchronize with the process running the `read()` call; they proceed independently once the call is made. Asynchronous input is useful when you may not need the data right away.

Non-blocking input is useful when the lack of input itself is a significant condition to be identified by the program. For example, it might indicate that a user has left the terminal and is no longer responding, or that a connection to a remote host has been broken, or that a pipe is empty. It may also imply that the user is choosing to not supply input because supplying input may mean making something happen that the user does not want to happen, as in a video game. Very often the process requesting the input has other work to do and it can simply check later whether the input is available. For this reason, non-blocking reads are usually placed inside loops where the condition is tested and an appropriate action can be taken. In most programs that use non-blocking input, the state of the program is changing without the user's intervention. This might be because animation is taking place, or a computation is being performed, or something else entirely.

The following listing is of a program that uses non-blocking input and pretends to do a simple animation. It draws "dots" on the screen, nothing more. Because it uses the same `modify_termios()`, `save_restore_tty()` and `get_winsize()` functions from the `simplevi.c` program, their definitions are omitted. It uses a function to put a file descriptor into non-blocking mode, `set_non_block()`.

```
Listing nonblockdemo.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/stat.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

#define RETRIEVE        1                   /* action for set_tty */
#define RESTORE         2                   /* action for set_tty */

// Defined elsewhere:
void modify_termios(int fd, int echo, int canon );
void save_restore_tty(int fd, int action);
void  get_winsize( int fd, int *rows, int *cols );

void set_non_block(int fd )
{
    int flagset;

    flagset    = fcntl(fd, F_GETFL);
    flagset    |= O_NONBLOCK;
    fcntl(fd, F_SETFL, flagset);
}


int main (int argc, char *argv[])
```

```
{
    char     ch;          // stores user's char
    char   period = '.';
    size_t bytecount;
    int    count = 0;
    int    done  = 0;    /* to control when to stop loop */
    int    pause = 0;    /* to control pausing of output */
    char   PARK[20];     /* ANSI escape sequence for parking cursor */
    int    numrows;      /* number of rows in window */
    int    numcols;      /* number of columns in window */
    const  char CURSOR_HOME[]   = "\033[1;1H";
    const  char CLEAR_SCREEN[]  = "\033[2J";
    const  char SAVE_CURSOR[]   = "\033[s";
    const  char REST_CURSOR[]   = "\033[u";
    const  char MENU[] = "Type q to quit or p to pause or r to resume.";
    char   dots[20];


    /* Check whether input or output has been redirected */
    if ( !isatty(0) || !isatty(1) ) {
        fprintf(stderr,"Output has been redirected!\n");
        exit(EXIT_FAILURE);
    }

    /* Save the original tty state */
    save_restore_tty(STDIN_FILENO, RETRIEVE);

    /* Modify the terminal -
       turn off echo, keybd sigs, and canonical mode */
    modify_termios( STDIN_FILENO, 0, 0);

    /* Turn off blocking mode */
    set_non_block( STDIN_FILENO );

    /* Get the window's size */
    get_window_size(STDIN_FILENO, &numrows, &numcols);

    /* Create string to park cursor */
    sprintf(PARK, "\033[%d;1H", numrows+1);

    /* Clear the screen and put cursor in upper left corner */
    write(STDOUT_FILENO,CLEAR_SCREEN, strlen(CLEAR_SCREEN));
    write(STDOUT_FILENO,CURSOR_HOME, strlen(CURSOR_HOME));

    /* Start drawing. Stop when the screen is full */
    while ( !done  ) {
        if ( ! pause ) {
            count++;
            /* Is screen full except for bottom row? */
            if ( count > (numcols * (numrows-1)) ) {
                pause = 1;
                count--;
            }
            else
```

```
                write(STDOUT_FILENO, &period, 1);
        }
        usleep(10000);  /* delay a bit */
        sprintf(dots, "  dots:   %d ", count);

        /* Save the cursor, park it, write the menu prompt */
        write(STDOUT_FILENO,SAVE_CURSOR, strlen(SAVE_CURSOR));
        write(STDOUT_FILENO, PARK, strlen(PARK));
        write(STDOUT_FILENO, MENU, strlen(MENU) );
        write(STDOUT_FILENO, dots, strlen(dots));
        /* Do the read. If nothing was typed, do nothing */
        if ( (bytecount = read(STDIN_FILENO, &ch, 1) ) > 0 ) {
            if ( ch == 'q' )
                done = 1;
            else if ( ch == 'p' )
                pause = 1;
            else if ( ch == 'r' )
                pause = 0;
        }
        /* Restore the cursor so the next dot follows the previous */
        write(STDOUT_FILENO,REST_CURSOR, strlen(REST_CURSOR));
    }
    /* Cleanup -- flush queue, clear the screen, and restore terminal */
    tcflush(STDIN_FILENO,TCIFLUSH);
    write(1,CLEAR_SCREEN, strlen(CLEAR_SCREEN));
    write(1,CURSOR_HOME, strlen(CURSOR_HOME));
    save_restore_tty(STDIN_FILENO, RESTORE);
    return 0;
}
```

The program has a loop of the form

```
while ( !done ) {
    /* actual work here */
    if ( (bytecount = read(STDIN_FILENO, &ch, 1) ) > 0 ) {
        if ( ch == 'q' )
            done = 1;
        else if ( ch == 'p' )
            pause = 1;
        else if ( ch == 'r' )
            pause = 0;
        }
}
```

This is the second paradigm shown in Section 5.2.1. The user's input has one of three possible effects: (1) entering `'q'` terminates the loop, (2) entering `'p'` allows the loop to continue but stops output, giving the illusion that the program is paused, and (3) entering `'r'` resumes the output if it is paused and has no effect otherwise. This is a very inefficient method of pausing of course, because the program is gobbling up CPU cycles while it is pretending to do nothing. However, we do not know enough as yet to do otherwise.

The part of the loop with the comment labeled "actual work here" is the part in which it

1. increments the period count and checks how many periods have been written so far,
2. delays a bit,
3. if still room for another period, writes a period, and
4. moves the cursor to the bottom row, writing the prompt and a count of the periods.

This section of the code is a form of animation – it is changing the state of the screen as a function of time. In this case, the timing is achieved by pausing a constant amount of time between redraws. This is a simple, and inefficient, form of animation. Later we will see that there are better means of achieving this.

5.2.4 Allowing Time-Outs

Sometimes we would like to write a program that has time-outs: if the user does not respond within a certain amount of time, it will take this condition to be a significant event in itself. Many programs have some kind of time-out or delay feature like this, so that if the user does not respond within a certain amount of time, the program will either terminate or take some other default action.

We can add a time-out feature to a program by setting `MIN` to 0 and `TIME` to the number of deciseconds we would like it to wait before it decides that there is no input to wait for. For the sake of curiosity, though, we will design a program that will allow us to set the `MIN` and `TIME` terminal attributes to any values we choose, so that we can see how it behaves with different values.

5.2.5 A Test Program

We will build a program in which we can test the effects of changing both the terminal driver's attributes and the open connection's attributes. The main program will be a test driver that allows the user to control the state of the terminal and terminal connection by various command line options, and repeatedly runs a simple function, which we will call `get_response()`, that reads user input in the given state of the terminal and connection. The main program will have a few bells and whistles besides.

The program will have separate functions for controlling the state of the control terminal and for changing the attributes of the file connection to the terminal device. The main program will have a loop to allow us to experiment with the `get_response()` function until we are satisfied that we understand how it behaves under the given settings. There are several pieces to the program, which we present in a bottom-up approach.

First, we combine the `save_restore_tty()` and `set_non_block()` functions into a single function that saves and restores both the terminal settings and the file descriptor flags. It uses the same macros as before:

```
void save_restore_tty(int fd, int action, struct termios *copy)
{
    static struct termios original_state;
    static int            original_flags = -1;
    static int            retrieved = FALSE;
```

```
    if ( action == RETRIEVE ){
        retrieved = TRUE;
        tcgetattr(fd, &original_state);
        original_flags = fcntl(fd, F_GETFL);
        if ( copy != NULL )
            *copy = original_state;
    }
    else if (retrieved && action == RESTORE ) {
        tcsetattr(fd, TCSADRAIN, &original_state);
        fcntl( fd, F_SETFL, original_flags);
    }
    else
        fprintf(stderr, "Bad action to save_restore_tty().\n");
}
```

We will change our `modify_termios()`, function so that it can be given a structure whose members describe the terminal settings:

```
typedef struct tty_opts_tag {
    int min;     /* value to assign to MIN */
    int time;    /* value to assign to TIME */
    int echo;    /* value to assign to echo [0|1] */
    int canon;   /* value to assign to canon [0|1] */
} tty_opts;
```

`modify_termios()` will options in the `tty_opts` parameter that is passed to it to the given `termios` structure.

```
void modify_termios( struct termios *cur_tty, tty_opts ts )
{
    if ( ts.canon )
        cur_tty->c_lflag    |= ICANON;
    else
        cur_tty->c_lflag    &= ~ICANON;
    if ( ts.echo )
        cur_tty->c_lflag    |= ECHO;
    else
        cur_tty->c_lflag    &= ~ECHO;
    cur_tty->c_cc[VMIN]      =  ts.min;
    cur_tty->c_cc[VTIME]     =  ts.time;
}
```

The second, `apply_termios_settings()`, applies the values in the `termios` structure to the terminal line associated to the given file descriptor (which should be standard input.)

```
void apply_termios_settings( int fd, struct termios cur_tty )
{
```

```
    tcsetattr(fd, TCSANOW, &cur_tty);
}
```

The `set_non_block()` function is the same as the one we used above and is omitted.

The `get_response()` function will prompt the user to type a character and will return a value that indicates what the user typed. For simplicity, it will ask for yes or no answers. It prints a question on the screen and gives the user a chance to give a valid response. If the user types a valid response or `max_tries` attempts were made, it returns.

```
int get_response( FILE* fp, ui_params  uip )
{
    int input, n;
    unsigned char c;
    time_t time0, time_now;

    time(&time0);
    while ( TRUE ){
        printf( "%s (y/n)?", uip.prompt);
        fflush(stdout);
        if ( !uip.isblocking )
             sleep(uip.sleeptime);
        if ( (n = read(fileno(fp), &c, 1)) > 0 ) {
            tcflush(fileno(fp), TCIFLUSH);
            input = tolower(c);
            if ( input == 'y' || input == 'n' ) {
                return input;
            }
            else {
                printf("\nInvalid input: %c\n", input);
                continue;
            }
        }
        time(&time_now);
        printf("\nTimeout waiting for input: %d secs elapsed,"
               " %d timeouts left\n",
                (int)(time_now- time0), uip.maxtries);
        if ( uip.maxtries-- == 0 ) {
            printf("\nTime is up.\n");
            return 0;
        }
    }
}
```

Comments

- The `fflush()` call flushes the buffers associated with the file stream passed to it. The C Standard I/O Library provides buffered I/O for file streams. When a program is started,

by default, the streams `stdin`, `stdout`, and `stderr` are *line buffered*. This means that the characters are transmitted to the terminal only when a newline is placed onto the stream. Since functions such as `printf()`, `puts()`, and the others that act on `stdout`, act on file streams, they are line buffered. The preceding `printf()` call

```
printf("%s (y/n)?", uip.prompt);
```

sends a string to `stdout` without a terminated newline and therefore this string will not appear immediately. To force the characters to be delivered to the terminal device, we use `fflush(stdout)`, which empties the buffer. If we comment out the `fflush()` call, the prompt will not appear on the screen until after a `read()` runs.

Note that the buffering provided for streams is independent of the buffering done by the terminal within the line discipline. Even if you put the terminal into non-canonical mode, if you use the higher-level C library functions, C will continue to line buffer. You must use the lower-level file descriptor operations to avoid the buffering.

- The call to flush the terminal's input queue, `tcflush()`, is needed in case the program is run in canonical mode and input is buffered. In this case the user has to enter a newline before the terminal will deliver the characters to the `read()` call, and `get_response()` needs to remove that newline character, otherwise it will be used as the next input character when it is called again.

- `get_response()` calls `sleep()` to block itself for the number of seconds given by `uip.sleeptime`. The `sleep()` function's prototype is

  ```
  unsigned int sleep(unsigned int seconds);
  ```

  The process will sleep until either the given time has elapsed or it receives a signal that it does not ignore. (Signals will be covered soon.) This gives the user a chance to type a response in case non-blocking input is in effect. Without this delay, the `read()` call would return faster than the user could blink an eye. The delay is not needed when input is blocking. There are alternatives to `sleep()` with finer granularity, such as `usleep()` and `nanosleep()`.

- Lastly, `get_response()` computes the total time elapsed since the original prompt was displayed by calling the `time()` function initially and each time that the user fails to enter any character before time runs out, and computing the difference in seconds.

The main program, with includes and a few other utilities omitted, follows. This is `term_demo1.c`.

Listing: term_demo1.c

```
#define PROMPT              "Do you want to continue?"
#define MAX_TRIES           3          // max tries
#define SLEEPTIME_DEFAULT   2          // delay after prompt
#define BEEP                putchar('\a')    // alert user
#define RETRIEVE            1          // action for save_restore
#define RESTORE             2          // action for save_restore
#define FALSE               0
```

```
#define TRUE                1

int main(int argc, char* argv[])
{
    int             response;
    tty_opts        ttyopts = {1,0,1,1};
    ui_params       ui_parameters = {
                       SLEEPTIME_DEFAULT, TRUE,
                       MAX_TRIES, PROMPT
                    };
    int             fflags = 0;
    struct termios  ttyinfo;
    int             fd;
    FILE*           fp;

    get_options(&argc, &argv, &fflags, &ttyopts,
                       &(ui_parameters.sleeptime));

    if ( !isatty(0) || !isatty(1) )
        exit(1);
    fp = stdin;

    save_restore_tty(fileno(fp), RETRIEVE, &ttyinfo);
    modify_termios( &ttyinfo, ttyopts);
    apply_termios_settings( fileno(fp), ttyinfo);

    // If fflags != 0 then the O_NONBLOCK flag is set on fp
    if ( fflags ) {
        ui_parameters.isblocking = FALSE;
        set_non_blocking_mode(fileno(fp), fflags);
    }
    while ( TRUE ) {
        response = get_response(fp, ui_parameters );
        if ( response ) {
            printf("\nMain: Response from user = %c\n",
                    response );
            if ( 'n' == response )
                break;
            }
        else
            printf("\nMain: No response from user\n" );
    }
    save_restore(fileno(fp), RESTORE, NULL);
    tcflush(0,TCIFLUSH);
    return response;
}
```

Comments

1. One problem with the program as it stands is that it will not respond to any key presses until it wakes up from its sleep. You cannot kick it into waking up. It will sleep for the specified time, come hell or high water. This means that a user with a fast response time will be unhappy with this solution. It is a one-size-fits-all solution to making a program responsive to user response rates. It is, in essence, a polling solution because it just repeatedly checks in on the user. An alternative is to somehow make the program sleep until the user actually does something.

2. Another problem with the program is that if it is terminated abnormally, as when the user types `Ctrl-C`, it will not have a chance to restore the terminal to its original settings. If the program has turned off canonical input and echo, for example, and it is killed before reaching the instruction in the main program to restore the terminal, then when the shell resumes execution upon the program's untimely death, the terminal will still be in non-canonical mode with no echo. Fortunately for present-day UNIX programmers, modern shells such as `bash` and `tcsh` automatically reset the terminal when processes invoked from the shell are killed, so these users will not see this happen.

3. However, even with the shell's ability to reset the terminal, it is still not necessarily immune to the problem that occurs when the program turns on non-blocking I/O[2]. If the program turns on non-blocking reads and is subsequently killed, there is a good possibility that the shell will be killed too. This is because, when a program is invoked from the shell, it shares the file descriptors of its parent shell. In other words, file descriptor 0 in the program points to the same file structure in the kernel as it does in the shell. Suppose that the program turns on the `O_NONBLOCK` flag on the standard input connection. It is actually modifying the connection that its parent, i.e., the shell, uses as well. In fact, this standard input connection is shared by all related processes – siblings, cousins, and so on. Any process that is sharing this file connection can potentially make changes that affect all other processes that use this terminal. Once the changes are made, unless they are undone before the process terminates, the shell's connection has those changed properties.

   Now think about the shell for a moment. A shell is basically running a loop of the form

   ```
   while ( not end of input ) {
       display the shell prompt
       read the user's command line
       carry out the instruction
   }
   ```

   Thus, if the program turns on non-blocking reads and is killed before turning it off, when the shell resumes, it executes a read command. Usually the shell is in blocked input mode, so when it tries to read input but none is there, it enters a blocked state waiting for the user to press the Enter key, which sends input to the shell process. Since non-blocking input is still on, instead of waiting for the user's input, it receives an `EOF` from the function call that it uses to retrieve the input. This `EOF` may cause the shell itself to terminate because most shells are killed by the `EOF` character. You should set `ignoreeof` to prevent this. In `bash`, you add the line

[2]Some shells appear to have fixed this "bug" as well. If a spawned process leaves the `O_NONBLOCK` flag on standard input, they clear it.

```
set -o ignoreeof
```

in your `.bashrc` to do this. Anyway, the result is that your shell is killed, and if this is a login shell, you will be logged out. On my host machine, `bash` gets caught in a segmentation fault, which should not happen.

4. The problem with non-blocking reads causing the shell to exit can also be solved by opening a new file connection to the terminal instead of using standard input. In the demo program, explore the effect of replacing the line

```
fp = stdin;
```

by

```
fp = fopen(ctermid(NULL), "r");
```

5.3 Signals

Signals are, as Richard Stevens once put it, software interrupts. They are a mechanism for handling asynchronous events, such as when a user types Ctrl-C at a terminal. Most non-trivial applications need to handle signals. In this section we provide an overview of signals, including what they are, how they are generated, how they are named, and how processes can deal with them.

From a strictly technical point of view, a signal is a message that has a type but does not have content. Messages are usually defined to be containers for data. Signals are not containers. Signals outside of the computer are things like traffic lights, hand gestures that mean "please stop, taxi driver" or "give me the check", alarm clock rings, or warning lights like "you're about to run out of gas". They do not have contents. It is enough that they have identity, so a particular signal type has well-defined meaning. When a combatant raises a white flag, the enemy knows that this signal means "I give up." In UNIX, a signal is simply an integer with a mnemonic name. For example, `SIGINT` is the interrupt signal.

5.3.1 Typing Ctrl-C at a Terminal

When you type a Ctrl-C, the effect is to terminate the currently running process. Why? When you type the Ctrl-C, the character code for it is sent by hardware and then, within the kernel, to the terminal's device driver. The device driver checks the character code and sees that it matches the `INTR` code[3]. Since it knows this is a control character that is supposed to cause delivery of an interrupt signal, it checks whether the `isig` attribute is set for the terminal. If `isig` is set, then the corresponding signal must be delivered, which is `SIGINT`, so it calls the signal subsystem of the kernel to notify it to send the `SIGINT` signal to all processes whose control terminal[4] is the one that received the Ctrl-C. If any of these processes has not explicitly notified the kernel of how it wants to handle this signal, it will terminate upon receipt of the signal, because by default, processes are killed if they do not catch `SIGINT`. Soon we shall see that a process can declare what the disposition of a delivered signal should be while it is running.

[3] In SunOS it is `INTR`. On other systems, it might be `VINTR`.
[4] The operating system keeps track of which processes are attached to which terminals.

5.3.2 Sources of Signals

All signals are sent by the kernel to processes. There is no other way for a process to receive a signal; the kernel is like the central signal processing station inside the machine. The kernel will send a signal to one or more processes if it receives a request to do so. Requests can come from a few different types of sources.

The terminal A user can type a key combination that causes the terminal driver to ask the kernel to send a signal. This is an asynchronous signal, since it can arrive at a process at any time, independent of what the process might be doing. Examples include Ctrl-C, Ctrl-Z, Ctrl-S.

Hardware Hardware exceptions can generate signals. The kernel detects when the exception occurs and sends a signal to the offending process. These may be synchronous or asynchronous. Synchronous events are things such as floating-point exceptions, illegal instructions, addressing exceptions (such as attempts to access addresses outside of the process's address space), and other events generally caused by the process itself. They are synchronous because if the process is run again, they will occur again at the same point in the process's execution. Asynchronous events are things like power loss and terminal hang-ups.

Software Software conditions can generate signals when something noteworthy happens. This can happen when out-of-band data arrives over a network connection, or when a process writes to a pipe after the reader of the pipe has terminated, or when an alarm clock set by the process expires.

Processes Processes themselves can request the kernel to send signals to processes, even themselves. This is not so strange; an alarm clock is a way for you to send a signal to yourself in the future; you set the alarm and wait for it to signal you. Similarly, a process can ask the kernel to send a wake-up call to itself at some future time. A process can also ask the kernel to send a signal to other processes to which it has permission to send signals. For ordinary user processes, these include any processes with the same real or effective user-id.

5.3.3 Signal Types

UNIX systems define signals in the header file `<signal.h>`. More accurately, the header file `<signal.h>` includes the header file that contains the signal definitions. Different UNIX systems store this file in different places. In Linux, the file `<bits/signum.h>` is where the signals are defined. The kernel includes this header file, but user level program are supposed to include `<signal.h>`. The idea is to keep separate headers for the kernel and the user-level programs.

The exact set of signals varies from one system to another, but some of them are standard across all systems. Signal names are just names for small integers such as `SIGINT`, `SIGKILL`, `SIGHUP`, and `SIGCHLD`. All names begin with the prefix `SIG`. `SIGHUP` is the *hang-up* signal. It is sent to a process when its control terminal has been disconnected. `SIGCHLD` is the signal sent to a parent by the kernel when it detects that one of its child processes has terminated. There are typically about 30 to 35 different signals defined in any UNIX system. The list of signals has changed over the years. The first 30 signals listed below are found in Linux and Solaris 9 ; the last 4 only in Solaris 9. The constant `NSIG` is the the total number of signals defined. Since the signal numbers are allocated consecutively, `NSIG` is also one greater than the largest defined signal number.

Name	Value	Default	Event	Note	Category
SIGHUP	1	Exit	Hangup		Termination
SIGINT	2	Exit	Interrupt		Termination
SIGQUIT	3	Core	Quit		Termination
SIGILL	4	Core	Illegal Instruction		Program Error
SIGTRAP	5	Core	Trace or Breakpoint Trap		Program Error
SIGABRT	6	Core	Abort		Program Error
SIGEMT	7	Core	Emulation Trap		Program Error
SIGFPE	8	Core	Arithmetic Exception		Program Error
SIGKILL	9	Exit	Killed		Termination
SIGBUS	10	Core	Bus Error	1	Program Error
SIGSEGV	11	Core	Segmentation Fault		Program Error
SIGSYS	12	Core	Bad System Call	1	Program Error
SIGPIPE	13	Exit	Broken Pipe		Operation Error
SIGALRM	14	Exit	Alarm Clock		Alarm
SIGTERM	15	Exit	Terminated		Termination
SIGUSR1	16	Exit	User Signal 1	1	Miscellaneous
SIGUSR2	17	Exit	User Signal 2	1	Miscellaneous
SIGCHLD	18	Ignore	Child Status Changed	1	Job Control
SIGPWR	19	Ignore	Power Fail or Restart		Hardware
SIGURG	21	Ignore	Urgent Socket Condition		Asynchronous I/O
SIGPOLL	22	Exit	Pollable Event		Asynchronous I/O
SIGSTOP	23	Stop	Stopped (signal)	1	Job Control
SIGTSTP	24	Stop	Stopped (user)	1	Job Control
SIGCONT	25	Ignore	Continued	1	Job Control
SIGTTIN	26	Stop	Stopped (tty input)	1	Job Control
SIGTTOU	27	Stop	Stopped (tty output	1	Job Control
SIGVTALRM	28	Exit	Virtual Timer Expired	1	Alarm
SIGPROF	29	Exit	Profiling Timer Expired	1	Alarm
SIGXCPU	30	Core	CPU time limit exceeded	1	Operation Error
SIGXFSZ	31	Core	File size limit exceeded	1	Operation Error
SIGWINCH	20	Ignore	Window Size Change	2	Miscellaneous
SIGWAITING	32	Ignore	Concurrency signal	3	Miscellaneous
SIGLWP	33	Ignore	Inter-LWP signal	3	Miscellaneous
SIGFREEZE	34	Ignore	Check point Freeze	3	Miscellaneous
SIGTHAW	35	Ignore	Check point Thaw	3	Miscellaneous

Notes

1. In Linux the numerical value of the signal is architecture-dependent.
2. This is only found in Sun OS and BSD.
3. These are in Solaris 9.

The above list has four columns. The first is the mnemonic name for the signal, i.e., the name that can be used in a program. The second is the integer value, which you do not need to know. The third is the default action that happens to a process. For example, `SIGCHLD` is ignored by default,

`SIGSTOP` causes the program to stop by default, and `SIGINT` causes the process to terminate. The last column indicates the cause or condition that leads to this signal.

5.3.4 Sending Signals

In UNIX, the kernel can send a signal to a process when some hardware error condition arises. For example, if a program attempts to execute an illegal instruction, the kernel will receive the hardware notification and will send the `SIGILL` (illegal instruction signal) to the offending process. A process can also send a signal to one or more processes (or even itself) by using the `kill()` system call. The form of the call is

```
int kill(int processid, int signal);
```

The first parameter stores a means to specify the process id of the process to receive the signal. The second parameter is the kind of signal to send. In the simplest case,

```
kill(942, SIGTERM);
```

sends the `SIGTERM` signal to the process whose process-id is 942. A process cannot send a signal to another process if they do not share the same real or effective user-id1. If a process does not have permission to issue the kill call, `kill()` returns –1.

`processid` can be 0, -1, or a negative number, and it means something different in each case. If `processid` is 0, the signal will be sent to all processes in the same process group, whereas if it is –1, and the sender is not the superuser, it is sent to all processes for which it has permission to send signals, which are those processes with the same real or effective user-id. If `processid` $<$ -1, it is sent to all processes in the process group with id `-processid`.

A process can also send a signal to itself using

```
int raise( int signal);
```

which is equivalent to

```
kill(getpid(), signal);
```

The call to `raise()` will return only when the process has handled the signal.

5.3.5 Signal Generation and Delivery

UNIX systems generally distinguish between the generation of a signal and its delivery. According to the *Open Group Base Specification Issue 6* (IEEE Std 1003.1),

> "A signal is said to be ***'generated'*** for (or sent to) a process or thread when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, signals generated via the sigevent structure and terminal activity, as well as invocations of the kill() and sigqueue() functions. In some circumstances, the same event generates signals for multiple processes.
>
> "A signal is ***delivered*** to a process when the appropriate action for that process and signal is taken."

What this means is that delivery takes place when the process receives the signal and responds by either

- *ignoring* it,
- taking the *default* action, or
- executing a *signal handler* for it.

Signal handling is described below. The point to remember now is that from the moment that a signal is generated for a process until the moment that the signal is delivered, the signal is *pending*. Pending signals are managed by the operating system[5].

5.3.6 Signal Handling

A process does not have to accept the default action caused by a signal. It can choose to respond differently to all signals except for `SIGKILL` and `SIGSTOP`. These signals always terminate the process. To handle a signal, the programmer defines a function called a *signal handler*. The signal handler is executed when the signal is received, provided that it has been installed.

The program notifies the operating system that it has a handler for a specific signal by executing a system call to install the handler. The original system call for installing a signal handler was the `signal()` system call. The `signal()` system call was unreliable because it was possible to miss signals when using it. It was replaced by a reliable signal installing call named `sigaction()`. We will explore the `sigaction()` call later. For now, we start with the `signal()` system call, partly because it is an easier one to use, and partly so that you understand its weaknesses. Once you do, you should avoid using it.

The `signal()` call has the form

```
signal( signal_number, handler_action)
```

The first parameter is the number of a signal, but you should always use its mnemonic name such as `SIGINT` or `SIGQUIT`. The second parameter is one of the following:

`SIG_DFL` Take the default action, which is usually to terminate the process.

`SIG_IGN` Ignore the signal completely and continue.

user-defined function Address of a user-defined function

[5]This discussion of signals overlooks the complexity entailed because of threads and multi-threaded processes. Until we discuss threads in general, we have to overlook this topic. But you should bear in mind that the operating system has to make decisions when signals are generated as to whether they are to be sent to every thread in a process or just to a single thread in particular, and that certain signals must always be sent to one choice or the other.

Examples

The following program, `signal_demo1.c`, shows how signal handlers for Ctrl-C (`SIGINT`) and Ctrl-\ (`SIGQUIT`) are installed.

```
#include        <stdio.h>
#include        <signal.h>

void catch1(int signum)
{
    printf("You can do better than that!\n");
}

void catch2(int signum)
{
    printf("I'm no quitter!\n");
}

int main()
{
    int i;

    signal( SIGINT, catch1 );
    signal( SIGQUIT,catch2 );
    for(i = 20; i > 0; i-- ) {
        printf("Try to kill me with ^C or ^\\.\n");
        sleep(1);
    }
    return 0;
}
```

The call `signal(signum, f)` installs `f()` as the signal handler for the signal `signum`. When `signal()` is executed, `f()` is installed. Until that point, `f()` is not installed. When you run `signal_demo1` and enter a Ctrl-C, the `SIGINT` signal is sent to the process executing `signal_demo1`; as a result, the handler `f()` runs, and when it terminates, the program resumes execution. In `signal_demo1.c`, the only action taken by either handler is to print a message on the screen, simply to show that the function was executed.

The next program, `signal_demo2.c`, is almost the same as `signal_demo1.c` with one exception: `SIGINT` and `SIGQUIT` are ignored by calling `signal()` with `SIG_IGN` as the second argument.

```
#include        <stdio.h>
#include        <signal.h>

int main()
{
    signal( SIGINT, SIG_IGN );    // ignore Ctrl-C
```

```
    signal( SIGQUIT, SIG_IGN );  // ignore Ctrl-\

    for(i = 20; i > 0; i-- ) {
        printf("Try to kill me with ^C or ^\\.\n");
        sleep(1);
    }
    return 0;
}
```

5.3.7 Putting It Together

We revise `term_demo1.c` so that it handles Ctrl-C and Ctrl-\ interrupts (whether from the keyboard or sent via a `kill()` from another process.) This program is `term_demo2.c`. The listing below shows only the changed portion of the code.

```
....
#include        <signal.h>
....  (snip)

    if ( fflags ) {
        ui_parameters.isblocking = FALSE;
        set_nodelay(fileno(fp), fflags);
    }

    signal(SIGINT, interrupt_handler);
    signal(SIGQUIT, interrupt_handler);

    while ( TRUE ) {
        response = get_response(fp, ui_parameters );
.... (snip)

char * signame( int signo)
{
    static char  name[16];
    switch ( signo ) {
    case  SIGINT:
        strcpy(name, "SIGINT");
        break;
    case  SIGQUIT:
        strcpy(name, "SIGQUIT");
    }
    return name;
}

void interrupt_handler(int signum)
{
```

```
    printf("Exiting with signal %s\n", signame(signum));
    save_restore(fileno(stdin), RESTORE, NULL);
    exit(2);
}
```

The major changes are that the `signal()` function is used to install handlers for `SIGINT` and `SIGQUIT`. In this program they share the same handler, named `interrupt_handler()`. This handler prints a message on the standard output and then restores the `termios` structure's flags and the file status flags to what they were before the program was run.

5.3.8 Weaknesses of the Signal Mechanism

Signals in this form do not carry any information other than their particular values. Therefore, they are of limited use. They were never intended to be a robust form of communication, and they are still not completely reliable. The early form of signal handling using the signal() system call was very unreliable. While a process was in the midst of catching a signal, it was unable to detect the arrival of another signal of the same type; any new signals of that type were lost. This means that if two signals of the same type were sent in rapid succession to a process, the second might be lost. Later versions of the signal() function in BSD and in SVR corrected this problem in different ways, so that it now has at least two different behaviors. The modern version in Linux 2.6 combines the semantics of each. It is best to avoid the signal() call for that reason.

5.3.9 Signal Handling The "Right" Way

Signals are a primitive form of interprocess communication by today's standards, but at the time they were conceived, they provided a simple, efficient method of solving the most important interprocess communication problems. The `signal()` system call was early UNIX's method of defining and installing signal handlers. One problem with the `signal()` call is that it needs to be reset each time, like a mouse trap – once it catches a signal, arriving signals are missed. Another problem with `signal()` is that its behavior was left unspecified in the case when multiple signals arrived, and different implementations of UNIX provided different semantics to handle multiple signals.

5.3.10 Multiple Signals

Suppose that a signal handler is in the midst of handling a signal that has been delivered when a second signal is generated and is pending. There are a few possible ways to dispose of this new signal:

- Ignore it completely, effectively losing the new signal;
- Put it in a queue and handle it when the current signal has been handled completely, effectively blocking pending signals while handling the current one;
- Interrupt the processing of the current signal, handle the new signal, and return to the old signal when the new one has been handled, effectively treating the handler like an involuntary recursive function;

In any one UNIX system, you might have found one of these solutions employed rather than the others, without any consistency. The POSIX standard introduced a uniform solution to the problem in the `sigaction()` interface.

5.3.11 The sigaction() call

The `sigaction()` system call allows a process to install a signal handler and to specify how it will respond to multiple arriving signals. Its prototype is

```
#include <signal.h>
int (sigaction (int signum, const struct sigaction *act,
struct sigaction *oldaction);
```

where

`signum` is the value of the signal to be handled

`act` is a pointer to a `sigaction` structure that specifies the handler, masks, and flags for the signal

`oldact` is a pointer to a structure to hold the currently active `sigaction` data.

We will examine the `sigaction` structure first to see how flexible this interface is. Notice that the function name is the same as the name of the structure whose address is passed to it, like the `stat()` function and the `stat` structure.

5.3.12 The sigaction struct

The `sigaction` structure is defined in `<signal.h>`. The definition is unusual because it has two members (`sa_handler` and `sa_sigaction`) that are allowed to overlap in memory and must be used in mutual exclusion. The simplest way to present it is as if it were two different overloaded definitions of the same structure:

```
struct sigaction              // backward-compatible, old-style handler
{
    void (*sa_handler) (int); // the action to take
    sigset_t sa_mask;         // additional signals to block
                              // during handling of the signal
    int  sa_flags;            // flags that affect behavior
};
```

or

```
struct sigaction           // POSIX compliant, new-style handler
{
void (*sa_sigaction) (int, siginfo_t *, void *);
```

```
                                    // pointer to signal handler
    sigset_t sa_mask;               // additional signals to block
                                    // during handling of the signal
    int  sa_flags;                  // flags that affect behavior
    };
```

In other words, there are two different forms of the `sigaction` structure. The first one uses the old-style of handler, and the second uses the newer POSIX compliant method. The structs are otherwise identical.

Notes

In the old-style, the `sa_handler` member does not have to be a pointer to a function. It can also be one of the two flags `SIG_IGN` or `SIG_DFL`. If it is `SIG_IGN`, the signal is ignored; if `SIG_DFL`, then the default action is taken. If a pointer to a handler is supplied, that handler will be run. The handler must have a single integer argument.

In the new style, the `sa_handler` is replaced by a pointer to a function that has three parameters as follows:

- An `sa_mask`, which defines which signals should be blocked while the handler is processing the signal. By default, the signal that caused the handler to run will always be blocked. Adding signals to `SA_MASK` is a way to block other signals as well.
- The `sa_flags` is a flag set that can be used to control how subsequent signals of the same type as the one that caused the handler to run are handled. For example, if a handler is handling a `SIGINT` signal and another `SIGINT` arrives while the process is in the handler, the `sa_flags` will determine how to dispose of the second `SIGINT`. It has no effect on other arriving signals. The flags determine how arriving signals are handled after the handler has been invoked. All flags are independent and `sa_flags` is their bitwise-or. The most important flags are:

  `SA_RESETHAND` If this bit is set, the signal action is reset to `SIG_DFL`. This means that as soon as the signal is delivered, the default action will take place. This flag implies the `SA_NODEFER` flag; signals are not blocked, instead causing the process to take whatever is the default action for the type of signal. The intention is to make the handler behave like the old-style `signal()` handler, since any signal arriving after the first will cause the default behavior.

  `SA_NODEFER` If this bit is set, the kernel will not automatically block the signal while it is being caught. This means that an arriving signal will cause the handler itself to be interrupted and re-entered with the second signal. This is involuntary recursion.

  `SA_RESTART` If this bit is set, certain system calls that would otherwise be terminated if a signal were delivered during their execution, will be automatically restarted. This bit allows the BSD style handling.

  `SA_SIGINFO` If this bit is set, two additional arguments are passed to the signal-catching function. If the second argument is not `NULL`, it points to a `siginfo_t` structure containing the reason why the signal was generated; the third argument points to a `ucontext_t` structure containing the receiving process's context when the signal was delivered.

Note. If multiple instances of an individual signal are delivered while that signal is currently blocked, then only one instance is queued. For example, if you issue multiple `SIGINT` signals to a process, without setting the `SA_NODEFER` flag in the handler, then the first one will cause the handler to run, the second will be queued, but all those after that will be lost.

5.3.12.1 Example

The first example, sigaction_demo1.c, shows how the `SA_SIGINFO` flag can be used. A signal handler for `SIGINT` is installed with the `SA_SIGINFO` flag set. The program delays itself using `sleep(60)` so that the user has time to enter a Ctrl-C. When the Ctrl-C is entered and the program is delivered a `SIGINT` signal, the handler runs and accesses the `siginfo_t` structure to print the values of its members as a result of the signal.

Listing: sigaction_demo1.c

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <bits/siginfo.h>
#include <stdio.h>
#include <stdlib.h>

void sig_handler (int signo, siginfo_t *info,  void *context)
{
    printf ("Signal number: %d\n", info->si_signo);
    printf ("Error number:  %d\n", info->si_errno);
    printf ("PID of sender: %d\n", info->si_pid);
    printf ("UID of sender: %d\n", info->si_uid);
    exit(1);
}

int main (int argc, char* argv[])
{
   struct sigaction the_action;

   the_action.sa_flags = SA_SIGINFO;
   the_action.sa_sigaction = sig_handler;

   sigaction(SIGINT, &the_action, NULL);

   printf ("Type Ctrl-C within the next minute"
           "or send signal 2. \n");
   sleep(60);
   return 0;
}
```

5.3.12.2 Example

The following program will demonstrate the use of the `sigaction` structure for old-style handlers as well as the new-style handlers. The program begins by defining the `sigaction` structure that will be passed to the `sigaction` function. The handler function, `interrupt_handler`, is assigned to the `sa_handler` field. The `sa_flags` field is initialized with the bitwise-or of the flags referenced in the command-line. The `sa_mask` is built step by step. There are a few ways to do this. You can start with an empty set and add to it or start with a full set and remove from it. In this case, I am allowing the user to interactively add signal numbers to the blocked set, so I start with an empty set and add to it. The method of creating these sets is discussed below.

```
Listing sigaction_demo2.c
/*
 * Usage:
 *        sigaction_demo  [ reset | noblock | restart ]*
 *
 * i.e., 0 or more of the words reset, noblock, and restart may appear
 * on the command line, and multiple instances of the same word as the same
 * effect as a single instance.
 *
 * reset   turns on SA_RESETHAND
 * noblock turns on SA_NODEFER
  * restart turns on SA_RESTART
 *
 * NOTES
 * (1) If you supply the word "reset" on the command line it will set the
 *     handling to SIG_DFL for signals that arrive when the process is
 *     in the handler. If noblock is also set, the signal will have the
 *     default behavior immediately. If it is not set, the default will
 *     delay until after the handler exits. If noblock is set but reset is
 *     not, it will recursively enter the handler.
 * (2) The interrupt_handler purposely delays for a few seconds in order to
 *     give the user time to enter a few interrupts on the keyboard.
 * (3) interrupt_handler is the handler for both SIGINT and SIGQUIT, so if it
 *     is not reset, neither Ctrl-C nor Ctrl-\ will kill it .
 * (4) It will ask you to enter the numeric values of signals to block. If
 *     you don't give any, no signals are blocked.
 *
 */

#include   <stdio.h>
#include   <string.h>
#include   <signal.h>
#include   <time.h>
#include   <fcntl.h>

#define INPUTLEN        100

int main(int ac, char* av[])
{
    struct sigaction newhandler;        /* new settings         */
    sigset_t     blocked;               /* set of blocked sigs */
    void         inthandler();          /* the handler          */
```

```
    char          x[INPUTLEN];
    int           flags = 0;
    int           signo, n;
    char          s[] = "Entered text: ";
    int           s_len = strlen(s);

    while ( 1 < ac ) {
        if (0 ==  strncmp("reset", av[ac-1], strlen(av[ac-1])) )
            flags |= SA_RESETHAND;
        else if (0 ==  strncmp("noblock", av[ac-1], strlen(av[ac-1])) )
            flags |= SA_NODEFER;
        else if (0 ==  strncmp("restart", av[ac-1], strlen(av[ac-1])) )
            flags |= SA_RESTART;
        ac--;
    }
    /* load these two members first */
    newhandler.sa_sigaction = interrupt_handler;/* handler function    */
    newhandler.sa_flags = SA_SIGINFO | flags;   /* new style handler   */

    /* then build the list of blocked signals */
    sigemptyset(&blocked);                  /* clear all bits      */

    printf("Type the numeric value of a signal to block (0 to stop):");
    while ( 1 ) {
        scanf( "%d", &signo );
        if ( 0 == signo )
             break;
        sigaddset(&blocked, signo);         /* add signo to list */
        printf("next signal number (0 to stop): ");
    }
    newhandler.sa_mask = blocked;           /* store blockmask     */

    // install inthandler as the SIGINT handler
    if ( sigaction(SIGINT, &newhandler, NULL) == -1 )
        perror("sigaction");

    // if successful, install inthandler as the SIGQUIT handler also
    else if ( sigaction(SIGQUIT, &newhandler, NULL) == -1 )
        perror("sigaction");
    else
        while( 1 ) {
            x[0] = '\0';
            tcflush(0,TCIOFLUSH);
            printf("Enter text then <RET>: (quit to quit)\n");
            n = read(0, &x, INPUTLEN);
            if ( n == -1 && errno == EINTR ) {
                printf("read call was interrupted\n");
                x[n] = '\0';
                write(1, &x, n+1);
            }
            else if ( strncmp("quit", x, 4) == 0 )
                break;
            else {
                x[n] = '\0';
```

```
                write(1, &s, s_len);
                write(1, &x, n+1);
                printf("\n");
            }
        } //while
    return 0;
}

void interrupt_handler (int signo, siginfo_t *info, void *context)
{
    int      localid;     /* stores a number to uniquely identify signal */
    time_t   timenow;     /* current time -- used to generate id */
    static   int ticker = 0;  /* used for id also */
    struct tm *tp;

    time(&timenow);
    tp = localtime(&timenow);
    localid = 36000*tp->tm_hour + 600*tp->tm_min + 10*tp->tm_sec +
              ticker++ % 10;
    printf("Entered handler: sig = %d \tid = %d\n",
            info->si_signo, localid );
    sleep(3);
    printf("Leaving handler: sig = %d \tid = %d\n",
            info->si_signo, localid );
}
```

The while loop in the main program exists just to demonstrate that the system calls to perform I/O are restarted when the interrupts occur. You can run this program with any combination of `SA_RESTART`, `SA_RESETHAND`, and `SA_NODEFER` to see the combined effect of the flags. You can add any signal to the blocked set, but you will only be able to send `SIGINT` and `SIGQUIT` unless you open a separate window and use the `kill` command to send arbitrary signals to the process. You can try this – put signal 4 in the blocked set and issue `kill -4` with the process id from a second window while the process is handling a Ctrl-C. You will see that signal 4 is blocked until `interrupt_handler()` finishes.

The while loop is designed so that you do not have to kill this program to terminate it. If you type `"quit"` it will terminate.

5.3.13 Creating Signal Mask Sets

There are four functions that modify signal mask sets:

```
sigemptyset(sigset_t *setp);
sigfillset(sigset_t *setp);
sigaddset(sigset_t *setp, int signum);
sigdelset(sigset_t *setp, int signum);
```

The first two create empty and full mask sets respectively. The next two add or delete specific signals from the specified sets. You can either create an empty mask and add to it, or create a full mask and delete from it. If you plan on having more than half of the signals in it, then do the latter, otherwise do the former.

5.3.14 Blocking Signals Temporarily around Critical Sections

The `sigprocmask()` system call can be used to block or unblock signals sent to a process. This is useful if you need to temporarily turn off all signals in a small section of code. This is one way to create something like a critical section, in the sense that the process will not be interrupted in the middle of the code fragment. It does not prevent the kernel from preempting the process and letting another process run on the CPU, so it does not provide a means of protecting shared variables that might be accessed while the process is removed from the CPU, but it does allow the process to complete some critical sequence of statements without any signal handlers running in the interim, and without being terminated in the midst of it. The prototype is

```
int  sigprocmask( int how, const sigset_t *sigs, sigset_t *prev);
```

where `how` is one of `SIG_BLOCK`, `SIG_UNBLOCK`, or `SIG_SETMASK`. `SIG_BLOCK` will block the specified signal set, i.e., prevent those signals from reaching until the procmask is changed. `SIG_UNBLOCK` allows the signals in the set to be unblocked, and `SIG_SETMASK` is used to change the mask completely, i.e., assign a new mask to the procmask. The following program can be run to demonstrate how the blocking works. If you type Ctrl-C during the first loop, the process will continue to loop and it will exit before the second loop is executed. If you change the `SIG_BLOCK` to `SIG_UNBLOCK` then the Ctrl-C will kill the process when you type it.

```
Listing: procmask_demo.c
#include <signal.h>
#include <stdio.h>

int main()
{
    int i;
    sigset_t sigs, prevsigs;

    sigemptyset(&sigs);
    sigaddset(&sigs, SIGINT);
    sigprocmask(SIG_BLOCK, &sigs, &prevsigs);

    for ( i = 0; i < 5; i++) {
        printf("Waiting %d\n", i);
        sleep(1);
    }
    sigprocmask(SIG_SETMASK, &prevsigs, NULL);
    for ( i = 0; i < 5; i++) {
         printf("After %d\n", i);
        sleep(1);
    }
}
```

5.4 Summary

An understanding of terminals and signal handling is essential to being able to write UNIX System applications. You now have almost all of the tools at your disposal to write terminal-based interactive programs that can use the full terminal window and allow the user to interact with it at will. As you will discover in the next chapter, you are still missing a few more pieces. For one, we still need to manage timing more accurately and allow events to happen at specific times as controlled by the program. For another, we do not have the means of placing characters anywhere on the screen. This is what comes next.

Chapter 6 Event Driven Programming

Concepts Covered

The NCurses library,
Alarms and interval timers
Signals revisited, signal-driven I/O
Asynchronous I/O

AIO Library
alarm, pause, nanosleep,
setitimer, getitimer, aio_read,
aio_return, aio_error

6.1 Introduction

An event-driven program is a program in which the flow of control of the program depends upon the occurrence of external events. The typical event-driven program remains in a state in which it listens for or awaits events, selects which events to respond to next, responds to them, and then returns to its listening state. Event driven programs must have some type of event recognition mechanism and event handling mechanism. Unlike sequential programs, event-driven programs must work correctly in an environment in which unexpected, dynamic, external stimuli come from sources such as users, hardware, or other processes.

6.2 Common Features of Event-Driven Programs

Event-driven programs include programs with graphical user interfaces, operating systems, device drivers, control system software, and video games, to name a few. Writing video games is a good means to master event-driven programming, because their requirements include those commonly encountered in other event-driven programs (*EDPs*), and because it is generally fun to write them. Typical video games need to handle the following:

Spatial control Like many other EDPs, video games have to manage the two-dimensional screen image, maintaining information about where all of its objects are located.

Timing Video games, like many EDPs, usually have moving images whose velocities are monitored and controlled by the game. Games may also time the user's inputs. They often have to keep track of clock time and cause certain events to happen at specific times or at specific intervals of time.

Asynchronous inputs and signals Video games, like all EDPs, have to respond to unpredictable user inputs such as mouse clicks, mouse motion, and keystrokes, as well as inputs from other sensors. These events are asynchronous with respect to the execution of the program.

Process synchronization Video games usually have multiple threads of control. One or more objects might be moving independently across the screen while the user independently types or uses a tracking device. The program has to keep track of and synchronize these independent processes and objects.

6.3 Terminal-based Games

Early UNIX systems often came bundled with a large variety of terminal-based games, i.e., games that ran in a pseudo-terminal window rather than a console window. The distinction between these is that a terminal window is a character I/O device that treats its display area as a two-dimensional array whose cells can contain characters, whereas a console window is a bit-mapped display device each of whose pixels can be accessed individually. Historically, these games were located in `/usr/games`.

These days, system administrators do not install the games, one reason being because they know that users tend to use up system resources having fun instead of working[1]. Another reason for not installing the terminal-based games is that there are now many free games that run on top of the X Windows system, using bit-mapped displays, making the older games seem less fun to those accustomed to the advanced technology. Perhaps those who appreciate the old terminal-based games are the same people who still appreciate black-and-white movies.

The great advantage of writing a terminal-based game over one that uses a GUI, is that it is easier to concentrate on the principles rather than the details of the windowing system. Although it is more exciting to create a game that runs in a graphics window, that requires an entirely different set of topics to learn and it would be a distraction from the objective of learning how to control and use signals, how to use time and synchronization, and how to control what the user is able to do with the keyboard. If you also had to learn about video cards, X Windows, and widgets and windows, your time would be consumed with that instead.

We will write a game similar to the game of `pong`, which runs in a terminal window. The game of `pong` is a simplification of an arcade game. In `pong`, there are two controlled objects: a ball and a paddle. The ball is a small circle or square that moves across the screen at some fixed speed. The paddle is a vertical line segment that the user can move up and down with keystrokes. The edges of the terminal window are walls off of which the ball bounces.

6.4 The Curses (NCurses) Library

Recall that in Chapter 1 we saw that we could configure the terminal by sending various escape sequences to it, such as "\033[7m" to reverse the colors of the video display. We also saw how we could move the cursor around, clear various portions of the screen, and do other things by sending escape sequences to the terminal. With different kinds of terminals requiring different escape sequences, the task of writing a program that controls a terminal becomes complex, if this is the only means of configuring and controlling terminals.

Fortunately, in UNIX, it is relatively easy to write programs that control the terminal, because UNIX systems come bundled with a character-oriented graphics library called Curses, the header file for which is `<curses.h>`. Curses is basically a library that wraps the complexity of terminal capabilities into an easy to use interface. According to Eric Raymond in the September issue of the Linux Journal,

"The first Curses library was hacked together at the University of California at Berkeley in about 1980 to support a screen-oriented dungeon game called `rogue`. It leveraged an earlier facility called `termcap`, the terminal capability library, which was used in the `vi` editor and elsewhere."

[1] In the past, people would spend idle time playing snake, worm, hangman, chess, or even rogue. The first thing one did when given an account on a UNIX system was to check the contents of `/usr/games`.

AT&T Bell Labs saw the virtues of Curses and developed their own version and incorporated it into SVR1. The SVR1 Curses library had many attractive features, but it was proprietary and it was based on a binary file format called `terminfo`, while the BSD version was free and based on the `termcap` file, a plain text file. Programmers were torn between the proprietary, enhanced Curses of SVR1 and the free, but limited feature BSD version. In 1982, Pavel Curtis solved the problem by rewriting a version of Curses based on the SVR1 version, but his was free and text-based. This made it possible for hackers to improve on it. To shorten the story, from Curses eventually came Ncurses (new curses), with more features and multi-terminal capabilities.

We will use the Ncurses library. The Ncurses library is a library of more than one hundred graphics functions for manipulating a character-oriented display device. The functions treat the display device as a two-dimensional array of characters, with coordinate (0,0) in the upper left corner. The library also contains routines for creating and manipulating windows, sub-windows and panels, and menus. We will focus our attention on a small set of features of the library.

6.4.1 NCurses Basics

Most window managers, whether on UNIX, the Macintosh operating systems, or the various Windows operating systems, follow the same principle of drawing: they maintain two data structures representing the canvas on which they draw. One, the *visible canvas*, is what is currently in view on the physical display device, and the other, the *hidden canvas*, is a canvas stored in memory, on which drawing operations take place. This terminology is not standardized and goes by various names, depending on the particular system one uses. I will use the term *double-buffering* to describe this method of rendering, which is what it is called in graphics applications.

In double buffering, applications draw on the hidden canvas, and when it is ready to be displayed, it is drawn onto the screen. In effect the hidden canvas becomes the visible canvas, and the memory used for the visible canvas becomes the hidden canvas. The operation of drawing the hidden canvas on the screen is known by various names, but the most common is *screen updating*. In reality, screen updating is optimized to redraw only those portions of the screen that are different than what is on display.

NCurses uses a form of double buffering. Because NCurses manages the content of a terminal window, the concept of a "screen" in this context is not the monitor's full visible area, but the area enclosed within the terminal emulation window. Henceforth, a screen refers to the content area of a terminal window. In NCurses, screen updating is called *refreshing*.

NCurses, like many graphical libraries, uses a coordinate system derived from matrix coordinates rather than Cartesian coordinates. The origin is at the upper left corner of the screen, and the pair (y,x) representing the coordinates of a point (or character in this case) is the row number followed by the column number, as shown in Figure 6.1.

It defines a data structure called a `WINDOW` to represent a *window*. A `WINDOW` structure describes a sub-rectangle of the screen, possibly the entire screen. It includes the window's starting position on the screen (the (y, x) coordinates of the upper left hand corner), its size, and various properties. It is opaque to the programmer; you do not have access to its members. You can write to a window as though it were a miniature screen, scrolling independently of other windows on the physical screen. A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen.

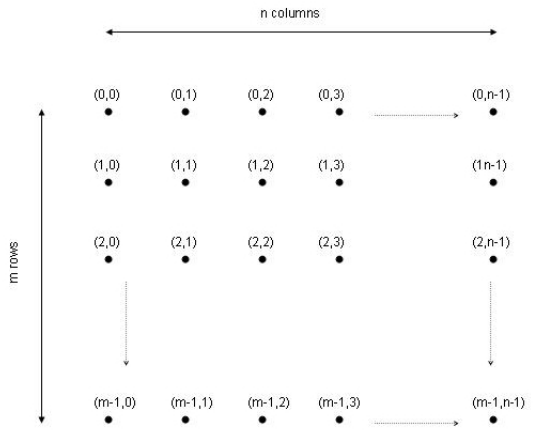


Figure 6.1: NCurses coordinate system

A *screen* is a special window that is the size of the terminal screen, i.e., it starts at the upper left hand corner and extends to the lower right hand corner. There are two predefined screens[2]: `stdscr`, which is automatically provided for the programmer, and `curscr`, for current screen, which is a screen image of what the terminal currently looks like. The programmer can draw on `stdscr`, but not on `curscr`.

6.4.2 Screen Updating

Each time that the application makes changes to a window that it wants to become visible on the screen, it needs to refresh the screen. There are two functions that make the section of the terminal screen corresponding to a window reflect the contents of the window structure: `refresh()` and `wrefresh()`. If the application is drawing on `stdscr`, which is the default screen, then it simply calls

```
refresh()
```

If it is drawing on a `WINDOW` named `win`, and it wants to draw that window's content on the screen, it calls

```
wrefresh(win)
```

`refresh()` is equivalent to `wrefresh(stdscr)`. It is actually a macro.

A piece of screen "real estate" may be within the extent of any number of overlapping windows. If two windows, `win1` and win2, overlap, and `wrefresh(win2)` is called, the library determines how to redraw the screen most efficiently, replacing those portions of the screen within the intersection of `win1` and `win2`. It only redraws a window if that window's content has changed in some way. You can call `touchwin(win)` to tell NCurses that the entire window `win` has changed, forcing a redraw when `wrefresh(win)` is called.

[2]There is a third, hidden screen that represents the logical screen on which the hidden drawing takes place. NCurses documentation calls it the *virtual screen*.

6.4.3 Building Programs

All programs using NCurses must include the `<ncurses.h>` header file and the standard C I/O library header file `<stdio.h>`. The header file `<ncurses.h>` is often just a symbolic link to `<curses.h>`, so they are often interchangeable. Because the NCurses library is not in the linker's standard set of libraries, you have to build explicitly with `-lncurses` in the command (following all files that reference NCurses symbols):

```
$ gcc -o myprog myprog.c -lncurses
```

6.4.4 A Core Repertoire of Functions

This is not intended to be a comprehensive tutorial on NCurses. For that you should consult any of the several on-line reference manuals or tutorials. The objective here is to explain the underlying concepts of the core library and to describe many of the functions that it provides. Below is a collection of the most important, and basic, representative functions for terminal configuration, cursor movement, output and input.

Configuration Functions

`initscr()`	Initialize the curses library and create a logical screen.
`endwin()`	Turn off curses and reset the screen.
`wrefresh(win)`	Draw what is in the logical window `win` into the physical display. Remember that `refresh()` is the same as `wrefresh(stdscr)`.
`clear()`	Clear the screen.
`keypad(stdscr,TRUE)`	Enable use of function and keypad keys (arrows, F1, ...)

Output and Cursor Movement

`move(r,c)`	Move the cursor to screen position (r,c).
`getyx(win,y,x)`	Get the current cursor position. This is a macro so y and x do not have to be passed by address.
`addch(c)`	Draw character `c` on the screen at the current cursor position, advancing the cursor to the end of the character. If `c` is a tab, newline, or backspace, the cursor is moved appropriately within the window. It wraps if it reaches the right margin.
`addstr(str)`	Draw the character string `str` on the screen at the current cursor position, advancing the cursor to the end of the string. It is equivalent to calling `addch()` for every character in the string.
`addnstr(str, n)`	Like addstr(), except that at most `n` characters of `str` will be written. If `n` is -1, then the entire string will be added, up to the maximum number of characters that will fit on the line, or until a terminating null is reached. Thus, `addstr(str)` is the same as `addnstr(str,-1)`.

`mvaddch(x,y,c)`	Move the cursor to position (x,y) on the screen and draw the character at that position, advancing cursor to the end of the character.
`addchstr(str)`	Like `addstr(str)`, except that: the cursor does not advance, it does not perform any kind of checking (such as for the newline, backspace, or carriage return characters), it does not expand other control characters to ^-escapes, and it truncates the string if it crosses the right margin, rather than wrapping it around to the new line.
`printw(fmt, ...)`	The same as `printf()` in C but prints to current cursor position.

Input

`getch()`	Read a character from the window.
`getstr(str)`	Read characters until a newline is received and store (without newline) in `str` (allocated by caller)
`scanw(fmt, ...)`	The same as `scanf()` in C – like calling `getstr()`, passing to `sscanf()`.

Window Functions

`win = newwin(l,c,y,x)`	Create a new window with `l` lines, `c` columns, whose upper left corner is at (`y,x`). Returns pointer to new window.
`mvwin(win,y,x)`	Move window pointed to by `win` to position (`y,x`)
`win = dupwin(oldwin)`	Make a duplicate of `oldwin`, returning pointer to the new window.
`delwin(win)`	Delete the window `win`, releasing all of its resources.
`putwin(win, filep)`	Write all data associated with the window pointed to by `win` into the `FILE` stream to which `filep` points. Returns `ERR` if the underlying write fails.
`win = getwin(filep)`	Read all window data stored into the file by `putwin`, and create and initialize a new window with that data. This returns a pointer to the new window.

Synopsis.

The `initscr()` function initializes the terminal in curses mode. It may also clear the screen in certain implementations. This always must be called first. It initializes the NCurses system and allocates memory for the `stdscr` and `curscr` windows and some other data-structures. When a program is finished, it should always call `endwin()` to reset the terminal and release curses resources.

After initializing curses, there are several functions that can be used to configure the terminal. It is usual to clear the screen with `clear()`, and if the program wants to receive key-presses from the keypad and function keys, then it should call `keypad(stdscr, TRUE)`. Other functions not shown above include functions that affect the terminal driver processing modes – `raw()` and `cbreak()`, `echo()` and `noecho()`, and `halfdelay()`. We will discuss these later.

Output functions can be divided into three families:

`addch()` Print a character at the cursor position, advancing cursor

`addstr()` Print a string at the cursor position, advancing cursor

`printw()` Print formatted output similar to `printf()`, advancing cursor

Thus, `addch()` adds a character, `addstr()` adds a string, and `printw()` prints formatted text. For each of these there are many variants, which are described below.

The cursor can be moved without output using the `move()` function. Its current position can be retrieved using `getyx()`. The `mvaddch()` function is a representative of a class of functions that perform a cursor movement prior to an output operation. Generally speaking, for each output function such as `addch()`, there is a corresponding function of the form `mvaddch()`. For example, there is a `mvaddstr()` function and a `mvprintw()` function.

The basic input functions are `getch()` and the string counterpart, `getstr()`, and the C-like `scanw()`, which is like `scanf()`. Notice that `getch()` has no argument, but `getstr()` expects a pointer to an allocated buffer in which to store the entered text.

Finally, all of the above functions work on the standard screen, `stdscr`. *They are all macros.* For each of them, there is a function that operates on arbitrary windows, and a naming convention that makes it pretty easy to guess what they are. For example, `wgetch(win)` is an input function that reads a character from the current window, `win`, and `getch()` is defined as `wgetch(stdscr)`. Similarly, `waddch(win,ch)` puts a character at the cursor position in `win` and `addch(ch)` is defined as `waddch(stdscr,ch)`.

One can create windows using `newwin()`, which allocates the memory on the heap and returns a pointer to it. If `newwin()` is passed 0 for either lines or columns, that dimension is set to the maximum it can be and fit within the terminal window. The function makes sure that the new window does not extend beyond the bounds of the terminal screen in all cases. A window can be moved using `mvwin()`; you have to refresh to see the change. This does not erase the old window from the screen, which you have to do yourself. You can make a copy of a window with `dupwin()`, and delete a window with `delwin()`.

6.4.5 Important Points About Windows and Refreshing

- It is a good idea to call `refresh()` or `wrfresh()` whenever you make changes to the screen, but you should bear in mind several important points.
- The functions of the `addch()` and `addstr()` families that write strings and characters to the screen always call `wrefresh()` themselves, so that it is not necessary to refresh when adding strings or characters. This is not true of the `printw()` functions.
- When drawing many windows to the screen, if `wrefresh()` is called for each window, it can cause bursty output and poor performance. The `wrefresh()` function actually calls two functions, `wnoutrefresh()` and `doupdate()`. A call to `wnoutrefresh(win)` copies the `WINDOW` pointed to by `win` onto the logical screen, and `doupdate()` copies the logical screen to the physical screen. Therefore, it is better to call `wnoutrefresh(win)` for each window to be written to the screen, followed by a single call to `doupdate()`.

- The input functions of the `getch()` and `getstr()` families will call `wrefresh()` on the given window if it has been moved or modified since its last refresh. If echo is on, then automatically a refresh will take place, since this is a modification. To be clear, `getch()` will call `refresh()`, and `wgetch(win)` will call `wrefresh(win)`. This can have serious consequences on the behavior of your program, since the cursor will move into the window on which `wgetch(win)` is being called, and refreshes may have unexpected consequences as well.

- Lastly, as a general rule, you should never write NCurses programs that mix the use of the standard screen, `stdscr`, and other windows. The functions that perform input and output and refreshing on the standard screen interact in unexpected ways with other windows. If you want to write simple programs, do not use windows in them, and conversely, if you feel that the program would benefit from using windows, then do not use any functions that operate on the standard screen.

6.4.6 A Few Simple Programs

In keeping with the tradition, we start with a hello-world program.

Listing 6.1: helloworld.c

```
Listing. helloworld.c
#include <ncurses.h>

int main()
{
    initscr();                    /* initialize the library */
    printw("Hello World !!!\n"); /* print at cursor */
    refresh();                    /* update screen (unnecesssary) */
    getch();                      /* wait for a keypress */
    endwin();                     /* clean up and quit curses */
    return 0;

}
```

The input call `getch()` is used so that the screen does not disappear before we can see it. The next program is a bit more interesting.

Listing 6.2: drawpattern.c

```
#include <stdio.h>
#include <curses.h>

int main()
{
    char   pattern[] = "1234567890";
    int    i;

    /* Initialize NCurses and clear the screen */
    initscr();
    clear();
```

```
    /* This will wrap across the screen */
    move(LINES/2,0);
    for ( i = 1; i <= 8; i++ ) {
        addstr( pattern );
        addch(' ');
    }

    /* Park the cursor at bottom */
    move(LINES-1,0);
    addstr("Type any char to quit:");
    refresh(); /* not needed */

    /* Wait for the user to type something, otherwise
       the screen will clear.  */
    getch();
    endwin();
    return 0;
}
```

Comments

1. NCurses has a predefined constant, `LINES`, that contains the number of rows in the terminal window, and a constant `COLS` that stores the number of columns.

2. Notice too that the program calls `refresh()` each time it changes the screen. This is unnecessary, because `addstr()` forces the refresh automatically.

3. If you delete the call to `getch()`, you will not see anything, and if you delete the call to `endwin()`, the screen will not be restored.

The next program draws a grid of periods centered on the screen.

Listing 6.3: drawgrid.c

```
#define  CENTERY    (LINES/2 -2)     /* The middle line in terminal */
#define  CENTERX    (COLS/2 -2)      /* The middle column in terminal */
#define  NUMROWS    (LINES/2)
#define  NUMCOLS    (COLS/2)

int main()
{
    int r, c;
    char  MESSAGE[] = "Press any character to exit:";
    int   length, i, j;
    length = strlen(MESSAGE);

    initscr();      /* Initialize screen */
    clear();        /* Clear the screen */
    noecho();       /* turn off character echo */

    char grid[NUMROWS][NUMCOLS];
```

```
    for ( i = 0; i < NUMROWS; i++ ) {
        for ( j = 0; j < NUMCOLS-1; j++ )
            grid[i][j] = '.';
        grid[i][NUMCOLS-1] = '\0';
    }

    /* move to center to draw grid */
    r = CENTERY - (NUMROWS/2);
    c = CENTERX - (NUMCOLS/2);
    move(r,c);

    /* Draw each row of grid as a string    */
    for ( i = 0; i < NUMROWS; i++ ) {
        mvaddstr(r+i,c,grid[i]);
    }

    /* Move to bottom of screen, post message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    getch();            /* wait for the user to type something */
    clear();            /* clear the screen */
    endwin();           /* delete curses window and quit */;
    return 0;
}
```

NCurses makes it easy to save any window to a file. The `putwin()` function will write the contents of a window to a `FILE` stream, and this can be read back into a program using `getwin()`. The next two listings show how to do both. The first is a program that draws a face in a window and also saves it to a file specified on the command line.

Listing 6.4: Saving a window: drawface2.c

```
#include <stdio.h>
#include <string.h>
#include <ncurses.h>
#include <stdlib.h>

#define  CENTERY    (LINES/2 -2)      /* The middle line in terminal     */
#define  CENTERX    (COLS/2 -2)       /* The middle column in terminal   */

void addhappyface(int * y, int * x)
{
    int orig_y = *y;
    addstr(" ^ ^ "); move(++(*y),*x);
    addstr(" o o "); move(++(*y),*x);
    addstr("  ^  "); move(++(*y),*x);
    addstr("\\___/");move(++(*y),*x);
    addstr("      ");
    *y = orig_y;
    *x = (*x) + 5;
    move( *y, *x);
}
```

```
int main(int argc, char *argv[])
{
    int   r, c;
    char  MESSAGE[] = "Press any character to exit:";
    int   length;
    FILE *fp;       /* for writing window contents */

    length = strlen(MESSAGE);
    if ( argc < 2 ) {
        printf("usage: %s window-file\n", argv[0] );
        return 0;
    }

    fp = fopen(argv[1], "w");
    if ( NULL == fp ) {
        printf("Error opening %s for writing.\n", argv[1]);
        return 0;
    }

    initscr();  /* Initialize curses library and the drawing screen */
    clear();    /* Clear the screen */

    /* Move to bottom of screen and post  message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* move to center of screen - width of face */
    r = CENTERY;
    c = CENTERX - 5;
    move(r,c);
    addhappyface(&r, &c);
    addhappyface(&r, &c);
    addhappyface(&r, &c);

    /* Park cursor at bottom at the right side of the message */
    move(LINES-1,length);
    refresh();

    /* Write the standard screen to a file */
    if ( ERR == putwin(stdscr, fp) ) {
        printw("Error saving window.\n");
    }
    fclose(fp);

    getch();          /* wait for the user to type something */
    clear();          /* clear the screen */
    endwin();         /* delete curses window and quit */
    return 0;
}
```

The next listing is of a program that can read any file created by an NCurses program that saved data using `putwin()`. It tries to open the file and display the window stored there. As `getwin()` returns a `NULL` pointer on failure, it checks that the returned pointer is not `NULL` before displaying the data.

Listing 6.5: Retrieving a saved window: getdrawing.c

```
#include <stdio.h>
#include <string.h>
#include <curses.h>

int main(int argc, char *argv[])
{
    FILE   *fp;
    WINDOW *win;

    if ( argc < 2 ) {
        printf("usage: %s window-file\n", argv[0] );
        return 0;
    }

    fp = fopen(argv[1], "r");
    if ( NULL == fp ) {
        printf("Error opening %s.\n", argv[1]);
        return 0;
    }

    initscr();      /* Initialize curses library and the drawing screen */
    cbreak();       /* So that the character is available immediately */
    noecho();       /* Turn off echo */
    clear();        /* Clear the screen */

    move(LINES-1,0);
    addstr("Enter a character to see the faces:");
    getch();

    win = getwin(fp);
    if ( NULL == win ) {
        clear();
        move(LINES-2,0);
        printw("The file %s was not created using putwin()."
               " Type any character to exit.\n",
                argv[1]);
    }
    wrefresh(win);
    fclose(fp);

    getch();          /* wait for the user to type something */
    clear();          /* clear the screen */
    endwin();         /* delete curses window and quit */
    return 0;
}
```

6.5 User Input in NCurses

NCurses has functions to put the terminal into a few different input modes. The following table summarizes the different models of input.

Function Call	Line Buffering	Erase/Kill Processing	Signal Interpretation	Blocking
`raw()`	No	No	No	Yes
`cbreak()`	No	No	Yes	Yes
`halfdelay(n)`	No	No	Yes	Timed (0.1*n seconds)
`nodelay(stdscr, TRUE)`	No	No	Yes	No

Raw mode, established with the `raw()` function, is similar to non-canonical mode in which, in addition, keyboard signal processing is disabled. Note though that it is a blocking input mode. Cbreak mode, established with `cbreak()`, is like raw mode except that keyboard signals are processed. Cbreak mode is also blocking.

The `halfdelay()` function and the `nodelay()` function both turn off blocking mode, but the `halfdelay()` function has a timeout whereas `nodelay()` is mercilessly unforgiving and does not. Neither is line-buffered nor allows editing functions. The `halfdelay()` function takes a single integer argument that represents the number of tenths of a second to block for terminal input. If no input arrives within that time, then it returns the `ERR` value, which is an integer value. In our demos directory, in chapter07, you can find demo programs named `raw_demo.c`, `cbreak_demo.c`, `halfdelay_demo.c`, and `nodelay_demo.c`, that show how these input modes work.

There are two other functions worth remembering: `noraw()` and `nocbreak()`. If the terminal has been put into raw, cbreak, or halfdelay mode, `noraw()` undoes that effect, turning on line buffering, line editing, blocking, and signal processing. It will not undo the effect of `nodelay()`, which can only be undone by calling

```
nodelay(stdscr, FALSE);
```

The `nocbreak()` function restores line-buffering and line-editing, but does not restore signal processing if it had been disabled by raw mode previously. For that you need to call `noraw()`, which, turns signal processing back on. `nocbreak()` also ends halfdelay mode.

Example

We will begin with a relatively simple program that puts the terminal into cbreak mode. The program will go into a user-controlled loop that terminates only when the user enters a specific character. To make it a bit more interesting, and realistic, we will use the F1 function key to terminate the program. The program will also show how user input can be used to modify the current window state other than by displaying text. It will let the user move the cursor around on the screen with the arrow keys. Finally, it will create a status bar at the bottom of the screen and write the current cursor position into it as the cursor moves, as well as the user's instructions for what to do.

The listing follows. The comments explain the logic within the program.

Listing 6.6: cursortrack.c

```
/* LINES and COLS are NCurses variables that get initialized when     */
/* initscr() is called. */
```

```
#define  CENTERY   (LINES/2 -2)      /* The middle line in terminal   */
#define  CENTERX   (COLS/2 -2)       /* The middle column in terminal */

int main(int argc, char *argv[])
{
    int x,y;      /* to retrieve coordinates of cursor */
    int ch;       /* to receive user input character */
    int r, c;     /* to store coordinates of cursor */

    /* A string to display in the "status bar" at the bottom of the screen */
    char  MESSAGE[] = "Use the arrow keys to move the cursor. "
                      "Press F1 to exit";
    int   length;
    length = strlen(MESSAGE);  /* compute this once. */

    initscr();              /* Initialize screen */
    clear();                /* Clear the screen */
    noecho();               /* turn off character echo */
    cbreak();               /* disable line buffering */
    keypad(stdscr, TRUE); /* Turn on function keys */

    /* Move to bottom left corner of screen, write message there */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* Start the cursor at the screen center  */
    r = CENTERY;
    c = CENTERX;
    move(r,c);

    /* Print the cursor's coordinates at the lower right */
    move(LINES-1,COLS-8);
    printw("(%02d,%02d)",r,c);
    refresh();

    /* Then move the cursor back to the center */
    move(r,c);

    /* Repeatedly wait for user input using getch(). Because we turned off */
    /* echo and put curses into cbreak mode, getch() will return without   */
    /* needing to get a newline char and will not echo the character.      */
    /* When the user presses the F1 key, the program quits.                */
    while((ch = getch()) != KEY_F(1)) {
        switch(ch) {

        /* When keypad() turns on function keys, the arrow keys are enabled */
        /* and are named KEY_X, where X is LEFT, RIGHT, etc.                 */
        /* This switch updates the row or column as needed, modulo COLS      */
        /* horizontally to wrap, and LINES-1 to wrap vertically without      */
        /* entering the sanctity of the status bar. */
            case KEY_LEFT:
                c = ( 0 == c )? COLS-1:c-1;
                break;
            case KEY_RIGHT:
```

```
                c = ( c == COLS-1 )? 0 : c+1;
                break;
            case KEY_UP:
                r = ( 0 == r )? LINES-2 : r-1;
                break;
            case KEY_DOWN:
                r = ( r == LINES-2 )? 0 : r+1;
                break;
            }

        /* Now we move the cursor to the new position, get its coordinates */
        /* and then move to the lower right to print the new position      */
        move(r,c);
        getyx(stdscr,y,x);
        move(LINES-1,COLS-8);
        printw("(%02d,%02d)",y,x);
        refresh();
        /* Now we have to move back to where we were ince the cursor was   */
        /* in the lower right after the printw(). */
        move(r,c);
      }

      endwin();         /* exit curses  */
      return 0;
}
```

6.6 Multiple Windows in NCurses

The next example program demonstrates how to use multiple windows. Note that this program does not use the standard screen.

Listing 6.7: drawmanygrids.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curses.h>

#define  CENTERY    (LINES/2 -2)        /* The middle line in terminal   */
#define  CENTERX    (COLS/2 -2)         /* The middle column in terminal */
#define  NUMROWS    (LINES/2)           /* number of rows we use         */
#define  NUMCOLS    (COLS/2)            /* number of columns we use      */
#define  REPEATS    5
#define  GRIDCHARS ".*@+#"              /* should have REPEATS many chars */

int main(int argc, char *argv[])
{
    char     MESSAGE[] =
             "Type the character of the grid to bring it forward, 'q' to exit:";
    int      length, i, j, k;
    WINDOW *mssge_win;
    WINDOW *windows[REPEATS];
```

```
    char     gridchar[REPEATS] = GRIDCHARS;
    int      rowshift, colshift;
    int      ch;

    initscr();     /* Initialize screen */
    noecho();      /* turn off character echo */

    /* Make sure that the window is wide enough for message at the bottom.*/
    length = strlen(MESSAGE);
    if ( length > COLS - 2 ) {
        endwin();
        printf("This program needs a wider window.\n");
        exit(1);
    }

    /* Calculate the amount by which we shift each window when we draw it */
    rowshift = (LINES - NUMROWS)/5;
    colshift = (COLS - NUMCOLS)/5;

    /* In this loop, we create a new window, fill it with a grid of a unique
       characters
    */
    for ( j = 0; j < REPEATS; j++ ) {
        /* Create a new window at an offset from (0,0) determined by the
           row and column shift. */
        windows[j] = newwin(NUMROWS, NUMCOLS, rowshift*j, colshift*j);
        if ( NULL == windows[j] ) {
            endwin();
            fprintf(stderr, "Error creating window\n");
            exit(1);
        }

        /* Draw each grid row  as a string into windows[j] */
        for ( i = 0; i < NUMROWS; i++ ) {
            for ( k = 0; k < NUMCOLS; k++ ) {
                wmove(windows[j], i,k );
                if ( ERR == waddch(windows[j],  gridchar[j]) )
                    /* Ignore the error; it means we are in the
                       bottom right corner of the window and the
                       cursor was advance to a non-window position
                    */
                    ;
            }
        }
        /* Update the virtual screen with this window's content */
        wnoutrefresh(windows[j]);
    }
    /* Now send the virtual screen to the physical screen */
    doupdate();

    /*Create a window to hold a message and put it in the bottom row */
    mssge_win = newwin(1, COLS, LINES-1, 0);

    /* Write the message into the window; mvwaddstr positions the cursor */
```

```
    mvwaddstr(mssge_win,0,0, MESSAGE);
    wrefresh(mssge_win);

    while ( 1 ) {
        /*
           Read a character from the message window, not from stdscr. The
           call to wgetch forces a refresh on its window argument. If we
           refresh stdscr, our  grids will disappear.
        */
        ch = wgetch(mssge_win);  /* wait for the user to type something */
        if ( ch == 'q' )  /* time to quit */
            break;
        /* Check if they typed a grid character */
        for ( j = 0; j < REPEATS; j++ ) {
            if ( ch == gridchar[j]  ) {
                wmove(mssge_win,0,length);    /* move cursor to bottom */
                touchwin(windows[j]);         /* force the update */
                wrefresh(windows[j]);         /* refresh, bringing it forward */
                break;
            }
        }
    }
    clear();              /* clear the screen */
    endwin();             /* delete curses window and quit */
    return 0;
}
```

6.7 Adding Timing to Programs: Sleeps

To make images move or animate on the screen, the program has to control the rate at which images are changed or displayed, which implies their being able to access a time-of-day clock or a timer. You have already seen the `sleep()` system call. It is one method of controlling time. The problem with `sleep()` is that its base unit is a one-second interval, which is too coarse for most video. An alternative is the `usleep()` system call; `usleep()` has a granularity of one microsecond[3]. The problem with `usleep()` though is that it uses the real timer of the process, of which there is just one, so multiple simultaneous calls to `usleep()` will have unexpected results. Both `sleep()` and `usleep()` suffer from the fact that they may share the same timer as the `alarm()` system call. POSIX requires a call named `nanosleep()`, which has even finer granularity and is guaranteed not to interact with any other timers. Therefore, we will use `nanosleep()`:

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

The `timespec` structure is defined by

```
struct timespec {
```

[3]This does not mean that it will be implemented accurately to within a microsecond. The implementation of the timer may be inaccurate for small intervals because of context-switching.

```
        time_t tv_sec;      /* seconds */
        long   tv_nsec;     /* nanoseconds */
    };
```

The first argument specifies the amount of time that the caller should be suspended. The second argument can be used to store the amount of time remaining in case the caller is awakened by a signal. If it is a `NULL` pointer, it is ignored. For now we will pass a `NULL` pointer as the second argument.

The first example of animation alternates two images on the screen at regular intervals determined by the `nanosleep()` timer. It uses three functions defined in file `faces.c` :

```
void addsadface(int * y, int * x);       // Draws a "sad" face at (y,x)
void addhappyface(int * y, int * x);     // Draws a "happy" face at (y,x)
void eraseface(int * y, int * x);        // Erases face at (y,x)
```

that draw, respectively, a "sad face", a "happy face", and a blank face. The coordinates are initially the upper left corner of the rectangle enclosing the face. On return they store the upper right hand corner.

The main loop will repeatedly draw a face, park the cursor in the lower left-hand corner of the screen, call `refresh()`, and then sleep a bit. The sad and happy faces will alternate. The first version of the program, whose listing follows, uses a loop that runs forever and must be killed by the user's entering a `Ctrl-C`.

Listing 6.8: animateface0.c

```
#include <stdio.h>
#include <curses.h>
#include <string.h>
#include <time.h>      /* for struct timespec */
#include "faces.h"     /* The set of face drawing functions */


int main(int argc, char* argv[] )
{
    int r, c;
    int i = 0;
    char  MESSAGE[] = "Type Ctrl-C to exit:";
    char  BLANKS[] = "                              ";
    int   length;
    struct timespec sleeptime = {0,500000000};     /* 1/2 second */


    initscr();     /* Initialize curses library and the drawing screen */
    clear();       /* Clear the screen */

    /* Move to bottom of screen and post  message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* Loop repeatedly until user types any character */
```

```
    while (1) {
        /* move to center of screen */
        r = CENTERY;
        c = CENTERX;
        move(r,c);     /* move to that position to draw */

        /* Draw either a happy face or sad face at (r,c) */
        if (0 == i ) {
            addsadface(&r, &c);
            i = 1;
        }
        else {
            addhappyface(&r, &c);
            i = 0;
        }

        /* Park cursor at bottom */
        move(LINES-1,length);
        refresh();
        nanosleep(&sleeptime, NULL);     /* sleep  1/2 second */
    }

    /* Cleanup -- erase the face first */
    r = CENTERY;
    c = CENTERX;
    move(r,c);
    eraseface(&r, &c);
    /* erase the message at the bottom of the screen */
    move(LINES-1,0);
    addstr(BLANKS);
    refresh();

    endwin();          /* Delete NCurses window and quit */
    return 0;
}
```

6.8 Combining User Input and Timing

We can use the `halfdelay()` function in combination with timed sleeps to animate the face and also let the user enter input. Our program can call `halfdelay(1)` to cause reads to wait one-tenth of a second and use a controlled loop whose entry condition is simply `(ERR == getch())` to allow the user to type a character to stop the loop. As soon as the user types, the character will be buffered, and the next time the `getch()` is executed, the character will be removed and returned, and the condition will be false, breaking the loop.

We can also turn off echo within NCurses with the `noecho()` function. Putting this all together, we have the makings of `animateface.c` below.

Listing 6.9: animateface.c

```
#include <stdio.h>
#include <curses.h>
```

```
#include <string.h>
#include <time.h>      /* for struct timespec */
#include "faces.h"     /* The set of face drawing functions */


int main(int argc, char* argv[] )
{
    int r, c;
    int i = 0;
    char  MESSAGE[] = "Press any character to exit:";
    char  BLANKS[] = "                              ";
    int   length;
    length = strlen(MESSAGE);
    struct timespec sleeptime = {0,500000000};     /* 1/2 second */


    initscr();     /* Initialize curses library and the drawing screen */
    clear();       /* Clear the screen */
    noecho();      /* Turn  off character echo */
    halfdelay(1); /* Turn on timed delay of 0.1 second -- if no char  */
                   /* within 0.1 sec,  getch() returns ERR */

    /* Move to bottom of screen and post  message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* Loop repeatedly until user types any character */
    while (ERR == getch()) {
        /* move to center of screen */
        r = CENTERY;
        c = CENTERX;
        move(r,c);     /* move to that position to draw */

        /* Draw either a happy face or sad face at (r,c) */
        if (0 == i ) {
            addsadface(&r, &c);
            i = 1;
        }
        else {
            addhappyface(&r, &c);
            i = 0;
        }

        /* Park cursor at bottom */
        move(LINES-1,length);
        refresh();
        nanosleep(&sleeptime, NULL);     /* sleep  1/2 second */
    }

    /* Cleanup -- erase the face first */
    r = CENTERY;
    c = CENTERX;
    move(r,c);
    eraseface(&r, &c);
```

```
    /* erase the message at the bottom of the screen */
    move(LINES-1,0);
    addstr(BLANKS);
    refresh();

    endwin();          /* Delete NCurses window and quit */
    return 0;
}
```

6.9 Timing with the alarm() and pause() system calls

The `sleep()` system call is based upon the use of *alarms*. An alarm in UNIX is essentially the software equivalent of a timer. (A timer goes off after a designated time interval; an alarm clock goes off at a designated clock time; in UNIX alarms are like timers.) When you want to snooze for an hour, you set a timer to wake you in an hour. In UNIX, a process can set an alarm to send itself a signal at some future time. It does this by calling `alarm()`, whose prototype is

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
```

`alarm()` sets a timer to expire in the number of seconds specified as its argument and returns immediately. If there is no pending alarm, the return value is 0. Otherwise the return value is the number of seconds remaining in the pending alarm. An alarm is *pending* if `alarm()` was called previously but the time period for which it was set has not yet elapsed. For example, suppose that at time 0 an alarm is set for 10 seconds:

```
alarm(10);
```

and that 4 seconds later, the same process calls `alarm()` again, this time asking for 20 seconds:

```
seconds_left = alarm(20); // called with 6 seconds remaining
```

The value returned by this call to `alarm()`, which is stored in `seconds_left`, would be 6. The alarm is reset to 20, and the alarm will signal the process 20 seconds later. The demo program `snoozealot.c` demonstrates how this works and how to use that return value. Before you look at `snoozealot.c`, take a look at the simpler program, `snooze.c`, which will be described shortly.

When an alarm's timer expires, a `SIGALRM` (there is no "A" between the L and R) signal is sent to the process that set the alarm. If the process does not provide a signal handler for the `SIGALRM`, or if for some other reason, the signal is not caught, then the `SIGALRM` will kill the process. The "other reason" can be that the process is in a system call that cannot be interrupted, or that it is handling some other signal at the time the `SIGALRM` hits it, and the particular handler is not designed to allow multiple signals to be received.

From this discussion you should realize that the `alarm()` call can be used to maintain at most one alarm at a time. If you want the effect of multiple alarms, then you have to code this into the `SIGALRM` handler; i.e. you have to reset the alarm for the new time.

In case it is not yet apparent, the `alarm()` system call has several different uses. A process can set an alarm prior to starting a long task that might not complete if the input data is unexpectedly large. The alarm will prevent the process from spending too much time on potentially endless tasks. It can also set an alarm to do a task asynchronously after a specific amount of time, perhaps based upon the state of its data.

A system call that is often used with alarms is the `pause()` call. When a process calls `pause()`, it is suspended and remains suspended until it receives a signal. Any signal will do to waken it. The prototype of `pause()` is:

```
#include <unistd.h>
int pause(void);
```

If a process calls `pause()` without having scheduled an alarm that will expire after the call to `pause()`, it will most likely never run again[4]. For example:

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    pause();
    printf("You will never see this message!\n");
    return 0;
}
```

This program, when run, will stay blocked until either the kernel sends it a signal or a user does, and because there is no handler, it will take the default action on receiving the signal, which is usually to terminate, not ever reaching the `printf()` statement.

The demos directory contains several different examples to demonstrate the `alarm()`, `signal()`, and `pause()` calls. The `snooze.c` demo is similar to the UNIX `sleep` command. The `snoozealot.c` demo demonstrates how the alarm can be reused, how a signal handler for `SIGALRM` and for `SIGINT` can do program cleanup, and how to allow non-blocking user input while in a programmed loop that is counting down an alarm. The following demo is another example that focuses only on alarms but also records the times that they occur.

The program in the listing below, `alarmdemo1.c`, uses the `signal()` system call to install signal handlers. There is a second version of this program in the demos directory that does the exact same thing using `sigaction()` instead. It is useful to compare them.

Listing 6.10: alarmdemo1.c

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <stdlib.h>

/* This is the SIGALRM handler. When the SIGALRM is delivered to this */
```

[4]It might run if some other signal is delivered to it, for which it has a handler.

```
/* process, it resets the handler and displays the current time.      */
void catchalarm(int signo )
{
    signal(SIGALRM, catchalarm); /* install handler again */
    time_t t;                    /* time in seconds since the Epoch */
    struct tm *tp;               /* time struct with years, months, days,...*/

    time(&t);                    /* get current time as a time_t in t */
    tp = localtime(&t);          /* convert time to a tm struct */
    printf("Caught alarm at %d:%d:%d\n", tp->tm_hour, tp->tm_min ,tp->tm_sec );
}

int main(int argc, char * argv[])
{
    int k, sec;
    struct tm *tp;      /* time struct with years, months, days, etc */
    time_t t;           /* time in seconds since the Epoch */

    /* check proper usage */
    if (2 > argc) {
        printf("Usage: %s n\n",argv[0]);
        return -1;
    }

    k = atoi(argv[1]);             /* convert argv[1] to int (no error check) */
    signal(SIGALRM, catchalarm);   /* install catchalarm as the handler */

    time (&t);                     /* store current time in t */
    tp = localtime(&t);            /* store t as day,hours,minutes, etc. */

    /* print time at which alarm is set an how long it is set for */
    printf("Time is %d:%d:%d\n", tp->tm_hour, tp->tm_min ,tp->tm_sec );
    printf( "Alarm is set for %d seconds.\n",  k);
    sec = alarm(k);                /* set alarm  */

    pause();                       /* wait for a signal to arrive */
    return 0;
}
```

Explanation

The main program begins by installing a `SIGALRM` handler using the `signal()` call:

```
signal(SIGALRM, catchalarm);
```

The `catchalarm()` handler is unlike the earlier examples. Before it does anything else, it calls `signal()` to reinstall the handler. This is because signals of the same type will be lost while the process is handling a signal. The only way to catch a `SIGALRM` while in the handler for `SIGALRM` is to reissue the `signal()`. Although this particular program cannot issue another alarm, in general, signal handlers should be designed so that if a second signal of the same type arrives while they

are processing the first, they are not caught "by surprise" and possibly killed by the second signal. This handler is just demonstrating that technique.

The handler uses the `time()` and `localtime()` system calls to get the current time, convert it to a human readable format, and display it on the console. The main program displays the current time and immediately turns on the alarm by calling

```
alarm(k)
```

where `k` is the command line argument's numeric value. It then calls `pause()` to wait for the `SIGALRM` to be received. If a `SIGALRM` arrives before any other signal, it will cause `catchalarm()` to run, which will display the time. If another signal arrives first, the process will probably be killed.

6.10 Interval Timers

The time granularity, or resolution, of the `alarm()` system call is too coarse to be useful for many applications. Furthermore, `alarm()` must be called repeatedly if an alarm is to go off at regular intervals, such as when a process is timing the progress of some task. (Suppose you wanted to display some sort of speed indicator on the console, where instantaneous speed was measured by the amount of data written in a fixed time interval. You would need a timer of fine resolution and a `SIGALRM` catcher that would measure the amount of data processed and reinstall itself, but this would be slightly inaccurate because of time lapsed between the start of the handler and the time it took to reinstall itself.)

Interval timers were introduced in later Berkeley distributions of UNIX (4.2BSD) as well as in the SVR4(1170) versions of UNIX as a solution to this problem. An interval timer has two components: an *initial delay* and a *repeat interval*. The value of the initial delay is the amount of time the kernel should delay before sending the first signal to the process. The value of the repeat interval is the amount of time the kernel should wait between successive signals sent to the process. In other words, if an interval timer is started at time t_0, with initial delay $= x$ and repeat interval y, then it will generate signals at times $t_0 + x$, $t_0 + x + y$, $t_0 + x + 2y$, $t_0 + x + 3y$, $t_0 + x + 4y$, ... until the process terminates.

6.10.1 Three kinds of timers: Real, Virtual, and Profile

There are three different types of interval timers. One type of timer ticks during all elapsed time (like the clock on the wall); this is the *real timer*. The second ticks only when the process is in user mode (like the timer in a sporting event, which stops when play is paused for various reasons); this is the *virtual timer*. The last ticks when the process is in user mode or in system calls (like the timer in a professional chess game, which is stopped when one person has stopped it and the other has not yet started it[5]); it is called the *prof timer*. The constants used to define these timers, as you will shortly see in the documentation are:

`ITIMER_REAL` ticks always and sends a `SIGALRM` when it expires

[5] If the two people decide to take a coffee break, the timer is in the off state. You can think of user mode as your time and kernel mode as your opponent's time. Then this analogy fits.

`ITIMER_VIRTUAL` only ticks when the process is in user mode, i.e., not in system calls. It sends a `SIGVTALRM` when it expires.

`ITIMER_PROF` ticks during user mode and in system calls; on expiration sends a `SIGPROF` signal

The "`PROF`" in `SIGPROF` and `ITIMER_PROF` is short for *profile*, which is a snapshot of a process's time usage across all user mode and kernel mode activities. From these definitions, it follows that the time the process spends sleeping is its real time less its profile time, and that the time it spends in kernel mode is profile time less virtual time. Our interest is in real interval timers, those that tick like an ordinary alarm clock.

6.10.2 The Initial and Repeating Values

An `itimerval` structure contains two members: the value of the initial delay, and the value of the repeat interval:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
};
```

The initial delay is stored in the `it_value` element and the repeat interval is stored in `it_interval`. As the timer ticks, the `it_value` element is decremented; when it reaches zero, a signal is sent to the process and the value of `it_interval` is copied into `it_value`.

Each member is of type `timeval`. A `timeval` structure represents a time interval using two elements: the number of *seconds* and the number of *microseconds* in the interval. There is no milliseconds field:

```
struct timeval {
    long tv_sec;            /* seconds */
    long tv_usec;           /* microseconds */
};
```

As a long integer is usually either 32 or 64 bits, depending upon the implementation, the `tv_usec` member is large enough to represent any number of microseconds from 0 to one million. Since it is common to work with time in milliseconds, you need to convert a time measured in milliseconds to a `timeval` with seconds and microseconds units. Mathematically, if t is a time expressed in milliseconds, then

- $\lfloor t/1000 \rfloor$ is the number of whole seconds in t, and
- $1000 \cdot (t \mod 1000)$ is the number of microseconds in $t - \lfloor t/1000 \rfloor$

Therefore, the following C code fragment sets a `timeval` structure's fields, given an integer number `m` of milliseconds

```
timeval t;
t.tv_sec = m / 1000;
t.tv_usec = ( m - t.tv_sec * 1000 ) * 1000;
```

This avoids a second division using the modulo operator.

The `getitimer()` and `setitimer()` system calls work with interval timers. The former gets a timer's current value and the latter sets a timer's value.

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
struct itimerval *ovalue);
```

The first parameter to both is an integer constant that specifies the type of timer, one of the constants, `ITIMER_REAL`, `ITIMER_VIRTUAL`, or `ITIMER_PROF`. The second parameter to `getitimer()` is a pointer to the `itimerval` structure to be filled with its current values. The `it_value` element of this structure is given the time remaining on the timer, not the time it was originally set to be.

The `setitimer()` function's second parameter, `value`, is the address of the `itimerval` structure with which to set the timer, and the third, `ovalue`, if it is not `NULL`, is the address of a structure to be filled with its current values.

To stop a timer, set the initial and repeat intervals to 0. If the repeat is 0 but the initial delay is not, the timer sends a single signal and then stops. If the initial value is 0 the timer never starts, no matter what the repeat interval is.

The function `set_timer()`, below, can be used to set the value of an interval timer, given a time value expressed in milliseconds. It has three parameters, the type of timer to set, the number of milliseconds in the initial delay, and the number of milliseconds in the repeat interval.

Listing 6.11: set_timer()

```
int set_timer( int which, long initial, long repeat )
{
    struct itimerval itimer;
    long secs;

    /* initialize initial delay */
    secs = initial / 1000 ;
    itimer.it_value.tv_sec     = secs;
    itimer.it_value.tv_usec    = (initial - secs*1000 ) * 1000 ;

    /* initialize repeat interval */
    secs = repeat / 1000 ;
    itimer.it_interval.tv_sec  = secs;
    itimer.it_interval.tv_usec = (repeat - secs*1000 ) * 1000 ;

    return setitimer(which, &itimer, NULL);
}
```

The demo program, `timerdemo.c`, demonstrates how this function can be used. It accepts command line arguments so that you can control the initial and repeat delays, and the signal handler is designed to simply count how many signals are received and to quit after a pre-specified number of signals.

Listing 6.12: timerdemo.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <signal.h>
#include "timers.h"

void    count_alarms(int);

int main(int argc, char* argv[])
{
    int initial = 250;  /* default value */
    int repeat  = 500;  /* default value */

    if (argc >= 3) {
        initial = atoi(argv[1]);
        repeat  = atoi(argv[2]);
    }
    if ( initial == 0 || repeat == 0 ) {
        printf("Setting either interval to 0 hangs the process.\n");
        printf("Bailing out...\n");
        return 0;
    }

    signal(SIGALRM, count_alarms);
    if ( set_timer(ITIMER_REAL, initial, repeat) == -1 )
        perror("set_timer");
    else
        while( 1 )
            pause();
    return 0;
}

void count_alarms(int signum)
{
    int alarmsaccepted = 10;
    static int count = 0;
    printf("alarm %d \n", ++count);
    fflush(stdout);
    if ( alarmsaccepted == count ){
        printf("No more alarms allowed!\n");
        exit(0);
    }
```

```
}
```

If you run this program with various values as arguments, you will see how it works.

6.10.3 How Timers Are Implemented in UNIX

There is only one system clock. In contrast, there are many processes, and more than one of these might have active timers. The kernel maintains a data structure containing the timers of all processes. With each tick of the system clock, the kernel decrements each of the currently active per-process timers. If a process's timer reaches 0, the kernel sends the appropriate signal to the process and copies the `it_interval` value into `it_value`, provided that `it_interval` is not 0, effectively resetting the timer. If the `it_interval` is 0, the timer is stopped.

6.10.4 Timer Limitations and Precautions

Each process can have one of each kind of timer: a real timer, a virtual timer, and a profile timer, but only one of each. Although both the seconds value and the microseconds value are used to set the timer parameters, most operating systems will not give a process an interval of that exact amount of time because these are not real-time timers and because the operating system typically uses a time resolution on the order of a few milliseconds, not microseconds. UNIX systems that conform to SVR4 and to 4.4BSD specs do guarantee, though, to generate a signal no sooner than the requested time interval. The signal's delivery may be delayed on very heavily loaded systems. In addition, if a system is very heavily loaded, it is even a possibility that a later signal may fail to be delivered because the signal from an earlier timer expiration has not yet been delivered and so the second will be lost.

6.11 Timers and Signals in Video Games

So far we have seen how to create the illusion of movement on the screen using the NCurses library by erasing, repositioning, and drawing the same object, with a small time delay between repeated drawing. The `animateface.c` program used the `nanosleep()` function to achieve this time delay because `nanosleep()` provided a small enough time resolution and does not interfere with `SIGALRM` signals. If we want a video game to be interactive, however, then it has to respond to user inputs while creating the illusion that the action on the screen is independent of the user's actions. The method used by `animateface.c` will not work because when the program is waiting in the `nanosleep()` call, it is unable to respond to user inputs.

Instead, we can let a timer run in the background. At regular intervals, it can interrupt the process by sending a `SIGALRM` signal. All of the functionality to update the drawing can be put into the signal handler for the `SIGALRM` signal. However, there are dangers with extrapolating these ideas to programs in general, as is explained below.

6.11.1 Cautions About Signal Handler Design

The signal generation and delivery mechanism is a complex system with many nuances, and the programmer must be aware of them and must design the handlers with utmost care. The first issue is with respect to potential race conditions within the handlers themselves.

Because signal handlers cannot have any parameters other than the signal number or the structures passed to it by the kernel in the case of the newer `sa_sigaction()` style handlers, the only way that they can share data with the rest of the program is through global variables. For example, if the `SIGALRM` signal handler has to update the position of an object on the screen, then the handler needs read and write access to a variable that stores the object's current position. This variable must be either a static variable within the handler, or a global variable in the program. It must be a global if the variable needs to be accessed by other parts of the program outside of the handler. In either case, the variable cannot reside on the runtime stack because if it did, it would be destroyed between invocations of the handler. If the variable's contents are destroyed between invocations of the handler, there will be no means of animating the object.

In general, using signal handlers that have to access either global data or data that is not on the stack is a dangerous thing. If handled correctly in our video programs, there is little risk, but if this same strategy is used for programs in general, it can lead to unreliable and insecure programs. It can open up a Pandora's box of problems associated with the possibility of race conditions within the handler itself. This is because the handler might be re-entered as a result of another signal arriving while the handler is active. For example, if the handler is registered with the `SA_NODEFER` flag set, then it can be interrupted in the middle of its execution and variables within the handler might be in an inconsistent state as a result. Still worse, under certain circumstances, intruders could find ways to send the appropriate signal sequences to the program to force it to core dump and could use these dumps to gain root access (see Zalewski [3]).

A second issue pertains to certain system calls and library functions. Certain system calls and library functions are marked as safe, and the rest are unsafe. If a signal handler makes a call to a library function or a system call, and another signal causes it to be re-entered (because the signal was not masked or blocked) during the time the handler is in the call, the second invocation of the signal handler may also enter that same function. If it does, then the function will be re-entered as well, by the same process. If this function is not safe, then the data state of the handler will be corrupted and its execution no longer predictable. For example, in the following handler

```
void sighandler(int signum)
{
    ....
    printf("Running with uid=%d euid=%d\n",getuid(),geteuid());
    ...
}
```

if a second signal arrives while the first is in the `printf()` function, then both invocations will be using the `printf()` code, which is not re-entrant, and hence not safe. This means that the output of `printf()` may be corrupted. Far worse scenarios can result, making a system vulnerable to attack. POSIX.1-2004 requires that the following functions can be safely called within a signal handler:

```
_Exit(), _exit(), abort(), accept(), access(), aio_error(), aio_return(),
aio_suspend(), alarm(), bind(), cfgetispeed(), cfgetospeed(), cfsetispeed(),
cfsetospeed(), chdir(), chmod(), chown(), clock_gettime(), close(), connect(),
creat(), dup(), dup2(), execle(), execve(), fchmod(), fchown(), fcntl(),
fdatasync(), fork(), fpathconf(), fstat(), fsync(), ftruncate(), getegid(),
geteuid(), getgid(), getgroups(), getpeername(), getpgrp(), getpid(),
```

```
getppid(), getsockname(), getsockopt(), getuid(), kill(), link(), listen(),
lseek(), lstat(), mkdir(), mkfifo(), open(), pathconf(), pause(), pipe(),
poll(), posix_trace_event(), pselect(), raise(), read(), readlink(), recv(),
recvfrom(), recvmsg(), rename(), rmdir(), select(), sem_post(), send(),
sendmsg(), sendto(), setgid(), setpgid(), setsid(), setsockopt(), setuid(),
shutdown(), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(),
sigismember(), signal(), sigpause(), sigpending(), sigprocmask(), sigqueue(),
sigset(), sigsuspend(), sleep(), socket(), socketpair(), stat(), symlink(),
sysconf(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetpgrp(),
tcsendbreak(),tcsetattr(), tcsetpgrp(), time(), timer_getoverrun(),
timer_gettime(), timer_settime(), times(), umask(), uname(), unlink(),
utime(), wait(), waitpid(), write().
```

POSIX.1-2008 removes `fpathconf()`, `pathconf()`, and `sysconf()` from the preceding list, and adds the following functions to it:

```
execl(), execv(), faccessat(), fchmodat(), fchownat(), fexecve(), fstatat(),
futimens(), linkat(), mkdirat(), mkfifoat(), mknod(), mknodat(), openat(),
eadlinkat(), renameat(), symlinkat(), unlinkat(), utimensat(), utimes()
```

In general, I/O functions are not safe to invoke inside signal handlers.

Non-reentrant functions are functions that cannot safely be called, interrupted, and then recalled before the first call has finished without resulting in memory corruption. An easy way to think of a function being re-entrant is that every single variable used by that function is stored on the run time stack, including any return value. It uses no static variables and no globals. Each time it is invoked, the new invocation has its own set of variables.

A signal handler would have to completely remove all possibility of its being interrupted if it contained a call to an unsafe function within it. This is not realistic. If an unsafe function is in the middle of execution when a signal arrives, and the handler for this signal also calls an unsafe function, then the result of execution becomes undefined, meaning all bets are off about what will happen. This is an even more compelling reason to avoid unsafe functions within handlers.

Three general rules to follow when designing signal handlers, whenever possible, as recommended by Wheeler[2], are:

1. Where possible, have your signal handlers unconditionally set a specific flag and do nothing else.

2. If you must have more complex signal handlers, use only calls specifically designated as being safe for use in signal handlers. In particular, don't use `malloc()` or `free()` in C (which on most systems aren't protected against signals), nor the many functions that depend on them (such as the `printf()` family and `syslog()`). You could try to "wrap" calls to insecure library calls with a check to a global flag (to avoid re-entry), but I wouldn't recommend it.

3. Block signal delivery during all non-atomic operations in the program, and block signal delivery inside signal handlers.

In addition to these recommendations, I would add one more:

- Use the `SA_RESTART` flag when possible, to avoid the possibility of system calls being interrupted and terminated, which may then cause the program to exit abnormally, and check the return value of all system calls when using signal handlers in a program.

We will not be able to adhere to rule 1 in some of our demo programs because they are designed to produce output or change program state during the handler calls to illustrate various principles, but we can stick to rules 2 and 3.

6.11.2 A Demonstration

We will develop a simple program to illustrate the first method of animation. Our program, called `bouncestr.c`, moves a string, in this case a worm-like fellow, horizontally across the screen, from left to right and then back again. It is called `bouncestr.c` because each time the poor guy hits the "wall", he bounces back in the opposite direction. The program allows the user to control the game with three different keys:

- Typing '`f`' speeds up the motion;
- Typing '`s`' slows down the motion;
- Typing a space character reverses the direction of the worm.

The *speed* is the number of character positions that our object will move each second. For example, a speed of 6 means that it moves 6 columns (i.e., characters) per second. When the user presses the `f` or `s` key, the speed should increase or decrease linearly, up to some reasonable limits. For example, if v is the current speed, then one press of `f` should mean $v = v + 2$. The changes in speed are handled by changing the intervals in the interval timer and resetting it with `setitimer()` whenever the user presses the '`f`' or '`s`' key.

This first method of animation will make the main program in charge of getting user input, and use timers and signal handlers to interrupt the main program and update the worm's position on the screen. Because timers will interrupt the main program loop whenever they occur, there is a good chance they will interrupt the `read()` system call that is invoked within the `getch()` code to get user input. For this reason, they must be established with the `SA_RESTART` flag, to restart these calls and not lose the user's input. There is no need to make the input non-blocking; doing so would waste needless CPU cycles asking the kernel if input is available. But line buffering should be disabled, as well as echo and line-editing. Therefore, the program will turn on `cbreak` mode and turn off echo.

The program needs one global variable:

```
int direction;
```

to store the direction of movement (left or right, by one cell), and the signal handler needs two static variables:

```
int row, col;
```

which store the current position at which to draw. The logic in the main program's loop handles the input events, as follows.

Listing 6.13: Main processing loop of bouncestr.c

```
while ( !done )  {
    is_changed = FALSE;
    c = getch();
    switch (c) {
        case 'Q':
        case 'q':
            done = 1;
            break;
        case ' ':
            direction = ( direction == LEFT)? RIGHT:LEFT;
            break;
        case 'f':
            if ( 1000/speed > 2 ) {          /* if interval > 2 */
                speed = speed + 2;           /* increase        */
                is_changed = TRUE;
            }
            break;
        case 's':
            if (1000/speed <=  500 ) {       /* if interval <= 500 */
                speed = speed - 2 ;          /* decrease           */
                is_changed = TRUE;
            }
            break;
    }
    if ( is_changed ) {
        set_timer( ITIMER_REAL, 1000/speed, 1000/speed );
    }
}
```

Notes

1. `is_changed` lets us know whether to reset the timer.
2. Blocking input is on, so the `getch()` can never return without data. The loop just has to check which character was typed.
3. Speed is the reciprocal of the interval length, in the same way that frequency is the inverse of period with a periodic function (like a wave). If we want a frequency of k signals every 1000 milliseconds, then the interval between each signal must be $1000/k$. Similarly, if we want an object to be moved k times each second (equivalently k times each 1000 ms), then the interval to give to the interval timer must be $1000/k$ ms. Since speed contains the current number of chars per second for moving the object, the interval to give to the timer is $1000/speed$, since the `set_timer()` function (defined in Listing 6.11) is expecting the interval expressed in milliseconds.

4. As a concrete example, if *speed* = 10, then the timer must expire every 1000/10 = 100 milliseconds, in order that we can move the object 10 times per second. If speed is increased by 2, to 12 chars/second, then the interval must be 1000/12 ≈ 83 milliseconds, so that the timer will expire every 83 milliseconds.

5. We make sure we avoid a division by zero, by preventing the user from decrementing speed below 1, which is achieved by making sure $1000/speed \leq 500$. We set an upper bound on the speed simply because high speeds are not easy to watch.

The logic of redrawing is now in the signal handler, `move_msg()`, shown below:

Listing 6.14: move_msg()

```
void move_msg(int signum)
{
    static int row = ROW;
    static int col = 0;
    char   mssge[40];
    mvaddstr( row, col, BLANK );  /* erase old string      */
    col += dir;                   /* advance one column    */
    move( row, col );             /* move to new locataion */
    if ( RIGHT == dir ) {
        addstr( MESSAGE );        /* add forward string    */
        if ( col+strlen(MESSAGE) >= COLS-1 )
            dir = LEFT;           /* reverse if hitting edge */
    }
    else {
        addstr( REVMSSGE );       /* add reverse string      */
        if ( col <= 0 )
            dir = RIGHT;          /* reverse if hitting edge */
    }

    move( LINES-1, 0);
    sprintf(mssge, "Current speed: %d (chars/sec)", speed);
    addstr(mssge);
    refresh();
}
```

Note that

- The handler uses static variables, a.k.a globals, making it non-re-entrant.
- It makes calls to several functions that are not safe.

However, because this is a `SIGALRM` handler, and the time intervals are extremely long relative to the length of the code, it is essentially impossible for a `SIGALRM` signal to be delivered while the handler is running. That is why it is effectively safe. Of course you can send it multiple Ctrl-C's and it will be unsafe for them.

If the signal handler were installed using the `signal()` system call, the handler would have to reset itself by calling `signal()` immediately. We use the `sigaction()` call instead. If the user were allowed to speed up the animation enough, the `SIGALRM` signals might arrive so fast that they would arrive before the handler has finished executing. In this case, the handler's behavior would

be unsafe. By using `sigaction()`, we can make sure that signals are blocked while the program is executing inside the handler.

The program without the main processing loop from Listing 6.13 is below.

Listing 6.15: Main program of bouncestr.c

```
#include  <stdio.h>
#include  <string.h>
#include  <curses.h>
#include  <signal.h>
#include  "timers.h"

#define     INITIAL_SPEED    50
#define     RIGHT            1
#define     LEFT             -1
#define     ROW              12
#define     MESSAGE          "ooooooo=>"
#define     REVMSSGE         "<=ooooooo"
#define     BLANK            "         "

int         dir;          /* Global variable to store direction of movement */
int         speed;        /* Current speed in chars/second */

int main()
{
    int    done;
    int    is_changed;
    int    c;

    /* Set up signal handling  */
    struct sigaction newhandler;       /* for installing handlers       */
    sigset_t         blocked;          /* to set mask for handler */

    newhandler.sa_handler = move_msg; /* name of handler  */
    newhandler.sa_flags = SA_RESTART; /* flag is just RESTART */
    sigemptyset(&blocked);            /* clear all bits of blocked set */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */

    if ( sigaction(SIGALRM, &newhandler, NULL) == -1 ){  /* try to install */
        perror("sigaction");
        return (1);
    }

    /* Prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
    cbreak();       /* turn off line buffering and editing */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    dir    = RIGHT;
    done   = 0;
    speed  = INITIAL_SPEED ;
```

```
    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    /* main processing loop omitted but would be here */

    endwin();
    return 0;
}
```

6.12 Non-polling Input

The `bouncestr.c` program uses a timer to generate interrupts to update the screen, but it obtains the user's input through what is essentially a polling loop: the main program repeatedly polls the terminal for input. This is fine if the CPU is not going to be used by any other process or if the program does not have other tasks to perform in the main loop, but it is an inefficient method of checking for the availability of input, which is extremely infrequent in the life of a processor. We should be unsatisfied with the idea that our program is a CPU hog, stuck in a polled I/O loop, even if the process is blocked each time it calls `getch()` to check for user input. The process basically calls `getch()`, blocks, is awakened when the user types, does a bit of work and blocks again, over and over. It would be more efficient if the input part of the program were also signal-driven, meaning that the program would ask the kernel to notify it when input could be delivered to it, perhaps through the signal-handling mechanism. In this case, the program would be essentially idle, waiting for a signal of any kind, either from the timer to update the screen, or from the kernel because input was available.

There are two different types of non-polling input: *signal-driven* and *asynchronous*. To understand the difference between them, it is important to know that input is first moved from a device to a buffer in the kernel's address space, and from there to the process's address space.

- In signal-driven I/O, the program tells the kernel to notify it when input has been placed into the kernel's address space. Once the process is notified that the input is in the kernel's address space, if it makes a `read()` system call, because the data is immediately available, it will not block. In other words, a `read()` executed after the process is notified is guaranteed to return immediately with data.
- In asynchronous I/O, the process tells the kernel to notify it when input has been moved from the device to the kernel's address space and then into a buffer in the process's address space. In this type of I/O, when the process receives the signal, the `read()` has already been executed, and the user process has the data already, but not necessarily in the memory location into which it must go.

Signal-driven I/O is available in UNIX by setting the `O_ASYNC` flag in the file descriptor and then establishing appropriate signals. Asynchronous I/O is available through the POSIX Asynchronous I/O Interface (AIO). It is a bit confusing that the flag to enable signal-driven I/O is called `O_ASYNC`. We will first explore signal-driven I/O by modifying the `bouncestr.c` program. Then we will create a version of the `bouncestr.c` program that uses asynchronous I/O with the AIO interface.

6.12.1 Non-polling I/O Using the `O_ASYNC` Flag

When you set the `O_ASYNC` flag on a file descriptor, it causes input from the descriptor's file connection to be partially delivered asynchronously. To be precise, it means that when input is available on the device, it is copied by the kernel into a location in the kernel's address space, after which the kernel sends a `SIGIO` signal to the process. To set up signal-driven input by this method, the program must do the following:

1. Tell the kernel which process should be sent the `SIGIO` signal when the data is ready to read by calling

   ```
   fcntl(SETOWN, getpid());
   ```

   The `SETOWN` operation makes the process-id in the second argument the owner of the signal to be received. Usually the program wants to receive the signal itself, so it calls this with `getpid()`.

2. Retrieve the existing flags on the standard input device with

   ```
   fcntl(0, F_GETFL);
   ```

3. Set the `O_ASYNC` flag on the connection with

   ```
   fcntl(0,F_SETFL,(fd_flags|O_ASYNC));
   ```

4. Assuming that `on_input()` is the function that will handle the `SIGIO` signal, register that signal handler:

   ```
   struct sigaction newhandler;
   sigset_t         blocked;
   newhandler.sa_handler = on_input;
   newhandler.sa_flags = SA_RESTART;
   sigemptyset(&blocked);
   newhandler.sa_mask = blocked;
   sigaction(SIGIO, &newhandler, NULL);
   ```

   The `on_input()` handler can call the NCurses `getch()` function and will be guaranteed to receive the single character input by the user. This way it does not have to be in a loop doing a blocking read and can instead do other things in the loop.

This is all put together in the program `bouncestr_async.c`. The first three of the above steps can be put into a function called `enable_keybd_signals()`:

```
void enable_keybd_signals()
{
    int   fd_flags;

    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));
}
```

6.12.2 The `bouncestr.c` Program Using `O_ASYNC`: Flawed Version

Sometimes it is worth writing a bad program in order to understand how to write a good program. This is such an exercise. The program (excluding the `#includes` and parts that are identical to the `bouncestr.c` program's) is shown below. The program is terminated within the `on_input()` handler when it receives the quit input character. This is because, once the program has started, it cannot be terminated by turning off the timer or by setting the control variable of the loop to 1. This will be explained later.

The tasks of the main program are:

1. Establish the signal handlers.
2. Initialize NCurses (in `cbreak` mode with no echo).
3. Initialize the data state of the program (speed, direction, rows, columns, etc)
4. Set up keyboard signals.
5. Start the interval timer.
6. Display the messages on the last line and loop until it is time to quit.

The program:

Listing 6.16: A flawed bouncestr_async.c

```
...
/*  <---- snip ---->  */
int         row;            /* current row           */
int         col;            /* current column         */
int         dir;            /* Global variable to store direction of movement */
int         speed;          /* Current speed in chars/second */
volatile    sig_atomic_t    finished;

void     on_alarm(int);         /* handler for alarm   */
void     on_input(int);         /* handler for SIGIO   */
void     enable_kbd_signals(); /* setup for SIGIO     */

int main( int argc, char * argv[])
{
    struct sigaction newhandler;       /* for installing handlers       */
    sigset_t         blocked;          /* to set mask for handler */

    /* Set up signal handling  */
    newhandler.sa_handler = on_input; /* name of handler  */
    newhandler.sa_flags = SA_RESTART; /* flag is just RESTART */
    sigemptyset(&blocked);            /* clear all bits of blocked set */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */
    if ( sigaction(SIGIO, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        return (1);
    }
```

```
    sigemptyset(&blocked);              /* clear all bits of blocked set */
    sigaddset(&blocked, SIGIO);
    newhandler.sa_mask = blocked;       /* set this empty set to be the mask */
    newhandler.sa_handler = on_alarm; /* SIGALRM handler function */
    if ( sigaction(SIGALRM, &newhandler, NULL) == -1 ){  /* try to install */
        perror("sigaction");
        return (1);
    }

    /* Prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
    cbreak();       /* put terminal into non-blocking input mode */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    row         = ROW;
    col         = 0;
    dir         = RIGHT;
    finished    = 0;
    speed       = INITIAL_SPEED ;

    /* Turn on keyboard signals */
    enable_kbd_signals();

    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    mvaddstr(LINES-1, 0, "Current speed:");
    refresh();

    /* Put the message into the first position and start */
    mvaddstr(row, col,   MESSAGE);

    while( 0 == finished  ) {
        pause();
    }
    endwin();
    return 0;
}

void on_input(int signum)
{
    int     c;
    int     is_changed = 0;
    char    mssge[40];

    c = getch();
    switch (c) {
        case 'Q':
        case 'q':
            finished = 1;                   /* quit program */
            clear();
```

```
            endwin();
            /* exit(0);  UNCOMMENT THIS IF YOU WANT IT TO WORK!!! */
            break;
        case ' ':
            dir = (LEFT == dir)? RIGHT:LEFT;  /* reverse direction */
            break;
        case 'f':
            if ( 1000/speed > 2 ) {           /* if interval > 2 */
                speed = speed + 2;            /* increase  */
                is_changed = 1;
            }
            break;
        case 's':
            if (1000/speed <  500 ) {         /* if interval <= 500 */
                speed = speed - 2 ;           /* decrease */
                is_changed = 1;
            }
            break;
    }
    if ( is_changed ) {
        set_timer( ITIMER_REAL, 1000/speed, 1000/speed );
        sprintf(mssge, "Current speed: %d (chars/sec)", speed);
        mvaddstr(LINES-1, 0, mssge);
    }

    move( LINES-1, COLS-12);
    sprintf(mssge, "Last Char:%c", c);
    addstr(mssge);
    refresh();
}


void on_alarm(int signum)
/* same as in bouncestr.c, and so omitted here  */

void enable_keybd_signals()
{
    int  fd_flags;

    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));
}
```

Notes.

1. The biggest difference between this and the `bouncestr.c` program is that the input handling is entirely inside the `on_input()` handler.

2. The `enable_keybd_signals()` function sets up the asynchronous input on file descriptor 0.

3. This program must call `exit()` from within the handler, otherwise it will never terminate. If you modify the `on_input()` handler so that when a '`q`' is typed, all it does is to set the `finished` flag

to 1, the program will not stop. In fact, the main program will continue to see the value 0 stored in `finished`. You can go one step further and delete the main loop completely, and the program will animate forever. In other words, the signal handler for `SIGALRM` continues to run and the `endwin()` call is never reached. This problem is not related to NCurses, nor to the timers.

The problem is that, as the man page for `fcntl()` notes, " a `SIGIO` signal is sent whenever input or output becomes possible on that file descriptor." From various experiments I have carried out, I have determined that the problem is that, when the terminal is in non-canonical mode and signal-driven input has been set up on the input descriptor of the terminal (file descriptor 0), if the main program or the `SIGIO` handler attempts output on the terminal device, it corrupts the `SIGIO` signal mechanism so that when the `SIGIO` handler terminates, instead of returning to the main program, execution will resume in the handler again, as if the `SIGIO` signal was not cleared from the process's state. So the process continues to execute only in the input handler and the `SIGALRM` handler if it has not been blocked. If one removes all output instructions of any kind from the signal handler and the main program to "slow devices", meaning the screen, then the program will work correctly. The behavior of programs that issue writes within the handler or the main program to the screen is apparently undefined.

4. The signal handler for the `SIGIO` must not block `SIGALRM`, or else the animation will disappear. The `SIGALRM` handler can block `SIGIO` signals though.

What follows is a better version of this same program, also using a `SIGIO` signal handler that adheres to all safety rules noted above, and that works correctly.

6.12.3 The `bouncestr.c` Program Using `O_ASYNC` : A Proper Solution

In this version, all code has been removed from the `SIGIO` signal handler except to set the value of a state variable of type `volatile sig_atomic_t` that the main program checks. According to the *CERT Secure Programming Standard, SIG31* [1],

> Accessing or modifying shared objects in signal handlers can result in race conditions that can leave data in an inconsistent state. The exception to this rule is the ability to read and write to variables of `volatile sig_atomic_t`. The need for the `volatile` keyword is described in rule DCL34-C. Use `volatile` for data that cannot be cached. It is important to note that the behavior of a program that accesses an object of any other type from a signal handler is undefined.
>
> The type `sig_atomic_t` is the integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts. The type of `sig_atomic_t` is implementation defined, though it provides some guarantees. Integer values ranging from `SIG_ATOMIC_MIN` through `SIG_ATOMIC_MAX`, inclusive, may be safely stored to a variable of the type.

Further details can be found on the CERT website or in the cited reference.

In our program, if the state variable is set, then the main loop calls `getch()` to get the input, and then calls a function to process the input. Otherwise it blocks itself on `pause()`. The listing follows.

Listing 6.17: bouncestr_async2.c: A safe version of bouncestr_async.c

```
// #includes omitted here
```

```
#define     INITIAL_SPEED    30
#define     RIGHT            1
#define     LEFT             -1
#define     ROW              12
#define     MESSAGE          "ooooooo=>"
#define     REVMSSGE         "<=ooooooo"
#define     BLANK            "         "

int         dir;            /* Global variable to store direction of movement */
int         speed;          /* Current speed in chars/second */
volatile    sig_atomic_t    input_ready;

/************************************************************************/
/*                      Signal Handler Prototypes                       */
/************************************************************************/
void  on_alarm(int);                                    /* handler for alarm   */
void  on_input(int);                                    /* handler for keybd   */

/* This is not a signal handler -- it consolidates logic for updating */
int   update_from_input(int c,int *speed,int *dir );

/************************************************************************/
/*                                  Main                                */
/************************************************************************/
int main( int argc, char * argv[])
{

    struct sigaction newhandler;        /* for installing handlers */
    sigset_t         blocked;           /* to set mask for handler */
    int     fd_flags;
    int     c;
    int     finished;

    /* Set up signal handling  */
    newhandler.sa_handler = on_input; /* name of handler                  */
    newhandler.sa_flags = SA_RESTART; /* flag is just RESTART             */
    sigemptyset(&blocked);            /* clear all bits of blocked set    */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */
    if ( sigaction(SIGIO, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        return (1);
    }

    sigemptyset(&blocked);            /* clear all bits of blocked set */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */
    newhandler.sa_handler = on_alarm; /* SIGALRM handler function */
    if ( sigaction(SIGALRM, &newhandler, NULL) == -1 ){  /* try to install */
        perror("sigaction");
        return (1);
    }

    /* Prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
```

```
    cbreak();       /* put terminal into non-blocking input mode */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    dir         = RIGHT;
    finished    = 0;
    speed       = INITIAL_SPEED ;
    input_ready = 0;

    /* Turn on keyboard signals */
    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));

    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    /* Put a message in bottom row with current speed. */
    mvaddstr(LINES-1, 0, "Current speed:");

    /* Put the message into the first position and start */
    mvaddstr(ROW, 0,   MESSAGE);

    while( !finished  ) {
        if ( input_ready ) {
            c = getch();
            finished = update_from_input(c, &speed, &dir);
            input_ready = 0;
        }
        else
            pause();
    }
    clear();
    endwin();
    return 0;
}

/*****************************************************************************/

int update_from_input( int c, int *speed, int *dir )
{
    int is_changed = 0;
    char  mssge[40];
    switch (c) {
        case 'Q':
        case 'q':
            return 1;                          /* quit program */
        case ' ':
            *dir = (LEFT == *dir )? RIGHT:LEFT;  /* reverse direction */
            break;
        case 'f':
            if ( 1000/(*speed) > 2 ) {           /* if interval > 2 */
```

```
                *speed = *speed + 2;            /* increase  */
                is_changed = 1;
            }
            break;
        case 's':
            if (1000/(*speed) <  500 ) {        /* if interval <= 500 */
                *speed = *speed - 2 ;           /* decrease */
                is_changed = 1;
            }
            break;
    }
    if ( is_changed ) {
        set_timer( ITIMER_REAL, 1000/(*speed), 1000/(*speed) );
        sprintf(mssge, "Current speed: %d (chars/sec)", (*speed));
        mvaddstr(LINES-1, 0, mssge);
    }

    move( LINES-1, COLS-12);
    sprintf(mssge, "Last Char:%c", c);
    addstr(mssge);
    refresh();
    return 0;
}

/*****************************************************************************/

void on_input(int signum)
{
    input_ready = 1;
}

/*****************************************************************************/

void on_alarm(int signum)
{
    static int row = ROW;
    static int col = 0;

    mvaddstr( row, col, BLANK );  /* erase old string */
    col += dir;                   /* advance one column  */
    move( row, col );             /* move to new locataion */
    if ( RIGHT == dir ) {
        addstr( MESSAGE );        /* add forward string  */
        if ( col+strlen(MESSAGE) >= COLS-1 )
            dir = LEFT;           /* reverse if hitting edge */
    }
    else {
        addstr( REVMSSGE );       /* add reverse string        */
        if ( col <= 0 )
            dir = RIGHT;          /* reverse if hitting edge */
    }
    refresh();
}
```

6.12.4 The `bouncestr.c` Program Using AIO

The AIO interface is a POSIX interface that provides asynchronous I/O. Whereas setting the `O_ASYNC` flag on a file descriptor causes a signal to be sent when data is available to be read, using the AIO interface causes a signal to be sent when the data has actually been read and placed into a user buffer. The `aio_read()` call is an asynchronous read. In essence, it places a read request in the I/O device driver's queue and returns immediately, as indicated in the man page:

```
#include <aio.h>
int aio_read(struct aiocb *aiocbp);

The  aio_read()  function  requests  an asynchronous
     "n = read(fd, buf, count)"
with   fd,   buf,   count   given   by    aiocbp->aio_fildes, aiocbp->aio_buf,
aiocbp->aio_nbytes, respectively.  The return status n can be retrieved upon
completion using aio_return(3).

The data is read starting at the absolute file offset aiocbp->aio_offset,
regardless of the current file position.  After this request, the value of
the current file position is unspecified.

The "asynchronous" means that this call returns as soon as the  request has
been enqueued; the read may or may not have completed when the call returns.
One tests for completion using aio_error(3).
```

When the request is satisfied, the driver sends a `SIGIO` signal. The signal handler can process the input and then issue a new `aio_read()` call to get more data.

The program must

- create a buffer to store the input data, and
- fill an `aiocb` structure with appropriate values before issuing the first read.

An `aiocb` structure has the following members:

```
int             aio_fildes      //File descriptor.
off_t           aio_offset      // File offset.
volatile void  *aio_buf         //Location of buffer.
size_t          aio_nbytes      //Length of transfer.
int             aio_reqprio     //Request priority offset.
struct sigevent aio_sigevent    // Signal number and value.
int             aio_lio_opcode  //Operation to be performed.
```

A program does not have to assign a value to the `aio_reqprio` member, but all others must be initialized. The following function, `setup_aio_buffer()`, demonstrates how to set up a read of a single character at a time into a buffer named `input`. It is given a pointer to an `aiocb` structure and fills its members with the required data. The main program can then give the address of this structure to the `aio_read()` function.

Listing 6.18: setup_aio_buffer()

```
void setup_aio_buffer(struct aiocb *aio_buf)
{
    static char input[1];                   /* 1 char of input  */

    /* describe what to read  */
    aio_buf->aio_fildes     = 0;      /* file descriptor for I/O */
    aio_buf->aio_buf        = input;  /* address of buffer for I/O          */
    aio_buf->aio_nbytes     = 1;      /* number of bytes to read  each time */
    aio_buf->aio_offset     = 0;      /* offset in file to start reads   */

    /* describe what to do when read is ready  */
    aio_buf->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    aio_buf->aio_sigevent.sigev_signo  = SIGIO;       /* send SIGIO    */
}
```

The main program should declare the `aiocb` structure so that it is visible to the various functions that must access it. The program follows.

Listing 6.19: bouncestr_aio.c

```
#include  <unistd.h>
#include  <stdio.h>
#include  <curses.h>
#include  <signal.h>
#include  <string.h>
#include  <aio.h>
#include  "timers.h"

#define    INITIAL_SPEED    30
#define    RIGHT            1
#define    LEFT             -1
#define    ROW              12
#define    MESSAGE          "ooooooo=>"
#define    REVMSSGE         "<=ooooooo"
#define    BLANK            "         "

int       dir;            /* Global variable to store direction of movement */
int       speed;          /* Current speed in chars/second */
volatile  sig_atomic_t  input_ready;

struct aiocb kbcbuf;        /* an aio control buf    */

void move_msg(int signum);   /* handler for alarm     */
void on_input(int);          /* handler for keybd     */
int  update_from_input( int *speed, int *dir );


/*****************************************************************************/
/*                        Signal Handler Prototypes                          */
/*****************************************************************************/

/* SIGALRM signal handler -- it is responsible for animating the string */
void    move_msg(int);
```

```
/* SIGIO signal handler -- it is responsible for retrieving user input  */
void    setup_aio_buffer(struct aiocb *aio_buf);

/*************************************************************************/
/*                                  Main                                 */
/*************************************************************************/

int main(int argc, char* argv[])
{
     struct sigaction newhandler;           /* new settings         */
     sigset_t         blocked;              /* set of blocked sigs  */
     int              finished;

     newhandler.sa_handler = on_input;      /* handler function     */
     newhandler.sa_flags = SA_RESTART;      /* options      */

     /* then build the list of blocked signals  */
     sigemptyset(&blocked);                 /* clear all bits       */
     newhandler.sa_mask = blocked;          /* store blockmask      */
     if ( sigaction(SIGIO, &newhandler, NULL) == -1 )
         perror("sigaction");

     newhandler.sa_handler = move_msg;      /* handler function     */
     if ( sigaction(SIGALRM, &newhandler, NULL) == -1 )
         perror("sigaction");

    /* prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
    cbreak();       /* put terminal into non-blocking input mode */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    dir        = RIGHT;
    finished   = 0;
    speed      = INITIAL_SPEED ;

    /* initialize aio buffer for the first read and place call */
    setup_aio_buffer(&kbcbuf);
    aio_read(&kbcbuf);

    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    mvaddstr(LINES-1, 0, "Current speed:");
    refresh();

    /* Put the message into the first position and start */
    mvaddstr(ROW, 0,   MESSAGE);

    while( !finished )
        if ( input_ready ) {
            finished = update_from_input(&speed, &dir);
```

```
            input_ready = 0;
        }
        else
            pause();
    clear();
    endwin();
    return 0;
}

/*****************************************************************************/
/*                              SIGIO Signal Handler                         */
/*****************************************************************************/

void on_input(int signo)
{
    input_ready = 1;
}


/*  Handler called when aio_read() has stuff to read */
/*  First check for any error codes, and if ok, then get the return code */

int update_from_input( int *speed, int *dir )
{
    int  c;
    int is_changed = 0;
    char *cp = (char *) kbcbuf.aio_buf;      /* cast to char *  */
    char   mssge[40];
    int finished=0;

    /* check for errors  */
    if ( aio_error(&kbcbuf) != 0 )
        perror("reading failed");
    else
        /* get number of chars read  */
        if ( aio_return(&kbcbuf) == 1 ) {
            c = *cp;
            /*ndelay = 0; */
            switch (c) {
                case 'Q':
                case 'q':
                    finished = 1;                    /* quit program */
                    break;
                case ' ':
                    *dir = (*dir == LEFT)? RIGHT:LEFT;  /* reverse direction */
                    break;
                case 'f':
                    if ( 1000/(*speed) > 2 ) {         /* if interval > 2 */
                        *speed = *speed + 2;           /* increase  */
                        is_changed = 1;
                    }
                    break;
                case 's':
                    if (1000/(*speed) <  500 ) {       /* if interval < 500 */
```

```
                        *speed = *speed - 2 ;          /* decrease */
                        is_changed = 1;
                    }
                    break;
                }
            if ( is_changed ) {
                set_timer( ITIMER_REAL, 1000/(*speed), 1000/(*speed) );
                sprintf(mssge, "Current speed: %d (chars/sec)", *speed);
                mvaddstr(LINES-1, 0, mssge);
            }
            /* write the status line message */
            move( LINES-1, COLS-12);
            sprintf(mssge, "Last Char:%c", c);
            addstr(mssge);
            refresh();
        }
    /* place a new request  */
    aio_read(&kbcbuf);
    return finished;
}

/*****************************************************************************/
/*                            SIGALRM Signal Handler                         */
/*****************************************************************************/

/* SIGALRM handler -- moves string on the screen when the signal is received */
void move_msg(int signum)
{
    static int row = ROW;
    static int col = 0;
    mvaddstr( row, col, BLANK );  /* erase old string */
    col += dir;                   /* advance one column  */
    move( row, col );             /* move to new locataion */
    if ( RIGHT == dir ) {
        addstr( MESSAGE );        /* add forward string  */
        if ( col+strlen(MESSAGE) >= COLS-1 )
            dir = LEFT;           /* reverse if hitting edge */
    }
    else {
        addstr( REVMSSGE );       /* add reverse string        */
        if ( col <= 0 )
            dir = RIGHT;          /* reverse if hitting edge */
    }
    refresh();
}

/*****************************************************************************/
/*                        Asynchronous I/O Library Setup                     */
/*****************************************************************************/

/*  The following function initializes the AIO structure to enable */
/*  asynchronous I/O through the AIO library. */
void setup_aio_buffer(struct aiocb *aio_buf)
/* Same as in Listing above */
```

6.12.5 Simulating Multiple Timers

Even though a process can have only a single timer, it is still possible to animate an unlimited number of independently moving objects. The key is to simulate in the program exactly what the kernel does with its timers relatively to the system clock. The idea is to create an array of the objects to be animated. Each entry of the array can have all of the information needed to animate a single object, and in particular, the length of the interval between movements of that object. In the world of animation, these movable creatures are called *sprites*, so I will call them that here.

In essence, a sprite can be represented by a structure such as the one below.

```
struct  sprite
{
    int      interval;       // number of time units between redraws
    int      counter;        // counter for elapsed time between redraws
    char     shape;          // shape used to draw object
    position display_pos;    // current location on screen (int,int)
    position real_pos;       // current real location  (double, double)
    double   dx;             // current x-coordinate of direction
    double   dy;             // current y-coordinate of direction
};
```

For each tick of the process's single interval timer, it can iterate through an array of sprites, decrementing each counter. If any counter reaches 0, it copies the interval value into it and issues a request to move the sprite in the (`dx,dy`) direction from position `real_pos`. The particular implementation above uses two positions, a real position and a display position. The idea is to keep track of the actual position as a floating point value, and display it in the cell in which its center of mass resides. The real position is updated by the (`dx,dy`) value and then the display position is calculated from that. The (`dx,dy`) pair is a vector of length 1 that is added to the real position of the point. The display cell is obtained by rounding the `x` and `y` values to the nearest integer. Moving a sprite can be accomplished with the following function.

Listing 6.20: move_sprite()

```
void move_sprite(sprite *sp)
{
    erase_sprite(*sp);

    sp->real_pos.y += sp->dy;
    sp->real_pos.x += sp->dx;
    sp->display_pos.r = (int)( sp->real_pos.y + 0.5);
    sp->display_pos.c = (int)( sp->real_pos.x + 0.5);
    draw_sprite(*sp);

    if  ( ( sp->real_pos.y > LINES-0.5 ) && ( sp->dy > 0 ) )
        sp->dy = -sp->dy;
    else if ( ( sp->real_pos.y < 0.0 ) && ( sp->dy < 0 ) )
        sp->dy = -sp->dy;
    if ( ( sp->real_pos.x > COLS-0.5 ) && ( sp->dx > 0 ) )
```

```
        sp->dx = -sp->dx;
    else if ( ( sp->real_pos.x < 0.5 ) && ( sp->dx < 0 ) )
        sp->dx = -sp->dx;
}
```

An unsafe `SIGALRM` signal handler would be as follows:

```
void update_all(int signum)
{
    int k;
    for ( k = 0; k < NUM_OBJS; k++ )
        if ( --object[k].counter == 0 ){
            move_sprite(&(object[k]));
            object[k].counter = object[k].interval;
        }
    move(LINES-1, COLS-1);
    refresh();
}
```

The rest of this program is relatively easy to piece together.

6.12.6 Summary

This chapter introduced the NCurses library as a means for controlling the user's terminal in a simpler and more powerful way than was possible by modifying terminal driver attributes. It barely scratched the surface of the library's interface. It also introduced timers as a way of introducing timed events and motion. Finally, it introduced several different models of input/output, including signal-driven and asynchronous I/O, an well as several different models of terminal processing, such as raw, cbreak, and non-canonical mode.

More efficient processing and better control can be achieved by the use of multiple processes. This is the topic of the next chapter.

Bibliography

[1] Robert C. Seacord and Jason A. Rafail. The cert c secure coding standard, 2008.

[2] David A. Wheeler. Secure programming for linux and unix howto, 2003.

[3] Michal Zalewski. Delivering signals for fun and profit, 2001.

Chapter 7 Process Architecture and Control

Concepts Covered

Memory architecture of a process
Memory structures
Viewing memory layout
Process structure
Executable file format
Process creation
Process synchronization
file, nohup, pgrep, ps, psg, readelf, strings
API: atexit, brk, sbrk, malloc, calloc, fork, execve, execlp, ..., exit, on_exit, vfork, wait, waitpid, waitid

7.1 Introduction

In a typical operating systems course, a process is defined to be a program in execution, or something similar to that. This is a true and accurate abstraction. A program such as the bash shell can have many, many instances running on a multi-user machine and each individual instance is a separate and distinct process, although each and every one of these is executing the exact same executable file. What this definition does not tell you is what a process is in concrete terms. It is like saying that a baseball game is an instance of the implementation of a set of rules created by Alexander Cartwright in 1845 by which two teams compete against each other on a playing field. Neither definition gives you a mental picture of the thing being defined.

In this chapter, we focus on the concrete representation of a process in UNIX: how it is represented within the kernel, what kinds of resources it requires, how those resources are materialized and managed, what attributes it has, and what system calls are related to its control and management. As a first step we will look at processes from the command level. Afterward, we will look at how UNIX systems arrange their address spaces and manage them in virtual and physical memory. Then we will look at how processes are created and how they communicate and synchronize with each other.

7.2 Examining Processes on the Command Line

The `ps` command is used for viewing one or more processes currently known to the operating system. I say "currently known" as opposed to "active" because the list may include processes that are technically not active, so called *zombie* processes. The set of options available for the `ps` command is very system dependent. There were different versions of it in BSD systems and in Version 7, and then there are options added by GNU. RedHat Linux systems support all of the historical options, and so there are many different ways to use this command in Linux. In Linux, users also have the option of running the top command. The `top` command is very much like `ps`, except that it displays the dynamic, real-time view of the state of the system. It is like the Windows task manager, and also like a terminal-based version of the Gnome System Monitor. Here we will describe the standard syntax rather than the BSD style or GNU syntax.

The `ps` command without options displays the list of processes of the login id that executes it, in a short form:

```
$ ps
  PID TTY      TIME CMD
14244 pts/1    00:00:00 bash
14572 pts/1    00:00:00 ps
```

You will notice that it always contains a line for the command itself because it itself has to run to do its job, obviously. The `-f` option causes it to display a full listing:

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
sweiss    2508  2507  0 12:09 pts/8    00:00:00 -bash
sweiss    3132  2508  0 12:22 pts/8    00:00:00 ps -f
```

The `UID` column is the user login name. The `PID` column is the process id of the process. The `PPID` column is the parent process's process id. The `C` field is rarely of interest; it gives processor utilization information. The `STIME` field is the starting time in hours, minutes, and seconds. The `TIME` column is the cumulative execution time, which for short commands will be zero, since the smallest unit of measurement in this column is seconds. The `CMD` column is the command line being executed by this process; in this case there are two: `bash` and "`ps -f`". All command line arguments are displayed; if any are suppressed, the command will appear in square brackets:

```
root      3080     1  0 Jan29 ?        00:00:00 [lockd]
```

The `TTY` column is the controlling terminal attached to this process. Some processes have no terminal, in which case a "`?`" will appear.

The `-e` option displays all processes, which will be quite long. If I want to know which process is the parent of my `bash` process, I can use `ps -ef` and filter using `grep`:

```
$ ps -ef | grep  2507
sweiss    2507  2504  0 12:09 ?        00:00:00 sshd: sweiss@pts/8
sweiss    2508  2507  0 12:09 pts/8    00:00:00 -bash
sweiss    3207  2508  0 12:30 pts/8    00:00:00 grep 2507
```

From this output you see that I am connected via `ssh` on pseudo-terminal `pts/8` and that the `ssh` daemon `sshd` is the parent of my `bash` process.

You can learn a lot about a system just by running `ps`. For example, on our Linux system, the first few processes in the system are:

```
$ ps -ef | head -4
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Jan29 ?        00:00:01 init [5]
root         2     1  0 Jan29 ?        00:00:02 [migration/0]
root         3     1  0 Jan29 ?        00:00:00 [ksoftirqd/0]
```

whereas on our Solaris 9 server, the list is:

```
$ ps -ef | head -4
UID   PID  PPID  C    STIME TTY      TIME CMD
root    0     0  0   Mar 13 ?        0:23 sched
root    1     0  0   Mar 13 ?        0:00 /etc/init -
root    2     0  0   Mar 13 ?        0:00 pageout
```

Notice that in Solaris, the (CPU) process scheduler itself is the very first process in the system. It is absent in Linux. In all UNIX systems, the process with PID 1 is always init. In Solaris, the `pageout` process is responsible for writing pages to disk, and `fsflush` flushes system buffers to disk.

The `-u` and `-U` options are useful for viewing all of your processes or those of others in a supplied user list. The list of users must be comma-separated, with no intervening spaces. For example:

```
$ ps -f -U sweiss,wgrzemsk
UID        PID  PPID  C STIME TTY          TIME CMD
sweiss    2507  2504  0 12:09 ?        00:00:00 sshd: sweiss@pts/8
sweiss    2508  2507  0 12:09 pts/8    00:00:00 -bash
wgrzemsk  2572  2570  0 12:10 ?        00:00:00 sshd: wgrzemsk@notty
wgrzemsk  2575  2573  0 12:10 ?        00:00:00 /bin/sh
```

While there are dozens of other options, I will only mention one more: the `-o` option. You can customize the output of the `ps` command to include any of the dozens of attributes available to be displayed using `-o`. The man page gives the general format for this. Some examples from the man page:

```
ps -eo pid,tid,class,rtprio,ni,pri,psr,pcpu,stat,wchan:14,comm
ps axo stat,euid,ruid,tty,tpgid,sess,pgrp,ppid,pid,pcpu,comm
ps -eopid,tt,user,fname,tmout,f,wchan
```

Note that there are no spaces in the list. In general never use spaces in any of the lists because the shell will then treat them as separate words rather than a single word to be passed to the `ps` command itself.

A related command is `pgrep`. If you need the process id of a command or program that is running, typing `pgrep` <*executable name*> will give you a list of processes running that program, one per line. For example

```
$ pgrep bash
2508
3502
3621
```

showing that three instances of `bash` are running, with pids 2508, 3502, and 3621.

7.3 Process Groups

UNIX systems allow processes to be placed into groups. There are several reasons for grouping processes. One is that a signal can be sent to an entire process group rather than a single process. For example, the shell arranges that all processes created in order to carry out the command line are in a single group, so that if the user needs to terminate that command, a single signal sent via `Ctrl-C` will kill all processes in the group. The alternative would require using the `ps` command to find all processes that were created to carry out the command.

Every process has a *process group-id* (of type `pid_t`). There is a single process in each group that is considered to be the *leader* of the group. It can be identified easily, because it is the only process whose *process group-id is the same as its process-id.* You can view the process group-id of a process in the output of `ps` by using the `-o` option and specifying the format in either AIX format, such as

```
ps -o'%U %p %P %r %C %x %y %a'
```

or in standard format, as in

```
ps -ouser,pid,ppid,pgrp,%cpu,cputime,tty,args
```

If you run the command

```
$ cat | sort -u | wc
```

and then view the processes using one of the above `ps` commands, you will see the group formation:

```
$ psg -u sweiss | egrep 'TIME|cat|sort|wc'
USER PID PPID PGID %CPU TIME TTY COMMAND
sweiss 17198 17076 17198 0.0 00:00:00 pts/2 cat
sweiss 17199 17076 17198 0.0 00:00:00 pts/2 sort -u
sweiss 17200 17076 17198 0.0 00:00:00 pts/2 wc
```

Notice that the `cat` command's process group-id (`pgid`) is the same as its process-id (`pid`) and that the three processes belong to the same group. If the full listing were displayed you would see that no other process is in this group.

7.4 Foreground and Background Processes

UNIX allows processes to run in the *foreground* or in the *background.* Processes invoked from a shell command line are *foreground processes*, unless they have been explicitly placed into the background by appending an ampersand '`&`' to the command line. There can be only one process group in the foreground at any time, because you cannot enter a new command until the currently running one terminates and the shell prompt returns. Foreground processes can read from and write to the terminal.

In contrast, there is no limit to the number of background processes, but in a POSIX-compliant system, they cannot read from or write to the terminal. If they try to do either, they will be stopped by the kernel (via `SIGTTIN` or `SIGTTOU` signals). The default action of a `SIGTTOU` signal is to stop the process, but many shells override the default action to allow background processes to write to the terminal.

Background and foreground processes use the terminal as their control terminal, but background processes do not receive all signals from that terminal. A `Ctrl-C`, for example, will not cause a `SIGINT` to be sent to background processes. However, a `SIGHUP` will be sent to all processes that use that terminal as their control terminal, including background processes. This is so that, if a terminal connection is broken, all processes can be notified of it and killed by default. If you want to start a background process and then logout from a session, you can use the `nohup` command to run it while ignoring `SIGHUP` signals, as in

```
$ nohup do_backup &
```

which will let `do_backup` run after the terminal is closed. In this case, the `do_backup` program must not read or write a terminal.

7.5 Sessions

Every process belongs to a *session*. More accurately, every process group belongs to a session, and by transitivity, each process belongs to a session. Every session has a unique *session-id* of type `pid_t`. The primary purpose of sessions is to organize processes around their controlling terminals. When a user logs on, the kernel creates a session, places all processes and process groups of that user into the session, and links the session to the terminal as its controlling terminal. Sessions usually consist of a single foreground process group and zero or more background process groups. Just as process groups have leaders, *sessions have leaders*. The session leader can be distinguished because its process-id is the same as the session-id.

Processes may secede from their sessions, and unlike countries, they can do this without causing wars. Any process other than a process group leader can form a new session and automatically be placed into a new group as well. The new session will have no control terminal. This is exactly how a *daemon* is created – it detaches itself from the session into which it was born and goes off on its own. Later we will see how programs can do this.

You can add output to the `ps` command to see the session-id by adding the "`sid`" output format to the standard syntax, as in

```
$ ps -ouser,pid,ppid,pgrp,sid,%cpu,cputime,tty,args
```

7.6 The Memory Architecture of a Process

Although earlier chapters made allusions as to how a process is laid out in virtual memory, here the process layout is described in detail. In addition, we provide a program that displays enough information about the locations of its own symbols in virtual memory that one can infer its layout

from them. In particular, the program will display the addresses of various local and global symbols in hexadecimal and decimal notation. These locations are clues to how the process is laid out in its logical address space.

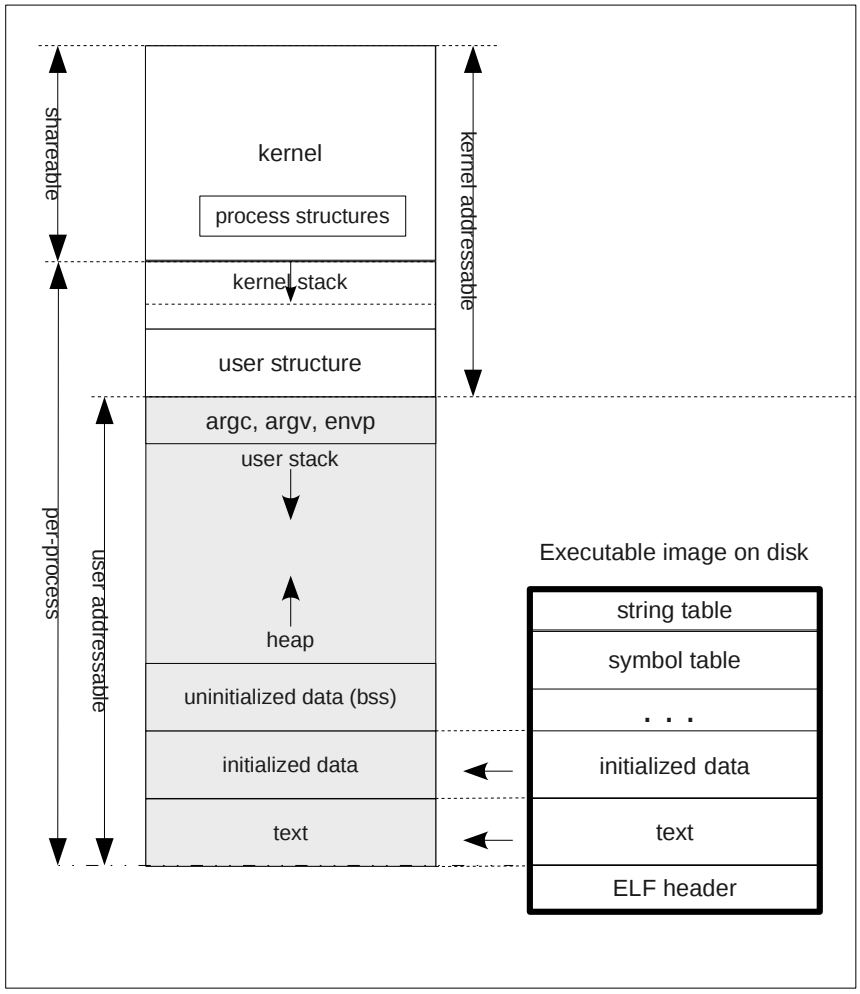


Figure 7.1: Typical layout of a process in virtual memory.

7.6.1 Overview

To start, we look at the big picture of what a process looks like in its logical address space. This picture should enable you to construct a mental image of the physical layout of a process in its own logical address space, how it looks in a file, and how that file relates to the virtual memory image of the process. Different UNIX systems use different layouts for processes, but for the most part, most modern systems adhere to a format known as the *Executable and Linkable Format* (*ELF*).

The resources needed by a process executing in user mode include the CPU state (general purpose

registers, program status register, stack related registers, etc.), some environment information, and three memory segments called the *text segment*, the *data segment*, and the *stack segment*. The text segment contains the program code. The stack segment is reserved for the run-time stack of the user phase[1] of the process, and the data segment is for program data. More will be said about these below.

The resources needed when the process is executing in kernel phase include the CPU state (same as above) as well as the resources needed for the kernel to provide the services for the process and schedule it appropriately. These resources include the parameters to the system call, the process's identity and properties, scheduling information, open file descriptors, and so on. This set of kernel resources is separated into two main structures, a *process structure* and a *user structure*, and a few minor ones. Together these structures constitute what is usually called the *process image*. The process structure and user structure are kept in kernel memory. The layout of the process address space is shown in Figure 7.1.

The process structure contains the information that must be memory-resident even when the process is swapped out, including the process privileges and rights, identifiers associated with the process, its memory map, descriptors, pending events, and maximum and current resource utilization. The user structure contains information that is not needed in memory when the process is swapped out, including the *process control block*[2], accounting and statistics, and a few other pieces of information, and is therefore swapped out along with the rest of the process.

7.6.2 The Process Structure

The kernel maintains a process structure for every running process. This structure contains the information that the kernel needs to manage the process, such as various process descriptors (process-id, process group-id, session-id, and so on), a list of open files, a memory map and possibly other process attributes. In all of the versions of UNIX with which I am familiar, this structure is known as the *process structure*. In Linux, the terms process structure and *task structure* are used interchangeably, and the actual data structure that represents it is the `task_struct`. In some versions of UNIX, there is much less information in the process structure and more in the user structure.

Up until the introduction of threads, or so called *light-weight processes* (*LWPs*), the process structure was a very "heavy" structure filled with a large amount of information. One reason that the concept of a light-weight process was invented was to reduce the amount of information associated with each executable unit, so that they did not take up as much memory and so that creating new ones would be faster. The process structure was redesigned in 4.4BSD to support these threads by moving much of the information that had been in it into smaller structures that could be pointed to by the process structure. Each thread could share the information in the substructures by pointing to them, rather than keeping complete copies of them. This way, each thread could have its own unique identifiers, such as a process-id, and also have access to the shared data, such as open files and memory maps.

The exact information present in any process structure will vary from one implementation to another, but all process structures minimally include

- Process id

[1] User phase and user mode are used interchangeably here.

[2] The process control block is used in UNIX only to store the state of the CPU – the contents of the registers and so on.

- Parent process id (or pointer to parent's process structure)
- Pointer to list of children of the process
- Process priority for scheduling, statistics about CPU usage and last priority.
- Process state
- Signal information (signals pending, signal mask, etc.)
- Machine state
- Timers

Usually the process structure is a very large object containing much more additional information. The task structure in Linux 2.6.x, for example, may contain over 150 different members. Typical substructures referenced in the process structure may include such things as the

- Process's group id
- User ids associated with the process
- Memory map for the process (where all segments start, and so on)
- File descriptors
- Accounting information
- Other statistics that are reported such as page faults, etc.
- Signal actions
- Pointer to the user structure.

The substructure generally contains information that all threads would share, and the process structure itself contains thread-specific information.

The process structure is located in kernel memory. Different versions of UNIX store it in different ways. In BSD, the kernel maintains two lists of process structures called the *zombie list* and the *allproc list*. Zombies are processes that have terminated but that cannot be destroyed, the reasons for which will be made clear a little later in this chapter. Zombie processes have their structures on the zombie list. The allproc list contains those that are not zombies. In Linux 2.6.x, the process or task structures are kept in one, circular, doubly-linked list. In Solaris, the process structure is in a `struct proc_t` and the collection of these are maintained in a table.

7.6.3 The User Structure

The user structure contains much less information than the process structure. The user structure gets swapped in and out with the process; keeping it small reduces the swapping overhead. Historically, the most important purpose of the user structure was that it contained the *per-process execution stack* for the kernel.

Every process in UNIX needs its own, small kernel stack. When a process issues a system call and the kernel phase begins, the kernel needs a stack for its function calls. Since the kernel might be interrupted in the middle of servicing a call, it must be able to switch from one process's service call to another. This implies that it needs a different stack for each process. The UNIX kernel designers carefully designed all functions in the kernel so that they are non-recursive and do not use large automatic variables. Furthermore, they can trace the possible sequences of call chains in the kernel so that they know exactly the largest possible stack size. Thus, unlike ordinary user programs, the kernel itself has a known upper bound on its required stack size. Because the stack size is known in advance, the kernel stack can be allocated in a fixed size chunk of virtual address space. This is why, in Figure 7.1, you see that it is bounded above and below by fixed boundaries. As you can see from the figure, the stack is at the high end of the address space, above the environment variables and program parameters. Its exact placement varies from one version of UNIX to another.

Depending on the version of UNIX, the user structure can contain various other pieces of information. In BSD, the memory maps are all in the process structure or its substructures. In Linux, the user structure, defined in `struct user` (in `<user.h>`) contains the memory maps. The memory maps generally include the starting and ending addresses of the text, data, and stack segments, the various base and limit registers for the rest of the address space, and so on1. The user structure usually contains the process control block. This contains the CPU state and virtual memory state (page table base registers and so on.)

7.6.4 The Text Segment

The text segment (also called the *instruction segment*) is the program's executable code. It is almost always a sharable, read-only segment, shared by all other processes executing the same program. The C compiler, by default, creates shared text segments. The advantage of using shared text segments is not so much that it conserves memory to do so, but that it reduces the overhead of swapping. When a process is active, its text segment resides in primary memory. There are various reasons why the process might be swapped to secondary storage. Often it is that it issues a wait for a slow event; in this case the system will swap it to secondary storage to make room in memory for other more productive processes. When it becomes active again, it is brought back into memory. Since a read-only text segment can never be modified, there is no reason to copy it to secondary storage when a process executing it is swapped out. Similarly, if a copy of a text segment already resides in primary memory, there is no reason to copy the text segment from secondary storage into primary memory. Thus, there is a savings in swapping overhead.

UNIX keeps track of the read-only text segment of each user process. It records the location of the segment in secondary storage and, if it is loaded, its primary memory address, and a count of the number of processes that are currently executing it. When a process first executes the segment, the segment is loaded from secondary storage, the count is set to one, and a table entry is created. When a process terminates, the count is decremented. When the count reaches zero, the segment is freed and its primary and secondary memory are de-allocated.

7.6.5 The Stack Segment

The user stack segment serves as the run-time stack for the user phase of the process. That is, when the process makes calls to other parts of the user code, the calls are stacked in this segment. The stack segment provides storage for the automatic identifiers and register variables, and serves

its usual role of managing the linkage of subroutines called by the user process. The stack is always "upside-down" in UNIX, meaning that pushes cause the top to become a smaller memory address. If the stack ever meets the top of the heap, it causes an exception. In a 32-bit architecture, a user process is typically allocated a virtual address space of 4 Gbytes. If the stack meets the heap, the process exceeds its virtual memory allotment and it is time to port the application to a 64-bit machine!

7.6.6 The Data Segment

The data segment is memory allocated for the program's initialized and uninitialized data. The initialized data is separated from the uninitialized data, which is stored in a section called the *bss*, which is an acronym for *Block Started by Symbol*, an old FORTRAN machine instruction. Initialized data are items such as named constants and initialized static variables. They come from the symbol table. Uninitialized data has no starting value. The system only needs to reserve the space for them, which it does by setting the address of the top of the data segment. The data segment grows or shrinks by explicit memory requests to shift its boundary. The system call to shift the boundary is the `brk()` call; `sbrk()` is a C wrapper function around `brk()`.The data segment always grows toward the high end of memory, i.e., the `brk()` call increases the boundary to increase memory. Most programmers use the C library functions `malloc()` and `calloc()` to allocate more memory instead of the low-level `brk()` primitives. These C routines call `brk()` to adjust the size of the data segment.

Programmers often refer to the part of memory that is used by `malloc()` and `calloc()`, the "*heap*". People usually think of the heap as the part of memory that is not yet allocated, lying between the top of the stack and the end of the bss.

7.6.7 From File to Memory

How is an executable program file arranged and how does it get loaded? When you run the `gcc` compiler and do not specifically name the executable file, as in

```
$ gcc myprog.c
```

the compiler (actually the linker) creates a file named `a.out`. The name `a.out` is short for "assembler output" and was not just the name of the output file, but was also name of the format of all binary executable files on UNIX systems for many years. The `a.out` file format could be read in the `a.out` man page as well as in the `<a.out>` header file.

In the mid 1990's, a more portable and extensible format known as *Executable and Linking Format* (*ELF*) was created by the UNIX System Laboratories as part of the *Application Binary Interface* (*ABI*). It was later revised by the Tool Interface Standards (TIS) Committee, an industry consortium that included most major companies (Intel, IBM, Microsoft, and so on). While compilers continue to create files named `a.out`, they are ELF files on most modern machines.

The ELF specification defines exactly how an executable file is organized, and is general enough to encompass three different types of files:

- *Relocatable files* holding code and data suitable for linking with other object files, to create an executable or a shared object file (`.o` files),

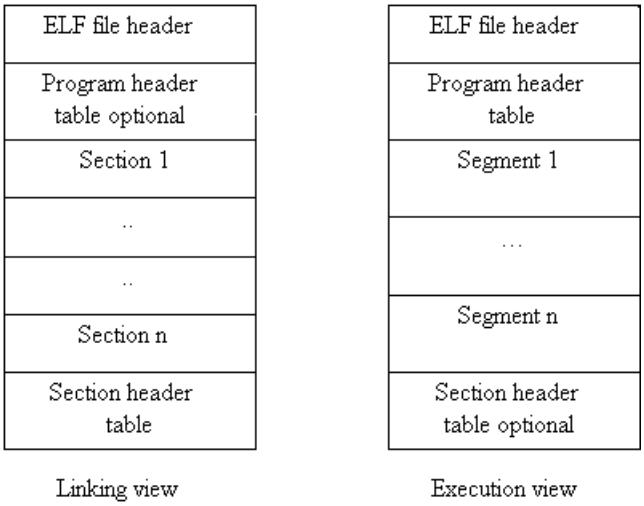


Figure 7.2: Linking and execution views of an ELF file.

- *Executable files* holding a program suitable for execution, and
- *Shared object files* that can be used by the dynamic linker to create a process image (`.so` files).

An ELF file begins with a structure called the *ELF header*. This header is essentially a road map to the rest of the file. It contains identification information and the addresses and sizes of the rest of the components of the file. An ELF file is characterized by the fact that it contains two different, parallel, yet overlapping views: the *linking view* and the *execution view* as shown in Figure 7.2.

The linking view is the view of the file needed by the link editor in order to link and relocate components in the file. Information within it is organized into *sections*, which contain such things as the instructions, data, symbol table, string table, and relocation information. A *section header table* serves as a table of contents for the sections and is generally located at one end of the file.

The execution view, in contrast, is the view of the file needed in order to execute it. It organizes its information in *segments*. Segments correspond conceptually to virtual memory segments; when the executable is loaded into memory, ELF segments are mapped to virtual memory segments. Thus, for example, for an executable program, there is a text segment containing instructions, an uninitialized data segment, and an initialized data segment, as well as several others. A *program header table* serves as a table of contents for the segments and usually follows the ELF header.

Neither the sections nor the segments in the file have to be in any particular order because the tables define their positions. The two separate views are overlapped in the file, as shown in Figure 7.3. The figure also shows that segments can consist of multiple sections.

The *symbol table* is a table that the compiler creates to map symbolic names to logical addresses and store the attributes of these symbols. The compiler uses the table to construct the code, but in addition, it is the symbol table that makes it possible for a debugger to associate memory addresses with the names of variables. When the debugger runs, the symbol table is loaded into memory with the program. The *string table* is a table containing all of the strings used in the program. The `strings` command is a handy command to know about – it displays a list of all of the strings in a binary file. The strings command works because the string table is part of the file and the command simple has examine it.

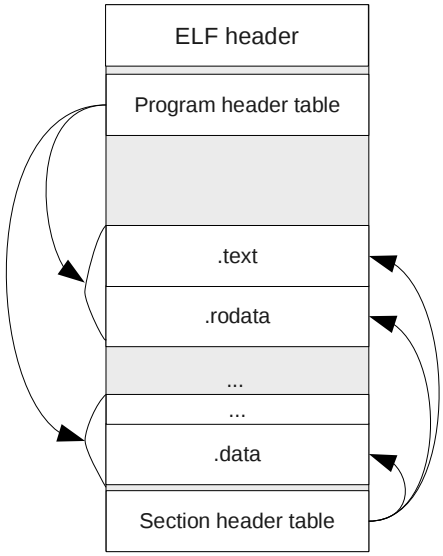


Figure 7.3: Overlapped views of the ELF file.

The `readelf` command can be used to examine an executable file.

```
$ readelf [options] elffile ...
```

The information displayed depends upon the options provided on the command line. With `-a`, all information is provided. The `-h` option displays the contents of the ELF header (in human readable form of course):

```
$ readelf -h /bin/bash
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x805c7b0
Start of program headers: 52 (bytes into file)
Start of section headers: 733864 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
```

```
Number of program headers: 8
Size of section headers: 40 (bytes)
Number of section headers: 32
Section header string table index: 31
```

You can see from this output that the ELF header has information about the executable's format: 32-bit, 2's complement, for Intel 80386 on UNIX System V. It also has the location of the program header (byte 52 into the file) and the section header table (733864 bytes into the file).

The `file` command uses the ELF header to provide its output:

```
$ file /bin/bash
/bin/bash: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
```

You can experiment with some of the executables to see how much you can learn about the structure of executable programs.

Notice in the output above that the entry point address of the executable is `0x805c7b0`. This is the address of the first executable instruction in the bash executable. The starting virtual address is not `0`. In modern UNIX systems, the starting address is always after `0x8048000`. The addresses below that are reserved for the system to use. One reason for this is that the debugger will run in the lower addresses when the program runs under the debugger.

Figure 7.4 shows the virtual addresses of a hypothetical executable, taken from a version of the ELF standard. Notice in this example that the text segment is not at the start of the virtual address space, and that each segment is padded as needed so that it aligns on `0x1000` (4096) byte boundaries, because page sizes are 4096 bytes. Notice too that the data segment and uninitialized data segment follow the text segment.

7.6.8 A Program To Display The Virtual Memory Boundaries

In this section we will explore the virtual address space of a process with the aid of a program that I found in a book on interprocess communication [1] and subsequently modified. It displays the boundaries of the different components of the virtual address space of its executable image. The program declares the following types of memory objects:

- Global, initialized pointer variable: `cptr`
- Global uninitialized string: `buffer1`
- Automatic variable in main program: `i`
- Parameters to main program: `argc`, `argv`, `envp`
- Static uninitialized local in main program: `diff`
- Main function `main()`
- Non-main function `showit()`

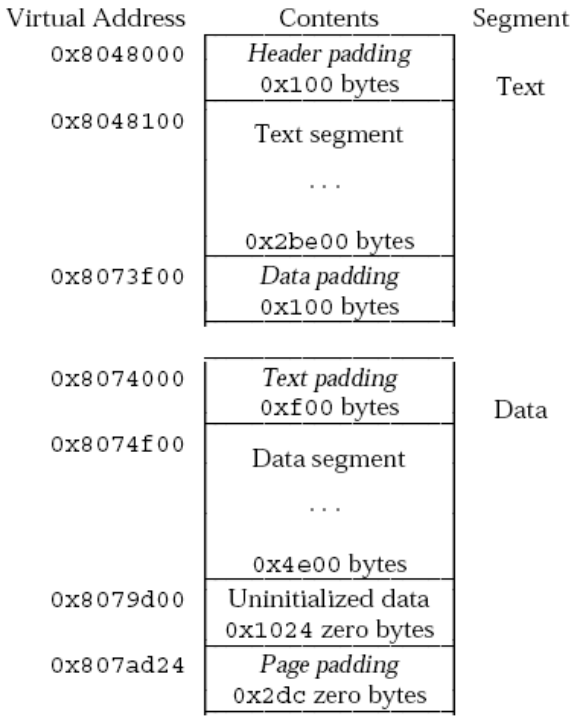


Figure 7.4: Example of ELF process image segments.

- Automatic pointer variable, dynamically allocated: `buffer2`
- Automatic variable in non-main function: `num`

It displays the locations of each of these objects as hexadecimal and decimal virtual addresses. The locations of these objects will lie within the ranges that are specified by the text, data, and stack segment positions described earlier. For example, the location of the uninitialized global `buffer1` and the initialized global `*cptr` will be between the first and last addresses of the data segment, one in the `bss` and the other, not. The remaining variables each pinpoint the location of a particular segment: `"Hello World"`, the pointer variable `cptr` itself, the symbols `main()` and `showit`(), which are globals, and the local variables `num` and `buffer2` in `showit`(). The location of a local variable, which is supposed to be on the run-time stack, will show where that stack is in virtual memory.

Three external integer variables `etext`, `edata`, and `end` are defined in global scope in the C library and may be accessed by any C program[3]. They are boundaries of three specific segments:

`etext` The address of `etext` is the first location after the program text.

`edata` The address of `edata` is the first location after the initialized data region.

`end` The address of `end` is the first location after the uninitialized data region.

The value of `etext` is an upper bound on the size of the executable image. The initialized and uninitialized data regions are the regions where constants and globals are stored. Uninitialized data

[3]They may be declared as macros.

are globals whereas initialized data are constants. These symbols are incorporated into the program, displayed in Listing 7.1.

Listing 7.1: displayvm.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

/*****************************************************************/
/*                          Global Constants                     */
/*****************************************************************/

#define SHW_ADR(ID,I) \
printf("       %s \t is at addr:%8X\t%20u\n",\
        ID,(unsigned int)(&I),(unsigned int)(&I))

/* These are system variables, defined in the unistd.h header file */
extern int    etext, edata, end;

char     *cptr = "Hello World\n";  /* cptr is an initialized global */
char     buffer1[40];              /* uninitialized  global         */

void showit(char *);         /* Function prototype -- has no storage */

int main(int argc, char* argv[], char* envp[])
{
    int  i = 0;          /* on stack */
    static int  diff;    /* global in uninitialized data segment */

    strcpy(buffer1, "    Layout of virtual memory\n");
    write(1,buffer1,strlen(buffer1)+1);

    printf("Adr etext: %8X \t Adr edata: %8X \t Adr end: %8X \n\n",
           (unsigned int)&etext,(unsigned int) &edata,(unsigned int) &end);

    printf("         ID \t            HEX_ADDR\t        DECIMAL_ADDR\n");
    SHW_ADR("main",  main);
    SHW_ADR("showit", showit);
    SHW_ADR("etext", etext);
    diff = (int) &showit - (int) &main;
    printf("           showit is %d bytes above main\n", diff);
    SHW_ADR("cptr", cptr);
    diff = (int) &cptr - (int) &showit;
    printf("           cptr is %d bytes above showit\n", diff);
    SHW_ADR("buffer1", buffer1);
    SHW_ADR("diff", diff);
    SHW_ADR("edata", edata);
    SHW_ADR("end", end);
    SHW_ADR("argc", argc);
    SHW_ADR("argv", argv);
    SHW_ADR("envp", envp);
    SHW_ADR("i", i);
```

```
    showit(cptr);
    return 0;
}
/*****************************************************************/

void showit(char * p)
{
    char  *buffer2;
    SHW_ADR("buffer2",buffer2);
    if ( (buffer2= (char *) malloc((unsigned)(strlen(p)+1))) != NULL){
        strcpy(buffer2, p);
        printf("%s", buffer2);
        free(buffer2);
    }
    else {
        printf("Allocation error.\n");
        exit(1);
    }
}
```

When you run this program, you should get output similar to the following. The last column is the decimal value of the location. Notice that `i` in `main()` and `buffer2` in `showit()` are high addresses. They are in the stack. Notice that `envp`, which is a pointer to an array of strings, is above the stack, as Figure 7.1 depicts. Notice that the addresses of these descend, because the stack grows downward. Notice too that `diff` in `main()` and `buffer1` lie between `edata` and `end`, showing that they are in the uninitialized data region, but that `cptr` is between `etext` and `edata` because it is initialized. You should see that `buffer1` is the last variable in the *bss*, since it is 40 bytes long and its address is 40 bytes from the end. Also, notice that the address of `etext` is the same as the value of `etext`. This is because `etext` is not a real variable. It is just a mnemonic name for the actual location, as determined by the linker.

```
     Layout of virtual memory
Adr etext:  8048958      Adr edata:  8049C90      Adr end:  8049CE8

        ID                  HEX_ADDR            DECIMAL_ADDR
       main         is at addr: 8048544              134513988
       showit       is at addr: 8048806              134514694
       etext        is at addr: 8048958              134515032
           showit is 706 bytes above main
       cptr         is at addr: 8049C8C              134519948
           cptr is 5254 bytes above showit
       buffer1      is at addr: 8049CC0              134520000
       diff         is at addr: 8049CA8              134519976
       edata        is at addr: 8049C90              134519952
       end          is at addr: 8049CE8              134520040
       argc         is at addr:BF9ECDC0             3214855616
       argv         is at addr:BF9ECDC4             3214855620
       envp         is at addr:BF9ECDC8             3214855624
       i            is at addr:BF9ECD9C             3214855580
       buffer2      is at addr:BF9ECD6C             3214855532
Hello World
```

Notice that the main program starts at virtual address `0x08048544`. The start of the virtual address space is at `0x08048000`. The difference is 1348 bytes. If you look at the output of the `readelf`

`-a` command, you will see that there are several sections of code that reside in this small pocket before the main program. There you will find the code that must be run before `main()` starts (what I like to call the "glue" routine), the code that runs when the program finishes, and special code for dynamic linking and relocation. When a program starts, before the operating system transfers control to the program, there are initializations and possible linking and relocation; the code there serves that purpose.

7.7 Creating New Processes Using fork

Now that you have a better idea of what processes actually are, we can start exploring their world. We will begin with process creation, since that is, after all, the beginning of all things.

Once the operating system has bootstrapped itself, the only way for any process to be created is via the `fork()` system call[4]. All processes are created with `fork()`:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

The `fork()` call is a hard one to accept at first; you probably have never seen a function quite like it before. It is very appropriately named, because the statement

```
pid_t processid = fork();
```

causes the kernel to create a new process that is almost an exact copy of the calling process, such that after the call, there are two processes, each continuing its execution at the point immediately after the call in the executing program! So before the instruction is executed, there is a single process about to execute the instruction, and by the time it has returned, there are two. There has been a fork in the stream of instructions, just like a fork in a road. It is almost like process mitosis.

The process that calls `fork()` is called the *parent process* and the new one is the *child process*. They are distinguished by the values returned by `fork()`. Each process is able to identify itself by the return value. The value returned to the parent is the process-id of the newly created child process, which is never 0, whereas the value returned to the child is 0. Therefore, each process simply needs to test whether the return value is 0 to know who it is. The child process is given an independent *copy* of the memory image of the parent and the same set of resources. It is essentially a clone of the parent and is almost identical in every respect[5]. Unlike cellular mitosis though, it is not symmetric.

The typical use of the `fork()` call follows the pattern

```
processid = fork();
```

[4]This is a simplification. There are other system calls, depending on the version of UNIX. For example, Linux revived the old BSD `vfork()` system call, which was introduced in 3.0BSD and later removed in 4.4BSD for good reason. Linux also provides a `clone()` library function built on top of the kernel's `sys_clone()` function. Section 7.7.1 below.

[5]There are a few minor differences. For example, a call to `getppid()` will return different values, since this returns the process-id of the parent process.

```
if (processid == 0)
// child's code here
else
// parent's code here
```

The true branch of the if-statement is code executed by the child and not by the parent. The false branch is just the opposite. This may seem like a useless way to create processes, since they always have to share the same code as the parent, so they do nothing different. The `fork()` call is valuable when used with `exec()` and `wait()`, as will be shown shortly.

Our first example, `forkdemo1.c`, in Listing 7.2 demonstrates a bit about how `fork()` works.

Listing 7.2: forkdemo1.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int global = 10;

int main(int argc, char* argv[])
{
    int    local = 0;
    pid_t  pid;

    printf("Parent process: (Before fork())");
    printf(" local = %d, global = %d \n", local, global);

    if ( ( pid = fork() ) == -1 ) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid ) {
        /* child executes this branch */
        printf("After the fork in the child:");
        local++;
        global++;
        printf(" local = %d, global = %d\n", local, global);
    }
    else {
        /* parent executes this branch */
        sleep(2);  /* sleep long enough for child's output to appear */
    }

    /* both processes execute this print statement */
    printf("pid = %d, local = %d, global = %d \n",
            getpid(), local, global);

    return 0;
}
```

In this program, the return value from `fork()` is stored in `pid`, which is tested in the if-statement. `fork()` returns `-1` on failure, and one should always check whether it failed or not.

The child's code is the next branch, in which `0 == pid` is true. The child increments the values of two variables, one, `global`, declared globally and one, `local`, locally in `main()`. It then prints out their values, jumps over the parent code and executes the `printf()`, obtaining its process-id using the `getpid()` system call and displaying the values of `local` and `global` again.

In the meanwhile, the parent sleeps for two seconds, enough time so that the child's output will appear first. It then executes the `printf()`, obtaining its process-id using the `getpid()` system call and displaying the values of `local` and `global`. The `sleep()` prevents intermingling of the output, which can happen because the child shares the terminal with the parent.

Bearing in mind that the child is given a *copy* of the memory image of the parent, what should the output of the parent be? Hopefully you said that `local = 0` and `global = 10`. This is because the child changed their values in the copy of the memory image, not in a shared memory. The point of this program is simply to demonstrate this important fact.

Let us make sure we understand `fork()` arithmetic before continuing. Before running the program `forkdemo2.c` shown in Listing 7.3, predict the number of lines of output. Do not redirect the output or pipe it. This will be explained afterward.

Listing 7.3: forkdemo2.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char* argv[] )
{
    int i;
    printf("About to create many processes...\n");
    for ( i = 0; i < N; i++ )
        if ( -1 == fork() )
            exit(1);

    printf("Process id = %d\n", getpid());
    fflush(stdout);       /* force output before shell prompt        */
    sleep(1);             /* give time to the shell to diplay prompt */
    return 0;
}
```

Did you correctly predict?

Each time the bottom of the loop is reached, the number of processes in existence is twice what it was before the loop was entered, because each existing process executes the `fork()` call, making a copy of itself. If N were 1, the loop would execute once and there would be 2 processes, each printing their process ids to the screen. If N were 2, the loop would execute a second time, and the 2 processes would make 2 more, 4 in total. In general, there will be 2^N processes when the loop finishes, and that many lines of output on the screen, together with the first line,

```
About to create many processes...
```

for a total, when N = 4, of 17 lines. Now try redirecting the output of the program to `wc`, to make it easier to count how many lines are there.

```
$ forkdemo2 | wc
     32     144     832
$
```

Why are there 32 lines and not 17? Try instead redirecting the output to a file and looking at it there:

```
$ forkdemo2 > temp; wc temp
 24 108 624 temp
$
```

If you look at the file `temp`, you will see something like

```
About to create many processes...
Process id = 6708
About to create many processes...
Process id = 6707
About to create many processes...
Process id = 6717
About to create many processes...
Process id = 6709
About to create many processes...
Process id = 6716
...
```

and there will be fewer than 16 lines with process ids. So what is going on here? Why is that line replicated for each process and why are there fewer than 16 lines stating the process ids?

- Remember this important fact about `fork()`: when a process is created by a call to `fork()`, it is an almost exact duplicate of the original process. In particular it gets copies of all open file descriptors and naturally, all of the process's user space memory image.
- What this means is that when a child process is created, its standard output descriptor points to the same open file structure as the parent and all other processes forked by the parent, and therefore the children and parent share the file position pointer.
- Operations such as `printf()` are part of the C I/O library and act on objects of type `FILE`, which are called *streams*. The C I/O Library uses *stream buffering* for all operations that act of `FILE` streams. (We noted this in Chapter 5.) There are three different kinds of buffering strategies:
  - *Unbuffered streams*: Characters written to or read from an unbuffered stream are transmitted individually to or from the file as soon as possible.
  - *Line buffered streams*: Characters written to a line buffered stream are transmitted to the file in blocks when a newline character is found.
  - *Fully buffered streams*: Characters written to or read from a fully buffered stream are transmitted to or from the file in blocks of arbitrary size.

- By default, when a stream is opened, it is fully buffered, except for streams connected to terminal devices, which are line buffered.
- The buffers created by the C I/O Library are in the process's own address space, not the kernel's address space. (When your program calls a function such as `printf()`, the library is linked into that program; all memory that it uses is in its virtual memory.) This means that when `fork()` is called, the child gets a copy of the parent's library buffers, and all children get copies of these buffers. *They are not shared*; *they are duplicated.*
- The C I/O library flushes all output buffers
  - When the process tries to do output and the output buffer is full.
  - When the stream is closed.
  - When the process terminates by calling `exit()`.
  - When a newline is written, if the stream is line buffered.
  - Whenever an input operation on any stream actually reads data from its file.
  - When `fflush()` is called on that buffer.
- As a corollary to the preceding statement, until the buffer has been flushed, it contains all characters that were written to it since the last time it was flushed.
- No C I/O Library function is atomic. It is entirely possible that output can be intermingled or even lost if the timing of calls by separate processes sharing a file position pointer leads to this.

Now we put these facts together. The `forkdemo2` program begins with the instruction

```
printf("About to create many processes...\n");
```

If output has not been redirected, then `stdout` is pointed to a terminal device and it is line buffered. The string `"About to create many processes...\n"` is written to the terminal and removed from the buffer. When the process forks the children, they get empty buffers and write their individual messages to the terminal. Unless by poor timing a line is written over by another process, each process will produce exactly one line of output. It is quite possible that this will happen if there are a large enough number, N, of processes, as the probability of simultaneous writes increases rapidly towards 1.0 as N increases.

Let us do a bit of mathematical modeling. The `printf()` instruction

```
printf("Process id = %d\n", getpid());
```

writes to standard output. If the fraction of time that each process spends in the portion of the `printf()` function in which a race condition might occur is p, then there is a probability of $1-(1-p)^N$ that at least two processes are in that portion of code at the same time. If, for example, $p = 0.05$ and $N = 16$, then the probability of a race (and hence lost output) is $1 - 0.95^16 \approx 0.56$. If $N = 32$, it is 0.81 and when $N = 64$, it is 0.96. So you see that as the number of processes increases, it becomes almost inevitable that lines will be lost, regardless of whether they are written

to the terminal or to a different file descriptor, because the race condition is independent of how the output stream is buffered.

If standard output is redirected to a file or to a pipe, it no longer points to a terminal device and the library will fully buffer it instead of line buffering it. The block size used for buffering is much larger than the total size of the strings given to the `printf()` function. The consequence is that the string `"About to create many processes...\n"` will remain in the buffers of all child processes when they are forked, and when they each call

```
printf("Process id = %d\n", getpid());
fflush(stdout);
```

each line of the output will be of the form

```
About to create many processes...
Process id = 8810
```

and there will be twice as many lines *written* as there were to the terminal.

Since the command

```
forkdemo2 | wc
```

redirects the standard output of `forkdemo2` to a pipe, `wc` will see twice as many lines as appear on the terminal. Similarly, the command

```
forkdemo2 > temp
```

redirects the standard output to a file, and the file will contain twice as many lines as what appears on the terminal.

The foregoing statement about the output to the file is true *only if* the executions of the `printf()` instructions do not overlap and no output is lost. We return to this issue shortly. The claim regarding the pipe is unconditionally true.

How can we make the behavior of the program the same regardless of whether it is to a terminal or is redirected? We can force the first string to be flushed from the buffer by calling `fflush(stdout)`. Since there is no need to do this if it is a terminal, we can insert the two lines

```
if ( !isatty(fileno(stdout)) );
fflush(stdout);
```

just after the first `printf()`.

What about the problem of lost output? How can we prevent this race condition? The answer is that we must not use the stream library but must use the lower level `write()` system call and file descriptors. Writes are unbuffered and we can set the `O_APPEND` flag on file descriptor 1 so that the race condition is eliminated. (Recall from Chapter 4 that this is how writes to the `utmp` file avoid race conditions.)

To use `write()`, we must first create the output string using the `sprintf()` function:

```
char str[32];
sprintf(str, "Process id = %d\n", getpid());
```

Then we can call `write()`:

```
write(1, str, strlen(str));
```

But first we must start the program by setting `O_APPEND` on standard output's descriptor:

```
int flags;
flags = fcntl(1, F_GETFL);
flags |= (O_APPEND);
if (fcntl( 1, F_SETFL,flags) == -1 )
    exit(1);
```

This solves the problems. The corrected program is listed below.

Listing 7.4: forkdemo3.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include   <fcntl.h>
#include <termios.h>

int main( int argc, char* argv[] )
{
    int     i;
    int     N;
    char    str[32];
    int     flags;

    /* Put standard output into atomic append mode */
    flags = fcntl(1, F_GETFL);
    flags |= (O_APPEND);
    if (fcntl( 1, F_SETFL,flags) == -1 )
        exit(1);

    /* Get the command line value and convert to an int.
       If none, use default of 4. If invalid, exit. */
    N = ( argc > 1 )? atoi(argv[1]):4;
    if ( 0 == N )
        exit(1);

    /* Print a message and flush it if this is not a terminal */
    printf("About to create many processes...\n");
    if ( !isatty(fileno(stdout)) )
        fflush(stdout);
```

```
    /* Now fork the child processes. Check return values and exit
       if we have a problem. Note that the exit() may be executed
       only for some children and not others. */
    for ( i = 0; i < N; i++ )
        if ( -1 == fork() )
            exit(1);

    /* Create the output string that the process wil/ write, and write using
       system call. */
    sprintf(str, "Process id = %d\n", getpid());
    write(1, str, strlen(str));
    fflush(stdout);          /* to force output before shell prompt */
    sleep(1);                /* to give time to the shell to diplay prompt */
    return 0;
}
```

7.7.1 Other Versions of fork()

The `vfork()` system call is a different version of the `fork()` call that is designed to be more efficient. Rather than making a complete copy of the address space of the old process, the `vfork()` call creates a new process without copying the data and stack segments of the parent and instead allows the child process to share these. This saves time and memory but also raises the possibility that the child will inadvertently corrupt the state of the parent process. It is not intended to be used to allow the child and parent to share data; on the contrary, its purpose is to avoid the extensive memory copying in the case that the child will replace its code anyway using `exec()` (to be discussed soon.)

There is also a `clone()` system call in Linux systems. The `clone()` function, which is technically a library routine wrapping a system call, allows the child to share the address space with its parent, and also lets the programmer pass a function and arguments for the child to execute. We will look at it in detail later.

7.7.2 Synchronizing Processes with Signals

The next program, `synchdemo1.c`, demonstrates how to use `fork()` and signals to synchronize a child and its parent.

Listing 7.5: synchdemo1.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

void  c_action(int signum)
{
    /* nothing to do here */
}

/*****************************************************************/
```

```
int main(int argc, char* argv[])
{
    pid_t  pid;
    int    status;
    static struct sigaction childAct;


    switch (pid = fork())  {
    case -1:
        /* fork failed! */
        perror("fork() failed!");
        exit(1);

    case 0: {
        /* child executes this branch   */
        /* set SIGUSR1 action for child */
        int i, x=1;
        childAct.sa_handler = c_action;
        sigaction(SIGUSR1, &childAct, NULL);
        pause();
        printf("Child  process: starting computation...\n");
        for ( i = 0; i < 10; i++ ) {
            printf("2^%d = %d\n", i, x);
            x = 2*x;
        }
        exit(0);
    }
    default:
        /* parent code */
        printf("Parent process: "
               "Will wait 2 seconds to prove child waits.\n");
        sleep(2); /* to prove that child waits for signal */
        printf("Parent process: "
               "Sending child notice to start computation.\n");
        kill(pid,SIGUSR1);

        /* parent waits for child to return here */
        if ((pid = wait(&status)) == -1)
        {
            perror("wait failed");
            exit(2);
        }
        printf("Parent process: child terminated.\n");
        exit(0);
    }
}
```

Comments.

- First note that the style of this program is slightly different. It uses the `switch` statement to distinguish the failed `fork()`, child, and parent.
- The `SIGUSR1` signal is a signal value that is reserved for user programs to use as they choose.

In this program, we can use it to synchronize two processes. One process delays itself using `pause()`, and waits for a signal to arrive. The second sends the signal to wake up the first. The signal handler does not have to do anything special in this case.

- The parent calls `wait()`, a function we will explore shortly. The `wait()` call makes the parent wait until the child terminates or is killed.
- The program displays output on the terminal just to demonstrate how the signaling works.

The next demo, `synchdemo2.c`, is a little more interesting than the preceding one. It demonstrates how the parent and child can work in lockstep using the `SIGUSR1` signal. It also shows that the child process inherits the open files of the parent, and that writes by the child and parent to the same descriptor advance the shared file position pointer.

Listing 7.6: synchdemo2.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*****************************************************************/
void p_action(int sig);
void c_action(int sig);
void on_sigint(int sig);
/*****************************************************************/

int                  nSignals = 0;
volatile sig_atomic_t  sigint_received = 0;

/*****************************************************************/
int main(int argc, char* argv[])
{
    pid_t pid, ppid;

    static struct  sigaction parentAct, childAct;
    int            fd;
    int            counter = 0;
    char           childbuf[40];
    char           parentbuf[40];

    if ( argc < 2 ) {
        printf("usage: synchdemo2 filename\n");
        exit(1);
    }

    if ( -1 == (fd = open(argv[1], O_CREAT|O_WRONLY|O_TRUNC, 0644 )) )
    {
        perror(argv[1]);
        exit(1);
```

```
    }
    switch( pid = fork() ) {
    case -1:
        perror("failed");
        exit(1);
    case 0:
        /* set action for child */
        childAct.sa_handler = c_action;
        sigaction(SIGUSR1, &childAct, NULL);
        ppid = getppid();  /* get parent id */
        for(;;) {
            sprintf(childbuf, "Child counter = %d\n", counter++);
            write(fd, childbuf, strlen(childbuf) );
            printf("Sending signal to parent -- ");
            fflush(stdout);
            kill(ppid, SIGUSR1);
            sleep(3);
        }

    default:
        /* set SIGUSR1 action for parent */
        parentAct.sa_handler = p_action;
        sigaction(SIGUSR1, &parentAct, NULL);

        /* set SIGINT handler for parent */
        parentAct.sa_handler = on_sigint;
        sigaction(SIGINT, &parentAct, NULL);

        for(;;) {
            sleep(3);
            sprintf(parentbuf, "Parent counter = %d\n", counter++);
            write(fd, parentbuf, strlen(parentbuf) );
            printf("Sending signal to child  -- ");
            fflush(stdout);
            kill(pid, SIGUSR1);
            if ( sigint_received ) {
                close(fd);
                exit(0);
            }
        }
    }
}

/*****************************************************************/

void p_action(int sig)
{
    printf("Parent caught signal %d\n", ++nSignals);
}

void c_action(int sig)
{
    printf("Child caught signal %d\n", ++nSignals);
}
```

```
void on_sigint( int sig )
{
    sigint_received = 1;
}
```

Comments.

- Writes to the file are in lockstep and there is no race condition because of the arrangement of the `sleep()` and `kill()` calls in the child and parent. The parent writes after it is awakened from its `sleep()` and before it signals the child, whereas the child writes before it signals the parent the first time, and then after it is awakened by the parent. If the parent is writing, the child must be sleeping, and vice verse.
- The use of the `sleep()` instead of `pause()` prevents deadlock. Had we used `pause()`, then there would be a very small but nonzero probability that one process could issue a `kill()` to the other and, before it then executes its `pause()`, the other is woken up, executes all of its code and issues a `kill()` to the first one. In this case, the signal would be lost, because it happened before the pause(). That process would then be blocked waiting for a second signal to wake it, but the other process will enter its `pause()` and never be able to send that signal. They are thus deadlocked. The `sleep()` will eventually terminate, so nether process will wait indefinitely.
- The call to `fflush()` is needed to force writes to the screen by each process to happen immediately, otherwise they will occur in the wrong order.
- The main program has a `SIGINT` handler so that the program can clean up after itself. When `Ctrl-C` is typed, both the parent and the child will receive it. The parent closes the open file descriptor before exiting, and the child is automatically killed.

We now turn to the question of how a process can change the code it executes.

7.8 Executing Programs: The exec family

Being able to create a new process is not so useful unless that new process has a way to execute a different program. The `exec()` family of calls fulfills that purpose. All versions of the `exec()` call have one thing in common – they cause the calling process to execute a program named in one way or another in the argument list, and they all are library wrappers for the `execve()` system call.

7.8.1 The execve() System Call

The man page for the `execve()` system call defines it as follows:

```
#include <unistd.h>
int execve(const char *filename, char *const argv[],
char *const envp[]);
```

`execve()` executes the program pointed to by its first argument. The filename must be a binary executable or a script whose first line is

```
#! interpreter [optional-arg]
```

The filename must be the absolute pathname or relative pathname of the program. `execve()` does not look at the `PATH` environment variable to resolve command names. The second and third arguments are `NULL`-terminated arrays of arguments and environment strings respectively. In other words, each is an array of strings followed by a `NULL` pointer. The environment strings are expected to be in the proper format: *key=value*.

You should remember that arrays are sometimes called vectors by computer scientists, and that the reason that the name of this system call is `execve` is that it expects vectors as its second and third arguments. In particular, you need to remember that `execve()` will pass these vectors to the program being executed, which will be able to access them in its own argument list:

```
int main( int argc, char* argv[], char* envp[])
```

Since all programs expect the program name in `argv[0]` and the first real argument in `argv[1]`, it is important that you arrange the argument list to satisfy this condition before you invoke `execve()`. The examples will demonstrate.

The man page provides all of the details about the call. In general, `execve()` causes the stack, data segment, bss, and text segment to be replaced, and pretty much clears all signals and closes anything the process had open before the call except for file descriptors, which remain open. The current working directory remains the same. Because the same process continues to execute, process relationships are also preserved. For the details consult the man page.

To start we will look at how to use the `execve()` system call, after which we will look at the different wrappers for it. The first program, `execdemo1.c`, is in Listing 7.7 below.

Listing 7.7: execdemo1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[], char * envp[])
{
    if ( argc < 2 ) {
        printf("usage: execdemo1 arg1 [arg2 ...]\n");
        exit(1);
    }
    execve("/bin/echo", argv, envp);
    fprintf(stderr, "execve() failed to run.\n");
    exit (1);
}
```

This program calls `execve()`, passing `/bin/echo` as the program to run, followed by its own command line arguments and environment strings. These are passed to `/bin/echo`. This program works correctly even though `argv[0]` contains the string "`execdemo1`" and not "`echo`" because `echo` pretty

much ignores `argv[0]` and only starts paying attention to arguments starting with `argv[1]`. This is not the best way to use `execve()` – it only works in a few circumstances.

Are you wondering about the `printf()` after the call? The `printf()` statement will only be executed if the `execve()` call fails; the only reason that `execve()` returns is failure to execute the program.

The next program, `execvedemo2.c`, uses `execve()` to execute the first command line argument of the program, passing to it the remaining arguments from the command line. In other words, if we supply a line like

```
$ execvedemo2 /bin/ls -l ..
```

it will execute it as if you typed the `/bin/ls -l ..` on a line by itself.

Listing 7.8: execvedemo2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[], char * envp[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s program args\n", argv[0]);
        exit (1);
    }
    execve(argv[1], argv+1, envp);
    fprintf(stderr,"execve() failed to run.\n");
    exit(1);
}
```

Notice that it uses pointer arithmetic to pass the array `argv[1 .. argc-1]` rather than `argv[0 .. argc-1]`.

7.8.2 The exec() Library Functions

Because it is a little inconvenient to arrange everything for `execve()`, the designers of UNIX created a family of five functions that act as front-ends to `execve()`, each expecting a different set of parameters. They are

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,..., char* const envp[]);
int execv(const char *path, char * const argv[]);
int execvp(const char *file, char * const argv[]);
```

Each of these contains either an 'l' or a 'v' in its name. The versions that contain an 'l', `execl()`, `execlp()` and `execle()`, expect a null-terminated *list* of null-terminated string arguments whereas

the versions that contain a 'v', `execv()` and `execvp()`, expect a *vector* of null-terminated string arguments. All versions cause the kernel to load the executable file whose name is either path or file, given above, overlaying the current program for the process, and passing it the remaining arguments.

The functions are also characterized by whether or not they contain a 'p' in their names. The versions that contain a 'p', `execlp()` and `execvp()`, expect the first argument to be a simple file name rather than a full path name, whereas the ones that do not contain a 'p': `execl()`, `execle()`, and `execv()`, require the full pathname for the first argument. The versions containing the 'p' will use the `PATH` environment variable to search for the file whose name is supplied, provided it does not contain any slashes. If it has a slash, then that is treated as a pathname, either relative or absolute, to the file to be loaded.

For all of these functions, the parameters named `arg` or `argv` above that follow the path or file parameter are passed to the executable as its own arguments. The first of these arguments must be a pointer to the executable file, because in UNIX, by convention, the first argument to a program (`argv[0]`) is always the name of the program itself, stripped of the preceding pathname. For example, to execute "`/bin/ls -l`" using `execl()`, you would use the syntax

```
execl( "/bin/ls", "ls", "-l", (char *) 0);
```

In other words, the name of the executable occurs twice – first with the full pathname, and second with just the name of the file itself.

The differences between the different versions can be summarized as follows:

`execl, execle, execlp` expect the arguments to be presented as a comma-separated list of strings, terminated by a `NULL` pointer cast to a string, as in

```
execl( "/bin/ls",  "ls", "-l", (char *) 0);
```

`execv, execvp` expect the arguments to be presented as a vector (array) whose last element is a `NULL` pointer as in:

```
strcpy(argvec[0], "ls");
strcpy(argvec[1], "-l");
argvec[2] = NULL;
execv( "/bin/ls", argvec );
```

`execlp, execvp` do not require a full path name:

```
execlp( "ls", "ls", "-l", (char *) 0);
strcpy(argvec[0], "ls");
strcpy(argvec[1], "-l");
argvec[2] = NULL;
execvp( "ls", argvec );
```

`execle` is the same as `execl()` except that it has a third argument that is an array of pointers to environment strings exactly as `execve()` expects, with a `NULL` pointer as its last entry.

The functions other than `execle()` obtain the environment settings for the new process from the values in the external `environ` variable.

To illustrate , the following program, `execvpdemo.c`, uses its first argument as the executable to run, and the remaining arguments as its arguments.

Listing 7.9: execvpdemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[], char * envp[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s program args\n", argv[0]);
        exit (1);
    }
    execvp( argv[1],  argv +1 );
    perror("execvp");
    exit (1);
}
```

Since it searches the `PATH` environment variable, it can be called, for example, with

```
$ execvpdemo cp origfile newfile ..
```

7.9 Synchronizing Parents and Children: wait and exit

7.9.1 Exit() Stage Left

We have used the `exit()` function many times in various programs, but the only reason for doing so was that it was a way to terminate the calling process and return a non-zero integer value when some error condition arose. The `exit()` function does much more than this. Its synopsis is

```
#include <stdlib.h>
void exit(int status);
```

Three actions take place when `exit()` is called:

1. The process's registered *exit functions* run;
2. the system gets a chance to clean up after the process; and
3. the process gets a chance to have a status value delivered to its parent.

By an *exit function*, we mean a function that is run when `exit()` is called.

Before continuing, you may wonder why we would want a special function to run when `exit()` is called. Imagine that when your program terminates, it has to update a log file. Suppose the function that does this is named `update_log()`. Suppose also that the program is very large, that there are

multiple points at which `exit()` is called, and that more than one programmer is maintaining this program. If the `exit()` function did not provide a means of invoking user-defined exit routines, then each time that anyone modified the program to insert a new call to `exit()`, he or she would have to remember to call `update_log()` first. However, by registering `update_log()` to run whenever `exit()` is called, it makes the programmer's job easier, since she does not have to worry about forgetting to include the call when the program is modified.

When `exit()` is called, the following actions take place in the given order:

1. All functions registered to run with the `atexit()` or `on_exit()` functions are run (in the reverse order in which they were registered with these routines).
2. All of the file streams opened through the Standard I/O Library are flushed and closed.
3. The kernel's `_exit()` function is called, passing the status argument to it.

Programmers can register a function to run when a process calls `exit()` using either `atexit()` or `on_exit()`. The preferred choice is `atexit()` since it is more portable. The man pages for both contain the details for how to register such exit functions. If more than one function is registered, they are run in the reverse order of the order in which they were registered (i.e., in last-in-first-out order). After the registered functions run, the `exit()` function flushes the streams and closes the files. The `exit()` function then calls `_exit(status)`. The kernel's `_exit()` function makes sure that

1. any open file descriptors are closed (not just those opened through Standard I/O Library functions),
2. all memory belonging to the process is released,
3. all children of the process (including zombies, defined below) are "adopted" by the `init()` process (meaning that `init()` is made the parent of these children,
4. the low-order eight bits of the integer argument to `exit()`, called its *exit status*, are made available to the parent process, and
5. under appropriate conditions, a `SIGCHLD` signal is sent to the parent process.

Actions (4) and (5) will be explained shortly. There are other actions that must take place within the clean-up routines of the kernel. This is just a partial list, including the most basic operations.

The following example shows how `atexit()` can be used to register a few exit functions:

Listing 7.10: atexitdemo.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void Worker(void)
{
    printf("Worker #1 : Finished for the day.\n");
}
```

```
void Foreman(void)
{
    printf("Foreman   : Workers can stop for the day.\n");
}

void Boss(void)
{
    printf("First Boss: Foreman, tell all workers to stop work.\n");
}

int main(void)
{
    long max_exit_functions = sysconf(_SC_ATEXIT_MAX);

    printf("Maximum number of exit functions is %ld\n",
            max_exit_functions);
    if ((atexit(Worker)) != 0) {
        fprintf(stderr, "cannot set exit function\n");
        return EXIT_FAILURE;
    }

    if ((atexit(Foreman)) != 0) {
        fprintf(stderr, "cannot set exit function\n");
        return EXIT_FAILURE;
    }

    if ((atexit(Boss)) != 0) {
        fprintf(stderr, "cannot set exit function\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

7.9.2 Waiting for Children to Terminate

After a process forks a child, how will it know if and when the child has finished whatever task it set out to accomplish? Typically, a process has to wait until the child or children finish completing their tasks before it can continue. The `fork()`, `exec()`, and `exit()` system calls need one more partner to form a complete ensemble, and that is the `wait()` family of calls. Generally speaking, the purpose of `wait()` is two-fold:

- to delay the parent until a child has terminated, and
- to obtain the status of a child that has terminated.

There are only two ways for a process to terminate: either "normally" by calling one of various exit functions[6] such as `exit()`, or "abnormally" and involuntarily as a result of receiving a signal that killed it, or calling `abort()`. (When a program either reaches the end of its code or executes a `return`, this results in an implicit call to `exit()`.) In either case, the parent can call `wait()`

[6]There are two other exit functions, including `_exit()` and `_Exit()`.

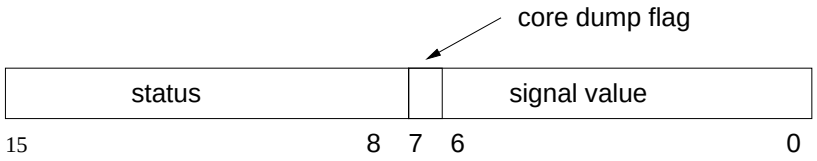


Figure 7.5: Two-byte exit status.

to determine the cause of termination. There are three different POSIX-compliant `wait()` system calls:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

The `wait()` function causes the executing process to suspend its execution until any one of its children terminates. When a child terminates, the kernel sends a `SIGCHLD` signal to its parent, unless the parent has indicated that it does not want these signals. Upon receipt of a `SIGCHLD` signal, the parent resumes in the `wait()` code. The return value of `wait()` is the process-id of the child that just terminated or was just killed. It does not matter which child terminates. The process is resumed if any child terminates. If a process calls `wait()` but it has no children, `wait()` returns immediately with a -1.

The purpose of the status parameter is to receive information about how the child terminated. It is a pointer to a two-byte integer. If the child terminated normally using the `exit()` call, then the high-order byte of the value received by `wait()` contains the low-order byte of the integer passed in the `exit()` call's argument, and the low-order byte of the received value is 0[7]. If the child terminated abnormally because of an unhandled signal, then the low-order byte of the received value contains the signal value. If the child was terminated by a signal, then in particular bit 7 is set if there was a core dump[8]. Figure 7.5 shows how the two bytes of the status are arranged.

Based on these facts, the following code can be used for querying and extracting the status and signal state:

```
if ((status & 0x000000FF) ==0)
    /* low-order byte is zero, so high-order byte has status */
    exitStatus = status >> 8;  /* exitStatus contains exit status */
else {
    signum = status%128;  /* signum contains signal */
    if ( status & 0x00000080 )
```

[7]The convention when using `exit()` is to supply a zero on success and some non-zero value on failure.

[8]The fact that a core dump is supposed to occur does not mean that there will be a core file in your working directory. If your shell has been configured so that core dumps are disabled, then you will not see the file. On some systems, you can enable the core dump by running the command "`ulimit -c unlimited`", which allows your processes to create core files of unlimited size. The `ulimit` command is part of `bash` and you can read the `bash` man page for more details.

```
            /* core dump took place */
    }
```

However, using the following macros, which are defined in `<sys/wait.h>`, makes the code more portable:

```
    if (WIFEXITED(status))
        /* true implies exit() was called to terminate the child   */
        exit_status = WEXITSTATUS(status);  /* extract exit status */
    else if ( WIFSIGNALED(status) ) {
        /* true if signal killed child */
        signum = WTERMSIG(status);  /* extract signal that killed child */
    #ifdef WCOREDUMP
        if ( WCOREDUMP(status) )
            /* true if a core dump took place  */
    #endif
    }
```

- The `WIFEXITED(status)` macro returns true if the child terminated normally, i.e., by calling `exit(3)` or `_exit(2)`, or by returning from `main()`. In this case the `WEXITSTATUS(status)` macro returns the exit status of the child. This macro should only be employed if `WIFEXITED()` returned true.
- The `WIFSIGNALED(status)` macro returns true if the child process was terminated by a signal. If it returns true, then the `WTERMSIG(status)` macro returns the number of the signal that caused the child process to terminate. The `WTERMSIG()` macro should only be used if `WIFSIGNALED()` returned true.
- The `WCOREDUMP(status)` macro returns true if the child produced a core dump. This macro should only be used if `WIFSIGNALED` returned true. *This macro is not specified in POSIX.1-2001* ; only use this enclosed in

  `#ifdef WCOREDUMP ... #endif.`

7.9.3 Using wait()

Listing 7.11 contains an example that puts together the use of `fork()`, `exit()`, and `wait()`. It is the typical way in which these three primitives are used. In this example the user is prompted to supply an exit value for the child, which is then passed to the `exit()` call, to show that the value is then available to the parent in the `wait()` call.

Listing 7.11: waitdemo2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
```

```
void child()
{
    int exit_code;
    printf("I am the child and my process id is %d.\n",getpid());
    sleep(2);
    printf("Enter a value for the child exit code:");
    scanf("%d",&exit_code);
    exit(exit_code);
}

int main(int argc, char* argv[])
{
    int pid;
    int status;

    printf("Starting up...\n");
    if ( -1 == (pid = fork())) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid )
        child();
    else {  /* parent code */
        printf("My child has pid %d and my pid is %d.\n", pid, getpid());
        if ((pid = wait(&status)) == -1) {
            perror("wait failed");
            exit(2);
        }
        if (WIFEXITED(status))  { /* low order byte of status equals 0 */
            printf("Parent: Child %d exited with status %d.\n",
                    pid, WEXITSTATUS(status));
        }
        else if ( WIFSIGNALED(status) ) {
            printf("Parent: Child %d exited with error code %d.\n",
                    pid, WTERMSIG(status));
#ifdef WCOREDUMP
            if ( WCOREDUMP(status) )
                printf("Parent: A core dump took place.\n");
#endif
        }
    }
    return 0;
}
```

Notes.

- When the child process displays a message such as

  ```
  I am the child and my process id is 5666.
  Enter a value for the child exit code:
  ```

  enter an exit code and observe that it is printed by the parent after the parent's call to `wait()` finishes. Then run the program again but this time send a signal to the child process from

another terminal using the kill command, i.e.,

```
$ kill -10 5666
```

and observe that the parent displays the message

```
Parent: Child 5666 exited with error code 6.
Parent: A core dump took place.
```

- The conditional compilation macro is used because the `WCOREDUMP` macro is not available on all UNIX systems, as noted above.

Sometimes a parent does not care very much about how its children terminate. A parent can explicitly tell the kernel that it doesn't care if and when its children terminate by setting the `SA_NOCLDWAIT` flag, which prevents the delivery of `SIGCHLD` signals to itself. It does this using the `sigaction()` call:

```
const struct sigaction act;
act.sa_flags = SA_NOCLDWAIT;
sigaction (SIGCHLD, &act, NULL);
```

Notice that it is not necessary to set a signal handler in this call; it is enough to just pass the `sa_flags` field. If a process has so indicated to the kernel its lack of interest in its children, then when a child terminates, the child's status will not be delivered to its parent. The child will be completely terminated immediately. Similarly, when the parent sets the action for `SIGCHLD` to `SIG_IGN`, the status will be discarded and the child completely terminated.

If the parent process has set neither `SA_NOCLDWAIT` nor the action for `SIGCHLD` to `SIG_IGN` and is presently executing any of the `wait()` calls described below, then the status will be delivered to the parent and a `SIGCHLD` signal sent to it. If the parent is not currently waiting, then when the parent does invoke `wait()`, it will receive the status. If the parent is not executing any form of `wait()`, though, the child process is transformed into a *zombie process*. A zombie process is an inactive process and it will be deleted at some later time when its parent process executes a `wait()` call. Zombie processes exist simply to provide their parents with their status values at a future time.

The flip side of this issue is what happens when a process terminates before its children. When a process terminates and has children, these children are not terminated also. In comes the `init()` process. The `init()` process adopts orphans, so if a process terminates and has any children, they are adopted by `init()`. When the children terminate, their exit status will be sent to `init()`.

7.9.4 Using waitpid()

The `waitpid()` function has three parameters. The first is the process-id of the child to wait for, the second is a pointer to the variable in which to store the status, and the last is an optional set of flags. If the pid is -1, then it tells the kernel that the process will wait for any child, like `wait()`. Setting the pid to 0 means to wait for only those children in the same process group as the parent. Setting the pid to -G, for some positive integer G, means to wait for any child whose process group-id is G.

There are three flags that can be passed to `waitpid()`:

`WNOHANG`	return immediately if no child has exited.
`WUNTRACED`	also return if a child has stopped (but not traced via `ptrace(2)`). Status for traced children which have stopped is provided even if this option is not specified.
`WCONTINUED`	(Since Linux 2.6.10) also return if a stopped child has been resumed by delivery of `SIGCONT`.

The program in Listing 7.12 below combines some of the preceding ideas and demonstrates the use of `waitpid()` with the `WNOHANG` flag.

Listing 7.12: waitpiddemo.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>

void child()
{
    int exit_code;

    printf("I am the child and my process id is %d.\n",getpid());
    sleep(2);
    printf("Enter a value for the child exit code followed by <ENTER>.\n");
    scanf("%d",&exit_code);
    exit(exit_code);
}

int main(int argc, char* argv[])
{
    pid_t pid;
    int status;
    int signum;

    printf("Starting up...\n");
    if ( -1 == (pid = fork())) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid ) {
        child();
    }
    else {
        /* wait for specific child process with waitpid()
           If no child has terminated, do not block in waitpid()
           Instead just sleep. (Would do something useful instead.)
        */
        while (0 == waitpid(pid, &status, WNOHANG) ) {
            printf("still waiting for child\n");
            sleep(1);
        }
```

```
        /* pid is the pid of the child that terminated */
        if (WIFEXITED(status)) {
            printf("Exit status of child %d was %d.\n",
                   pid, WEXITSTATUS(status));
        }
        else if ( WIFSIGNALED(status) ) {
            signum = WTERMSIG(status);
            printf("Parent: Child %d exited by signal %d.\n",pid,
                    signum);
#ifdef WCOREDUMP
            if ( WCOREDUMP(status) )
                printf("Parent: A core dump took place.\n");
#endif
        }
    }
    return 0;
}
```

Notes.

- The parent is in a busy waiting loop in this example, waiting for the child to terminate. The `WNOHANG` flag to `waitpid()` allows it to continue polling the `waitpid()` call and do something else in the meanwhile. The body of the loop would be replaced with a task that the parent could do while waiting for the child. The advantage of this is that the parent does not have to block, waiting for the child, but can instead do work. If there is no work to do, then this paradigm is not the one to use.
- You can send a signal to the child in the same way as for the program in Listing 7.11 to observe that the parent code detects the error return.

7.9.5 Non-blocking waits

Instead of calling `wait()` or `waitpid()`, a process can establish a `SIGCHLD` handler that will run when a child terminates. The `SIGCHLD` handler can then call `wait()`. This frees the process from having to poll the `wait()` function. It only calls `wait()` when it is guaranteed to succeed. The following example (`rabbits2.c` in the demos directory) demonstrates how this works.

Listing 7.13: rabbits2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <limits.h>
#include <sys/wait.h>
#include <termios.h>

#define  NUM_CHILDREN   5
#define  SLEEPTIME      30

/**  child()  The code that is executed by each child process
```

```
 *   All this does is register the SIGINT signal handler and then
 *   sleep SLEEPTIME seconds. If a child is delivered a SIGINT, it
 *   exits with the exit code 99. See on_sigint() below.
 */
void child       ();

/**  on_sigint()  Signal handler for SIGINT
 *   All it does it call exit with a code of 99.
 */
void on_sigint   ( int signo );

/**  on_sigchld()  Signal handler for SIGCHLD
 *   This calls wait() to retrieve the status of the terminated child
 *   and get its pid. These are both stored into global variables that
 *   the parent can access in the main program. It also sets a global
 *   flag.
 */
void on_sigchld  ( int signum );

/* These variables are declared with the volatile qualifier to tell the
   compiler that they are used in a signal handler and their values
   change asynchronously. This prevents the compiler from performing an
   optimization that might corrupt the program state. All three are
   shared by the main parent process and the SIGCHLD handler.
 */
volatile int          status;
volatile pid_t        pid;
volatile sig_atomic_t child_terminated;

/*************************************************************************
                                Main Program
*************************************************************************/

int main(int argc, char* argv[])
{
    int       count = 0;
    const int NumChildren = NUM_CHILDREN;
    int i;
    struct sigaction newhandler;       /* for installing handlers */

    printf("About to create many little rabbits...\n");
    for ( i = 0; i < NumChildren; i++) {
        if ( -1 == (pid = fork())) {
            perror("fork");
            exit(-1);
        }
        else if ( 0 == pid ) { /* child code */
            /* Close standard output so that children do not print
               parent's output again. */
            close(1);
            child();
            exit(1);
        }
        else { /* parent code */
```

```
            if ( 0 == i )
               printf("Another ");
            else if (i < NumChildren-1 )
               printf("and another ");
            else
               printf("and another.\n");
        }
    }
    /* parent continues here*/
    /* Set up signal handling  */
    newhandler.sa_handler = on_sigchld ;
    sigemptyset(&newhandler.sa_mask);
    if ( sigaction(SIGCHLD, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        return (1);
    }

    /* Enter a loop in which work could happen while the global flag
       is checked to see if any child has terminated. */

    child_terminated = 0;           /* Set flag to 0 */
    while( count < NumChildren ){
        if ( child_terminated ) {
            if ( WIFEXITED(status) )
                printf("Rabbit %d died with code %d.\n",
                        pid, WEXITSTATUS(status));
            else if (  WIFSIGNALED(status) )
                printf("Rabbit %d was killed by signal %d.\n",
                        pid, WTERMSIG(status));
            else
                printf("Rabbit %d dies with status %d.\n",pid,status);
            child_terminated = 0;
            count++;
        }
        else  {
            /* do something useful here. for now just delay a bit */
            sleep(1);
        }
    }

    printf("All rabbits have terminated and been laid to rest.\n");
    return 0; /* main returns; child never reaches here */
}

void on_sigint( int signo )
{
    exit(99);
}

void child()
{
    struct sigaction newhandler;

    newhandler.sa_handler = on_sigint;
```

```
    sigemptyset(&newhandler.sa_mask);
    if ( sigaction(SIGINT, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        exit (1);
    }
    sleep(SLEEPTIME);
}

void on_sigchld ( int signum )
{
    int child_status;

    if ((pid = wait(&child_status)) == -1) {
        perror("wait failed");
    }
    child_terminated = 1;
    status = child_status;
}
```

Notes.

- The C Standard I/O Library by default uses buffered streams. This means that when a process uses the C output functions such as `printf()`, the output is placed into a buffer before being delivered to the terminal device. When each child is created, it is given a copy of the parent's buffers at the time of creation. If we did not close the file descriptor in the child immediately, then when the child terminated, its buffer would be flushed and multiple copies of the parent's output would appear in the terminal window. We cannot use `fclose(stdout)` because `fclose()` is designed to flush the buffers and would also cause the output to appear. We must use the lower-level file descriptors. Try commenting out the line "`close(1)`" and running the program.
- The child is designed to catch a `SIGINT` and exit within the signal handler. The reason for this is to allow the user to send a `SIGINT` (by issuing a `kill -2` to the child on the command line) in order to show that even if a signal caused the signal handler to run, the fact that the child called `exit()` means that the parent will see that the child died by calling `exit()`, not as a result of being sent a signal.
- If the child is not killed by a signal, then it terminates normally after a 30 second sleep.
- The `on_sigchld()` handler, after calling `wait()`, sets an atomic flag and lets the main program do the work of handling the child's exit.
- The program does not test the return value of `fork()` for failure, just to save space here.
- Although POSIX does not permit it, some systems allow signals to be lost, and it is possible to lose a `SIGCHLD` signal in this code. If multiple children terminate in a small time sequence, and the parent is in the `SIGCHLD` handler, then some of the `SIGCHLD` signals may be merged into a single signal. GNU C allows this for example. The fix requires much more complex code that maintains a list of the child processes and which inspects that list within the handler.

7.9.6 Waiting for State Changes in Children

The `wait()` family was extended in Linux 2.6.9 with the inclusion of `waitid()`, which can be used to gather information about other changes in the state of child processes besides their termination:

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

The `idtype` can be one of `P_PID`, `P_PGID`, or `P_ALL`. If it is `P_PID`, it waits for the process whose process-id is passed as the second argument. If it is `P_PGID`, it waits for any process whose group-id is the second argument. If `P_ALL`, the second argument is ignored and it acts like the ordinary `wait()`.

The options parameter is the OR of one or more of the flags

`WEXITED`	Wait for children that have terminated.
`WSTOPPED`	Wait for children that have been stopped by delivery of a signal.
`WCONTINUED`	Wait for (previously stopped) children that have been resumed by delivery of `SIGCONT`.

and optionally the `WNOHANG` flag described earlier as well as the `WNOWAIT` flag, which leaves the child process as if the parent never called `wait()`, in case it wants to retrieve the status at a later time.

If waitid() completes successfully, it fills in the `siginfo_t` structure pointed to by the `infop` parameter . The `siginfo_t` structure is the same structure used by the `sigaction()` function. It is a union, so the members filled in by `sigaction()` are not exactly the same as those filled in by `waitid()`. `waitid()` provides the following members:

`si_pid` The process-id of the child

`si_uid` The real user-id of the child

`si_signo` `SIGCHLD`

`si_status` Either the exit status or the signal that caused the child's state to change.

`si_code` Exactly one of `CLD_EXITED` if the child exited, `CLD_KILLED` if the child was killed by a signal, `CLD_STOPPED` if the child was stopped by a signal, or `CLD_CONTINUED` if it was continued by a `SIGCONT`.

The way to use `waitid()` is to inspect the value of `infop->si_code` to determine the state of the child before accessing `infop-status`. The following listing, modified from the one in the man page for `wait()`, demonstrates how to use `waitid()`.

Listing 7.14: waitiddemo.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>

#define SLEEPTIME  60
```

```
int main(int argc, char* argv[])
{
    pid_t       pid;
    siginfo_t   siginfo;

    if ( -1 == (pid = fork())) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid ) {
        printf("Child pid is %d\n", getpid());
        sleep(SLEEPTIME);
        exit(0);
    }
    /* Parent code */
    else do {
        /* Zero out si_pid in case the sig_info_t struct does not get */
        /* initialized because no children are waitable. */
        siginfo.si_pid = 0;

        /* Wait for changes in the state of the child created above,  */
        /* specifically, stopping, resuming, exiting, and return */
        /*immediately if no child is waitable. */
        if (-1 == waitid(P_PID, pid, &siginfo,
                  WEXITED | WSTOPPED | WCONTINUED | WNOHANG) ) {
            perror("waitid");
            exit(EXIT_FAILURE);
        }
        if ( siginfo.si_pid == 0 )
            /* no child is waitable. */
            continue;
        switch ( siginfo.si_code ) {
                case CLD_EXITED:
                printf("Child exited with status %d\n",
                        siginfo.si_status );
                break;
                case CLD_KILLED:
            case CLD_DUMPED:
                printf("Child killed by signal %d\n",
                        siginfo.si_status );
                break;
                case CLD_STOPPED:
                printf("Child stopped by signal %d\n",
                        siginfo.si_status );
                break;
                case CLD_CONTINUED:
                printf("Child continued\n");
                break;
        }
    } while ( siginfo.si_code != CLD_EXITED &&
                    siginfo.si_code != CLD_KILLED &&
                    siginfo.si_code != CLD_DUMPED );
    return 0;
}
```

Notes.

- The `si_status` field does not need to be bit-manipulated to extract its value. The value contained there has already been shifted as necessary.
- The while-loop in the parent continues until the child is killed or terminated to give you a a chance to stop and continue the child.
- If you run this program in one terminal, and then from another issue `kill` commands, or run it in the background on one terminal and issue kill commands on the same terminal, you will see output like the following:

```
$ waitiddemo 77
Child pid is 15243
Child exited with status 77
$
```

  Then do it again without the command-line argument:

```
$ waitiddemo &
Child pid is 15245
$ kill -STOP 15245
Child stopped by signal 19
$ kill -CONT 15245
Child continued
$ kill -TERM 15245
Child killed by signal 15
[1]+ Done waitid2
$
```

7.10 Summary

The four principal tools in process creation and control are `fork()`, `exec()`, `exit()`, and `wait()` and their related functions. Add to the toolbox the things you have learned about signals and signal handling and you have the means of creating and managing processes effectively. What is still lacking is a means for these processes to exchange and share data effectively. At this point the only way they can share data other than provided by the signal mechanism is through the file system, which is extremely slow. Inter-process communication is the topic of the next chapter.

Because the new process is a copy of the parent process, it shares all open files and all library buffers. When the two processes both use the C I/O Library, care must be taken to prevent unexpected consequences of this.

Process creation is a time-consuming activity in the kernel, with high overhead in memory copying. When the objective is to use shared variables and common code, light-weight processes, or threads, are the better solution. This topic follows as well.

Bibliography

[1] Keith Haviland, Marcus Gray, and Ben Salama. *Unix System Programming.* Addison-Wesley Longman, Inc., 2nd edition, 1998.

Chapter 8 Interprocess Communication, Part I

Concepts Covered

Pipes
I/O Redirection
FIFOs
Concurrent Servers
Daemons
Multiplexed I/O with select()
API: dup, dup2, fpathconf, mkfifo, mknod, pipe, pclose, popen, select, setsid, shutdown, syslog, tee.

8.1 Introduction

Processes that cooperate in order to complete one or more tasks almost always need to communicate with each other. Sometimes the communication requires sharing data. One method of sharing data is by sharing a common file. If at least one of the processes modifies the file, then the file must be accessed in mutual exclusion. Sharing a file is essentially like sharing a memory-resident resource in that both are a form of communication that uses a shared resource that is accessed in mutual exclusion. Another paradigm involves passing data back and forth through some type of communication channel that provides the required mutual exclusion. A pipe is an example of this, as is a socket. This type of communication is broadly known as a *message-passing* solution to the problem.

This chapter is concerned only with message-passing types of communication. We will begin with *unnamed pipes*, after which we will look at *named pipes*, also known as *FIFO*'s, and then look at *sockets*. Part I is exclusively related to pipes.

8.2 Unnamed Pipes

You are familiar with how to use pipes at the command level. A command such as

```
$ last | grep 'reboot'
```

connects the output of `last` to the input of `grep`, so that the only lines of output will be those lines of `last` that contain the word '`reboot`'. The '`|`' is a `bash` operator; it causes `bash` to start the `last` command and the `grep` command *simultaneously*, and to direct the standard output of `last` into the standard input of `grep`.

Although '`|`' is a `bash` operator, it uses the lower-level, underlying *pipe* facility of UNIX, which was invented by Douglas Mcilroy, and was incorporated into UNIX in 1973. You can visualize the pipe mechanism as a special file or buffer that acts quite literally like a physical pipe, connecting the output of `last` to the input of `grep`, as in Figure 8.1.

The `last` program does not know that it is writing to a pipe and `grep` does not know that it is reading from a pipe. Moreover, if `last` tries to write to the pipe faster than `grep` can drain it, `last`

Figure 8.1: A pipe connecting `last` to `grep`.

will block, and if `grep` tries to read from an empty pipe because it is reading faster than `last` can write, `grep` will block, and both of these actions are handled behind the scenes by the kernel.

What then is a pipe? Although a pipe may seem like a file, it is not a file, and there is no file pointer associated with it. It is conceptually like a conveyor belt consisting of a fixed number of logical blocks that can be filled and emptied. Each write to the pipe fills as many blocks as are needed to satisfy it, provided that it does not exceed the maximum pipe size, and if the pipe size limit was not reached, a new block is made available for the next write. Filled blocks are conveyed to the read-end of the pipe, where they are emptied when they are read. These types of pipes are called `unnamed pipes` because they do not exist anywhere in the file system. They have no names.

An unnamed pipe[1] in UNIX is created with the `pipe()` system call.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

The system call `pipe(fd)`, given an integer array `fd` of size 2, creates a pair of file descriptors, `fd[0]` and `fd[1]`, pointing to the "read-end" and "write-end" of a pipe inode respectively. If it is successful, it returns a 0, otherwise it returns -1. The process can then write to the write-end, `fd[1]`, using the `write()` system call, and can read from the read-end, `fd[0]`, using the `read()` system call. The read and write-ends are opened automatically as a result of the `pipe()` call. Written data are read in *first-in-first-out* (*FIFO*) order. The following program (`pipedemo0.c` in the demos directory) demonstrates this simple case.

Listing 8.1: pipedemo0.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define  READ_END  0
#define  WRITE_END 1
#define  NUM       5
#define  BUFSIZE   32

int main(int argc, char* argv[] )
{
    int    i, nbytes;
    int    fd[2];
    char   message[BUFSIZE+1];
```

[1] Unless stated otherwise, the word "pipe" will always refer to an unnamed pipe.

```
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    for ( i = 1; i <= NUM; i++ ) {
        sprintf(message, "hello #%2d\n", i);
        write(fd[WRITE_END], message, strlen(message));
    }
    close(fd[WRITE_END]);

    printf("%d messages sent; sleeping a bit. Please wait...\n", NUM);
    sleep(3);

    while ( ( nbytes = read( fd[READ_END], message, BUFSIZE)) != 0 )
    {
        if ( nbytes > 0 ) {
            message[nbytes] = '\0';
            printf("%s", message);
        }
        else
            exit(1);
    }
    fflush(stdout);
    exit(0);
}
```

Notes.

- In this program, the write calls are not error-checked, which they should be. The `read()` in the while loop condition is error-checked: if it returns something strictly less than zero, `exit(1)` is executed.
- The `read()` call is a blocking read by default; you have to explicit make it non-blocking if you want it to be so. By design, a blocking read on a pipe will block waiting for data as long as the write-end of the pipe is held open. If the program does not close the write-end of the pipe before the read-loop starts, it will hang forever, because `read()` will continue to wait for data. This could be avoided if the read-loop knew in advance exactly how many bytes to expect, because in that case it could just read exactly that many bytes and then exit the loop, but it is rarely the case that one knows how much data to expect. Naturally, the process has to write the data into the pipe *before* the read loop begins, otherwise there will be nothing to read!
- Notice that the `read()` calls always read the same amount of data. This example demonstrates that the reader can read fixed-size chunks and assemble them into larger chunks, because *the data arrives in the order it was sent* (unlike data sent across a network.) Pipes have no concept of message boundaries – they are simply byte streams.
- Finally, observe that before calling `printf()` to print the string on the standard output, the string has to be null-terminated.

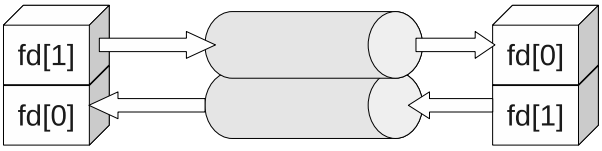


Figure 8.2: Parent and child sharing a pipe.

The semantics of reading from a pipe are much more complex than reading from a file. The following table summarizes what happens when a process tries to read n bytes from a pipe that currently has p bytes in it that have not yet been read.

Pipe Size (p)	**At least one process has the pipe open for writing**			**No processes have the pipe open for writing**
	Blocking read		**Non-blocking read**	
	At least one writer is sleeping	**No writer is sleeping**		
p = 0	Copy n bytes and return n, waiting for data as necessary when the pipe is empty.	Block until data is available, copy it and return its size.	Return `-EAGAIN`.	Return 0.
0 < p < n		Copy p bytes and return p, leaving the buffer empty.		
p ≥ n	Copy n bytes and return n leaving p-n bytes in the pipe buffer.			

The semantics depend upon whether or not a writer has been put to sleep because it tried to write into the pipe previously but the pipe was full. On a non-blocking read request, if the number of bytes requested, n, is greater than what is currently in the pipe and at least one writer is in this sleeping state, then the read will attempt to read n bytes, because as he pipe is emptied, the writer will be awakened to write into the pipe. If no writer is sleeping and the pipe is empty, however, then the read will block until some data becomes available.

8.2.1 Parent and Child Sharing a Pipe

Of course there is little reason for a process to create a pipe to write messages to itself. Pipes exist in order to allow two different processes to communicate. Typically, a process will create a pipe, and then fork a child process. After the fork, the parent and child will each have copies of the read and write-ends of the pipe, so there will be two data channels and a total of four descriptors, as shown in Figure 8.2.

On some Unix systems, such as System V Release 4 Unix, pipes are implemented in this full-duplex mode, allowing both descriptors to be written into and read from at the same time. POSIX allows only *half-duplex mode*, which means that data can flow in only one direction through the pipe, and each process must close one end of the pipe. The following illustration depicts this half-duplex mode.

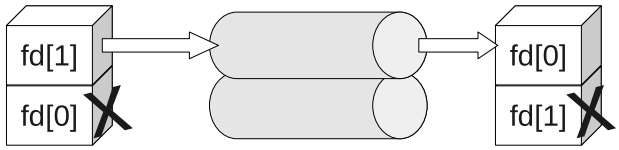


Figure 8.3: Pipe in half-duplex mode.

The paradigm for half-duplex use of a pipe by two processes is as follows:

```
if ( -1 == pipe(fd))
    exit(2);  // failed to create pipe
switch ( fork() ) {
    // child process:
case 0:
    close(fd[1]);  // close write-end
    bytesread = read( fd[0], message, BUFSIZ);
    // check for errors afterward of course
    break;
// parent process:
default:
    close(fd[0]);  // close read-end
    byteswritten = write(fd[1], buffer, strlen(buffer) );
    // and so on
    break;
}
```

Linux follows the POSIX model but does not require each process to close the end of the pipe it is not going to use. However, for code to be portable, it should follow the POSIX model. All examples here will assume half-duplex mode. The following is the first example of two-process communication through a pipe. The parent process reads the command line arguments and sends them to the child process, which prints them on the screen. As we get more deeply involved with pipes, you will discover that it is easy to make mistakes when coding for them, as there are many intricacies to be aware of. This first program exposes a few of them.

Listing 8.2: pipedemo1.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define  READ_FD  0
#define  WRITE_FD 1

int main(int argc, char* argv[] )
```

```
{
    int i;
    int bytesread;
    int fd[2];
    char  message[BUFSIZ];

    /* check proper usage */
    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s message\n", argv[0]);
        exit(1);
    }

    /* try to create pipe */
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    /* create child process */
    switch ( fork() ) {
    case -1:
        /* fork failed -- exit */
        perror("fork()");
        exit(3);

    case 0: /* child code */
        /* Close write end, otherwise child will never terminate */
        close(fd[WRITE_FD]);
        /* Loop while not end of file or not a read error    */
        while ( ( bytesread = read( fd[READ_FD], message, BUFSIZ) )
                != 0 )
            if ( bytesread > 0 ) {  /* more data */
                message[bytesread] = '\0';
                printf("Child received the word: '%s'\n", message);
                fflush(stdout);
            }
            else {  /*read error */
                perror("read()");
                exit(4);
            }
        exit(0);

    default: /* parent code */
        close(fd[READ_FD]);   /* Close read end, since parent is writing */
        for ( i = 1; i < argc; i++ )
            /* send each word separately */
            if ( write(fd[WRITE_FD], argv[i], strlen(argv[i]) ) != -1 )
            {
                printf("Parent sent the word: '%s'\n", argv[i]);
                fflush(stdout);
            }
            else {
                perror("write()");
                exit(5);
```

```
            }
        close(fd[WRITE_FD]);

        /* wait for child so it does not remain a zombie */
        /* don't care about it's status, so pass a NULL pointer */
        if (wait(NULL) == -1) {
            perror("wait failed");
            exit(2);
        }
    }
    exit(0);
}
```

Notes.

- It is now critical that the child closes the write-end of its pipe before it starts to read. As was noted earlier, reads are blocking by default and will remain waiting for input as long as ANY write-end of the pipe is open, including its own. Therefore, not only do we want to close the unused end of the pipe for the code to be more portable, but also for it to be correct!
- The parent waits for the child process because if it does not, the child will become a zombie in the system. You should make a habit of waiting for all processes that you create.
- The output of the parent and child on the terminal may occur in any order. This program makes no attempt to coordinate the use of the terminal simply because it would distract from its purpose as a demonstration of how to use pipes.

8.2.2 Atomic Writes

In a POSIX-compliant system, a single write will be executed atomically as long as the number of bytes to be written does not exceed `PIPE_BUF`. This means that if several processes are each writing to the pipe at the same time, as long as each limits the size of each write to N $\leq$ `PIPE_BUF` bytes, the data will not be intermingled. If there is not enough room in the pipe to store N $\leq$ `PIPE_BUF` bytes, and writes are blocking (the default), then `write()` will be blocked until room is available. On the other hand, if N $>$ `PIPE_BUF`, there is no guarantee that the writes will be atomic.

To use the value of `PIPE_BUF` in a program, include the header file `<limits.h>`. For example,

```
#include <limits.h>
char chunk[PIPE_BUF];
```

In the event that your `<limits.h>` header file does not define `PIPE_BUF`, it means that the value is greater than the POSIX minimum value for this constant, which is `POSIX_PIPE_BUF`, and is usually 512 bytes. POSIX does not require that `PIPE_BUF` be defined in this case. Therefore, you should write the above code snippet as

```
#include <limits.h>
#ifndef PIPE_BUF
```

```
        #define PIPE_BUF  POSIX_PIPE_BUF;
    #endif

    char chunk[PIPE_BUF];
```

An alternative that may work on your system is to use the `fpathconf()` system call to determine the value of the atomic write size dynamically. The `fpathconf()` system call returns the value of various system dependent configuration values associated with an open file descriptor. The `fpathconf()` function's synopsis and description is

```
#include <unistd.h>

long fpathconf(int filedes, int name);
long pathconf(char *path, int name);

DESCRIPTION
fpathconf() gets a value for the configuration option name
for the open file descriptor filedes.
```

The second argument to `fpathconf()` is a mnemonic name defined in the man page. These are names such as `_PC_NAME_MAX`, `_PC_PATH_MAX`, and `_PC_PIPE_BUF`. Each name has a different usage, and its validity depends upon whether the given file descriptor is that of a file, a directory, a pipe, or a terminal. If `filedes` is a pipe, then the constant `_PC_PIPE_BUF` *is supposed to tell* `fpathconf()` to return the maximum number of bytes that may be written atomically to that pipe. It may not return this value. This will be explained below.

Although it is not necessary to know the value of `PIPE_BUF`, it is an interesting exercise to learn its value and make sure that it has the magical properties it is supposed to have. The following program is designed to demonstrate (but not prove) that writes of up to `PIPE_BUF` bytes are atomic, and that larger writes may not be atomic. It also demonstrates how to create multiple writers and a single reader. The program creates two writer processes and one reader. One writer writes 'X's into the pipe, the other 'y's. They each write the same number of characters each time. The command line argument specifies the number of writes that each makes to the pipe. The idea is that if the number is large enough the scheduler will time slice them often enough so that one will write for a while, then the next, and so on. The parent is the reader. It reads the data from the pipe and stores it in a file. The parent reads smaller chunks since it does not matter.

Listing 8.3: pipedemo2.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <sys/wait.h>
```

```
#define  READ_FD  0
#define  WRITE_FD 1
#define  RD_CHUNK    10
#define  ATOMIC

#ifndef PIPE_BUF
    #define PIPE_BUF POSIX_PIPE_BUF
#endif

void do_nothing( int signo)
{
  return;
}


int main(int argc, char* argv[] )
{
    int i, repeat;
    int bytesread;
    int mssglen;
    pid_t child1, child2;
    int fd[2];
    int outfd;
    char  message[RD_CHUNK+1];
    char  *Child1_Chunk, *Child2_Chunk;
    long Chunk_Size;

    static struct sigaction sigact;

    sigact.sa_handler = do_nothing;
    sigfillset(&(sigact.sa_mask));
    sigaction(SIGUSR1, &sigact, NULL);


    /* check proper usage */
    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s size\n", argv[0]);
        exit(1);
    }

    /* try to create pipe */
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    repeat     = atoi(argv[1]);
#if defined ATOMIC
    Chunk_Size = PIPE_BUF;
#else
    Chunk_Size = PIPE_BUF + 200;
#endif
    printf("Chunk size = %ld\n", Chunk_Size);
    printf("Value of PIPE_BUF is %d\n", PIPE_BUF);
```

```
Child1_Chunk =  calloc(Chunk_Size, sizeof(char));
Child2_Chunk =  calloc(Chunk_Size, sizeof(char));
if ( ( NULL == Child1_Chunk  ) ||
     ( NULL == Child2_Chunk  ) ) {
    perror("calloc");
    exit(2);
}

/* create the string that child1 writes */
Child1_Chunk[0] = '\0'; /* just to be safe */
for ( i = 0; i < Chunk_Size-2; i++)
    strcat(Child1_Chunk, "X");
strcat(Child1_Chunk,"\n");

/* create the string that child2 writes */
Child2_Chunk[0] = '\0'; /* just to be safe */
for ( i = 0; i < Chunk_Size-2; i++)
    strcat(Child2_Chunk, "y");
strcat(Child2_Chunk,"\n");

/* create first child process */
switch ( child1 = fork() ) {
case -1:  /* fork failed -- exit */
    perror("fork()");
    exit(3);

case 0:    /* child1 code */
    mssglen = strlen(Child1_Chunk);
    pause();
    for ( i = 0; i < repeat; i++ ) {
        if ( write(fd[WRITE_FD], Child1_Chunk, mssglen )
             != mssglen ) {
            perror("write");
            exit(4);
        }

    }
    close(fd[WRITE_FD]);
    exit(0);
default:  /* parent creates second child process */
    switch ( child2 = fork() ) {
    case -1:  /* fork failed -- exit */
        perror("fork()");
        exit(5);

    case 0:    /* child2 code */
        mssglen = strlen(Child2_Chunk);
        pause();
        for ( i = 0; i < repeat; i++ ) {
            if ( write(fd[WRITE_FD], Child2_Chunk, mssglen )
                 != mssglen ) {
                perror("write");
                exit(6);
```

```
                }

            }
            close(fd[WRITE_FD]);
            exit(0);
        default:  /* parent code */
            outfd = open("pd2_output",
                         O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if ( -1 == outfd ) {
                perror("open");
                exit(7);
            }

            close(fd[WRITE_FD]);
            kill(child1, SIGUSR1);
            kill(child2, SIGUSR1);
            while ( ( bytesread = read( fd[READ_FD], message, RD_CHUNK) )
                      != 0 )
                if ( bytesread > 0 ) {  /* more data */
                    write(outfd, message, bytesread);
                }
                else {                  /*read error */
                    perror("read()");
                    exit(8);
                }

            close(outfd);
            /* collect zombies */
            for ( i = 1; i <= 2; i++ )
                if ( wait(NULL) == -1) {
                    perror("wait failed");
                    exit(9);
                }
            close(fd[READ_FD]);
            free(Child1_Chunk);
            free(Child2_Chunk);
        }
        exit(0);
    }
}
```

Notes.

- The parent process is the reader; the two child processes are writers. Each child calls `pause()` to start so that neither gets to grab the processor immediately. The parent sends a `SIGUSR1` signal to them when it is ready to start reading from the pipe.
- Each child write a chunk of size `Chunk_Size` into the pipe. `Chunk_Size` is either `PIPE_BUF` or 200 bytes larger than it.
- The parent reads from the pipe and writes the data into a file named `pd2_output`. When the `read()` returns 0, the children have finished writing and closed the pipe, so the parent closes the output file and calls `wait()` to collect the exit status of the children.

- The program prints the value of `PIPE_BUF` and the actual chunk size before the pipe operations begin.

First compile the program as it is written, naming the executable `pipedemo2`. When `pipedemo2` is run, the output will show that writes are atomic – each string written by each child in a single write has a newline at its end, and in the output, every sequence of `X`'s will be terminated by a newline and every sequence of `y`'s will end in a newline. There will be no occurrence of the string `Xy` or `yX` in the output because the kernel serializes the concurrent writes, and each time a child process writes, it writes its entire string, either `X`'s or `y`'s. The output does not prove it is atomic; it just shows that no output was intermingled, and thus no write was interrupted.

Each child should write two thousand times or more in order for us to see the possibility of their each competing for the shared pipe, so the program should be run with a command line argument of 2000 or more. It would be tedious to check the output by hand to determine whether there are any lines with intermingled output. The following script is designed to do this automatically:

```
#!/bin/bash
if [[ $# < 1 ]]
then
    printf "Usage: %b repeats\n" $0
    exit
fi

pipedemo2  $1
printf "Number of X lines     : "
    grep X pd2_output | wc -l
printf "Number of y lines     : "
    grep y pd2_output | wc -l
printf "X lines in first %b : " $1
    head -$1 pd2_output | grep X | wc -l
printf "y lines in first %b : " $1
    head -$1 pd2_output | grep y | wc -l
printf "X lines in last %b  : " $1
    tail -$1 pd2_output | grep X | wc -l
printf "y lines in last %b  : " $1
    tail -$1 pd2_output | grep y | wc -l
printf "Xy lines              : "
    grep Xy pd2_output  | wc -l
printf "yX lines              : "
    grep yX pd2_output  | wc -l
```

The command line argument is the number of chunks that each child should write. The script summarizes the output. If `repeats` is 1000, You should see output something like

```
Number of X lines : 1000
Number of y lines : 1000
X lines in first 1000 : 515
```

```
y lines in first 1000 : 485
X lines in last 1000 : 485
y lines in last 1000 : 515
Xy lines : 0
yX lines : 0
```

The last two lines show that all writes were atomic because there are no lines that contain an `Xy` or `yX` combination. Now edit the program by commenting out the line

```
#define ATOMIC
```

and recompile it. This flag determines how large the chunk is. When it is turned off, the chunk is larger than `PIPE_BUF` bytes. Run the script again. The output will most likely look something like this:

```
Number of X lines : 1443
Number of y lines : 1437
X lines in first 1000 : 758
y lines in first 1000 : 718
X lines in last 1000 : 685
y lines in last 1000 : 719
Xy lines : 577
yX lines : 586
```

which shows that when the chunk size of a write exceeds `PIPE_BUF`, the writes will not be atomic.

8.2.2.1 More About fpathconf()

Almost all systems comply with the POSIX requirement that result of the call

```
pipe_size = fpathconf(fd, _PC_PIPE_BUF);
```

is the system's current value of `PIPE_BUF`. But not all do. Some systems using recent versions of the GNU C Library will use a different version of `fpathconf()`. This version returns the pipe capacity, not the value of `PIPE_BUF`, but only if the kernel supports it. Linux kernels after 2.6.35 do for certain. What this implies is that you cannot reliably use the result of `fpathconf()` to determine the maximum number of bytes in an atomic write on all systems.

8.2.3 Pipe Capacity

The capacity of a pipe may be larger than `PIPE_BUF`. There is no exposed system constant that indicates the total capacity of a pipe; however, the following program, based on one from [Haviland *et al*], can be run on any system to test the maximum capacity of a pipe, and also to prove that a process cannot write to a pipe unless it has at least `PIPE_BUF` bytes available.

Listing 8.4: pipesizetest.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <limits.h>

int          count = 0;
sig_atomic_t full  = 0;

/**
 *  The SIGALRM handler. This sets the full flag to indicate that the
 *  write call blocked, and it prints the number of characters written
 *  to the pipe so far.
 */
void on_alarm( int signo)
{
    printf("\nwrite() blocked with %d chars in the pipe.\n", count);
    full = 1;
}

int main(int argc, char* argv[])
{
    int fd[2];
    int pipe_size;
    int bytesread;
    int amount_to_remove;

    char buffer[PIPE_BUF];
    char c = 'x';
    static struct sigaction sigact;

    sigact.sa_handler = on_alarm;
    sigfillset(&(sigact.sa_mask));
    sigaction(SIGALRM, &sigact, NULL);

    if ( -1 == pipe(fd) ) {
        perror("pipe failed");
        exit(1);
    }

    /* Check whether the _PC_PIPE_BUF constant returns the pipe capacity
       or the atomic write size
    */
    pipe_size = fpathconf(fd[0],  _PC_PIPE_BUF);

    while (1) {
        /* Set an alarm long enough that if write fails it will fail  */
        /* within this amount of time. 8 seconds is long enough. */
        alarm(4);
        write(fd[1], &c, 1);
        /* Unset the alarm */
        alarm(0);
```

```
    /* Did alarm expire? If so, write failed and we stop the loop */
    if ( full )
        break;

    /* Report how many chars written so far */
    if ( (++count % 1024) == 0 )
        printf ( "%d chars in pipe\n", count);
}

printf( "The maximum number of bytes that the pipe stored is %d.\n",
        count);
printf( "The value returned by fpathconf(fd,_PC_PIPE_BUF) is %d.\n\n",
        pipe_size);

printf( "Now we remove characters from the pipe and demonstrate that"
        " we cannot\n"
        "write into the pipe unless it has %d (PIPE_BUF) free bytes.\n",
         PIPE_BUF);

amount_to_remove = PIPE_BUF-1;

printf( "First we remove %d characters (PIPE_BUF-1) and try to "
        "write into the pipe.\n", amount_to_remove);

full = 0;
bytesread = read(fd[0], &buffer, amount_to_remove);
if ( bytesread < 0 ) {
    perror("error reading pipe");
    exit(1);
}
count = count - bytesread;
alarm(4);
write(fd[1], &c, 1);
/* Unset the alarm */
alarm(0);
if ( full )
    printf( "We could not write into the pipe.\n");
else
    printf( "We successfully wrote into the pipe.\n");

amount_to_remove = PIPE_BUF - amount_to_remove;
full = 0;

printf( "\nNow we remove one more character and try to "
        "write into the pipe.\n");

bytesread = read(fd[0], &buffer, amount_to_remove );
if ( bytesread < 0 ) {
    perror("error reading pipe");
    exit(1);
}
count = count - bytesread;
alarm(4);
```

```
    write(fd[1], &c, 1);
    /* Unset the alarm */
    alarm(0);
    if ( full )
        printf( "We could not write into the pipe.\n");
    else
        printf( "We successfully wrote into the pipe.\n");

    return 0;
}
```

Notes.

- The main program is the only process, and within its loop it repeatedly writes a single character to the pipe.
- Since the program never reads from the pipe, the pipe will eventually fill up. When the process attempts to write to the pipe after it is full, it will be blocked. To prevent it from being blocked forever, it sets an alarm before each `write()` call and unsets it afterwards. The alarm interval, 10 seconds, is long enough so that the alarm will never expire before a successful write finishes. When the pipe is full however, the write will be blocked indefinitely, and therefore the alarm will expire, interrupting the `write()`, and the alarm handler will display the total number of bytes written so far and then terminate the program.
- The program reports the value of `fpathconf(fd,_PC_PIPE_BUF)` in order to compare it to the actual pipe capacity. On Linux systems using recent versions of the GNU C Library, this value will be the pipe capacity, not the current value of `PIPE_BUF`.
- Once the pipe is full, the program removes `PIPE_BUF`-1 bytes from the pipe and attempts to write to it. This will fail. It then removes one more byte so that the pipe has `PIPE_BUF` bytes free, and writes to it again. This time the write will succeed.
- The program displays messages to indicate the various successes and failures.

8.2.4 Caveats and Reminders Regarding Blocking I/O and Pipes

Quite a bit can go wrong when working with pipes. These are some important facts to remember about using pipes and non-blocking reads and writes. Some of these have been mentioned already, some not. This section consolidates them into a single place.

1. If a `write()` is made to a pipe that is not open for reading by any process, a `SIGPIPE` signal will be sent to the writing process, which, if not caught, will terminate that process. If it is caught, after the `SIGPIPE` handler finishes, the `write()` will return with a -1, and `errno` will be set to the value `EPIPE`.
2. If there are one or more processes writing to a pipe, if a reading process closes its read-end of the pipe and no other processes have the pipe open for reading, each writer will be sent the `SIGPIPE` signal, and the same rules mentioned above regarding handling of the signal apply to each process.

3. As long as one writer has a pipe open for writing, a call to `read()` will remain blocked until there is data in the pipe. Therefore, if all writers finish writing to the pipe, but a single writer fails to close the write-end of the pipe, if a reader calls `read()`, the reader will remain permanently blocked. Once all writers close the write-ends of the pipe, the `read()` will return zero.

4. A `write()` to a full pipe will block the writer until there are `PIPE_BUF` free bytes in the pipe.

5. Unlike reads from a file, `read()` requests to a pipe drain the pipe of the data that was read. Therefore, when multiple readers read from the same pipe, no two read the same data.

6. Writes are atomic as long as the number of bytes is smaller than `PIPE_BUF`.

7. Reads are atomic in the sense that, if there is any data in the pipe when the call is initiated, the `read()` will return with as much data as is available, up to the number of bytes requested, and it is guaranteed not to be interrupted.

8. Processes cannot `seek()` on a pipe.

The situation is entirely different with non-blocking reading and writing. These will be discussed later. However, before continuing with the discussion of pipes, we will take a slight detour to look at I/O redirection in general, because studying I/O redirection will give us insight into some of the ways in which pipes are used.

8.3 I/O Redirection Revisited

8.3.1 Simulating Output Redirection

How does the shell implement I/O redirection? The key to understanding this rests on one simple principle used by the kernel: the `open()` system call always chooses the lowest numbered available file descriptor.

Suppose that you have entered the command

```
$ ls > listing
```

The steps taken by the shell are

1. `fork()` a new process.

2. In the new process, `close()` file descriptor 1 (standard output).

3. In the new process, `open()` (with the `O_CREAT` flag) the file named `listing`.

4. Let the new process `exec()` the `ls` program.

After step 1, the child and parent each have copies of the same file descriptors. After step 2, the child has closed standard output, so file descriptor 1 is free. In step 3, the kernel sees descriptor 3 is free, so it uses descriptor 3 to point to the file structure for the file named `listing`. Then the

child calls `exec()` passing it the string `"ls"`. The `ls` program writes to file descriptor 1, usually standard output, but in fact it is really writing to the file named `listing`. In the meanwhile, the shell continues to have descriptor 1 pointing to the standard output device, so it is unaffected by this secret trick it played on the `ls` command.

The following program, called `redirectout.c`, illustrates how this works. It simulates the shell's '`>`' operator. It forks a child, closes standard output descriptor 1, opens the output file specified in `argv[2]` for writing, and execs `argv[1]`. The parent simply waits for the child to terminate. Compile it and name it `redirectout`, and then try a command such as the following:

```
$ redirectout who whosloggedon
```

Redirecting standard input works similarly. The only difference is that the process has to close the standard input descriptor 0, and then open a file for reading.

Listing 8.5: redirectout.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char* argv[])
{
    int    fd;

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command output-file\n", argv[0]);
        exit(1);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:    /* child code */
        /* Close standard output */
        close(1);
        /* Open the file into which to redirect standard output */
        /* and check that it succeeds */
        if ( (fd = open( argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644 ))
            == -1 )
            exit(1);

        /* execute the command in argv[1] */
        execlp( argv[1],argv[1], NULL );

        /* should not reach here!      */
        perror("execlp");
        exit(1);
    default:    /* parent code; just waits for child */
        wait(NULL);
```

```
    }
    return 0;
}
```

8.3.2 Simulating the '|' Shell Operator

The pertinent question now is, how can we write a similar program that can simulate how the shell carries out a command such as

```
$ last | grep 'pts/2'
```

This cannot be accomplished using just the `open()`, `close()`, and `pipe()` system calls, because we need to connect one end of a pipe to the standard output for `last`, and the other end to the standard input for `grep`. Just closing file descriptors cannot do this. There are two system calls that can be used for this purpose: `dup()` and `dup2()`. `dup()` is the progenitor of `dup2()`, which superseded it. We will first look at a solution using `dup()`.

The `dup()` system call duplicates a file descriptor. From the man page:

```
#include <unistd.h>
int dup(int oldfd);

After a successful return from dup(), the old and new file descriptors
may be used interchangeably. They refer to the same open file description
(see open(2)) and thus share file offset and file status flags;
for example, if the file offset is modified by using lseek(2) on one
of the descriptors, the offset is also changed for the other.
```

In other words, given a file descriptor, `oldfd`, `dup()` creates a new file descriptor that points to the same kernel file structure as the old one. But again the critical feature of `dup()` is that it returns the lowest-numbered available file descriptor. Therefore, consider the following sequence of actions.

```
int fd[2];                  /* Declare descriptors for a pipe  */
pipe(fd);                   /*  Create the pipe */
switch ( fork() )           /* Fork a child */
case 0:                     /* In the child: */
    close(fileno(stdout));      /* close standard output  */
    dup(fd[1]);                 /* dup write-end of pipe  */
    close(fd[0]);               /* close read-end of pipe */
    exec("last", "last", NULL); /* exec the command that writes to the pipe */
```

The `dup()` call will find the standard output file descriptor available, and since that is the lowest numbered available descriptor, it will make that point to the same structure as `fd[1]` points to. Therefore, when the `last` command writes to standard output, it will really be writing to the write-end of the pipe.

Now it is not hard to imagine what the parent's job is. It has to close the standard input descriptor, then `dup(fd[0])`, and exec the `grep` command. We can put these ideas together in a more general program, called `shpipe1.c`, which follows.

Listing 8.6: shpipe1.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
    int  fd[2];

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:
        close(fileno(stdout));
        dup(fd[1]);
        close(fd[0]);   /* close read end since child does not use it */
        close(fd[1]);   /* close write end since it is not needed now */
        execlp( argv[1],argv[1], NULL );
        perror("execlp");
        exit(1);
    default:
        close(fileno(stdin));
        dup(fd[0]);
        close(fd[1]);  /* close write end to prevent child from blocking */
        close(fd[0]);  /* close read end since it is not needed now */
        execlp( argv[2],argv[2], NULL );
        exit(2);
  }
    return 0;
}
```

If you compile this and name it `shpipe1`, then you can try commands such as

```
$ shpipe1 last more
```

and

```
$ shpipe1 ls wc
```

There is a problem here. For one, the parent cannot wait for the child because it uses `execlp()` to replace its image. This can be solved by forking two children and letting the second do the work of the reading process. More importantly, this solution is not general, because there are two steps – close standard output and then `dup()` the write end of the pipe. There is a small window of time between closing standard output and duplicating the write-end of the pipe in which the child could be interrupted by a signal whose handler might close file descriptors so that the descriptor returned by `dup()` will not be the one that was just closed.

This is the reason that `dup2()` was created. `dup2(fd1, fd2)` will duplicate `fd1` in `fd2`, closing `fd2` if necessary, as a single atomic operation. In other words, if `fd2` is open, it will close it, and make `fd2` point to the same file structure to which `fd1` pointed. Its man page entry is

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
.....
dup2() makes newfd be the copy of oldfd, closing newfd first if necessary.
```

(`dup()` and `dup2()` share the same page. I deleted `dup2()`'s description above. This is the relevant part of it.)

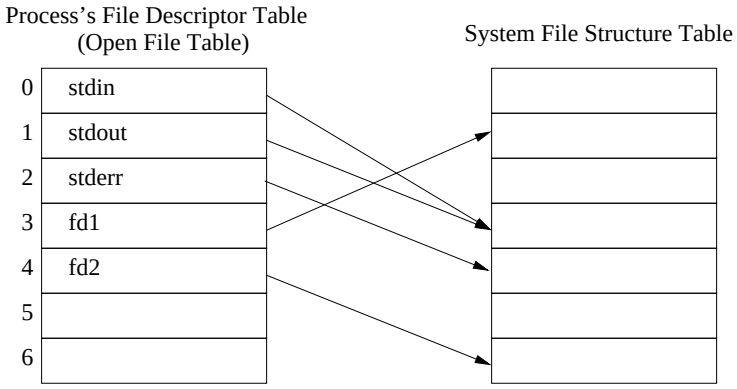


Figure 8.4: Initial state of open file table.

A picture best illustrates how `dup2()` works. Assume the initial state of the file descriptors for the process is as shown in Figure 8.4. Now suppose that the process makes the call

```
dup2( fd2 , fileno(stdin));
```

Then, after the call the table is as shown in Figure 8.5. Descriptor 0 (standard input) became a copy of `fd2` as a result of the call. Descriptor `fd2` is now redundant and can be closed if `stdin` is going to be used instead.

The following listing is of a program, `shpipe2.c` , which is an improved version of `shpipe1.c`. that uses the `dup2()` call instead of `dup()`.

Listing 8.7: shpipe2.c

```
#include <stdio.h>
#include <unistd.h>
```

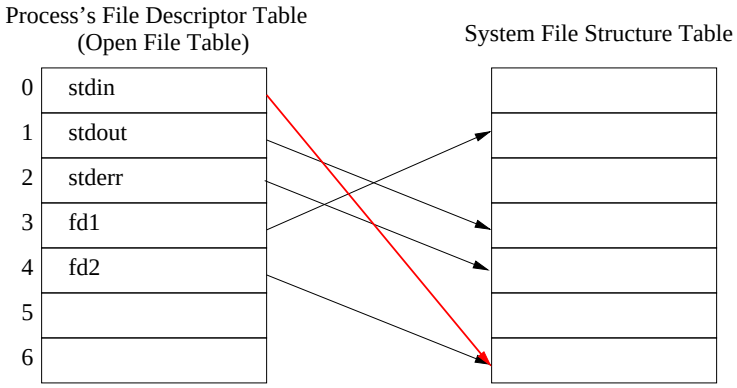


Figure 8.5: State of open file table after dup2(fd2,fileno(stdin));

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char* argv[])
{
    int     fd[2];
    int     i;
    pid_t   child1, child2;

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    switch ( child1 = fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:        /* child1 */
        dup2(fd[1],fileno(stdout));         /* now stdout points to fd[1] */
        close(fd[0]);                       /* close input end of pipe */
        close(fd[1]);                       /* close output end of pipe */
        execlp( argv[1],argv[1], NULL );    /* run the first command    */
        perror("execlp");
        exit(1);
    default:
        switch ( child2 = fork()) {
        case -1:
            perror("fork");
            exit(1);
        case  0:        /* child2 */
            dup2(fd[0],fileno(stdin));      /* now stdin points to fd[0] */
```

```
            close(fd[0]);                    /* close input end of pipe */
            close(fd[1]);                    /* close output end of pipe */
            execlp(argv[2],argv[2], NULL ); /* run the first command */
            perror("execlp");
            exit(2);
        default:
            close(fd[0]);  /* parent must close its ends of the first pipe */
            close(fd[1]);
              for ( i = 1; i <= 2; i++ )
                  if ( wait(NULL) == -1) {
                        perror("wait failed");
                        exit(3);
                    }
            return 0;
        }
    }
    return 0;
}
```

There are a couple of things you can try to do at this point to test your understanding of pipes.

1. There is a UNIX utility called `tee` that copies its input stream to standard output as well as to its file argument:

   ```
   $ ls -l | tee listing
   ```

   will copy the output of "`ls -l`" into the file named listing as well as to standard output. Try to write your own version of `tee`.

2. Extend `shpipe2` to work with any number of commands so that

   ```
   $ shpipe3 cmmd cmmd ... cmmd
   ```

   will act like

   ```
   $ cmmd | cmmd | ... | cmmd
   ```

8.3.3 The popen() Library Function

The sequence

1. generate a pipe,
2. fork a child process,
3. duplicate file descriptors, and
4. execute a new program in order to redirect the input or output of that program to the parent,

is so common that the developers of the C library added a pair of functions, `popen()` and `pclose()` to streamline this procedure:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

The `popen()` function creates a pipe, forks a new process to execute the shell `/bin/sh` (which is system dependent), and passes the command to that shell to be executed by it (using the `-c` flag to the shell, which tells it to expect the command as an argument.)

`popen()` expects the second argument to be either `"r"` or `"w"`. If it is `"r"` then the process invoking it will be returned a `FILE` pointer to the *read-end* of the pipe and the write-end will be attached to the standard output of the command. If it is `"w"`, then the process invoking it will be returned a `FILE` pointer to the *write-end* of the pipe, and the read-end will be attached to the standard input of the command. The output stream is fully buffered.

File streams created with `popen()` must be closed with `pclose()`, which will wait for the invoked process to terminate and returns its exit status or -1 if `wait4()` failed.

An example will illustrate. We will write a third version of the `shpipe` program, called `shpipe3`, using `popen()` and `pclose()` instead of the `pipe()`, `fork()`, `dup()` sequence. See Listing 8.8 below.

Listing 8.8: shpipe3.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>

int main(int argc, char* argv[])
{
    int     nbytes;
    FILE  *fin;                 /* read-end of pipe */
    FILE  *fout;                /* write-end of pipe */
    char    buffer[PIPE_BUF];   /* buffer for transferring data */

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( (fin = popen(argv[1], "r")) == NULL ) {
        fprintf(stderr, "popen() failed\n");
        exit(1);
    }

    if ( (fout = popen(argv[2], "w")) == NULL ) {
        fprintf(stderr, "popen() failed\n");
        exit(1);
    }

    while ( (nbytes = read(fileno(fin), buffer, PIPE_BUF)) > 0 )
        write(fileno(fout), buffer, nbytes);

    pclose(fin);
```

```
    pclose(fout);
    return 0;
}
```

8.4 Named Pipes

Unnamed pipes are an elegant mechanism, however, they have several drawbacks. They can only be shared by processes with a common ancestor, such as a parent and child, or multiple children or descendants of a parent that created the pipe. Also, they cease to exist as soon as the processes that are using them terminate, so they must be recreated every time they are needed. If you are trying to write a server program with which clients can communicate, the clients will need to know the name of the pipe through which to communicate, but an unnamed pipe has no such name.

Named pipes make up for these shortcomings. A named pipe, or *FIFO*, is very much like an unnamed pipe in how you use it. You read from it and write to it in the same way. It behaves the same way with respect to the consequences of opening and closing it when various processes are either reading or writing or doing neither. In other words, the semantics of opening, closing, reading, and writing named and unnamed pipes are the same.

What distinguishes named pipes from unnamed pipes is that

- They exist as directory entries in the file system and therefore have associated permissions and ownership[2].
- They can be used by processes that are not related to each other.
- They can be created and deleted at the shell level or at the programming level.

8.4.1 Named Pipes at the Command Level

Before we look at how they are created within a program, let us look at how they are created at the user level. There are two commands to create a FIFO. The older command is `mknod`. `mknod` is a general purpose utility for creating device special files. There is also a `mkfifo` command, which can only be used for creating a FIFO file. We will look at how to use `mknod`. You can read about the `mkfifo` command in the man pages.

```
$ mknod PIPE p
```

creates a FIFO named `"PIPE"`. The lowercase `p` , which must follow the file name, indicates to `mknod` that `PIPE` should be a FIFO (`p` for pipe.) After typing this command, look at the working directory:

```
$ ls -l PIPE
prw-r--r-- 1 stewart stewart 0 Apr 30 22:29 PIPE|
```

[2]Although they have directory entries, they do not exist in the file system. They have no disk blocks and their data is not on disk when they are in use.

The 'p' file type indicates that `PIPE` is a FIFO. Notice that it has 0 bytes. Try the following command sequence:

```
$ cat < PIPE &
$ ls -l > PIPE; wait
```

If we do not put the `cat` command into the background it will hang because a process trying to read from a pipe will block until there is at least one process trying to write to it. The `cat` command is trying to read from `PIPE` and so it will not return and you will not get the shell prompt back without backgrounding it. The `cat` command will terminate as soon as it receives the return value 0 from its `read()` call, which will be delivered when the writer closes the file after it is finished writing. In this case the writer is the process that executes "`ls -l`". When the output of `ls -l` is written to the pipe, `cat` will read it and display it on the screen. The `wait` command's only purpose is to delay the shell's prompt until after `cat` exits.

By the way, if you reverse this procedure:

```
$ ls -l > PIPE &
$ ls -l PIPE
$ cat < PIPE; wait
```

and expect to see that the `PIPE` does not have 0 bytes when the second `ls -l` is executed, you will be disappointed. That data is not stored in the file system.

8.4.2 Programming With Named Pipes

We turn to the creation and use of named pipes at the programming level. A named pipe can be created either by using the `mknod()` system call, or the `mkfifo()` library function. In Linux, according to the `mknod()` (2) man page,

"*Under Linux, this call cannot be used to create directories. One should make directories with mkdir(2), and FIFOs with mkfifo(3).*"

Therefore, we will stick to using `mkfifo()` for creating FIFOs. The other advantage of `mkfifo()` over `mknod()` is that it is easier to use and does not require superuser privileges:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

The call `mkfifo("MY_PIPE", 0666)` creates a FIFO named `MY_PIPE` with permission `0666 & ~umask`. The convention is to use UPPERCASE letters for the names of FIFOs. This way they are easily identified in directory listings.

It is useful to distinguish between *public* and *private* FIFOs. A *public FIFO* is one that is known to all clients. It is not that there is a specific function that makes a FIFO public; it is just that it is given a name that is easy to remember and that its location is advertised so that client programs know where to find it. Some authors call these *well-known FIFOs*, because they are analogous to

well-known ports used for sockets, which are covered later. A *private FIFO*, in contrast, is given a name that is not known to anyone except the process that creates it and the processes to which it chooses to divulge it. In our first example, we will use only a single public FIFO. In the second example, the server will create a public FIFO and the clients will create private FIFOs that they will each use exclusively for communicating with the server.

8.4.2.1 Example

This is a simple example that demonstrates the basic principles. In it, the server creates a public FIFO and listens for incoming messages. When a message is received, it just prints it on the console. Client programs know the name of the FIFO because its pathname is hard-coded into a publicly available header file that they can include. In fact, for this example, the server and the clients share this common header file. Ideally the FIFO's name should be chosen so that no other processes in the system would ever choose the same file name, but for simplicity, we use a name that may not be unique. [3].

The server will execute a loop of the form

```
while ( 1 ) {
    memset(buffer, 0, PIPE_BUF);
    if ( ( nbytes = read( publicfifo, buffer, PIPE_BUF)) > 0 ) {
        buffer[nbytes] = '\0';
        printf("Message %d received by server: %s", ++count, buffer);
        fflush(stdout);
    }
    else
         break;
}
```

In each iteration, it begins by zeroing the buffer into which it will copy the FIFO's contents. It reads at most `PIPE_BUF` bytes at a time into the buffer. When `read()` returns, if `nbytes` is positive, it null-terminates the buffer and writes what it received onto its controlling terminal. Because the input data may not have a terminating newline, it forces the write by calling `fflush(stdout)`. If `nbytes` is negative, there was an error and the server quits. If `nbytes` is 0, it means that `read()` returned without any data, and so there is nothing for it to write. We could design the loop so that it does not exit in this case but just re-executes the `read()`, but there are reasons not to, as we now explain.

The server has to perform blocking reads (the `O_NONBLOCK` and `O_NDELAY` flags are clear), otherwise it would continually run in a loop, needlessly calling `read()` until a client actually wrote to the FIFO. This would be a waste of CPU cycles. By using a blocking read, it relinquishes the CPU so that it can be used for other purposes. The problem is that the `read()` call will return 0 when there are no processes writing to the FIFO, so if no clients attempt to write to the server, or if all clients that were writing close their ends of the FIFO and exit, the server would receive a 0 from the `read()`. If we designed the loop so that it was re-entered when `read()` returned 0, this would not be a problem. However, it is a cleaner design to let the server open the FIFO for writing, so that

[3] There are programs that can generate unique keys of an extremely large size that can be used in the name of the file. If all applications cooperate and use this method, then all pipe names would be unique.

there is always at least one process holding the FIFO open for writing, and so that the return value of `read()` will be either positive or negative, unless there is some unanticipated error condition.

Therefore, the server begins by creating the FIFO and opening it for both reading and writing, even though it will only read from it. Since the server never writes to this pipe, it does not matter whether or not writes are non-blocking, but POSIX does not specify how a system is supposed to handle opening a file in blocking mode for both reading and writing, so it is safer to open it with the `O_NONBLOCK` flag set, since POSIX does not specify how a system is supposed to handle opening a file in blocking mode for both reading and writing, we avoid possibly undefined behavior.

The server is run as a background process and is the process that must be started first, so that it can create the FIFO. If the server is not running and a client is started up, it will exit, because the FIFO does not exist.

The common header file is listed first, in Listing 8.9, followed by the server code.

Listing 8.9: fifo1.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include <sys/stat.h>

#define  PUBLIC         "/tmp/FIFODEMO1_PIPE"
```

Listing 8.10: rcvfifo1.c

```
#include <signal.h>
#include "fifo1.h"

int    dummyfifo;        /* file descriptor to write-end of PUBLIC */
int    publicfifo;       /* file descriptor to read-end of PUBLIC */

/** on_signal()
 *  This closes both ends of the FIFO and then removes it, after
 *  which it exits the program.
 */
void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    unlink(PUBLIC);
    exit(0);
}

int main( int argc, char *argv[])
{
    int              nbytes;          /* number of bytes read from popen() */
    int              count = 0;
    static char      buffer[PIPE_BUF];/* buffer to store output of command */
```

```
    struct sigaction handler;          /* sigaction for registering handlers*/

    /* Register the signal handler to handle a few signals */
    handler.sa_handler = on_signal;       /* handler function    */
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    /* Create public FIFO. If it exists already, the call will return -1 and
       set errno to EEXIST. This is not an error in our case. It just means
       we can reuse an existing FIFO that we created but never removed. All
       other errors cause the program to exit.
    */
    if ( mkfifo(PUBLIC, 0666) < 0 )
        if (errno != EEXIST ) {
            perror(PUBLIC);
            exit(1);
        }

    /*
      We open the FIFO for reading, with the O_NONBLOCK flag clear. The POSIX
      semantics state the the process will be blocked on the open() until some
      process (to be precise, some thread) opens it for writing. Therefore, the
      server will be stuck in this open() until a client starts up.
    */
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 )  {
        perror(PUBLIC);
        exit(1);
    }

    /*
      We now open the FIFO for writing, even though we have no intention of
      writing to the FIFO.  We will not reach the call to open()
      until a client runs, but once the client runs, the server opens the FIFO
      for writing. If we do not do this, when the client terminates and closes
      its write-end of the FIFO, the server's read loop would exit and the
      server would also exit.  This "dummy" open keeps the server alive.
    */
    if ( (dummyfifo = open(PUBLIC, O_WRONLY | O_NONBLOCK )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }

    /* Block waiting for a message from a client */
    while ( 1 ) {
        memset(buffer, 0, PIPE_BUF);
        if ( ( nbytes = read( publicfifo, buffer, PIPE_BUF)) > 0 ) {
            buffer[nbytes] = '\0';
```

```
            printf("Message %d received by server: %s", ++count, buffer);
            fflush(stdout);
        }
        else
             break;
    }
    return 0;
}
```

Comments.

- The server reads from the public FIFO and displays the message it receives on its standard output, even though it may be put in the background; it is not detached from the terminal. The best way to run it is to leave it in the foreground and open a few clients in other terminal windows.
- The server increments a counter and displays each received message with the value of the counter, so that you can see the order in which the messages were received. As noted above, it flushes standard output just in case there is no newline in the message.
- It does detect a few signals, so that any of them are delivered to it, it will close its ends of the FIFO, remove the file, and bail out.

The client opens the public FIFO for writing and then enters a loop where it repeatedly reads from standard input and writes into the write-end of the public FIFO. It uses the library function `memset()`, found in `<string.h>`, to zero the buffer where the user's text will be stored, and it declares the buffer to be `PIPE_BUF` chars, so that the write will be atomic. (If the locale uses two-byte chars, this will not work properly.) When it is finished, it closes its write-end.

Listing 8.11: sendfifo1.c

```
#include "fifo1.h"
#define   QUIT          "quit"

int main( int argc, char *argv[])
{
    int     nbytes;          /* num bytes read */
    int     publicfifo;      /* file descriptor to write-end of PUBLIC */
    char    text[PIPE_BUF];


    /* Open the public FIFO for writing */
    if ( (publicfifo = open(PUBLIC, O_WRONLY) ) == -1) {
        perror(PUBLIC);
        exit(1);
    }

    printf("Type 'quit' to quit.\n");

    /* Repeatedly prompt user for command, read it, and send to server */
    while (1) {
```

```
        memset(text, 0, PIPE_BUF);                  /* zero string */
        nbytes = read(fileno(stdin), text, PIPE_BUF);
        if ( !strncmp(QUIT, text, nbytes-1))        /* is it quit? */
            break;

        if ( ( write(publicfifo, text, nbytes)) < 0 ) {
            perror("Server is no longer running");
            break;
        }
    }
    /* User quit, so close write-end of public FIFO */
    close(publicfifo);
    return 0;
}
```

Comments.

- The client code allows the user to type "`quit`" to end the program.
- It is not very robust; it does not handle any terminal interrupts or signals and does no clean-up if it is killed by a signal. If the server stops running though, it will detect this and exit, closing its end of the FIFO.

8.4.3 An Iterative Server

In this example, we create a server that has two way communication with each client, processing incoming client requests one after the other. Such a server is called an *iterative server*. In order to achieve this, the server creates a public FIFO that it uses for reading incoming messages from clients wishing to use its services. Each incoming message is a structure with a member that contains the name of the private FIFO that the client creates when it starts up, and which should be used by the server for sending a reply. The message structure also contains another field that the client can use to supply data for the server.

When the server receives a message, it looks at the FIFO name in it and tries to open it for writing. If successful, the server will use this FIFO for sending data to the client. After the client sends its message to the server, it opens its private FIFO for reading. It will block until the server opens the write end of this FIFO. When the server opens the write end, the client will read from it until it receives a return value of 0, indicating that the server has finished writing and closed its end of the pipe. Figure 8.6 depicts the relationship between the clients and the server with respect to the shared pipes.

In this particular example, the server provides lowercase-to-uppercase translation for clients. The clients send it a piece of text and the server sends back another piece of text identical to the first except that every lowercase letter has been converted to uppercase. The server will be named `upcased1` (for *uppercase daemon*), and the client, `upcaseclient1`.

The message structure used by the server and client, as well as all necessary include files and common definitions, is contained in the header file `upcase1.h`, displayed in the following listing.

Listing 8.12: upcase1.h

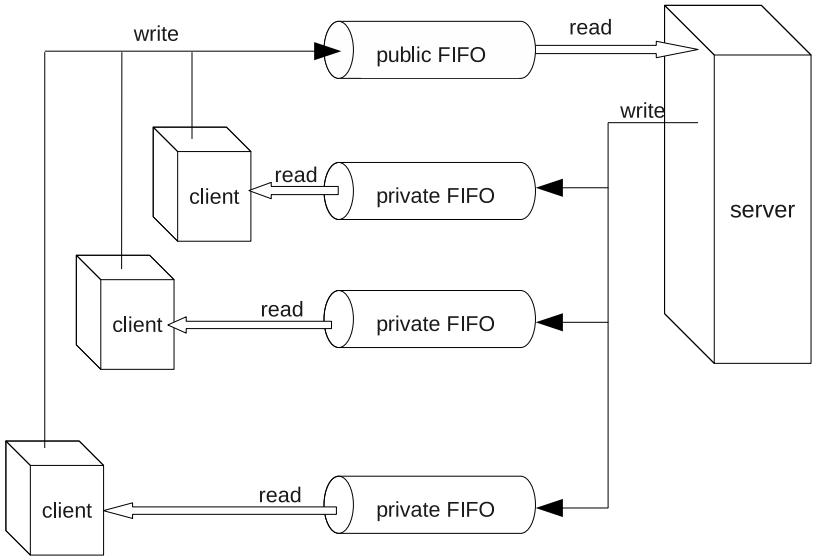


Figure 8.6: The FIFOs used in the iterative server.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <ctype.h>

#define PUBLIC          "/tmp/UPCASE1_PIPE"
#define HALFPIPE_BUF    (PIPE_BUF/2)

typedef struct _message {
    char    fifo_name[HALFPIPE_BUF];   /* private FIFO pathname */
    char    text[HALFPIPE_BUF];        /* message text          */
} message;
```

Because the message must be no larger than `PIPE_BUF` bytes, and because it should be general enough to allow FIFO pathnames of a large size, the structure is split equally between the length of the FIFO name and the length of the text to be sent to the server. Thus, `HALFPIPE_BUF` is defined as one half of `PIPE_BUF` and used as the maximum number of bytes in the string to be translated.

We begin with the client code this time. The basic steps that the client takes are as follows.

1. It makes sure that neither standard input nor output is redirected.
2. It registers its signal handlers.
3. It creates its private FIFO in `/tmp`.

4. It tries to open the public FIFO for writing in non-blocking mode.
5. It enters a loop in which it repeatedly
   (a) reads a line from standard input, and
   (b) repeatedly
      i. gets the next `HALFPIPE_BUF-1` sized chunk in the input text,
      ii. sends a message to the server through the public FIFO,
      iii. opens its private FIFO for reading,
      iv. reads the server's reply from the private FIFO,
      v. copies the server's reply to its standard output, and
      vi. closes the read-end of its private FIFO.
6. It closes the write-end of the public FIFO and removes its private FIFO.

The client listing follows.

Listing 8.13: upcaseclient1.c

```
#include "upcase1.h"  /* All required header files are included in  */
                      /* this shared header file. */

#define  PROMPT   "string: "
#define  UPCASE   "UPCASE: "
#define  QUIT     "quit"

const char   startup_msg[] =
"upcased1 does not seem to be running. "
"Please start the service.\n";

volatile sig_atomic_t  sig_received = 0;
struct message   msg;

/*************************************************************************/
/*                          Signal Handlers                              */
/*************************************************************************/

void on_sigpipe( int signo )
{
    fprintf(stderr, "upcased is not reading the pipe.\n");
    unlink(msg.fifo_name);
    exit(1);
}

void on_signal( int sig )
{
    sig_received = 1;
}
/*************************************************************************/
/*                            Main Program                               */
/*************************************************************************/
```

```
int main( int argc, char *argv[])
{
    int             strLength;      /* number of bytes in text to convert     */
    int             nChunk;         /* index of text chunk to send to server  */
    int             bytesRead;      /* bytes received in read from server     */
    int             privatefifo;    /* file descriptor to read-end of PRIVATE */
    int             publicfifo;     /* file descriptor to write-end of PUBLIC */
    static char     buffer[PIPE_BUF];
    static char     textbuf[BUFSIZ];

    struct sigaction handler;

    /* Only run if we are using the terminal. */
    if ( !isatty(fileno(stdin)) || !isatty(fileno(stdout)) )
        exit(1);

    /* Register the on_signal handler to handle all keyboard signals */
    handler.sa_handler = on_signal;       /* handler function    */
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
            perror("sigaction");
            exit(1);
    }

    /* Create hopefully unique name for private FIFO using process-id */
    sprintf(msg.fifo_name, "/tmp/fifo%d",getpid());

    /* Create the private FIFO  */
    if ( mkfifo(msg.fifo_name, 0666) < 0 ) {
        perror(msg.fifo_name);
        exit(1);
    }

    /* Open the public FIFO for writing */
    if ( (publicfifo = open(PUBLIC, O_WRONLY | O_NONBLOCK) ) == -1) {
        if ( ENXIO == errno )
            fprintf(stderr,"%s", startup_msg);
        else
            perror(PUBLIC);
        exit(1);
    }
    printf("Type 'quit' to quit.\n");

    /* Repeatedly prompt user for input, read it, and send to server */
```

```
    while (1) {
        /* Check if SIGINT received first, and if so, close write-end */
        /* of public fifo, remove private fifo and then quit */
        if ( sig_received ) {
            close(publicfifo);
            unlink(msg.fifo_name);
            exit(1);
        }

        /* Display a prompt on the terminal and read the input text */
        write( fileno(stdout), PROMPT, sizeof(PROMPT));
        memset(msg.text, 0x0, HALFPIPE_BUF);          /* zero string */
        fgets(textbuf, BUFSIZ, stdin);
        strLength = strlen(textbuf);
        if ( !strncmp(QUIT, textbuf, strLength-1))    /* is it quit? */
            break;

        /* Display label for returned upper case text */
        write(fileno(stdout), UPCASE, sizeof(UPCASE));

        for ( nChunk = 0; nChunk < strLength; nChunk += HALFPIPE_BUF-1 ) {
            memset(msg.text, 0x0, HALFPIPE_BUF);
            strncpy(msg.text, textbuf+nChunk, HALFPIPE_BUF-1);
            msg.text[HALFPIPE_BUF-1] = '\0';
            write(publicfifo, (char*) &msg, sizeof(msg));

            /* Open the private FIFO for reading to get output of command */
            /* from the server. */
            if ((privatefifo = open(msg.fifo_name, O_RDONLY) ) == -1) {
                perror(msg.fifo_name);
                exit(1);
            }

            /* Read maximum number of bytes possible atomically */
            /* and copy them to standard output. */
            while ((bytesRead= read(privatefifo, buffer, PIPE_BUF)) > 0) {
                write(fileno(stdout), buffer, bytesRead);
            }
            close(privatefifo);       /* close the read-end of private FIFO */
        }
    }
    /* User quit, so close write-end of public FIFO and delete private FIFO */
    close(publicfifo);
    unlink(msg.fifo_name);
    return 0;
}
```

Comments.

- The program registers `on_signal()` to handle all signals that could kill it and that can be generated by a user. If any of these signals is sent to the process, the handler simply sets an atomic flag. In its main loop, it checks whether the flag is set, and if it is, it closes the

write-end of the public FIFO and removes its private FIFO. The server will get a `SIGPIPE` signal the next time it tries to write to this FIFO, which it will handle.

- The program will get a `SIGPIPE` signal if it tries to write to the public FIFO but it is not open for reading. This can only happen if the server is not running. The `SIGPIPE` handler, `on_sigpipe()`, displays a message on standard error and terminates the program.
- The reason that the client opens the public FIFO with `O_NONBLOCK` set is that, in this case, if the server is not reading the FIFO, the client, instead of blocking, will return with a `ENXIO` error, so that it can gracefully exit.
- Inside the client's main loop, it displays a prompt and uses `fgets()` to read a line from the terminal.
- This client has been designed to handle the highly improbable case that the user enters a string that is larger than the allowed number of bytes in an atomic write to a pipe[4]. It does this by breaking the string into "chunks" that are small enough to send atomically. It send each chunk in sequence. It has to open and close the private FIFO before and after each chunk is sent because the server is designed primarily for handling the most likely case in which the string is small enough to fit into a single chunk. (The server only opens the client's private FIFO after receiving a message from the client with the name of the FIFO; if the client tries to open the FIFO for reading before sending any chunks, it will block on the `open()` call. To prevent this, the `open()` would have to be non-blocking, which would complicate its read loop. It is not worth the complication to save the run-time cost in this unusual case.)

Now we turn to the server, which is simpler than the client in this example. The steps that the server takes can be summarized as follows.

1. It registers its signal handlers.
2. It creates the public FIFO. If it finds it already exists, it displays a message and exits.
3. It opens the public FIFO for both reading and writing, even though it will only read from it.
4. It enters its main-loop, where it repeatedly
   (a) does a blocking read on the public FIFO,
   (b) on receiving a message from the `read()`, tries to open the private FIFO of the client that sent it the message. (It tries 5 times, sleeping a bit between each try, in case the client was delayed in opening it for writing. After 5 attempts it gives up on this client.)
   (c) converts the message to uppercase,
   (d) writes it to the private FIFO of the client, and
   (e) closes the write-end of the private FIFO.

It will loop forever because it will never receive an end-of-file on the pipe, since it is keeping the write-end open itself. It is terminated by sending it a signal. The code follows.

[4]Since `BUFSIZ`, the maximum size string allowed in the Standard I/O Library, may be larger than `PIPE_BUF`, it is possible to read a string much larger than can be sent in the pipe atomically.

Listing 8.14: upcased1.c

```
#include "upcase1.h"

#define  WARNING  "\nNOTE: SERVER ** NEVER ** accessed private FIFO\n"
#define  MAXTRIES 5

int         dummyfifo;     /* file descriptor to write-end of PUBLIC */
int         privatefifo;   /* file descriptor to write-end of PRIVATE */
int         publicfifo;    /* file descriptor to read-end of PUBLIC */

void on_sigpipe( int signo )
{
    fprintf(stderr, "Client is not reading the pipe.\n");
}

void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    if ( privatefifo != -1 )
        close(privatefifo);
    unlink(PUBLIC);
    exit(0);
}

/*************************************************************************/
/*                              Main Program                             */
/*************************************************************************/

int main( int argc, char *argv[])
{
    int              tries;         /* num tries to open private FIFO */
    int              nbytes;        /* number of bytes read from popen()  */
    int              i;
    int              done;          /* flag to stop loop */
    struct message   msg;           /* stores private fifo name and command */
    struct sigaction handler;       /* sigaction for registering handlers */


    /* Register the signal handler  */
    handler.sa_handler = on_signal;
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
```

```
        perror("sigaction");
        exit(1);
}

/* Create public FIFO */
if ( mkfifo(PUBLIC, 0666) < 0 ) {
    if (errno != EEXIST )
        perror(PUBLIC);
    else
        fprintf(stderr, "%s already exists. Delete it and restart.\n",
                PUBLIC);
    exit(1);
}


/* Open public FIFO for reading and writing so that it does not get an
   EOF on the read-end while waiting for a client to send data.
   To prevent it from hanging on the open, the write-end is opened in
   non-blocking mode. It never writes to it.
*/
if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
     ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
    perror(PUBLIC);
    exit(1);
}

/* Block waiting for a msg struct from a client */
while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {
    /* A msg arrived, so start trying to open write end of private FIFO */
    tries = done = 0;
    privatefifo = -1;
    do {
        if ( (privatefifo = open(msg.fifo_name,
                                 O_WRONLY | O_NDELAY)) == -1 )
            sleep(1);   /* sleep if failed to open */
        else {
            /* Convert the text to uppercase  */
            nbytes = strlen(msg.text);
            for ( i = 0; i < nbytes; i++ )
                if ( islower(msg.text[i]))
                    msg.text[i] = toupper(msg.text[i]);

            /* Send converted text to client  */
            if ( -1 == write(privatefifo, msg.text, nbytes) ) {
                if ( errno == EPIPE )
                    done = 1;
            }
            close(privatefifo);  /* close write-end of private FIFO */
            done = 1;            /* terminate loop */
        }
    } while (++tries < MAXTRIES && !done);

    if ( !done)
        /* Failed to open client private FIFO for writing */
```

```
            write(fileno(stderr), WARNING, sizeof(WARNING));
    }
    return 0;
}
```

Comments.

This server handles all user-initiated terminating signals by closing any descriptors that it has open and removing the public FIFO and exiting. It sets `privatefifo` to -1 at the start of each loop, and if it opens the private FIFO successfully, `privatefifo` is no longer -1. This way, in the signal handler, it can determine whether it had a private FIFO open for writing and needs to close it.

If it gets a `SIGPIPE` because a client closed its read end of its private FIFO immediately after sending a message but before the server wrote back the converted string, it handles `SIGPIPE` by continuing to listen for new messages and giving up on the write to that pipe.

8.4.4 Concurrent Servers

The preceding server was an iterative server; it handled each client request one after the other. If some client requests could be very time-consuming, then the server would be busy servicing one client to the exclusion of all others, and the others would experience delays. This can be avoided by allowing the server to handle multiple clients simultaneously. A server that can process requests from more than one client simultaneously is called a *concurrent server*.

The easiest way to create a concurrent server is to fork a child process for each client[5]. The server's role then amounts to little more than "listening" to the public pipe for incoming requests, forking a child process to handle a new request, and waiting for its children to finish. The waiting must be accomplished through a `SIGCHLD` handler, because, unlike a shell-style application, this process has to return immediately to the task of reading the public pipe. The basic outline of the server program's main process is therefore roughly:

1. It registers its signal handlers.
2. It creates the public FIFO. If it finds it already exists, it displays a message and exits.
3. It opens the public FIFO for both reading and writing, even though it will only read from it.
4. It enters its main-loop, where it repeatedly
   (a) does a blocking `read()` on the public FIFO,
   (b) on receiving a message from the `read()`, forks a child process to handle the client request.

Aside from spawning child processes, there are a few major differences between the way this server works and the way the sequential server worked:

Each client will have two private FIFOs: one into which it writes raw text to be translated, and a second from which it reads text that the server translated and sent back to it. The names of these two FIFOs must be sent to the server's public FIFO when a client wishes its services. Therefore,

[5]When we cover threads, you will see that threads are another means of accomplishing this.

the message structure is different in this program than it was in the iterative server. We will call this message a *connection message*, because its only purpose is to establish the means by which the client and the server can communicate privately. A connection message contains only the information needed to establish this two-way private communication between the server and the client:

```
typedef struct _message {
    char raw_text_fifo [HALFPIPE_BUF];
    char converted_text_fifo[HALFPIPE_BUF];
} message;
```

Each child process forked by the server begins by opening the read-end of the client's "raw_text" FIFO, and then it repeatedly reads from the this raw_text FIFO, translates the text into uppercase, opens the write-end of the client's converted_text FIFO, writes the converted text into it, and closes the write-end of the converted_text FIFO, until it received an end-of-file from the client.

8.4.4.1 The Concurrent Server Client

The client is also structurally different from the previous client. The major steps that it takes are as follows.

1. It registers its signal handlers.
2. It creates two private FIFOs in the `/tmp` directory with unique names.
3. It opens the server's public FIFO for writing.
4. It sends the initial message structure containing the names of its two FIFOs to the server to establish the two-way communication.
5. It attempts to open its raw_text FIFO in non-blocking, write-only mode. If it fails, it delays a second and retries. It retries a few times and then gives up and exits. If it fails it means that the server is probably terminated.
6. Until it receives an end-of-file on its standard input, it repeatedly
   (a) reads a line from standard input,
   (b) breaks the line into `PIPE_BUF`-sized chunks,
   (c) sends each chunk successively to the server through its raw_text FIFO,
   (d) opens the converted_text FIFO for reading,
   (e) reads the converted_text FIFO, and copies its contents to its standard output, and
   (f) closes the read-end of the converted_text FIFO
7. It closes all of its FIFOs and removes the files.

Figure 8.7 shows how the client processes and the server parent and child processes use the various FIFOs. Compare this to Figure 8.6.

The code for the client is displayed first, in the following Listing.

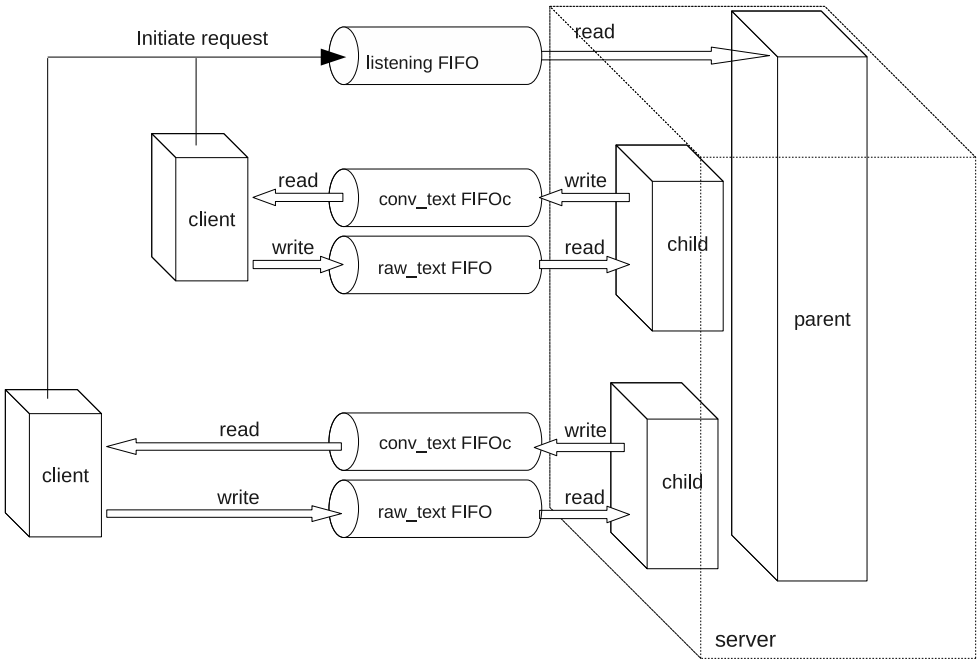


Figure 8.7: Concurrent server and client communication.

Listing 8.15: upcaseclient2.c

```
#include "upcase2.h"  /* All required header files are included in this */
                      /* shared header file.  */

#define  MAXTRIES 5

const char   startup_msg[] =
       "The upcased2 server does not seem to be running. "
       "Please start the service.\n";

const char   server_no_read_msg[] =
       "The server is not reading the pipe.\n";

int          convertedtext_fd; /* file descriptor for READ PRIVATE FIFO  */
int          dummyreadfifo;    /* to hold fifo open                     */
int          rawtext_fd;       /* file descriptor to WRITE PRIVATE FIFO  */
int          dummyrawfifo_fd;  /* to hold the raw text fifo open        */
int          publicfifo;       /* file descriptor to write-end of PUBLIC */
FILE*        input_srcp;       /* File pointer to input stream          */
message      msg;              /* 2-way communication structure         */

/*  Signal Handlers and Utilities  */

void on_sigpipe( int signo )
{
    fprintf(stderr, "upcased is not reading the pipe.\n");
    unlink(msg.raw_text_fifo);
```

```
    unlink(msg.converted_text_fifo);
    exit(1);
}

void on_signal( int sig )
{
    close(publicfifo);
    if ( convertedtext_fd != -1 )
        close(convertedtext_fd);
    if ( rawtext_fd != -1 )
        close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
    exit(0);
}

void clean_up()
{
    close(publicfifo);
    close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
}

/*************************************************************************/
/*                              Main Program                             */
/*************************************************************************/

int main( int argc, char *argv[])
{
    int              strLength;     /* number of bytes in text to convert   */
    int              nChunk;        /* index of text chunk to send to server*/
    int              bytesRead;     /* bytes received in read from server   */
    static char      buffer[PIPE_BUF];
    static char      textbuf[BUFSIZ];
    struct sigaction handler;
    int              tries;      /* for counting tries to open rawtext fifo */

    /* Check whether there is a command line argument, and if so, use it as
       the input source. */
    if ( argc > 1 ) {
        if ( NULL == (input_srcp  = fopen(argv[1], "r")) ) {
            perror(argv[1]);
            exit(1);
        }
    }
    else
        input_srcp = stdin;

    /* Initialize the file descriptors for error handling */
    publicfifo = -1;
    convertedtext_fd   = -1;
    rawtext_fd = -1;
```

```
    /* Register the on_signal handler to handle all signals */
    handler.sa_handler = on_signal;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    /* Create unique names for private FIFOs using process-id */
    sprintf(msg.converted_text_fifo, "/tmp/fifo_rd%d",getpid());
    sprintf(msg.raw_text_fifo, "/tmp/fifo_wr%d",getpid());

    /* Create the private FIFOs */
    if ( mkfifo(msg.converted_text_fifo, 0666) < 0 ) {
        perror(msg.converted_text_fifo);
        exit(1);
    }

    if ( mkfifo(msg.raw_text_fifo, 0666) < 0 ) {
        perror(msg.raw_text_fifo);
        exit(1);
    }

    /* Open the public FIFO for writing */
    if ( (publicfifo = open(PUBLIC, O_WRONLY | O_NDELAY) ) == -1) {
        if ( errno == ENXIO )
            fprintf(stderr,"%s", startup_msg);
        else
            perror(PUBLIC);
        exit(1);
    }

    /* Send a message to server with names of two FIFOs */
    write(publicfifo, (char*) &msg, sizeof(msg));

    /* Try to open the raw text FIFO for writing. After MAXTRIES
       attempts we give up. */
    tries = 0;
    while (((rawtext_fd = open(msg.raw_text_fifo,
             O_WRONLY | O_NDELAY )) == -1 ) && (tries < MAXTRIES ))  {
            sleep(1);
            tries++;
    }
    if ( tries == MAXTRIES ) {
        fprintf(stderr, "%s", server_no_read_msg);
```

```
        clean_up();
        exit(1);
    }

    /* Get one line of input at a time from the input source */
    while (1) {
        memset(textbuf, 0x0, BUFSIZ);
        if ( NULL == fgets(textbuf, BUFSIZ, input_srcp) )
            break;

        strLength = strlen(textbuf);

        /* Break input lines into chunks and send them one at a */
        /* time through the client's write FIFO */
        for ( nChunk = 0; nChunk < strLength; nChunk += PIPE_BUF-1 ) {
            memset(buffer, 0x0, PIPE_BUF);
            strncpy(buffer, textbuf+nChunk, PIPE_BUF-1);
            buffer[PIPE_BUF-1] = '\0';
            write(rawtext_fd, buffer, strlen(buffer));

            /* Open the private FIFO for reading to get output of command */
            /* from the server. */
            if ((convertedtext_fd = open(msg.converted_text_fifo, O_RDONLY) )
                  == -1) {
                perror(msg.converted_text_fifo);
                exit(1);
            }
            memset(buffer, 0x0, PIPE_BUF);
            while ((bytesRead= read(convertedtext_fd, buffer, PIPE_BUF)) > 0)
                write(fileno(stdout), buffer, bytesRead);

            close(convertedtext_fd);
            convertedtext_fd   = -1;
        }
    }
    /* User quit, so close write-end of public FIFO and delete private FIFO */
    close(publicfifo);
    close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
    return 0;
}
```

Comments.

- The order of events here is important, and in some cases critical. After the client creates its private FIFOs without error, it opens the write-end of the server's public FIFO. It then sends a message containing the names of its private FIFOs. After sending the names of the private FIFOs, it tries to open the write-end of its raw_text FIFO in non-blocking mode. This will fail if the server has not opened the read-end yet. Assuming that the server is running, the client will succeed in opening the raw_text FIFO. The server can open its read-end without the write-end being open, so this works well. If we were to reverse the order and open the

raw_text FIFO before sending the server the message, we would need to open it in read-write mode since the server is blocked on its read of the public FIFO and the two processes would deadlock otherwise. But if we open the raw_text FIFO in read-write mode, then if the server terminates unexpectedly and never reads the raw_text FIFO again, the client will not get a `SIGPIPE` signal because the client itself has a read-end open, preventing the kernel from generating the signal. The client would never be notified that the server died.

- The client then keeps the write-end of its raw_text FIFO open for the duration of its main loop.
- Within the loop, the client first writes to its raw_text FIFO, and then opens its converted_text FIFO, after which, if all goes well, it reads and closes it again. Thus, it repeatedly opens and closes this FIFO within the loop. We could just let it stay open for the duration of the loop, but closing it and re-opening it we give ourselves the chance to detect in the `open()` call that the server closed its write end of the FIFO unexpectedly.
- The error handling in the client is similar to what it was in the iterative server's client. The code has redundant error checks such as guards to prevent closing a FIFO that is not open (setting the file descriptors to -1 unless they are in use), and closing descriptors before unlinking the files. On the other hand, it should really check the return values of the `close()` calls. A `clean_up()` function simplifies the error-handling, consolidating the cleaning up code.

8.4.4.2 The Concurrent Server

The server code is in the next listing.

Listing 8.16: upcased2.c

```
#include "upcase2.h"
#include "sys/wait.h"

#define  WARNING    "Server could not access client FIFO\n"
#define  MAXTRIES 5

int    dummyfifo;          /* file descriptor to write-end of PUBLIC */
int    client_convertedtext_fd; /* file descriptor to write-end of PRIVATE */
int    client_rawtext_fd;  /* file descriptor to write-end of PRIVATE */
int    publicfifo;         /* file descriptor to read-end of PUBLIC */
FILE*  upcaselog_fp;       /* points to log file for server */
pid_t  server_pid;         /* to store server's process id */

/*************************************************************************/
/*                      Signal Handler Prototypes                        */
/*************************************************************************/
/** on_sigpipe()
 *  This handles the SIGPIPE signals, just writes to standard error.
 */
void on_sigpipe( int signo );

/** on_signal()
 *  This handles the interrupt signals. It closes open FIFOs and files,
 *  removes the public FIFO and exits.
```

```
 */
void on_signal( int sig );

/** on_sigchild()
 *  Because this is a concurrent server, the parent process has to collect the
 *  exit status of each child. The SIGCHLD handler issues waits and writes to
 *  the log file.
 */
void on_sigchld( int signo );

/*****************************************************************************/
/*                              Main Program                                 */
/*****************************************************************************/

int main( int argc, char *argv[])
{
    int              tries;      /* num tries to open private FIFO  */
    int              nbytes;     /* number of bytes read from private FIFO */
    int              i;
    struct message   msg;        /* message structure with FIFO names */
    struct sigaction handler;    /* sigaction for registering handlers */
    char             buffer[PIPE_BUF];
    char             logfilepath[PATH_MAX];
    char             *homepath;      /* path to home directory */
    pid_t            child_pid;      /* pid of each spawned child */
    /* Open the log file in the user's home directory for appending. */
    homepath  = getenv("HOME");
    sprintf(logfilepath, "%s/.upcase_log", homepath );

    if ( NULL == (upcaselog_fp  = fopen(logfilepath, "a")) ) {
        perror(logfilepath);
        exit(1);
    }

    /* Register the interrupt signal handler  */
    handler.sa_handler = on_signal;
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigchld;
    if ( sigaction(SIGCHLD, &handler, NULL) == -1 ) {
```

```
        perror("sigaction");
        exit(1);
    }

    /* Create public FIFO */
    if ( mkfifo(PUBLIC, 0666) < 0 ) {
        if (errno != EEXIST )
            perror(PUBLIC);
        else  {
            fprintf(stderr, "%s already exists. Delete it and restart.\n",
                    PUBLIC);
        }
        exit(1);
    }

    /* Open public FIFO for reading and writing so that it does not get */
    /* EOF on the read-end while waiting for a client to send data.     */
    /* To prevent it from hanging on the open, the write-end is opened  */
    /* in non-blocking mode. It never writes to it. */
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }

    server_pid = getpid();

    /* Block waiting for a msg structure from a client */
    while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {

        /* spawn child process to handle this client */
        if ( 0 == fork() ) {

            /* We get the pid for message identification. */
            child_pid = getpid();

            /* We use the value of client_rawtext_fd to detect errors */
            client_rawtext_fd = -1;

            /* Client should have opened rawtext_fd for writing before
               sending the message structure, so the following open should
               succeed immediately. If not it blocks until the client opens
               it.  */
            if ( (client_rawtext_fd = open(msg.raw_text_fifo, O_RDONLY))
                  == -1 ) {
                fprintf(upcaselog_fp,
                        "Client did not have pipe open for writing\n");
                exit(1);
            }
            /* Clear the buffer used for reading the client's text */
            memset(buffer, 0x0, PIPE_BUF);

            /*
              Attempt to read from client's raw_text_fifo. This read will
```

```
                block until either input is available or it receives an EOF.
                An EOF is delivered only when the client closes the write-end
                of its raw_text_fifo.
              */
              while ( (nbytes = read(client_rawtext_fd, buffer,
                       PIPE_BUF)) > 0 ) {
                  /* Convert the text to uppercase */
                  for ( i = 0; i < nbytes; i++ )
                      if ( islower(buffer[i]))
                          buffer[i] = toupper(buffer[i]);

                  /* Open client's convertedtext FIFO for writing. To allow for
                     delays, we try 5 times. Here it is critical that the
                     O_NONBLOCK flag is set, otherwise it will hang in the loop
                     and we will not be able to abandon the attempt if the client
                     has died.  */
                  tries = 0;
                  while (((client_convertedtext_fd = open(msg.converted_text_fifo,
                           O_WRONLY | O_NDELAY)) == -1 ) && (tries < MAXTRIES ))
                  {
                       sleep(2);
                       tries++;
                  }
                  if ( tries == MAXTRIES ) {
                      /* Failed to open client convertedtext FIFO for writing */
                      fprintf(upcaselog_fp, "%d: " WARNING, child_pid);
                      exit(1);
                  }

                  /* Send converted text to client in its readfifo */
                  if ( -1 == write(client_convertedtext_fd, buffer,
                       nbytes) ) {
                      if ( errno == EPIPE )
                          exit(1);
                  }
                  /* See the notes below. */
                  close(client_convertedtext_fd);
                  client_convertedtext_fd = -1;

                  /* Clear the buffer used for reading the client's text */
                  memset(buffer, 0x0, PIPE_BUF);
              }
              exit(0);
          }
      }
      return 0;
}
```

The signal handlers for the server are below. The `SIGCHLD` handler uses `waitpid()` to wait for all children, and it remains in its loop as long as there is a zombie to be collected. The `WNOHANG` flag is used to prevent it from blocking in the `waitpid()` code. This way, if multiple `SIGCHLD` signals arrive while it is in the handler, the children whose deaths caused them will be collected. (Remember that signals may not be reliably handled on all systems, and even though in a POSIX compliant system,

each `SIGCHLD` will be delivered if we set `SA_NODEFER`, it is safer to collect them in this loop.)

```
void on_sigchld( int signo )
{
    pid_t pid;
    int   status;

    while ( (pid = waitpid(-1, &status, WNOHANG) ) > 0 )
        fprintf(upcaselog_fp, "Child process %d terminated.\n", pid);
    fflush(upcaselog_fp);
    return;
}

void on_sigpipe( int signo )
{
    fprintf(stderr, "Client is not reading the pipe.\n");
}

void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    if ( clientreadfifo != -1 )
        close(clientreadfifo);
    if ( clientwritefifo != -1)
        close(clientwritefifo);
    if ( getpid() == server_pid )
        unlink(PUBLIC);
    fclose(upcaselog);
    exit(0);
}
```

Comments.

- All of the work is performed by the child processes. Each child begins by trying to open the client's raw_text FIFO for reading. If successful, it enters a loop in which it repeatedly reads, converts the text to uppercase, opens the client's converted_text FIFO, writes the converted text to it, and closes it.
- Since the client may not have the converted_text open for reading for any number of reasons – it might have been terminated – the child process tries the `open()` a fixed number of times before it gives up. It uses the same technique as the iterative server did, using a non-blocking `open()`.
- When the child process does successfully open the FIFO, it still checks whether the `write()` failed, since anything can happen in between, and if so, the child exits. Otherwise, it writes the data, closes its end of the FIFO and waits to read more text from the client. When it receives the end-of-file, it exits.
- You may wonder why the server repeatedly opens and closes the write end of the client's converted_text FIFO. This is the only way that the client will receive an EOF in its `read()`.

If the client does not get the EOF, then it will remain blocked in its read of the converted_text FIFO, and will not be able to send any more data to the server. This would put the client and this server subprocess into deadlock, because this process would go back to the `read()` of the client's raw_text FIFO and block waiting for data from the client, which would never arrive. Therefore, although it seems inefficient to open and close this FIFO each time, it is the simplest means of preventing deadlock.

- The signal handler checks whether the parent process is executing it. If the parent has been signaled, then it should remove the public FIFO, otherwise not. We do not want child processes to remove this FIFO!

- If you are at all familiar with sockets, you might have noticed that the design of this server is easily converted to one that uses sockets. We will refer back to this example when we take up sockets.

8.5 Daemon Processes

As was mentioned earlier, a daemon is a process that runs in the background, has no controlling terminal. In addition, daemons set their working directory to "/". Usually daemons are started by system initialization scripts at boot-time. If you have written a server and want to turn it into a full-fledged daemon, it is not enough to put it into the background. This will only tell the shell not to wait for it; it will still have a control terminal and will still be killed by any signals from that terminal.

Some daemons are started by other programs. For example, some network daemons are started by the `inetd` or `xinetd` superserver. Some are started by programs such as the `crond` daemon, which runs scheduled jobs. Some are invoked at the user terminal. For example, sometimes the printer daemon is stopped and restarted at the terminal by the superuser.

Because daemons do not have a controlling terminal, they cannot write messages to standard output or to standard error. Instead they can use a system logging function named `syslog()`, which is a client that talks to the `syslogd` daemon, which write messages to specific log files. The `glibc` version of this function is `klogctl()`. Later we will look at an example of how it can be used. A server should be designed to turn itself into a daemon. In other words, when the server is run, it should take all of the steps necessary to become a daemon, which include:

1. Putting itself in the background. It does this by forking a new process and executing its code as the child and having the parent execute `exit()`. When the parent exits, the shell that started it collects its exit status and thinks the invoked program has terminated (which it has.) The child, which is now the server, is no longer in the foreground, but it is still controlled by the terminal.

2. Making itself a session leader. Recall that a process can detach itself from a terminal by becoming a session leader, but only processes that are neither session leaders nor process group leaders can do this. Since the server is now a child of the original process, it is neither, so it can call `setsid()`, which makes it a session leader of a new session and a group leader of a new process group.

3. Registering its intent to ignore `SIGHUP`.

4. Forking another child process, terminating in the parent again, and letting the new child, which is the grandchild of the original process, execute the server code. In some versions of UNIX, when a session leader opens a terminal device (which it may want to do sometimes), that terminal is automatically made the control terminal for the process. By running as the child of a session leader, the server is now immune from this eventuality. In Linux, a process can set the `O_NOCTTY` flag on `open()` to prevent this. The reason for ignoring `SIGHUP` is that when a session leader terminates, all of its children are sent a `SIGHUP`, which would otherwise kill them. Since the parent is a session leader, the child must ignore `SIGHUP`.

5. Changing the working directory to `"/"`.

6. Clearing the umask.

7. Closing any open file descriptors.

A procedure for doing all of these steps, based on one from [Stevens], is below.

Listing 8.17: daemon_init.c

```
void daemon_init(const char *pname, int facility)
{
    int         i;
    pid_t     pid;

    if ( (pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    else if (pid != 0)
        exit(0);        /* parent terminates  */

    /* Child continues from here */
    /* Detach itself and make itself a sesssion leader */
    setsid();

    /* Ignore SIGHUP */
    signal(SIGHUP, SIG_IGN);

    if ( (pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    else if ( pid != 0 )
        exit(0);    /* First child terminates */

    /* Grandchild continues from here */
    chdir("/");         /* change working directory  */

    umask(0); /* clear our file mode creation mask  */

    /* Close all open file descriptors */
    for (i = 0; i < MAXFD; i++)
        close(i);
```

```
    /* Start logging with syslog() */
    openlog(pname, LOG_PID, facility);

}
```

The final version of the `upcase` server incorporates this function and turns itself into a daemon. The only changes required are to include this function into the code and insert the line

```
daemon_init(argv[0], 0);
```

before the first executable statement.

8.6 Multiplexed I/O With select

Imagine the situation in which a process has multiple sources of input open for reading, such as a set of pipes as well as the terminal. Suppose the process has to respond to commands typed at the terminal as well as display messages that are available in the pipes. This is what is meant by *multiplexed input*: when a process has to obtain input available from multiple sources simultaneously. One solution would be to make all of the reads non-blocking and to continually poll each descriptor to see if there is data ready for reading on it. Polling, though, has many drawbacks, as we have seen, the most important of which is that it is wasteful of the CPU resource.

Another alternative would be to use asynchronous reads on each descriptor. This is also possible, but quite messy to code, and has the drawback that it relies on signals which may not be handled properly or reliably.

It is for these reasons that the `select()` system call was developed[6]. Basically, the `select()` call allows a process to listen to multiple descriptors at once and to be notified when any of them have pending input or output. Roughly put, `select()` is given a set of masks of file descriptors, representing I/O devices or files in which the process is interested. When input or output is ready on any of them, the appropriate bits in these masks are set. The process can check the masks to see which I/O is ready and can then read or write the ready descriptors. The `select()` call works with any file descriptor, so that it can be used with files, pipes, FIFOs, devices, and sockets.

select() is fairly complex:

```
/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

[6] There is a similar call named `poll()`.

The parameters have the following meanings:

`ndfs` The number of file descriptors of potential interest.

`readfds` The address of a file descriptor mask indicating which file descriptors the process is interested in *reading*.

`writefds` The address of a file descriptor mask indicating which file descriptors the process is interested in *writing*.

`exceptfds` The address of a file descriptor mask indicating which file descriptors the process is interested in checking for *out-of-band* data[7]. (Out-of-band messages or data should be thought of as exceptions or error conditions concerning any of the descriptors in the read or write descriptor masks.)

`timeout` The address of a `timeval` structure containing the amount of time to wait before completing the `select()` call. If timeout is `NULL`, it means wait forever, i.e., block until at least one descriptor is ready. If it is zero, it means return immediately with the status of all descriptors in the above sets. If it is non-zero, it will either wait the specified amount of time or return before if one of the specified descriptors is ready.

The return value of the `select()` call is the number of descriptors that are ready, or -1 if there was an error.

The `fd_set` data type is not necessarily a scalar. It is usually an array of long integers. If you do a little digging you will discover a constant, `FD_SETSIZE`, that defines the maximum number of descriptors in a `fd_set`, which is usually on the order of 1024 or more. Fortunately, you do not need to know how it is defined to use it, since there are macros and/or functions in the library for manipulating `fd_set` objects:

This turns off the bit for descriptor `fd` in the mask pointed to by `fdset`:

```
void FD_CLR(int fd, fd_set *fdset);
```

This turns on the bit for descriptor `fd` in the mask pointed to by `fdset`:

```
void FD_SET(int fd, fd_set *fdset);
```

This sets all bits to zero in the mask pointed to by `fdset`:

```
void FD_ZERO(fd_set *fdset);
```

This checks whether the bit for descriptor `fd` is set in the mask pointed to by `fdset`:

```
int FD_ISSET(int fd, fd_set *set);
```

[7]Out-of-band refers to data that is transferred in a separate communication channel. Out-of-band implies that the data does not arrive in sequence with the rest of the data, but in a parallel channel. It is used for transmitting error or control messages.

The value of the first parameter, `ndfs`, must be set to the value of the largest file descriptor + 1, since the file descriptor array is 0-based. The reason that the first argument is the maximum number of descriptors of interest is for efficiency. By supplying this number to the kernel, it saves the kernel the work of having to copy parts of the descriptor mask that are not needed. To give you an idea of how this call is used in a simple case, if we wanted to read from two different open file descriptors, we would use something like

```
#include <sys/time.h>
#include <sys/types.h>

...

int     fd1, fd2, maxfd;
fd_set  readset, tempset;

fd1   = open("file1", O_RDONLY);  /* open file1  */
fd2   = open("file2", O_RDONLY);  /* open file2*/
maxfd = fd1 > fd2 ? fd1+1 : fd2+1;

FD_ZERO(&readset);          /* clear the bits in the mask  */
FD_SET(fd1, &readset);      /* set the bit for fd1 (file1) */
FD_SET(fd2, &readset);      /* set the bit for fd2 (file2) */
tempset = readset;          /* copy into tempset           */

while ( select(maxfd, &tempset, NULL, NULL, NULL) > 0) {
    if ( FD_ISSET(fd1, &tempset) ) {
         /* read from descriptor fd1 */
    }
    if ( FD_ISSET(fd2, &tempset) ) {
        /* read from descriptor fd2  */
    }
    tempset = readset;
}
```

Notes.

- Although we are interested only in file descriptors `fd1` and `fd2`, the proper way to use select is to specify the full range of descriptors from 0 to the maximum of `fd1` and `fd2`. Since it is a zero-based array, this value is `max(fd1, fd2) + 1`.
- Because the return value of `select()` is positive as long as there is data to be read on either of `fd1` or `fd2`, the loop will continue until we get end-of-file on both files.
- The way that `select()` works, it resets the file descriptor masks to reflect the status of the descriptors of interest. In other words, the masks change after each call to `select()`. Therefore, you need to keep a copy of the original mask, and before each call, reset the masks to their original states.
- The masks are not modified if the `select()` call returned with an error.

- Inside the loop, you use the `FD_ISSET()` function to test each descriptor in which you expressed interest.
- It is a very common mistake to forget to add 1 to the largest descriptor in the first argument. It is also a common mistake to forget to reset the mask between each successive call.

We will put these ideas to work in a slightly more interesting example, borrowed from *[Haviland et al]*.

Listing 8.18: selectdemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/wait.h>

#define  MSGSIZE   6

char  msg1[] = "Hello";
char  msg2[] = "Bye!!";

void parent( int pipeset[3][2]);
int child( int fd[2] );

/*************************************************************************/
/*                              Main Program                             */
/*************************************************************************/

int main( int argc, char* argv[])
{
    int fd[3][2];  /* array of three pipes */
    int i;

    for ( i = 0; i < 3; i++ ) {
        /* create three pipes */
        if ( pipe(fd[i]) == -1 ) {
            perror("pipe");
            exit(1);
        }

        /* fork children */
        switch( fork() ) {
        case -1 :
            fprintf(stderr, "fork failed.\n");
            exit(1);
        case 0:
            child(fd[i]);
        }
    }
```

```
    parent(fd);
    return 0;
}

/*****************************************************************************/
/*                              Parent and Child                             */
/*****************************************************************************/

void parent( int pipeset[3][2])
{
    char    buf[MSGSIZE];
    char    line[80];
    fd_set  initial, copy;
    int     i, nbytes;

    for ( i = 0; i < 3; i++)
         close(pipeset[i][1]);

    /* create descriptor mask */
    FD_ZERO(&initial);
    FD_SET(0, &initial);                        /* add standard input */

    for ( i = 0; i < 3; i++)
        FD_SET(pipeset[i][0], &initial);     /* add read end of each pipe */

    copy = initial;                             /* make a copy */
    while (select( pipeset[2][0]+1, &copy, NULL, NULL, NULL ) > 0 ) {

        /* check standard input first */
        if ( FD_ISSET(0, &copy) ) {
            printf("From standard input: ");
            nbytes = read(0, line, 81);
            line[nbytes] = '\0';
            printf("%s", line);
        }

        /* check the pipe from each child */
        for ( i = 0; i < 3; i++ ) {
            if ( FD_ISSET(pipeset[i][0], &copy)) {
                /* it is ready to read */
                if ( read( pipeset[i][0], buf, MSGSIZE) > 0 ) {
                    printf("Message from child %d:%s\n", i, buf );
                }
            }
        }
        if (waitpid(-1, NULL, WNOHANG) == -1 )
            return;
        copy = initial;
    }
}

int child( int fd[2] )
{
    int count;
```

```
    close(fd[0]);

    for ( count = 0; count < 10; count ++) {
          write( fd[1], msg1, MSGSIZE);
          sleep(1 + getpid() % 6 );
    }

    write( fd[1], msg2, MSGSIZE);
    exit(0);
}
```

Comments.

- Each child writes a small string to the write-end of its pipe and then sleeps a bit so that the output does not flood the screen too quickly.
- The parent uses the `select()` call to query standard input and the read-ends of each child's pipe. The user can type a string on the keyboard and the parent will detect that standard input is ready. Within the while-loop each descriptor is tested, and if it is set, the `read()` can be done because input is waiting. This way the parent never holds up any child that is waiting for its message to be read.

There will be another, more interesting use of `select()` after the introduction to sockets.

8.7 Summary

Related processes can use unnamed pipes to exchange data. Unrelated processes running on the same host can use named pipes to exchange data. Unlike unnamed pipes, named pipes are entities in the file system. Both named and unnamed pipes are guaranteed by the kernel to be read and written atomically provide that the amount of data written is at most `PIPE_BUF` bytes.

Servers can be iterative or concurrent. A concurrent server creates a child process to handle every distinct client. An iterative server handles each client within a single process, sharing its time among them. Concurrent servers provide more reliable response time to the clients.

When a process has to handle I/O from multiple file descriptors, it can multiplex the I/O by means of the select() system call. This is one alternative of many, but it provides a relatively simple solution. Other alternatives include asynchronous I/O and the `poll()` call.

9 Interprocess Communication (IPC)

Concepts Covered

Pipes
I/O Redirection
FIFOs
Concurrent Servers
Daemons
Multiplexed I/O with select()
Sockets
mknod, tee,
API: *accept, bind, connect, dup, dup2, fpathconf, gethostbyname, listen, mkfifo, pipe, pclose, popen, recv, select, send, setsid, shutdown, socket, syslog,*

9.1 Introduction

Processes that cooperate in order to complete one or more tasks almost always need to communicate with each other. Sometimes the communication requires sharing data. One method of sharing data is by sharing a common file. If at least one of the processes modifies the file, then the file must be accessed in mutual exclusion. Sharing a file is essentially like sharing a memory-resident resource in that both are a form of communication that uses a shared resource that is accessed in mutual exclusion. Another paradigm involves passing data back and forth through some type of communication channel that provides the required mutual exclusion. A pipe is an example of this, as is a socket. This type of communication is broadly known as a message-passing solution to the problem.

We will begin with ***unnamed pipes***. After that we will look at ***named pipes***, also known as ***FIFO***'s, and then look at record-locking and file-locking schemes.

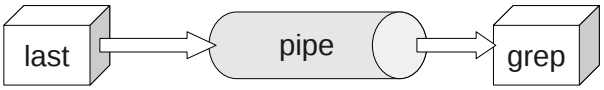
9.2 Unnamed Pipes

You are familiar with how to use pipes at the command level. A command such as

```
$ last | grep 'reboot'
```

connects the output of `last` to the input of `grep`, so that the only lines of output will be those lines of `last` that contain the word '`reboot`'. The '`|`' is a `bash` operator; it causes `bash` to start the `last` command and the `grep` command simultaneously, and to direct the standard output of `last` into the standard input of `grep`. Pipes were invented by Douglas Mcilroy. and were incorporated into UNIX in 1973.

We know that something called a *pipe* is involved in this mechanism; i.e., that there is some special file or buffer that we can visualize quite literally like a physical pipe, connecting the output of `last` to the input of `grep`:



The `last` program does not know that it is writing to a pipe and `grep` does not know that it is reading from a pipe. Moreover, if `last` tries to write to the pipe faster than `grep` can drain it, `last` will block, and if `grep` tries to read from an empty pipe because it is reading faster than `last` can write, `grep` will block, and both of these actions are handled behind the scenes by the kernel.

What then is a pipe? Although a pipe may seem like a file, *it is not a file*, and there is no file pointer associated with it. It is conceptually like a conveyor belt consisting of a fixed number of logical blocks that can be filled and emptied. Each write to the pipe fills as many blocks as are needed to satisfy it, provided that it does not exceed the maximum pipe size, and if the pipe size limit was not reached, a new block is made available for the next write. Filled blocks are conveyed to the read-end of the pipe, where they are emptied when they are read. These types of pipes are called ***unnamed pipes*** because they do not exist anywhere in the file system. They have no names.

An unnamed pipe[1] in UNIX is created with the `pipe()` system call.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

The system call `pipe(fd)`, given an integer array `fd` of size 2, creates a pair of file descriptors, `fd[0]` and `fd[1]`, pointing to the "read-end" and "write-end" of a pipe inode respectively. If it is successful, it returns a 0, otherwise it returns -1. The process can then write to the write-end, `fd[1]`, using the `write()` system call, and can read from the read-end, `fd[0]`, using the `read()` system call. The read and write-ends are opened automatically as a result of the `pipe()` call. Written data are read in first-in-first-out (FIFO) order. The following program (`pipedemo0.c` in `demos/chapter09`) demonstrates this simple case.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define  READ_END  0
```

1 Unless stated otherwise, the word "pipe" will always refer to an unnamed pipe.

```
#define  WRITE_END 1
#define  NUM       5
#define  BUFSIZE  32

int main(int argc, char* argv[] )
{
    int   i, nbytes;
    int   fd[2];
    char  message[BUFSIZE+1];

    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    for ( i = 1; i <= NUM; i++ ) {
        sprintf(message, "hello #%2d", i);
        write(fd[WRITE_END], message, strlen(message));
    }
    close(fd[WRITE_END]);     // MUST DO THIS!
    printf("%d messages sent; sleeping a bit. Please wait...\n", NUM);
    sleep(2);

    while ( (nbytes = read(fd[READ_FD], message, BUFSIZE)) != 0 ) {
        if ( nbytes > 0 ) {
            message[nbytes] = '\0';
            printf("%s", message);
        }
        else
           exit(1);  // read error
    }
    fflush(stdout);
    exit(0);
}
```

Notes.

- In this program, the read and write calls are not error-checked, which they should be.
- The `read()` is a blocking read by default. It will block waiting for data as long as the write-end of the pipe remains open. Naturally, the process has to write the data into the pipe before it reads it, otherwise it will block forever. If the program does not close the write-end of the pipe before the read-loop starts, it will hang forever, because the read will continue to wait for data. This could be avoided if the read-loop knew in advance exactly how many bytes to expect, but that is pretty pointless.

- Notice that the `read()` calls always read the same amount of data. This example demonstrates that the reader can assemble the data it reads from the pipe into larger chunks, because the data arrives in the order it was sent (unlike data sent across a network.) Pipes have no concept of message boundaries -- they are simply byte streams.

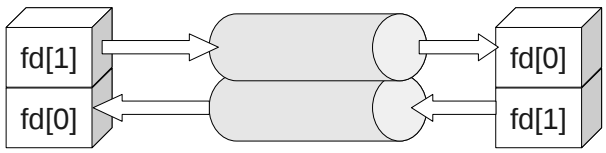
- Finally, observe that before calling `printf()` to print the string, the string has to be null-terminated .

The semantics of reading from a pipe are much more complex than reading from a file. The following table summarizes what happens when a process tries to read n bytes from a pipe that currently has p bytes in it that have not yet been read.

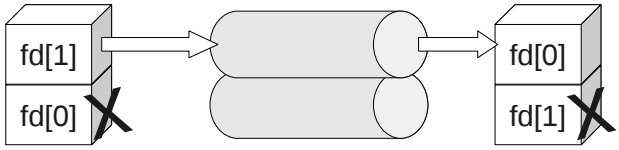
Pipe Size (p)	At one process has the pipe open for writing			No processes have the pipe open for writing
	Blocking read		Non-blocking read	
	At least one writer is sleeping	No writer is sleeping		
p = 0	Copy n bytes and return n, waiting for data as necessary when the pipe is empty.	Block until data is available, copy it and return its size.	Return -EAGAIN.	Return 0.
0 < p < n		Copy p bytes and return p, leaving the buffer empty.		
p >= n	Copy n bytes and return n leaving p-n bytes in the pipe buffer.			

9.2.1 Parent and Child Sharing a Pipe

Of course there is little reason for a process to create a pipe to write messages to itself. Pipes exist in order to allow two different processes to communicate. Typically, a process will create a pipe, and then fork a child process. After the fork, the parent and child will each have copies of the read and write-ends of the pipe, so there will be two data channels and a total of four descriptors:



On some Unix systems, such as **System V Release 4** Unix, pipes are implemented in this ***full-duplex mode***, allowing both descriptors to be written into and read from at the same time. POSIX allows only ***half-duplex mode***, which means that data can flow in only one direction through the pipe, and each process must close one end of the pipe. The following illustration depicts this half-duplex mode.



The paradigm for half-duplex use of a pipe by two processes is as follows:

```
if ( -1 == pipe(fd))
    exit(2);  // failed to create pipe

switch ( fork() ) {
// child process:
    case 0:
    close(fd[1]);          // close write-end
    bytesread    = read( fd[0], message, BUFSIZ);
                   // check for errors of course
    break;
// parent process:
    default:
    close(fd[0]);          // close read-end
    byteswritten = write(fd[1], buffer, strlen(buffer) );
                   // and so on
    break;
```

Linux follows the POSIX model but does not require each process to close the end of the pipe it is not going to use. However, for code to be portable, it should follow the POSIX model. All examples here will assume half-duplex mode. The following is the first example of two-process communication through a pipe. The parent process reads the command line arguments and sends them to the child process, which prints them on the screen. As we get more deeply involved with pipes, you will discover that it is easy to make mistakes when coding for them, as there are many intricacies to be aware of. This first program exposes a few of them.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
#include <stdlib.h>

#define  READ_FD  0
#define  WRITE_FD 1

int main(int argc, char* argv[] )
{
    int i;
    int bytesread;
    int fd[2];
    char  message[BUFSIZ];

    // check proper usage
    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s message\n", argv[0]);
        exit(1);
    }

    // try to create pipe
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    // create child process
    switch ( fork() ) {
    case -1:
        // fork failed -- exit
        perror("fork()");
        exit(3);

    case 0:
        // child code
        close(fd[WRITE_FD]);   // MUST DO THIS
        // Loop while not end of file or not a read error
        while ((bytesread = read( fd[READ_FD], message, BUFSIZ)) != 0)
            if ( bytesread > 0 ) {  // more data
                message[bytesread] = '\0';
                printf("Child received the word: '%s'\n", message);
                fflush(stdout);
            }
            else {                  //read error
                perror("read()");
                exit(4);
            }
```

```
        exit(0);

    default:
        // parent code
        close(fd[READ_FD]);   // parent is writing so close read-end
        for ( i = 1; i < argc; i++ )
            // send each word separately
            if (write(fd[WRITE_FD], argv[i], strlen(argv[i]) != -1 ) {
                printf("Parent sent the word: '%s'\n", argv[i]);
                fflush(stdout);
            }
            else {
                perror("write()");
                exit(5);
            }
        close(fd[WRITE_FD]);

        // wait for child so it does not remain a zombie
        // don't care about it's status, so pass a NULL pointer
        if (wait(NULL) == -1) {
            perror("wait failed");
            exit(2);
        }
    }
    exit(0);
}
```

Notes.

- It is now critical that the child closes the write-end of it's pipe before it starts to read. As was noted earlier, reads are blocking by default and will remain waiting for input as long as ANY write-end of the pipe is open, including its own. Therefore, not only do we want to close the unused end of the pipe for the code to be more portable, but also for it to be correct!

- The parent waits for the child process because if it does not, the child will become a zombie in the system. You should make a habit of waiting for all processes that you create.

- The output of the parent and child on the terminal may occur in any order. This program makes no attempt to coordinate the use of the terminal simply because it would distract from its purpose as a demonstration of how to use pipes.

9.2.2 Atomic Writes

In a POSIX-compliant system, a single write will be executed atomically as long as the number of bytes to be written does not exceed PIPE_BUF. This means that if several processes are each writing to the pipe at the same time, as long as each limits the size of each write to $N \leq$ PIPE_BUF bytes, the data will not be intermingled. If there is not enough room in the pipe to store $N \leq$ PIPE_BUF bytes, and writes are blocking (the default), then write() will be blocked until room is available. On the other hand, if $N >$ PIPE_BUF, there is no guarantee that the writes will be atomic.

There are two ways to obtain the value of PIPE_BUF. One is simply to include the header file <limits.h> and use the macro PIPE_BUF, as in

```
#include <limits.h>

char  chunk[PIPE_BUF];
```

A better solution is to obtain the actual value in use in the kernel, in case the header file is not up to date. A call to fpathconf() will return the value of various configuration values associated with an open file descriptor:

```
if ( -1 == pipe(fd) )
    exit(1);
long pipe_size = fpathconf(fd[0],  _PC_PIPE_BUF);
```

The fpathconf() function 's synopsis and description is

```
#include <unistd.h>

long fpathconf(int filedes, int name);
long pathconf(char *path, int name);

DESCRIPTION
fpathconf() gets a value for the configuration option name for the
open file descriptor filedes.
```

The second argument to fpathconf() is a mnemonic name defined in the man page. In the above example, the constant _PC_PIPE_BUF tells the function to return the value of PIPE_BUF. Consult the man page for a complete list of parameters that can be obtained at run-time with this function. POSIX demands that every PIPE_BUF be at least 512 bytes. Implementations are free to make it larger. On Linux, PIPE_BUF is 4096 bytes.

The following program is designed to demonstrate (but not prove) that writes of up to PIPE_BUF bytes are atomic, and that larger writes will not be. It also demonstrates how to create multiple writers and a single reader. The program creates two writer processes and one reader. One writer writes 'X's into the pipe, the other 'y's. They each write the same number of characters each time. The command line argument specifies the number of writes that each makes to the pipe. The idea

is that if the number is large enough the scheduler will time slice them often enough so that one will write for a while, then the next, and so on. The parent is the reader. It reads the data from the pipe and stores it in a file. The parent reads smaller chunks since it does not matter.

The output will show that writes are atomic -- each string written by each child in a single write has a newline at its end, and in the output, every sequence of X's and y's will also have a newline. There will be no occurrence of the string Xy or yX in the output because the kernel serializes the concurrent writes.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define  READ_FD  0
#define  WRITE_FD 1
#define  RD_SIZE  10
#define  ATOMIC

int main(int argc, char* argv[] )
{
    int i, repeat;
    int bytesread;
    int mssglen;
    int fd[2];
    int outfd;
    char  mssg[RD_SIZE+1];
    char  *Child1_Chunk, *Child2_Chunk;
    long Chunk_Size;

    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s repeats\n", argv[0]);
        exit(1);
    }

    // try to create pipe
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    repeat     = atoi(argv[1]);
    Chunk_Size = fpathconf(fd[0],  _PC_PIPE_BUF);
#ifndef ATOMIC
    Chunk_Size += 10;   // just make it bigger
```

```
#endif

    Child1_Chunk =  calloc(Chunk_Size, sizeof(char));
    Child2_Chunk =  calloc(Chunk_Size, sizeof(char));
    if ( ( Child1_Chunk == NULL ) || ( Child2_Chunk == NULL ) ) {
        perror("calloc");
        exit(2);
    }

    // create the string that child1 writes
    Child1_Chunk[0] = '\0'; // just to be safe
    for ( i = 0; i < Chunk_Size-2; i++)
        strcat(Child1_Chunk, "X");
    strcat(Child1_Chunk,"\n");

    // create the string that child2 writes
    Child2_Chunk[0] = '\0'; // just to be safe
    for ( i = 0; i < Chunk_Size-2; i++)
        strcat(Child2_Chunk, "y");
    strcat(Child2_Chunk,"\n");

    // create first child process
    switch ( fork() ) {
    case -1:  // fork failed -- exit
        perror("fork()");
        exit(3);

    case 0:   // child1 code
        mssglen = strlen(Child1_Chunk);
        for ( i = 0; i < repeat; i++ )
            if (write(fd[WRITE_FD],Child1_Chunk,mssglen) != mssglen) {
                perror("write");
                exit(4);
            }
        close(fd[WRITE_FD]);
        exit(0);

    default:  // parent creates second child process
        switch ( fork() ) {
        case -1:  // fork failed -- exit
              perror("fork()");
              exit(5);

        case 0:   // child2 code
            mssglen = strlen(Child2_Chunk);
            for ( i = 0; i < repeat; i++ )
```

```
                if (write(fd[WRITE_FD],Child2_Chunk,mssglen)!= mssglen){
                    perror("write");
                    exit(6);
                }
            close(fd[WRITE_FD]);
            exit(0);
        default:  // parent code
            outfd = open("pd2_output", O_WRONLY|O_CREAT|O_TRUNC,0644);
            if ( outfd == -1 ) {
                perror("open");
                exit(7);
            }
            close(fd[WRITE_FD]);
            while ((bytesread = read(fd[READ_FD],mssg,RD_SIZE)) != 0)
                if ( bytesread > 0 )
                    write(outfd, mssg, bytesread);
                else {                  //read error
                    perror("read()");
                    exit(8);
                }
            close(outfd);
              // collect zombies
            for ( i = 1; i <= 2; i++ )
                if ( wait(NULL) == -1) {
                    perror("wait failed");
                    exit(9);
                }
            close(fd[READ_FD]);
            free(Child1_Chunk);
            free(Child2_Chunk);
        }
        exit(0);
    }
}
```

Each child should write two thousand times or more in order for us to see the possibility of their each competing for the shared pipe, so the program should be run with a command line argument of 1000 or more. To check that the results are correct, i.e., that the output is not intermingled when writes are smaller than `PIPE_BUF` bytes each and possibly mingled when larger, first compile it with the `ATOMIC` macro enabled, and then comment it out and run it again. Try the following script each time, with a command line argument of 1000:

```
#!/bin/bash

if [[ $# < 1 ]]
then
    printf "Usage: %b repeats\n" $0
```

```
    exit
fi

pipedemo2  $1
printf "Number of X lines     : "
    grep X pd2_output | wc -l
printf "Number of y lines     : "
    grep y pd2_output | wc -l
printf "X lines in first %b : " $1
    head -$1 pd2_output | grep X | wc -l
printf "y lines in first %b : " $1
    head -$1 pd2_output | grep y | wc -l
printf "X lines in last %b  : " $1
    tail -$1 pd2_output | grep X | wc -l
printf "y lines in last %b  : " $1
    tail -$1 pd2_output | grep y | wc -l
printf "Xy lines              : "
    grep Xy pd2_output  | wc -l
printf "yX lines              : "
    grep yX pd2_output  | wc -l
```

You should see output such as this with `ATOMIC` enabled:

```
Number of X lines     : 1000
Number of y lines     : 1000
X lines in first 1000 : 515
y lines in first 1000 : 485
X lines in last 1000  : 485
y lines in last 1000  : 515
Xy lines              : 0
yX lines              : 0
```

and with `ATOMIC` disabled:

```
Number of X lines     : 1443
Number of y lines     : 1437
X lines in first 1000 : 758
y lines in first 1000 : 718
X lines in last 1000  : 685
y lines in last 1000  : 719
Xy lines              : 577
yX lines              : 586
```

which shows that when writes are too large, the writes will not be atomic, and a process can be interrupted in the middle of its write.

9.2.3 Pipe Capacity

The capacity of a pipe may be larger than `PIPE_BUF`. There is no exposed system constant that indicates the total capacity of a pipe; however, the following program, borrowed from [Haviland et al] and modified slightly, can be run on any system to test the maximum size of a pipe.

```
int count;

void on_alarm( int signo)
{
    printf("write() blocked after %d chars\n", count);
    exit(0);
}

int main()
{
    int fd[2];
    int pipe_size;
    char c = 'x';
    static struct sigaction sigact;

    sigact.sa_handler = on_alarm;
    sigfillset(&(sigact.sa_mask));
    sigaction(SIGALRM, &sigact, NULL);

    if ( -1 == pipe(fd) ) {
        perror("pipe failed");
        exit(1);
    }
    pipe_size = fpathconf(fd[0],  _PC_PIPE_BUF);
    printf("Max size of atomic write is %d bytes\n", pipe_size);

    while (1) {
        alarm(10);
        write(fd[1], &c, 1);
        alarm(0);
        if ( (++count % 1024) == 0 )
            printf ( "%d chars in pipe\n", count);
    }
    return 0;
}
```

Notes.

- Since there is only one process and it will never read from the pipe, the pipe will eventually fill up. When the process attempts to write to the pipe after it is full, it will be blocked. To prevent it from being blocked forever, it sets an alarm before each `write()` call and unsets it afterwards. The alarm interval, 10 seconds, is long enough so that the alarm will expire before the write finishes. When the pipe is full, the alarm will expire and the alarm handler will display the total number of bytes written and then terminate the program.

- The only purpose of displaying the value of `PIPE_BUF` is informational.

9.2.4 Caveats and Reminders Regarding Blocking I/O and Pipes

Quite a bit can go wrong when working with pipes. These are some important facts to remember about using pipes. Some of these have been mentioned already, some not.

- If a `write()` is made to a pipe that is not open for reading by any process, a `SIGPIPE` signal will be sent to the writing process, which, if not caught, will terminate that process. If it is caught, after the `SIGPIPE` handler finishes, the `write()` will return with a -1, and `errno` will be set to `EPIPE`.

- If there are one or more processes writing to a pipe, if a reading process closes its read-end of the pipe and no other processes have the pipe open for reading, each writer will be sent the `SIGPIPE` signal, and the same rules mentioned above regarding handling of the signal apply to each process.

- If when a writer is finished using a pipe, it fails to close the write-end of the pipe, and a reader is blocked on a `read()`, the reader will remain permanently blocked; as long as one writer has the pipe open for writing, the `read()` will remain blocked. As soon as all writers close the write-ends of the pipe, the `read()` will return zero.

- A `write()` to a full pipe will block the writer until there are `PIPE_BUF` free bytes in the pipe.

- Unlike reads from a file, `read()` requests to a pipe drain the pipe of the data that was read. Therefore, when multiple readers read from the same pipe, no two read the same data.

- Writes are atomic as long as the number of bytes is smaller than `PIPE_BUF`.

- Reads are atomic in the sense that, if there is any data in the pipe when the call is initiated, the `read()` will return with as much data as is available, up to the number of bytes requested, and it is guaranteed not to be interrupted.

- Processes cannot `seek()` on a pipe.

The situation is entirely different with non-blocking reading and writing. These will be discussed later. However, before continuing with the discussion of pipes, we will take a slight detour to look at I/O redirection in general, because studying I/O redirection will give us insight into some of the ways in which pipes are used.

9.3 I/O Redirection Revisited

9.3.1 Simulating Output Redirection

How does the shell implement I/O redirection? The key to understanding this rests on one simple principle used by the kernel: the `open()` system call ***always chooses the lowest numbered available file descriptor***.

Suppose that you have entered the command

```
$  ls  > list
```

The steps taken by the shell are

1. `fork()` a new process.
2. In the new process, `close()` file descriptor 1 (standard output).
3. In the new process, `open()` (with the `O_CREAT` flag) the file `list`.
4. Let the new process `exec()` the `ls` program.

After step 1, the child and parent each have copies of the same file descriptors. After step 2, the child has closed standard output, so file descriptor 1 is free. In step 3, the kernel sees descriptor 3 is free, so it uses descriptor 3 to point to the file structure for the file named `list`. Then the child `exec()`s "`ls`". The `ls` program thinks it is writing to standard output when it writes to descriptor 1, but it is really writing to the file named `list`. In the meanwhile, the shell continues to have descriptor 1 pointing to the standard output device, so it is unaffected by this secret trick it played on the `ls` command.

The following program, called `redirectout.c`, illustrates how this works. It simulates the shell's '>' operator. It creates a pipe, forks a child, closes standard output descriptor 1, opens the output file specified in `argv[2]` for writing, and `exec`s `argv[1]`. The parent simply waits for the child to terminate. Compile it and name it `redirectout`, and then try a command such as the following:

```
$ redirectout  who  whosloggedon
```

```
// redirectout.c
#include <stdio.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
    int  pid;
    int  fd;

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command output-file\n", argv[0]);
        exit(1);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:
        close(1);
        fd = open( argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644 );
        execlp( argv[1],argv[1], NULL );
        perror("execlp");
        exit(1);
    default:
        wait(NULL);
      }
    return 0;
}
```

Redirecting standard input works similarly. The only difference is that the process has to close the standard input descriptor 0, and then open a file for reading.

9.3.2 Simulating the '|' Shell Operator

The pertinent question now is, how can we write a similar program that can simulate how the shell carries out a command such as

```
$ last | grep 'pts/2'
```

This cannot be accomplished using just the `open()`, `close()`, and `pipe()` system calls. Somehow we need to connect one end of a pipe to the standard output for `last`, and the other end to the standard input for `grep`. There are two system calls that can be used for this purpose: `dup()` and `dup2()`. `dup()` is the progenitor of `dup2()`, which superseded it. We will first look at a solution using `dup()`.

The `dup()` system call duplicate a file descriptor. From the man page:

```
#include <unistd.h>

int dup(int oldfd);

After  a  successful  return from dup(), the old and new file
descriptors may be used interchangeably.  They refer to the  same
open  file description (see open(2)) and thus share file offset
and file status flags; for example,  if  the  file  offset  is
modified  by  using  lseek(2)  on one of the descriptors, the
offset is also changed for the other.
```

In other words, given a file descriptor, `oldfd`, `dup()` creates a new file descriptor that points to the same kernel file structure as the old one. But again the critical feature of `dup` is that it returns the lowest-numbered available file descriptor. Therefore, consider the following sequence of actions.

1. Declare descriptors for a pipe — `int fd[2];`
2. Create the pipe — `pipe(fd);`
3. Fork a child — `switch ( fork() )`
4. In the child: — `case 0:`
   i. close standard output — `close(fileno(stdout));`
   ii. dup write-end of pipe — `dup(fd[1]);`
   iii. close read-end of pipe — `close(fd[0]);`
   iv. exec the command that writes to the pipe — `exec("last", "last", NULL);`

The `dup()` call will find the standard output file descriptor available, and since that is the lowest numbered available descriptor, it will make that point to the same structure as `fd[1]` points to. Therefore, when the `last` command writes to standard output, it will really be writing to the write-end of the pipe.

Now it is not hard to imagine what the parent's job is. It has to close the standard input descriptor, `dup() fd[0]`, and `exec` the `grep` command. We can put these ideas together in a more general program, called `shpipe1.c`, which follows.

```
//shpipe1.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
```

```
    int  fd[2];

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:
        close(fileno(stdout));          // close stdout
        dup(fd[1]);                     // make stdout point to pipe
        close(fd[0]);                   // close read-end in child
        close(fd[1]);                   // not needed now
        execlp( argv[1],argv[1], NULL );// exec command that writes
        perror("execlp");
        exit(1);
    default:
        close(fileno(stdin));           // close stdin
        dup(fd[0]);                     // make stdin point to pipe
        close(fd[1]);                   // close write-end
        close(fd[0]);                   // not needed now
        execlp( argv[2],argv[2], NULL );// exec command that reads
        exit(2);
    }
    return 0;
}
```

If you compile this and name it `shpipe1`, then you can try commands such as

```
$ shpipe1 last  more
```

and

```
$ shpipe1 ls wc
```

There is a problem here. For one, the parent cannot wait for the child because it uses `execlp()` to replace its image. This can be solved by forking two children and letting the second do the work of the reading process. More importantly, this solution is not general, because there are two steps -- close standard output and then `dup()` the write end of the pipe. There is a small

window of time between closing standard output and duplicating the write-end of the pipe in which the child could be interrupted by a signal whose handler might close file descriptors so that the descriptor returned by `dup()` will not be the one that was just closed.

This is the reason that `dup2()` was created. `dup2(fd1, fd2)` will duplicate `fd1` in `fd2`, closing `fd2` if necessary, as a single atomic operation. In other words, if `fd1` is open, it will close it, and make `fd2` point to the same file structure to which `fd1` pointed. Its man page entry is

```
#include <unistd.h>
int dup2(int oldfd, int newfd);

.....

dup2()  makes newfd be the copy of oldfd, closing newfd first if
necessary.
```

(`dup()` and `dup2()` share the same page. I deleted `dup2()`'s description above. This is the relevant part of it.)

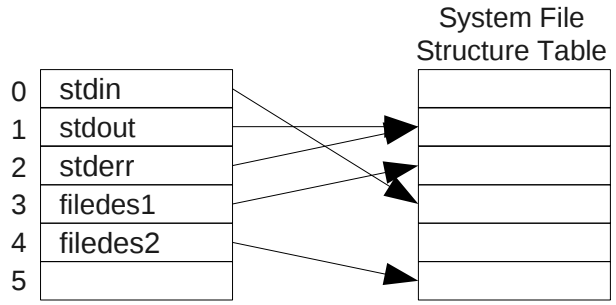


Illustration 1: Initial File Descriptor Table

A picture best illustrates how `dup2()` works. Assume the initial state of the file descriptors for the process is as shown in Illustration 1. Now suppose that the process makes the call

```
dup2( filedes2 , fileno(stdin));
```

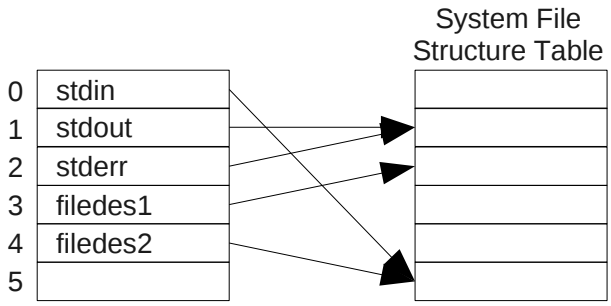


Illustration 2: File Descriptor Table After dup2()

Then, after the call the table is as shown in Illustration 2. Descriptor 0 (standard input) became a copy of `filedes2` as a result of the call. Descriptor `filedes2` is now redundant and can be closed if `stdin` is going to be used instead.

The following program, `shpipe2.c` , is an improved version of `shpipe1.c`.

```
//shpipe2.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
    int fd[2];

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:
        dup2(fd[1],fileno(stdout));     // copy fd[1] into stdout
        close(fd[0]);                    // close read-end of pipe
        close(fd[1]);                    // close write-end of pipe
        execlp( argv[1],argv[1], NULL ); // exec command1
        perror("execlp");
        exit(1);
    default:
        dup2( fd[0], fileno(stdin) );    // copy fd[0] into stdin
        close(fd[0]);                    // close read-end of pipe
        close(fd[1]);                    // close write-end of pipe
        execlp( argv[2],argv[2], NULL ); // exec command2
        exit(2);
    }
    return 0;
}
```

There are a couple of things you can try to do at this point to test your understanding of pipes.

1. There is a UNIX utility called `tee` that copies its input stream to standard output as well as to its file argument:

```
$ ls -l  | tee listing
```

will copy the output of "`ls -l`" into the file named listing as well as to standard output. Try to write your own version of `tee`.

2. Extend `shpipe2` to work with any number of commands so that

```
$ shpipe3 cmmd cmmd ... cmmd
```

will act like

```
$ cmmd  | cmmd | ... | cmmd
```

9.3.3 The popen() Library Function

The sequence of (1) generating a pipe, (2) forking a child process, (3) duplicating file descriptors, and (4) executing a new program in order to redirect the input or output of that program to the parent, is so common that the developers of the C library added a pair of functions, `popen()` and `pclose()` to streamline this procedure:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

The `popen()` function creates a pipe, forks a new process to execute the shell `/bin/sh` (which is system dependent), and passes the command to that shell to be executed by it (using the `-c` flag to the shell, which tells it to expect the command as an argument.)

`popen()` expects the second argument to be either "`r`" or "`w`". If it is "`r`" then the process invoking it will be returned a `FILE` pointer to the read-end of the pipe and the write-end will be attached to the standard output of the command. If it is "`w`", then the process invoking it will be returned a `FILE` pointer to the write-end of the pipe, and the read-end will be attached to the standard input of the command. The output stream is fully buffered.

File streams created with `popen()` must be closed with `pclose()`. `pclose()` will wait for the invoked process to terminate and returns its exit status or -1 if `wait4()` failed.

An example will illustrate. We will write a third version of the `shpipe` program called shpipe3 using `popen()` and `pclose()` instead of the `pipe()`, `fork()`, `dup()` sequence.

```
// shpipe3.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>
```

```
int main(int argc, char* argv[])
{
    int    nbytes;
    FILE  *fin;                // read-end of pipe
    FILE  *fout;               // write-end of pipe
    char   buffer[PIPE_BUF];   // buffer for transferring data

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }
    if ( (fin = popen(argv[1], "r")) == NULL ) {
        fprintf(stderr, "popen() failed\n");
        exit(1);
    }
    if ( (fout = popen(argv[2], "w")) == NULL ) {
        fprintf(stderr, "popen() failed\n");
        exit(1);
    }

    while ( (nbytes = read(fileno(fin), buffer, PIPE_BUF)) > 0 )
        write(fileno(fout), buffer, nbytes);

    pclose(fin);
    pclose(fout);
    return 0;
}
```

9.4 Named Pipes

Unnamed pipes are an elegant mechanism, however, they have several drawbacks. They can only be shared by processes with a common ancestor, such as a parent and child, or multiple children or descendants of a parent that created the pipe. Also, they cease to exist as soon as the processes that are using them terminate, so they must be recreated every time they are needed. If you are trying to write a server program that clients can communicate with, they will need to know the name of the pipe through which to communicate, but an unnamed pipe has no such name.

Named pipes make up for these shortcomings. A ***named pipe***, or ***FIFO***, is very much like an unnamed pipe in how you use it. You read from it and write to it in the same way. It behaves the same way with respect to the consequences of opening and closing it when various processes are either reading or writing or doing neither. In other words, the semantics of opening, closing, reading, and writing named and unnamed pipes are the same.

What distinguishes named pipes from unnamed pipes is that

- They exist as directory entries in the file system and therefore have associated permissions and ownership[2].
- They can be used by processes that are not related to each other.
- They can be created and deleted at the shell level or at the programming level.

9.4.1 Named Pipes at the Command Level

Before we look at how they are created programmatically, let us look at how they are created at the user level. There are two commands to create a FIFO. The older command is `mknod`. `mknod` is a general purpose utility for creating device special files. There is also a `mkfifo` command, which can only be used for creating a `FIFO` file. We will first look at how to use `mknod`.

```
$ mknod PIPE p
```

creates a `FIFO` named "`PIPE`". The lowercase `p` , *which must follow the file name*, indicates to `mknod` that `PIPE` should be a `FIFO` (`p` for *pipe*.) After typing this command, look at the directory:

```
$ ls -l PIPE

prw-r--r-- 1 stewart stewart 0 Apr 30 22:29 PIPE|
```

The '`p`' file type indicates that `PIPE` is a `FIFO`. Notice that it has 0 bytes. Try the following command sequence:

```
$ cat <  PIPE &

$ ls -l  > PIPE; wait
```

If we do not put the `cat` command into the background it will hang because a process trying to read from a pipe will block until there is at least one process trying to write to it. The `cat` command will terminate as soon as it receives a 0 from its `read()` call, which will be delivered when the writer closes the file after it is finished writing. In this case the writer is the process that executes "`ls -l`". When the output of `ls -l` is written to the pipe, `cat` will read it and display it on the screen. The `wait` command's only purpose is to delay the shell's prompt until after `cat` exits.

By the way, if you reverse this procedure:

```
$ ls -l >  PIPE &
```

2 Although they have directory entries, they do not exist in the file system. They have no disk blocks and their data is not on disk when they are in use.

```
$ ls -l PIPE

$ cat < PIPE; wait
```

and expect to see that the `PIPE` does not have 0 bytes, when the second `ls -l` is executed, you will be disappointed. That data is not stored in the file system.

9.4.2 Programming With Named Pipes

You can read about the `mkfifo` command in the man pages. We turn instead to the creation and use of named pipes at the programming level. A named pipe can be created either by using the `mknod()` system call, or the `mkfifo()` library function. In Linux, according to the `mknod()` (2) man page,

> "*Under Linux, this call cannot be used to create directories. One should make directories with mkdir(2), and FIFOs with mkfifo(3).*"

Therefore, we will stick to using `mkfifo()` for creating FIFOs. The other advantage of `mkfifo()` over `mknod()` is that it is easier to use and does not require superuser privileges:

```
#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

The call `mkfifo("MY_PIPE", 0666)` creates a FIFO named `MY_PIPE` with permission `0666 & ~umask`. The convention is to use UPPERCASE letters for the names of FIFOs. This way they are easily identified in directory listings.

It is useful to distinguish between ***public*** and ***private*** FIFOs. A ***public FIFO*** is one that is known to all clients. It is not that there is a specific function that makes a FIFO public; it is just that it is given a name that is easy to remember and that its location is advertised so that client programs know where to find it. A ***private FIFO***, in contrast, is given a name that is not known to anyone except the process that creates it and the processes to which it chooses to divulge it. In our first example, we will use only a single public FIFO. In the second example, the server will create a public FIFO and the clients will create private FIFOs that they will each use exclusively for communicating with the sever.

In this first example, the server creates a public FIFO. The server and client programs know the name of the public FIFO because they will share a common header file that hard-codes the pathname to the file in the file system. Ideally this name would be chosen so that no other processes in the system would ever choose the same file name[3]. This example supports multiple clients.

3 There are programs that can generate unique keys of an extremely large size that can be used in the name of the file. If all applications cooperate and use this method, then all pipe names would be unique.

The server creates the FIFO and opens it for both reading and writing, even if it only needs to read incoming messages on the pipe. This is because the FIFO needs to have at least one process that has it open for writing, otherwise the server will immediately receive an end-of-file on the FIFO and close its reading loop. By opening it up for writing also, the server will simply block on the `read()` to it, waiting for a client to send it data. Since the server never writes to this pipe, it does not matter whether writes are non-blocking or not, but by opening it with the `O_NDELAY` flag, since `POSIX` does not specify how a system is supposed to handle opening a file in blocking mode for both reading and writing, we avoid possibly undefined behavior. The server is run as a background process and is the process that must be started first, so that it can create the FIFO. If a client tries to write to a FIFO that does not exist, it will fail.

The client opens the public FIFO for writing and then enters a loop where it repeatedly reads from standard input and writes into the write-end of the public FIFO. It uses the library function `memset()`, found in `<string.h>`, to zero the buffer where the user's text will be stored, and it declares the buffer to be `PIPE_BUF` chars, so that the write will be atomic. (If the locale uses two-byte chars, this will not work properly.) When it is finished, it closes its write-end. The header file is listed first, followed by the client code.

```
// fifo1.h
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

#define  PUBLIC    "/tmp/FIFODEMO1_PIPE"
```

```
// sendfifo1.c
#include "fifo1.h"
#define  QUIT     "quit"

int main( int argc, char *argv[])
{
    int    nbytes;          // num bytes read
    int    publicfifo;      // file descriptor to write-end of PUBLIC
    char   text[PIPE_BUF];

    // Open the public FIFO for writing
    if ( (publicfifo = open(PUBLIC, O_WRONLY) ) == -1) {
        perror(PUBLIC);
        exit(1);
    }
```

```
    // Repeatedly prompt user for command, read it, and send to server
    while (1) {
        memset(text, 0x0, PIPE_BUF);    // zero string
        nbytes = read(fileno(stdin), text, PIPE_BUF);
        if ( !strncmp(QUIT, text, nbytes-1))   // is it quit?
            break;
        write(publicfifo, text, nbytes);
    }
    // User quit; close write-end of public FIFO
    close(publicfifo);
}
```

Comments.

- The client code allows the user to type "quit" to end the program.
- It is not robust. It does not handle any signals and does no clean-up if it is killed by a signal. If the server is not running it will hang and will need to be killed by a signal.

The server code follows.

```
// rcvfifo1.c
#include "fifo1.h"

int main( int argc, char *argv[])
{
    int        nbytes;       // number of bytes read from popen()
    int        n = 0;
    int        dummyfifo;    // file descriptor to write-end of PUBLIC
    int        publicfifo;   // file descriptor to read-end of PUBLIC
    static char buffer[PIPE_BUF];// buffer to store output of command

    // Create public FIFO
    if ( mkfifo(PUBLIC, 0666) < 0 )
        if (errno != EEXIST ) {
            perror(PUBLIC);
            exit(1);
        }
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }
    while ( 1 ) {
```

```
        memset(buffer, 0, PIPE_BUF);
        if ( ( nbytes = read( publicfifo, buffer, PIPE_BUF)) > 0 ) {
            buffer[nbytes] = '\0';
            printf("Message %d received by server: %s", ++n, buffer);
            fflush(stdout);
        }
        else
             break;
    }
    return 0;
}
```

Comments.

- The server reads from the public FIFO and displays the message it receives on its standard output, even though it may be put in the background; it is not detached from the terminal. The best way to run it is to leave it in the foreground and open a few clients in other terminal windows.
- The server increments a counter and displays each received message with the value of the counter, so that you can see the order in which the messages were received. It flushes standard output just in case there is no newline in the message.
- It, like the client, does not handle any error conditions other than the pre-existence of the public pipe. If it finds the pipe already exists, it just continues along.

This first example has several shortcomings that will be overcome in the next example.

9.4.3 An Iterative Server

In this example, we create a server that has two way communication with each client, processing incoming client requests one after the other. Such a server is called an ***iterative server***. In order to achieve this, the server creates a public FIFO that it uses for reading incoming messages from clients wishing to use its services. Each incoming message has a special field that contains the name of the private FIFO that the client creates when it starts up. Each message that a client sends has its private FIFO's name. The message structure also contains another field that the client can use to supply data for the server. When the server receives a message, it looks at the FIFO name in it and tries to open it for writing. If successful, the server will use this FIFO for sending data to the client. The client will, in turn, open its FIFO for reading. Illustration 3 depicts the relationship between the clients and the server with respect to the shared pipes.

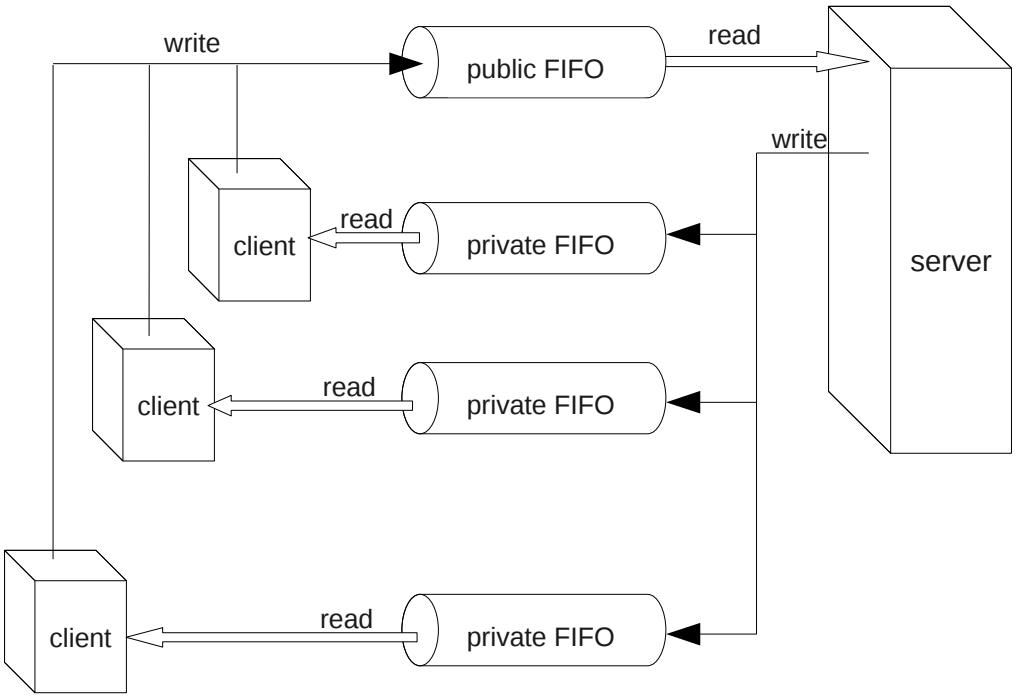


Illustration 3: Client-server Use of FIFOs

In this particular example, the server provides lowercase-to-uppercase translation for clients. The clients send it a piece of text and the server sends back another piece of text identical to the first except that every lowercase letter has been converted to uppercase. The server will be named `upcased` (for uppercase daemon), and the client, `upcaseclient`.

The `message struct` used by the `upcase1` server and client, as well as all necessary include files and common definitions, is contained in the header file `upcase1.h`, displayed below.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <signal.h>
#include <errno.h>

#define PUBLIC        "/tmp/UPCASE1_PIPE"
#define HALFPIPE_BUF  (PIPE_BUF/2)
```

```
struct message {
    char   fifo_name[HALFPIPE_BUF];
    char   text[HALFPIPE_BUF];
};
```

Because the message must be no larger than `PIPE_BUF` bytes, and because it should be flexible enough to allow FIFO pathnames of a large size, the `struct` is split equally between the length of the FIFO name and the length of the text to be sent to the server. Thus, `HALFPIPE_BUF` is defined as one half of `PIPE_BUF` and used as the maximum number of bytes in the string to be translated.

The basic steps that the client takes are as follows.

1. It makes sure that neither standard input nor output is redirected.
2. It registers its signal handlers.
3. It creates its private FIFO in `/tmp`.
4. It tries to open the public FIFO for writing in non-blocking mode.
5. It enters a loop in which it repeatedly
    i. reads a line from standard input, and
    ii. repeatedly
        a. gets the next `HALFPIPE_BUF-1` sized chunk in the input text,
        b. sends to the server through the public FIFO,
        c. opens its private FIFO for reading,
        d. reads the server's reply from the private FIFO,
        e. copies the server's reply to its standard output, and
        f. closes the read-end of its private FIFO.
6. It closes the write-end of the public FIFO and removes its private FIFO.

The client code follows.

```
// upcaseclient1.c
#include "upcase1.h"
#define  PROMPT   "string: "
#define  UPCASE   "UPCASE: "
#define  QUIT     "quit"
```

```
const char              startup_msg[] =
     "upcased does not seem to be running. Please start the service.\n";
volatile sig_atomic_t   sig_received = 0;
struct   message        msg;

void on_sigpipe( int signo )
{
    fprintf(stderr, "upcased is not reading the pipe.\n");
    unlink(msg.fifo_name);
    exit(1);
}

void on_signal( int sig )
{
    sig_received = 1;
}

int main( int argc, char *argv[])
{
    int      strLength;      // number of bytes in text to convert
    int      nChunk;         // index of text chunk to send to server
    int      bytesRead;      // bytes received in read from server
    int      privatefifo;    // file descriptor to read-end of PRIVATE
    int      publicfifo;     // file descriptor to write-end of PUBLIC
    static char      buffer[PIPE_BUF];
    static char      textbuf[BUFSIZ];
    struct sigaction handler;

    // Only run if we are using the terminal.
    if ( !isatty(fileno(stdin)) || !isatty(fileno(stdout)) )
        exit(1);

    // Register the on_signal handler to handle all signals
    handler.sa_handler = on_signal;      /* handler function    */
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1) ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigpipe;      /* handler function    */
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
```

```
    }

    // Create unique name for private FIFO using process-id
    sprintf(msg.fifo_name, "/tmp/fifo%d",getpid());

    // Create the private FIFO
    if ( mkfifo(msg.fifo_name, 0666) < 0 ) {
        perror(msg.fifo_name);
        exit(1);
    }

    // Open the public FIFO for writing
    if ( (publicfifo = open(PUBLIC, O_WRONLY | O_NDELAY) ) == -1) {
        if ( errno == ENXIO )
            fprintf(stderr,"%s", startup_msg);
        else
            perror(PUBLIC);
        exit(1);

    }

    // Repeatedly prompt user for input, read it, and send to server
    while (1) {
        // Check if a signal was received first, and if so, close
        // write-end of public fifo, remove private fifo and exit
        if ( sig_received ) {
            close(publicfifo);
            unlink(msg.fifo_name);
            exit(0);
        }

        // Display a prompt on the terminal and read the input text
        write( fileno(stdout), PROMPT, sizeof(PROMPT));
        memset(msg.text, 0x0, HALFPIPE_BUF);          // zero string
        fgets(textbuf, BUFSIZ, stdin);
        strLength = strlen(textbuf);
        if ( !strncmp(QUIT, textbuf, strLength-1))    // is it quit?
            break;

        // Display label for returned upper case text
        write(fileno(stdout), UPCASE, sizeof(UPCASE));

        for ( nChunk = 0; nChunk < strLength; nChunk += HALFPIPE_BUF-1 ) {
            memset(msg.text, 0x0, HALFPIPE_BUF);
            strncpy(msg.text, textbuf+nChunk, HALFPIPE_BUF-1);
            msg.text[HALFPIPE_BUF-1] = '\0';
```

```
            write(publicfifo, (char*) &msg, sizeof(msg));

            // Open the private FIFO for reading to get output of command
            // from the server.
            if ((privatefifo = open(msg.fifo_name, O_RDONLY) ) == -1) {
                perror(msg.fifo_name);
                exit(1);
            }

            // Read maximum number of bytes possible atomically
            // and copy them to standard output.
            while ((bytesRead= read(privatefifo, buffer, PIPE_BUF)) > 0) {
                write(fileno(stdout), buffer, bytesRead);
            }
        close(privatefifo);      // close the read-end of private FIFO
        }
    }
    // User quit; close write-end of public FIFO and delete private FIFO
    close(publicfifo);
    unlink(msg.fifo_name);
}
```

Comments.

- The program registers `on_signal()` to handle all signals that could kill it and that can be generated by a user. If any of these signals is sent to the process, the handler simply sets an atomic flag. In its main loop, it checks whether flag is set, and if it is, it closes the write-end of the public FIFO and removes its private FIFO. The server will get a `SIGPIPE` signal the next time it tries to write to this FIFO.

- The program will get a `SIGPIPE` signal if it tries to write to the public FIFO but it is not open for reading. This can only happen if the server is not running. The `SIGPIPE` handler, `on_sigpipe()`, displays a message on standard error and terminates the program.

- The reason that the client opens the public FIFO with `O_NDELAY` set is that, in this case, if the server is not reading the FIFO, the client, instead of blocking, will return with a `ENXIO` error, and it can gracefully exit.

- Inside the client's main loop, it displays a prompt and uses `fgets()` to read a line from the terminal.

- This client has been designed to handle the highly improbable case that the user enters a string that is larger than the allowed number of bytes in an atomic write to a pipe[4]. It does

4 Since `BUFSIZ`, the maximum size string allowed in the Standard I/O Library, may be larger than `PIPE_BUF`, it is possible to read a string much larger than can be sent in the pipe atomically.

this by breaking the string into "chunks" that are small enough to send atomically. It send each chunk in sequence. It has to open and close the private FIFO before and after each chunk is sent because the server is designed primarily for handling the most likely case in which the string is small enough to fit into a single chunk. (The server only opens the client's private FIFO after receiving a message from the client with the name of the FIFO; if the client tries to open the FIFO for reading before sending *any* chunks, it will block on the `open()` call. To prevent this, the `open()` would have to be non-blocking, which would complicate its read loop. It is not worth the complication to save the run-time cost in this unusual case.)

Now we turn to the server, which is simpler than the client in this example. The steps that the server takes can be summarized as follows.

1. It registers its signal handlers.
2. It creates the public FIFO. If it finds it already exists, it displays a message and exits.
3. It opens the public FIFO for both reading and writing, even though it will only read from it.
4. It enters its main-loop, where it repeatedly
   i. does a blocking read on the public FIFO,
   ii. on receiving a message from the `read()`, tries to open the private FIFO of the client that sent it the message. (It tries 5 times, sleeping a bit between each try, in case the client was delayed in opening it for writing. After 5 attempts it gives up on this client.)
   iii. converts the message to uppercase,
   iv. writes it to the private FIFO of the client, and
   v. closes the write-end of the private FIFO.

It will loop forever because it will never receive an end-of-file on the pipe, since it is keeping the write-end open itself. It is terminated by sending it a signal. The code follows.

```
//upcased1.c
#include "upcase1.h"   // fifo.h is shared by sender and receiver

#define   WARNING  "\nNOTE: SERVER ** NEVER ** accessed private FIFO\n"
#define   MAXTRIES 5

int       dummyfifo;       // file descriptor to write-end of PUBLIC
int       privatefifo;     // file descriptor to write-end of PRIVATE
int       publicfifo;      // file descriptor to read-end of PUBLIC
```

```
void on_sigpipe( int signo )
{
    fprintf(stderr, "Client is not reading the pipe.\n");
}

void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    if ( privatefifo != -1 )
        close(privatefifo);
    unlink(PUBLIC);
    exit(0);
}

int main( int argc, char *argv[])
{
    int              tries;       // num tries to open private FIFO
    int              nbytes;      // number of bytes read from popen()
    int              i;
    int              done;        // flag to stop loop
    struct message   msg;         // stores private fifo name and command
    struct sigaction handler;     // sigaction for registering handlers

    // Register the signal handler
    handler.sa_handler = on_signal;      /* handler function    */
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    handler.sa_handler = on_sigpipe;      /* handler function    */
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
            perror("sigaction");
            exit(1);
    }

    // Create public FIFO
    if ( mkfifo(PUBLIC, 0666) < 0 ) {
        if (errno != EEXIST )
```

```
            perror(PUBLIC);
        else
            fprintf(stderr, "%s already exists. Delete it and restart.\n",
                    PUBLIC);
        exit(1);
    }
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }

    // Block waiting for a msg struct from a client
    while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {
        tries = done = 0;
        privatefifo = -1;
        do {
            if ( (privatefifo = open(msg.fifo_name,
                                     O_WRONLY | O_NDELAY)) == -1 )
                sleep(1);   // sleep if failed to open
            else {
                // Convert the text to uppercase
                nbytes = strlen(msg.text);
                for ( i = 0; i < nbytes; i++ )
                    if ( islower(msg.text[i]))
                         msg.text[i] = toupper(msg.text[i]);

                if ( -1 == write(privatefifo, msg.text, nbytes) ) {
                    if ( errno == EPIPE )
                        done = 1;
                }
                close(privatefifo);  // close write-end of private FIFO
                done = 1;            // terminate loop
            }
        } while (++tries < MAXTRIES && !done);

        if ( !done)  // Failed to open client private FIFO for writing
            write(fileno(stderr), WARNING, sizeof(WARNING));
    }
    return 0;
}
```

Comments.

This server handles all user-initiated terminating signals by closing any descriptors that it has open and removing the public FIFO and exiting. It sets `privatefifo` to -1 at the start of each loop, and if it opens the private FIFO successfully, `privatefifo` is not -1. This way, in the

signal handler, it can determine whether it had a private FIFO open for writing and needs to close it.

If it gets a `SIGPIPE` because a client closed its read end of its private FIFO immediately after sending a message but before the server wrote back the converted string, it handles `SIGPIPE` by continuing to listen for new messages and giving up on the write to that pipe.

9.4.4 Concurrent Servers

The preceding server was an iterative server; it handled each client request one after the other. If some client requests could be very time-consuming, then the server would be busy servicing one client to the exclusion of all others, and the others would experience delays. This can be avoided by allowing the server to handle multiple clients simultaneously. A server that can process requests from more than one client simultaneously is called a ***concurrent server***.

The easiest way to create a concurrent server is to fork a child process for each client. The server's role then amounts to little more than "listening" to the public pipe for incoming requests, forking a child process to handle a new request, and waiting for its children to finish. The waiting must be accomplished through a `SIGCHLD` handler, because, unlike a shell-style application, this process has to return immediately to the task of reading the public pipe. The basic outline of the process is therefore roughly:

1. It registers its signal handlers.
2. It creates the public FIFO. If it finds it already exists, it displays a message and exits.
3. It opens the public FIFO for both reading and writing, even though it will only read from it.
4. It enters its main-loop, where it repeatedly
    i. does a blocking `read()` on the public FIFO,
    ii. on receiving a message from the `read()`, forks a child process to handle the client request.

Aside from spawning child processes, there are a few major differences between the way this server works and the way the sequential server worked:

- Each client will have two private FIFOs: one into which it writes raw text to be translated, and a second from which it reads text that the server translated and sent back to it.
- The public FIFO is used exclusively to send "connection" messages to the server. The connection message contains only the information needed to establish the two-way

private communication between the server and the client, namely the names of the two pipes:

```
struct message {
    char  raw_text_fifo [HALFPIPE_BUF];
    char  converted_text_fifo[HALFPIPE_BUF];
};
```

- Each child process forked by the server begins by opening the read-end of the client's "raw_text" FIFO, and then it repeatedly reads from the client's raw_text FIFO, translates the text into uppercase, opens the write-end of the client's converted_text FIFO, writes the converted text into it, and closes the write-end of the FIFO, until it received an end-of-file from the client.

The client is also structurally different from the previous client. The major steps it takes are:

1. It registers its signal handlers.
2. It creates two private FIFOs in the `/tmp` directory with unique names.
3. It opens the server's public FIFO for writing.
4. It sends the initial message structure containing the names of its two FIFOs to the server to establish the two-way communication.
5. It attempts to open its raw_text FIFO in non-blocking, write-only mode. If it fails, it delays a second and retries. It retries a few times and then gives up and exits. If it fails it means that the server is probably terminated.
6. Until it receives an end-of-file on its standard input, it repeatedly
   i. reads a line from standard input,
   ii. breaks the line into `PIPE_BUF`-sized chunks,
   iii. sends each chunk successively to the server through its raw_text FIFO,
   iv. opens the converted_text FIFO for reading,
   v. reads the converted_text FIFO, and copies its contents to its standard output, and
   vi. closes the read-end of the converted_text FIFO
7. It closes all of its FIFOs and removes the files.

Illustration 4 shows how the client processes and the server parent and child processes use the various FIFOs. Compare this to Illustration 3.

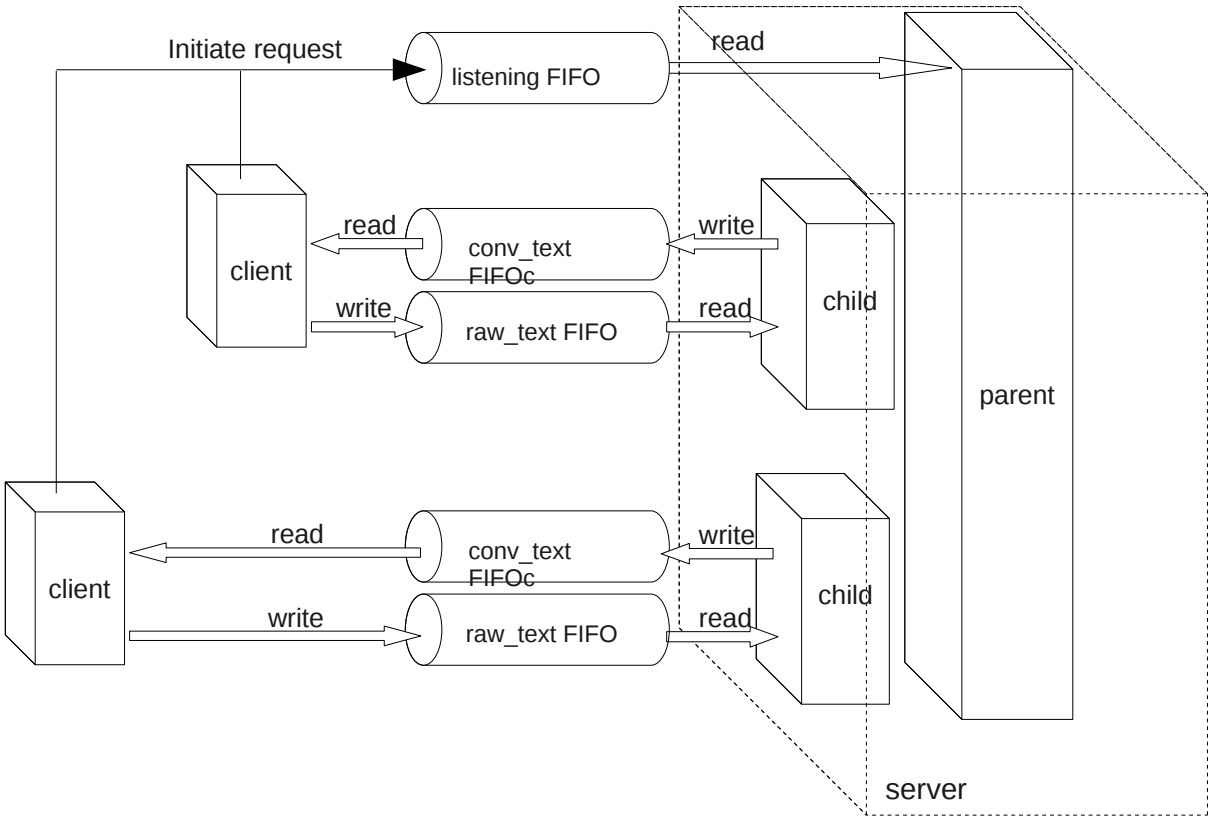


Illustration 4: Use of Pipes in a Concurrent Server

The code for the client is displayed first.

```
//upcaseclient2.c
#define  MAXTRIES 5
const char   server_no_read_msg[] =
"The server is not reading the pipe.\n";

const char   noserver_msg[] =
"The server does not appear to be running. "
"Please start the service.\n";

const char   missing_pipe_msg[] =
"cannot communicate with the server due to a missing pipe.\n"
"Check if the server is running and restart it if necessary.\n";

int            convertedtext_fd; // client's read FIFO
int            dummyreadfifo;    // to hold write-end open
int            rawtext_fd;       // client's write FIFO
```

```
int             publicfifo;       // server's public FIFO
FILE*           input_srcp;       // input stream
struct message msg;

// Signal Handlers
void on_sigpipe( int signo )
{
    fprintf(stderr, "%sExiting...\n",server_no_read_msg);
    unlink(msg.raw_text_fifo);
    unlink(msg.converted_text_fifo);
    exit(1);
}

void on_signal( int sig )
{
    close(publicfifo);
    if ( convertedtext_fd != -1 )
        close(convertedtext_fd);
    if ( rawtext_fd != -1 )
        close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
    exit(0);
}

// Clean up routine
void clean_up()
{
    close(publicfifo);
    close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
}

int main( int argc, char *argv[])
{
    int              strLength;
    int              nChunk;
    int              nbytes;
    int              tries = 0;
    static char      buffer[PIPE_BUF];
    static char      textbuf[BUFSIZ];
    struct sigaction handler;

    if ( argc > 1 ) {
        if ( NULL == (input_srcp  = fopen(argv[1], "r")) ) {
```

```
            perror(argv[1]);
            exit(1);
        }
    }
    else
        input_srcp = stdin;

    publicfifo = -1;
    convertedtext_fd   = -1;
    rawtext_fd = -1;

    // Register the on_signal handler to handle all signals
    handler.sa_handler = on_signal;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    // Create unique names for private FIFOs using process-id
    sprintf(msg.converted_text_fifo, "/tmp/fifo_rd%d",getpid());
    sprintf(msg.raw_text_fifo, "/tmp/fifo_wr%d",getpid());

    // Create the private FIFOs
    if ( mkfifo(msg.converted_text_fifo, 0666) < 0 ) {
        perror(msg.converted_text_fifo);
        exit(1);
    }
    if ( mkfifo(msg.raw_text_fifo, 0666) < 0 ) {
        perror(msg.raw_text_fifo);
        exit(1);
    }

    // Open the public FIFO for writing
    if ( (publicfifo = open(PUBLIC, O_WRONLY | O_NDELAY) ) == -1) {
        if ( errno == ENXIO )
            fprintf(stderr,"%s", noserver_msg);
```

```
        else if ( errno == ENOENT )
            fprintf(stderr,"%s %s", argv[0], missing_pipe_msg);
        else {
            fprintf(stderr,"%d: ", errno);
            perror(PUBLIC);
        }
        clean_up();
        exit(1);
    }

    // Send a message to server with names of two FIFOs
    write(publicfifo, (char*) &msg, sizeof(msg));

    // Open the raw text fifo for writing: if server is not reading
    // try a few times and then exit
    while (((rawtext_fd = open(msg.raw_text_fifo,
             O_WRONLY | O_NDELAY )) == -1 ) && (tries < MAXTRIES ))  {
            sleep(1);
            tries++;
    }
    if ( tries == MAXTRIES ) {
        // Failed to open client private FIFO for writing
        fprintf(stderr, "%s", server_no_read_msg);
        clean_up();
        exit(1);
    }

    // Get one line of input at a time from the input source
    while (1) {
        memset(textbuf, 0x0, BUFSIZ);
        if ( NULL == fgets(textbuf, BUFSIZ, input_srcp) )
            break;
        strLength = strlen(textbuf);

        // Break input lines into chunks and send them one at a
        // time through the client's raw_text FIFO
        for ( nChunk = 0; nChunk < strLength; nChunk += PIPE_BUF-1 ) {
            memset(buffer, 0x0, PIPE_BUF);
            strncpy(buffer, textbuf+nChunk, PIPE_BUF-1);
            buffer[PIPE_BUF-1] = '\0';
            write(rawtext_fd, buffer, strlen(buffer));
            if ( (convertedtext_fd = open(msg.converted_text_fifo,
                  O_RDONLY) ) == -1) {
                perror(msg.converted_text_fifo);
                clean_up();
                exit(1);
```

```
            }

            memset(buffer, 0x0, PIPE_BUF);
            while ((nbytes = read(convertedtext_fd, buffer,PIPE_BUF)) > 0)
                write(fileno(stdout), buffer, nbytes);

            close(convertedtext_fd);
            convertedtext_fd   = -1;
        } // end of for-loop
    }
    clean_up();
}
```

Comments.

- The order of events here is important, and in some cases critical. After the client creates its private FIFOs without error, it opens the write-end of the server's public FIFO. It then sends a message containing the names of its private FIFOs. After sending the names of the private FIFOs, it tries to open the write-end of its raw_text FIFO in non-blocking mode. This will fail if the server has not opened the read-end yet. Assuming that the server is running, the client will succeed in opening the raw_text FIFO. The server can open its read-end without the write-end being open, so this works well. If we were to reverse the order and open the raw_text FIFO before sending the server the message, we would need to open it in read-write mode since the server is blocked on its read of the public FIFO and the two processes would deadlock otherwise. But if we open the raw_text FIFO in read-write mode, then if the server terminates unexpectedly and never reads the raw_text FIFO again, the client will not get a `SIGPIPE` signal because the client itself has a read-end open, preventing the kernel from generating the signal. The client would never be notified that the server died.

- The client then keeps the write-end of its raw_text FIFO open for the duration of its main loop.

- Within the loop, the client first writes to its raw_text FIFO, and then opens its converted_text FIFO, after which, if all goes well, it reads and closes it again. Thus, it repeatedly opens and closes this FIFO within the loop. We could just let it stay open for the duration of the loop.

- The error handling in the client is similar to what it was in the iterative server's client. The code has redundant error checks such as guards to prevent closing a FIFO that is not open (setting the file descriptors to -1 unless they are in use), and closing descriptors before `unlink()`-ing the files. On the other hand, it should really check the return values of the `close()` calls. A `clean_up()` function simplifies the error-handling, consolidating the cleaning up code.

The server's code is next.

```
// upcased2.c
#include "upcase2.h"
#include "sys/wait.h"

#define    WARNING  "\nNOTE: SERVER ** NEVER ** accessed private FIFO\n"
#define    MAXTRIES 5
int        dummyfifo;        // file descriptor to write-end of PUBLIC
int        clientreadfifo;   // client's converted_text FIFO
int        clientwritefifo;  // client's raw_text FIFO
int        publicfifo;       // file descriptor to read-end of PUBLIC
FILE*      upcaselog;        // points to log file for server
pid_t      server_pid;       // stores parent pid

void on_sigpipe( int signo );
void on_sigchld( int signo );
void on_signal( int sig );

int main( int argc, char *argv[])
{
    int              tries;      // num tries to open private FIFO
    int              nbytes;     // number of bytes read from popen()
    int              i;
    struct message   msg;        // stores private fifo name and command
    struct sigaction handler;    // sigaction for registering handlers
    char             buffer[PIPE_BUF];
    char             logfilepath[PATH_MAX];
    char             *homepath;

    homepath  = getenv("HOME");
    sprintf(logfilepath, "%s/.upcase_log", homepath );

    if ( NULL == (upcaselog  = fopen(logfilepath, "a")) ) {
        perror(logfilepath);
        exit(1);
    }

    // Register the signal handlers
    handler.sa_handler = on_signal;
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
        perror("sigaction");
        exit(1);
```

```
    }
    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }
    handler.sa_handler = on_sigchld;
    if ( sigaction(SIGCHLD, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    // Create public FIFO
    if ( mkfifo(PUBLIC, 0666) < 0 ) {
        if (errno != EEXIST )
            perror(PUBLIC);
        else
            fprintf(stderr, "%s already exists. Delete it and restart.\n",
                    PUBLIC);
        exit(1);
    }

    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        unlink(PUBLIC);
        exit(1);
    }
    // Store the process id of the parent process for the signal handler
    // to compare to later.
    server_pid = getpid();

    while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {
        if ( 0 == fork() ) {
           clientwritefifo = -1;
           if ((clientwritefifo = open(msg.raw_text_fifo,O_RDONLY))==-1) {
               fprintf(stderr, "Client pipe not open for writing\n");
               exit(1);
            }
            // Clear the buffer used for reading the client's text
            memset(buffer, 0x0, PIPE_BUF);
            while ((nbytes = read(clientwritefifo,buffer,PIPE_BUF)) > 0) {
                 // Convert the text to uppercase
                for ( i = 0; i < nbytes; i++ )
                     if ( islower(buffer[i]))
                         buffer[i] = toupper(buffer[i]);
```

```
                tries = 0;
                // Open client's convertedtext_fd --
                // Try 5 times or until client is reading
                while (((clientreadfifo = open(msg.converted_text_fifo,
                        O_WRONLY|O_NDELAY)) == -1) && (tries < MAXTRIES))
                {
                     sleep(1);
                     tries++;
                }
                if ( tries == MAXTRIES ) {
                    // Failed to open client private FIFO for writing
                    write(fileno(stderr), WARNING, sizeof(WARNING));
                    exit(1);
                }

                // Send converted text to client in its readfifo
                if ( -1 == write(clientreadfifo, buffer, nbytes) ) {
                    if ( errno == EPIPE )
                        exit(1);
                }
                close(clientreadfifo);  // close write-end of private FIFO
                clientreadfifo = -1;

                // Clear the buffer used for reading the client's text
                memset(buffer, 0x0, PIPE_BUF);
              }
            exit(0);
        }
    }
    return 0;
}
```

The signal handlers for the server are below. The `SIGCHLD` handler uses `waitpid()` to wait for all children, and it remains in its loop as long as there is a zombie to be collected. The `WNOHANG` flag is used to prevent it from blocking in the `waitpid()` code. This way, if multiple `SIGCHLD` signals arrive while it is in the handler, the children whose deaths caused them will be collected. (Remember that signals may not be reliably handled on all systems, and even though in a POSIX compliant system, each `SIGCHLD` will be delivered if we set `SA_NODEFER`, it is safer to collect them in this loop.)

```
void on_sigchld( int signo )
{
    pid_t pid;
    int   status;
```

```
    while ( (pid = waitpid(-1, &status, WNOHANG) ) > 0 )
        fprintf(upcaselog, "child %d terminated.\n", pid);
        fflush(upcaselog);
    return;
}

void on_sigpipe( int signo )
{
    fprintf(stderr, "Client is not reading the pipe.\n");
}

void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    if ( clientreadfifo != -1 )
        close(clientreadfifo);
    if ( clientwritefifo != -1)
        close(clientwritefifo);
    // If this is the parent executing it, remove the public fifo
    if ( getpid() == server_pid )
        unlink(PUBLIC);
    fclose(upcaselog);
    exit(0);
}
```

Comments.

- All of the work is performed by the child processes. Each child begins by trying to open the client's raw_text FIFO for reading. If successful, it enters a loop in which it repeatedly reads, converts the text to uppercase, opens the client's converted_text FIFO, writes the converted text to it, and closes it.

- Since the client may not have the converted_text open for reading for any number of reasons -- it might have been terminated -- the child process tries the `open()` a fixed number of times before it gives up. It uses the same technique as the iterative server did, using a non-blocking `open()`.

- When the child process does successfully open the FIFO, it still checks whether the `write()` failed, since anything can happen in between, and if so, the child exits. Otherwise, it writes the data, closes its end of the FIFO and waits to read more text from the client. When it receives the end-of-file, it exits.

- The signal handler checks whether the parent process is executing it. If the parent has been signaled, then it should remove the public FIFO, otherwise not. We do not want child processes to remove this FIFO!

If you are at all familiar with sockets, you might have noticed that the design of this server is easily converted to one that uses sockets. We will refer back to this example when we take up sockets.

9.5 Daemon Processes

As was mentioned earlier, a ***daemon*** is a process that runs in the background, has no controlling terminal. In addition, daemons set their working directory to "/". Usually daemons are started by system initialization scripts at boot-time. If you have written a server and want to turn it into a full-fledged daemon, it is not enough to put it into the background. This will only tell the shell not to wait for it; it will still have a control terminal and will still be killed by any signals from that terminal.

Some daemons are started by other programs. For example, some network daemons are started by the `inetd` or `xinetd` superserver. Some are started by programs such as the `crond` daemon, which runs scheduled jobs. Some are invoked at the user terminal. For example, sometimes the printer daemon is stopped and restarted at the terminal by the superuser.

Because daemons do not have a controlling terminal, they cannot write messages to standard output or to standard error. Instead they can use a system logging function named `syslog()`, which is a client that talks to the `syslogd` daemon, which write messages to specific log files. The *glibc* version of this function is `klogctl()`. Later we will look at an example of how it can be used. A server should be designed to turn itself into a daemon. In other words, when the server is run, it should take all of the steps necessary to become a daemon, which include:

1. Putting itself in the background. It does this by forking a new process and executing its code as the child and having the parent execute `exit()`. When the parent exits, the shell that started it collects its exit status and thinks the invoked program has terminated (which it has.) The child, which is now the server, is no longer in the foreground, but it is still controlled by the terminal.

2. Making itself a session leader. Recall from Chapter 8 a process can detach itself from a terminal by becoming a session leader, but only processes that are neither session leaders nor process group leaders can do this. Since the server is now a child of the original process, it is neither, so it can call `setsid()`, which makes it a session leader of a new session and a group leader of a new process group.

3. Registering its intent to ignore `SIGHUP`.

4. Forking another child process, terminating in the parent again, and letting the new child, which is the grandchild of the original process, execute the server code. In some versions of UNIX, when a session leader opens a terminal device (which it may want to do sometimes), that terminal is automatically made the control terminal for the process. By running as the child of a session leader, the server is now immune from this eventuality. In Linux, a process can set the `O_NOCTTY` flag on `open()` to prevent this.

The reason for ignoring `SIGHUP` is that when a session leader terminates, all of its children are sent a `SIGHUP`, which would otherwise kill them. Since the parent is a session leader, the child must ignore `SIGHUP`.

5. Changing the working directory to "`/`".
6. Clearing the `umask`.
7. Closing any open file descriptors.

A procedure for doing all of these steps, based on one from [Stevens], is below.

```
#include <syslog.h>
#define  MAXFD  64
void daemon_init(const char *pname, int facility)
{
    int          i;
    pid_t   pid;

    if ( (pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    else if (pid != 0)
           exit(0);       // parent terminates

    // Child continues from here
    // Detach itself and make itself a sesssion leader
    setsid();

    // Ignore SIGHUP
    signal(SIGHUP, SIG_IGN);

    if ( (pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    else if ( pid != 0 )
         exit(0);   // First child terminates

    // Grandchild continues from here
    chdir("/");        // change working directory

    umask(0); // clear our file mode creation mask

    // Close all open file descriptors
    for (i = 0; i < MAXFD; i++)
```

```
        close(i);

    // Start logging with syslog()
    openlog(pname, LOG_PID, facility);
}
```

The final version of the `upcase` server incorporates this function and turns itself into a daemon. The only changes required are to include this function in the code and insert the line

```
daemon_init(argv[0], 0);
```

before the first executable statement.

9.6 Multiplexed I/O With select

Imagine the situation in which a process has multiple sources of input open for reading, such as a set of pipes as well as the terminal. Suppose the process has to respond to commands typed at the terminal as well as display messages that are available in the pipes. This is what is meant by ***multiplexed input***: when a process has to obtain input available from multiple sources simultaneously. One solution would be to make all of the reads non-blocking and to continually poll each descriptor to see if there is data ready for reading on it. Polling, though, has many drawbacks, as we have seen, the most important of which is that it is wasteful of the CPU resource.

Another alternative would be to use asynchronous reads on each descriptor. This is also possible, but quite messy to code, and has the drawback that it relies on signals which may not be handled properly or reliably.

It is for these reasons that the `select()` system call was developed[5]. Basically, the `select()` call allows a process to listen to multiple descriptors at once and to be notified when any of them have pending input or output. Roughly put, `select()` is given a set of masks of file descriptors, representing I/O devices or files in which the process is interested. When input or output is ready on any of them, the appropriate bits in these masks are set. The process can check the masks to see which I/O is ready and can then read or write the ready descriptors. The `select()` call works with any file descriptor, so that it can be used with files, pipes, FIFOs, devices, and sockets.

`select()` is fairly complex:

```
/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
```

5 There is a similar call named `poll()`.

```
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

The parameters have the following meanings:

`ndfs` The number of file descriptors of potential interest.

`readfds` The address of a file descriptor mask indicating which file descriptors the process is interested in ***reading***.

`writefds` The address of a file descriptor mask indicating which file descriptors the process is interested in ***writing***.

`exceptfds` The address of a file descriptor mask indicating which file descriptors the process is interested in checking for ***out-of-band*** data[6]. (Out-of-band messages or data should be thought of as exceptions or error conditions concerning any of the descriptors in the read or write descriptor masks.)

`timeout` The address of a `timeval` struct containing the amount of time to wait before completing the `select()` call. If timeout is `NULL`, it means wait forever, i.e., block until at least one descriptor is ready. If it is zero, it means return immediately with the status of all descriptors in the above sets. If it is non-zero, it will either wait the specified amount of time or return before if one of the specified descriptors is ready.

The return value of the `select()` call is the number of descriptors that are ready, or -1 if there was an error.

The `fd_set` data type is not necessarily a scalar. It is usually an array of long integers. If you do a little digging you will discover a constant, `FD_SETSIZE`, that defines the maximum number of descriptors in a `fd_set`, which is 1024 in the kernel we are using. Fortunately, you do not need to know how it is defined to use it, since there are macros and/or functions in the library for manipulating `fd_set` objects:

This turns off the bit for descriptor `fd` in the mask pointed to by `fdset`:

```
void FD_CLR(int fd, fd_set *fdset);
```

This turns on the bit for descriptor `fd` in the mask pointed to by `fdset`:

```
void FD_SET(int fd, fd_set *fdset);
```

This sets all bits to zero in the mask pointed to by `fdset`:

6 Out-of-band refers to data that is transferred in a separate communication channel. Out-of-band implies that the data does not arrive in sequence with the rest of the data, but in a parallel channel. It is used for transmitting error or control messages.

```
void FD_ZERO(fd_set *fdset);
```

This checks whether the bit for descriptor `fd` is set in the mask pointed to by `fdset`:

```
int  FD_ISSET(int fd, fd_set *set);
```

The value of the first parameter, `ndfs`, must be set to the ***value of the largest file descriptor + 1***, since the file descriptor array is 0-based. The reason that the first argument is the maximum number of descriptors of interest is for efficiency. By supplying this number to the kernel, it saves the kernel the work of having to copy parts of the descriptor mask that are not needed. To give you an idea of how this call is used in a simple case, if we wanted to read from two different open file descriptors, we would use something like

```
#include <sys/time.h>
#include <sys/types.h>
...
int     fd1, fd2, maxfd;
fd_set  readset, tempset;

fd1  = open("file1", O_RDONLY);  // open file1
fd2  = open("file2", O_RDONLY);  // open file2
maxfd = fd1 > fd2 ? fd1+1 : fd2+1;

FD_ZERO(&readset);          // clear the bits in the mask
FD_SET(fd1, &readset);      // set the bit for fd1 (file1)
FD_SET(fd2, &readset);      // set the bit for fd2 (file2)
tempset = readset;          // copy into tempset

while ( select(maxfd, &tempset, NULL, NULL, NULL) > 0) {
    if ( FD_ISSET(fd1, &tempset) ) {
         // read from descriptor fd1
    }

    if ( FD_ISSET(fd2, &tempset) ) {
        // read from descriptor fd2
    }

    tempset = readset;
}
```

Notes.

- Although we are interested only in file descriptors `fd1` and `fd2`, the proper way to use select is to specify the full range of descriptors from 0 to the maximum of `fd1` and `fd2`. Since it is a zero-based array, this value is `max(fd1, fd2)` + 1.

- Because the return value of `select()` is positive as long as there is data to be read on either of `fd1` or `fd2`, the loop will continue until we get end-of-file on both files.
- The way that `select()` works, it resets the file descriptor masks to reflect the status of the descriptors of interest. In other words, the masks change after each call to `select()`. Therefore, you need to keep a copy of the original mask, and before each call, reset the masks to their original states.
- The masks are not modified if the `select()` call returned with an error.
- Inside the loop, you use the `FD_ISSET()` function to test each descriptor in which you expressed interest.
- It is a very common mistake to forget to add 1 to the largest descriptor in the first argument. It is also a common mistake to forget to reset the mask between each successive call.

We will put these ideas to work in a slightly more interesting example, borrowed from [Haviland et al].

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/wait.h>

#define  MSGSIZE   6

char  msg1[] = "Hello";
char  msg2[] = "Bye!!";

void parent( int pipeset[3][2]);
int child( int fd[2] );

int main( int argc, char* argv[])
{
    int fd[3][2];  // array of three pipes
    int i;

    for ( i = 0; i < 3; i++ ) {
        // create three pipes
        if ( pipe(fd[i]) == -1 ) {
            perror("pipe");
            exit(1);
        }
```

```
        // fork children
        switch( fork() ) {
        case -1 :
            fprintf(stderr, "fork failed.\n");
            exit(1);
        case 0:
            child(fd[i]);
        }
    }
    parent(fd);
    return 0;
}

void parent( int pipeset[3][2])
{
    char    buf[MSGSIZE];
    char    line[80];
    fd_set  initial, copy;
    int     i, nbytes;

    for ( i = 0; i < 3; i++)
         close(pipeset[i][1]);

    // create descriptor mask
    FD_ZERO(&initial);
    FD_SET(0, &initial);                    // add standard input

    for ( i = 0; i < 3; i++)
        FD_SET(pipeset[i][0], &initial);    // add read end of each pipe

    copy = initial;                         // make a copy
    while (select( pipeset[2][0]+1, &copy, NULL, NULL, NULL ) > 0 ) {
        if ( FD_ISSET(0, &copy) ) {
            printf("From standard input: ");
            nbytes = read(0, line, 81);
            line[nbytes] = '\0';
            printf("%s", line);
        }

        // check the pipe from each child
        for ( i = 0; i < 3; i++ )
            if ( FD_ISSET(pipeset[i][0], &copy))
                // it is ready to read
                if ( read( pipeset[i][0], buf, MSGSIZE) > 0 )
                    printf("Message from child %d:%s\n", i, buf );
```

```
        if (waitpid(-1, NULL, WNOHANG) == -1 )
            return;
        copy = initial;
    }
}

int child( int fd[2] )
{
    int count;
    close(fd[0]);
    for ( count = 0; count < 10; count ++) {
         write( fd[1], msg1, MSGSIZE);
         sleep(getpid() % 10 );  // sleep a pseudo random # of seconds
    }
    write( fd[1], msg2, MSGSIZE);
    exit(0);
}
```

Comments.

- Each child writes a small string to the write-end of its pipe and then sleeps a bit so that the output does not flood the screen too quickly.
- The parent uses the `select()` call to query standard input and the read-ends of ech child's pipe. The user can type a string on the keyboard and the parent will detect that standard input is ready. Within the while-loop each descriptor is tested, and if it is set, the read() can be done because input is waiting. This way the parent never holds up any child that is waiting for its message to be read.

There will be another, more interesting use of `select()` after the introduction to sockets.

9.7 Sockets

Pipes are a good segue into sockets. ***Sockets*** are used like pipes in many ways but, unlike pipes, they can be used across networks. Sockets allow unrelated processes on different computers on a network to exchange data through a channel, using ordinary `read()` and `write()` system calls. We call this ***remote interprocess communication***. The development of sockets comes from the Berkeley distributions of UNIX in the early 1980's. AT&T developed a different API for remote interprocess communication, called the ***Transport Level Interface*** (TLI) in 1986. In many ways, TLI has advantages over sockets, however, sockets have been around a long time, there is a great deal of code that uses them, sockets can be accessed like files and work the same whether on a local machine or across a network, making programming them easier. TLI programming is more complex; it requires many more structures. This is why we will be looking at sockets here. In order to understand how to use sockets, you need to know the basics of networks.

9.7.1 Connections

There are two ways in which sockets can be used, corresponding roughly to the difference between making a telephone call and having an email conversation with someone. When you make a telephone call to someone, you have a conversation over a dedicated communication channel, the telephone line, and you stay connected with the person on the other end for the duration of the call. In socket parlance this is called a ***connection oriented model***. When you have an email conversation with someone, the messages are sent to the other person across different paths, and there is no dedicated connection. In fact there is no guarantee that the messages that you send will arrive in the order you send them, and the only way for the person who receives them to know who sent them is for them to have a return address in their message header. In socket parlance this is the ***connectionless model***.

The connection oriented model uses the ***Transmission Control Protocol***, known as ***TCP***. The connectionless model uses the ***User Datagram Protocol***, or ***UDP***. There are many important differences between TCP and UDP, or equivalently, between connection oriented and connectionless models, but we cannot go into them at length here. The most important differences are that TCP provides a reliable, full-duplex, sequenced channel with flow control. UDP can be full-duplex but it is not reliable (no guarantee of packet delivery), not sequenced (packets can arrive in different order than they were sent), and has no flow control (a sender can send faster than the receiver can receive).

9.7.2 Communication Basics

In order to understand how to program with sockets, you need to have a basic understanding of the important concepts that underly their use. This includes network addresses, communication domains (not internet domains), protocol families, and socket types.

Network Addresses

For two processes to communicate, they need to know each other's network addresses. At the level of socket programming, a network address consists of two parts: an internet (IP) address and a port number. The IP address, if 32 bits, consists of 4, 8-bit octets, and is expressed in the standard dot-notation as in "`146.95.2.131`", which happens to be the address of eniac. Some computers have multiple network interface cards and therefore may have multiple internet addresses. It used to be the case that internet addresses had a specific structure and were divided into address classes. That is no longer the case. They are now just flat addresses.

The kernel does not represent IP addresses as strings of octets -- that would be inefficient. It uses the `in_addr_t` data type, defined in `<arpa/inet.h>`, to represent an IP address. However, we will see that there are functions to convert from one format to the other.

The server on a machine has to have a specific ***port*** that it uses. There are many analogies that we could use, but if you think of an IP address as specifying a specific company's main telephone line, then the port is like a telephone extension within the company. The server uses a specific

port for its services and the clients have to know the port number in order to contact the server. A port is a 16-bit integer.

Certain port numbers are "***well-known***" and reserved by particular applications and services. For example, port 7 is for echo servers, 13 for daytime servers, 22 for SSH, 25 for SMTP, and 80 for HTTP. Port numbers from 1 to 1023 are the well-known ports. To see a list of the port numbers in use, take a look at the file, `/etc/services`.

Ports 1024 through 49151 are registered ports. These numbers are not controlled and a service can use one if it is not already in use.

Ports 49152 through 65536 cannot be used. They are called ***ephemeral ports***, which are assigned automatically by TCP or UDP for client use.

The lsof command can be used to view the ports that are currently open. The command is actually more general than this -- it can be used to view all open files.

To add: The lsof command for seeing ports in use, different definitions from BSD Solaris IANA,

Families

In order for two processes to communicate, they must use the same protocol. Part of the procedure for establishing communication involves specifying the address and protocol family. For example, the family might specify that the protocol is *IPv4* and the addresses are IP addresses, or that the family is *IPv6* with IP addresses, or that addresses are purely local, i.e., on the same machine using UNIX protocols. When a socket is created, the family is specified as one of the arguments to the function.

Socket Types

When a socket is created, its type must be specified. The type corresponds to the type of connection. A connection oriented communication has type `SOCK_STREAM`, whereas a connectionless communication is of type `SOCK_DGRAM`. There are also raw sockets, with type `SOCK_RAW`.

9.7.3 The Socket Interface

What then is a socket? A socket is an *endpoint of a two-way communication channel.* The `socket()` system call creates this endpoint and returns a file descriptor that represents it. We do not have to know how a socket is implemented to use it, however, you should think of a socket as something like the file structure that represents a file in UNIX. It is an internal structure in the kernel, accessed through a file descriptor, representing one end of a communication channel that has a specific network address, family (also called domain), port number, and socket type.

The `socket()` function creates what is called an unnamed socket:

```
#include <sys/socket.h>

int socket( int domain, int type, int protocol);
```

The domain is an integer specifying the address family and protocol. These families are defined in `<sys/socket.h>`. Some of the common domain values are

Name	*Purpose*
`PF_UNIX`	Local communication
`PF_INET`	IPv4 Internet protocols
`PF_INET6`	IPv6 Internet protocols

The `type` can be one of `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW` or several others. The `SOCK_STREAM` type is the connection model, with full-duplex, reliable, sequenced transmissions.

The `protocol` can be used to specify a particular protocol in the case that there is more than one choice for the particular type of socket and address family. Setting it to 0 ensures that the kernel will pick the appropriate protocol.

The return value of `socket()` is a file descriptor that can be used to read or write the socket. As an example,

```
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

creates a connection-oriented socket that can be used for communication over the internet.

9.7.4 Setting Up a Connection Oriented Service

We will go through the steps that a server must take in a connection-oriented model. The basic steps that a server must take are below. Details on how to use the specific functions will follow.

1. Create a socket using `socket()`. This creates an endpoint of communication, but does not associate any particular internet address or port number to it.

2. Bind the socket to a local protocol address using `bind()`. This gives a "name" to the socket.

3. The socket created so far is an active socket, one that can connect to other sockets actively, like dialing another telephone number. Since this is the server, the purpose of this socket is not to "dial-out" but to listen for incoming calls. Therefore, the server must now call `listen()` to tell the kernel that all it really wants to do is listen for incoming messages and set a limit on its queue size. This call will basically put the socket into the LISTEN state in the TCP protocol.

   After `listen()` has returned, two queues have been created for the server. One queue stores incoming connection requests that have not yet completed the TCP handshake

protocol. The other queue stores incoming requests that have completed the handshake. These requests are ready to be serviced.

4. Enter a loop in which it repeatedly accepts new connections and processes them. It can accept a new connection with the `accept()` call. The `accept()` function removes the request at the front of the completed connection queue and creates a second socket that the server can use for talking with this client. The return value of `accept()` is a file descriptor that represents this socket. The original socket continues to exist. The idea is that the original socket is just for listening, not talking to clients. In fact it is called the ***listening socket***, and the new socket is called the ***connected socket***. When the connection is closed, this connected socket is removed.

That is the essence of the server's tasks. Now what remains is to see how to program this.

9.7.5 Programming a Connection Oriented Server

We have already seen how the `socket()` call works. The step of binding a local protocol address to the socket is carried out with `bind()`, but before we look at `bind()` we need to see how these addresses are represented. A ***generic socket*** address is defined by the `sockaddr struct` defined in `<sys/socket.h>`:

```
struct sockaddr {
    sa_family_t sa_family;  /* address family */
    char        sa_data[];  /* socket  address */
};
```

This is a generic socket address structure because it is not specific to any one address family. When you call `bind()`, you will be specifying a particular family, such as `PF_INET` or `PF_UNIX`. For each of these there is a different form of socket address structure. The address structure for `PF_INET`, defined in `<netinet/in.h>`, would be

```
struct sockaddr_in {
    sa_family_t    sin_family;  /* internet address family */
    in_port_t      sin_port;    /* port number */
    struct in_addr sin_addr;    /* IP  address */
    unsigned char  sin_zero[8]; /* padding  */
};
```

The `bind()` system call takes the socket file descriptor returned by `socket()` and an address structure like the one above, and "binds" them together to form the end of a socket that can now be used by processes out there in internet land to find this server:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *address,
         socklen_t addrlen);
```

Putting these few steps together, we might start out with the following code:

```
int listenfd;
int size  = sizeof(struct sockaddr_in);

struct sockaddr_in server = {PF_INET, 25555, INADDR_ANY};

if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1 )
{
    perror("socket call failed");
    exit(1);
}

if ( bind( listenfd, ( struct sockaddr *) &server, size ) == -1 )
{
    perror(" bind call failed");
    exit(1);
}
```

The `sockaddr_in struct` is initialized to use the IPv4 protocol family with a port of `25555`, large enough to be safe for our purposes, and `INADDR_ANY` as the local IP address. Specifying this constant means that if there is more than one IP address for this host, any will do. The bind call is given the `listenfd` descriptor, the address of this struct, and its `size`.

The next step is to call `listen()`, which is defined by

```
#include <sys/socket.h>
int listen(int sockfd, int queue_size);
```

The first argument is the descriptor for the already bound socket, and the second is the maximum size of the queue of pending (incomplete) connections.

The `accept()` call is defined as follows:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

The `accept()` call expects the socket descriptor of a socket that has been created with `socket()`, bound to a local address with `bind()`, and set to listen with `listen()`. The second argument, if not `NULL`, is a pointer to a generic socket address structure, and the third is the address of a variable that stores its length in bytes. After the call, the address will be filled with the client's socket address, and the size will reflect the true size of the client's specific socket

address `struct`. The return value will be the descriptor of a connected socket. The `accept()` will block waiting for a connection.

We can put all of this together in a simple concurrent server that, yes, once again, does lower to upper case conversion. This time it will handle just one character at a time. We will move on to a more interesting task afterwards. This code is based on an example from [Haviland et al]. It forks a child process to handle each incoming connection. The client and the server will share a common header file, `sockdemo1.h`, which is displayed first, followed by the server code.

```
// sockdemo1.h
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <ctype.h>
#include <signal.h>
#include <errno.h>

#define SOCKADDR       (struct sockaddr*)
#define SIZE           sizeof(struct sockaddr_in)
#define DEFAULT_HOST   "eniac.geo.hunter.cuny.edu"
#define PORT           25555
#define ERROR_EXIT( _mssg, _num)  perror(_mssg);exit(_num);
```

Comments.

- (Perhaps out of laziness, I finally wrote a little macro (`ERROR_EXIT`) so that I do not have to keep typing the `perror(); exit()` combination on failures of system calls. It is included in this header file. Rather than making it a function, I made it a macro so that the code is faster.)
- The `SOCKADDR` macro reduces typing.
- The server and client are compiled with the same header so that the port number is hard-coded into each. The number 25555 appears to be unused on all of the machines I have run this example on.

The server code follows.

```
//sockdemo1_server.c
#include "sockdemo1.h"
#define LISTEN_QUEUE_SIZE   5

// The following typedef simplifies the function definition after it
typedef void    Sigfunc(int);   /* for signal handlers */

// override existing signal function to handle non-BSD systems
```

```
Sigfunc*  Signal(int signo, Sigfunc *func);

// Signal handlers
void on_sigpipe(int sig);
void on_sigchld(int sig);

// This needs to be global because the signal handler has to access it
int connectionfd;

int main(int argc, char* argv[])
{
    int listenfd;    // holds the file descriptor for the socket
    char c;
    struct sockaddr_in server = {PF_INET, PORT, INADDR_ANY};

    Signal(SIGCHLD, on_sigchld);
    Signal(SIGPIPE, on_sigpipe);

    if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1 ) {
        ERROR_EXIT("socket call failed",1)
    }

    if ( bind( listenfd, ( struct sockaddr *) &server, SIZE ) == -1 ) {
        ERROR_EXIT(" bind call failed",1)
    }

    if (listen(listenfd, LISTEN_QUEUE_SIZE ) == -1) {
        ERROR_EXIT(" listen call failed",1)
    }

    for ( ; ; ) {
        if ( (connectionfd = accept(listenfd, NULL, NULL ) ) == -1 ){
            if (errno == EINTR )
                continue;
            else
                perror(" accept call failed");
        }
        switch ( fork() ) {
        case -1:
            ERROR_EXIT("fork call failed",1)
        case  0:
            while ( recv(connectionfd, &c, 1, 0) > 0 ) {
                c = toupper(c);               // convert c to upeprcase
                send(connectionfd, &c, 1, 0 );// send it back
            }
            close(connectionfd);
```

```
            exit(0);
        default:
            // server code
            close(connectionfd);
        }
    }
}

void on_sigpipe( int sig)
{
    close(connectionfd);
    exit(0);
}

void on_sigchld(int sig)
{
    pid_t   pid;
    int     stat;

    pid  = wait(NULL);
    return;
}

Sigfunc*  Signal(int signo, Sigfunc *func)
{
    struct  sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags  = 0;
    if (SIGALRM != signo ) {
        act.sa_flags |= SA_RESTART;
    }
    if ( sigaction(signo, &act, &oact) < 0 )
        return ( SIG_ERR );
    return ( oact.sa_handler);
}
```

Comments.

- This program uses a user-defined `Signal()` function to encapsulate the logic of registering the signal handlers. Since we have been registering multiple handlers in most of our programs, it would have been a good idea to create this function earlier and put it in a library to reuse. The idea for this is from [Stevens].

- The program could have used ordinary `read()` and `write()` system calls. Instead, as a way to introduce two socket-specific communications primitives, it uses `recv()` and `send()`. `recv()` is one of a set of three socket-reading functions:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);

ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

- The `recvfrom()` and `recvmcg()` functions are the most general. The prototype for `recv()` is the same as that of `read()` except that it has a fourth argument that can be used to set various flags to control how the `recv()` behaves. The flags can be used to turn on non-blocking operation (`MSG_DONTWAIT`), to notify the kernel that the process wants to receive out-of-band data (`MSG_OOB`), or to peek at the data without reading it (`MSG_PEEK`), to name a few. In our program, no flags are used, so the fourth argument is zero, and `recv(s, buf, n, 0)` is identical to `read(s, buf, n)`.

- The `send()` function is also one of a set of three:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

- The `sendto()` and `sendmsg()` functions are the most general. The prototype of send is identical to that of `write()` except for the additional argument. Like `recv()`, the fourth argument of `send()` is a set of flags that can be or-ed together. Some of the flags are the same, such as `MSG_OOB`, which allows the process to send out-of-band data. See the man page for more details. We will return to the use of `sendto()` and `recvfrom()` when we look at a connection-less server.

The code for the client is next.

```
// sockclient1.c
#include "sockdemo1.h"

int main(int argc, char** argv)
{
```

```
    int                 sockfd;
    char                c, rc;
    char                ip_name[256] = "";
    struct sockaddr_in  server;
    struct hostent      *host;

    if ( argc < 2 )
       strcpy(ip_name, DEFAULT_HOST);
    else
       strcpy(ip_name, argv[1]);

    if ( (host = gethostbyname(ip_name)) ==  NULL) {
        ERROR_EXIT("gethostbyname", 1);
    }

    memset(&server, 0, sizeof(server));
    memcpy(&server.sin_addr, SOCKADDR *host->h_addr_list, SIZE);
    server.sin_family = AF_INET;
    server.sin_port   = PORT;

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) == -1 ) {
        ERROR_EXIT("socketcall failed",1)
    }

    if ( connect (sockfd, SOCKADDR &server, sizeof(server)) == -1) {
        ERROR_EXIT("connect call failed",1);
    }

    for ( rc = '\n';;) {
        if (rc == '\n')
            printf("Input a lowercase character\n");
        c = getchar();
        write(sockfd, &c, 1);
        if ( read(sockfd, &rc, 1) > 0 )
            printf("%c", rc);
        else {
            printf("server has died\n");
            close(sockfd);
            exit(1);
        }
    }
}
```

Comments.

- The client does hostname-to-address translation to make it more generic. The user can supply the name of the server on the command line rather than having to remember the IP

address. If the IP address changes, the program still works. The `gethostbyname()` call returns a pointer to a `hostent struct`, given a string that contains a valid hostname.

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
```

- The `hostent struct` is defined in the `<netdb.h>` header file:

```
struct hostent {
    char  *h_name;            /* official name of host */
    char **h_aliases;         /* alias list */
    int    h_addrtype;        /* host address type */
    int    h_length;          /* length of address */
    char **h_addr_list;       /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */
```

- The `h_name` field is the official name of the host. For example, if it is given the name "`eniac`" when running on a host on our network, it is able to resolve the name, and the `h_name` field will be filled in with "`eniac.geo.hunter.cuny.edu`". The `aliases` member is a pointer to a list of strings, each of which is an alias, i.e., another name listed in the hosts database for the same machine. The `h_addr_list` is a a pointer to a list of internet addresses for this host. If the host has just one network interface card, then only `h_addr_list[0]` is defined. Each entry is of type `in_addr`, which is why, in the client code, if can be assigned directly (with a cast) to the `sin_addr` field of the `sock_addr struct`. We do not use the `h_addrtype` or `h_length` fields here.

- The client uses ordinary `read()` and `write()` calls for its I/O operations.

9.7.6 A Connection-Oriented Client Using Multiplexed I/O

Consider the client from the `upcase` example in Section 9.4.4. It reads a line from standard input, writes it into a pipe, and then reads the converted text from a second pipe. In that example, two pipes were needed because a pipe cannot be used as a bi-directional channel. However, we can replace the pair of pipes by a single socket, which can then be used for both sending the raw text to the server and receiving the converted text from it. In addition, by making the socket an Internet-domain socket, the client and server can be on different machines.

If we keep the original design for the client but just replace the pipes by a socket, the client would read the raw text from standard input, write it to the socket, and then read the converted text from the same socket, in a loop of the form

```
while ( true ) {
    get text from standard input;
```

```
        write text to socket;
        read converted text from socket;
        write response on standard output;
    }
```

Since input can be redirected, it can arrive much faster than the responses that it receives from the server, because the server might be a long distance away. The client would spend most of its time blocked on the call to read the socket, even though both the server and the socket itself could handle much larger throughput. The same thing could happen in the interactive case as well if the user enters text very quickly but the round-trip time for the socket is large. In this case the client would be delayed in displaying a prompt to the user on the terminal. Therefore, it makes sense in this client to multiplex the standard input and the socket input using the `select()` call. By using the `select()` call, the client will only block if neither the user nor the server has data to read. As long as text arrives on the standard input stream, it will be forwarded to the server. If text arrives on standard input much faster than the round-trip time, the text will keep being sent to the server, which will process the lines one after the other and send them back in a steady stream.

An analogy will help. Imagine a thirty-person fire brigade trying to put out a fire with a single bucket. The bucket is filled with water and passed from one person to the next to the fire, poured on the fire, and then passed back to the water supply, where this is repeated. Suppose it takes one minute for the round trip and the bucket holds 5 gallons of water. This supplies 5 gallons per minute to the fire. Now suppose there are sixty buckets available. The first bucket is filled and handed to the next person, and the second bucket is filled, and so on, until all sixty buckets are filled. Assuming the people know how to pass the full buckets past the empty buckets and the exchange rate is uniform, although the round-trip time has not changed, there will be sixty buckets in the brigade at each instant, and each second, a full bucket will arrive at the fire. The fire will be supplied 5 gallons per second, or 300 gallons per minute.

This is how using `select()` can increase the throughput in the case that the bottleneck is the length of time it takes for the data to make a round trip from client to server and back. The code follows. It uses the same header file as was used in `sockdemo1`. This client will not accept a file name on the command line; it uses a single command line argument, which is the name of the host on which the server is running.

```
//sockdemo2_client.c
#include "sockdemo1.h"

#define  MAXFD( _x, _y)  ((_x)>(_y)?(_x):(_y))

int main(int argc, char** argv)
{
    int                sockfd;
    char               c, rc;
    char               ip_name[256] = "";
    struct sockaddr_in server;
    struct hostent     *host;
```

```
    fd_set              readset, copy;
    int                 maxfd, n;
    char                recvline[MAXLINE];
    char                sendline[MAXLINE];
    int                 stdin_eof = 0;

    if ( argc < 2 )
       strcpy(ip_name, DEFAULT_HOST);
    else
       strcpy(ip_name, argv[1]);

    if ( (host = gethostbyname(ip_name)) ==  NULL) {
        ERROR_EXIT("gethostbyname", 1);
    }

    memset(&server, 0, sizeof(server));
    memcpy(&server.sin_addr, SOCKADDR *host->h_addr_list, SIZE);
    server.sin_family = AF_INET;
    server.sin_port   = PORT;

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) == -1 ) {
        ERROR_EXIT("socketcall failed",1)
    }

    if ( connect (sockfd, SOCKADDR &server, sizeof(server)) == -1) {
        ERROR_EXIT("connect call failed",1);
    }

    maxfd = MAXFD(fileno(stdin), sockfd) +1;

    while ( 1 ) {
        FD_ZERO(&readset);
        if ( stdin_eof == 0 )
            FD_SET(fileno(stdin), &readset);
        FD_SET(sockfd, &readset);
        if ( select( maxfd, &readset, NULL, NULL, NULL ) > 0 ) {
            if ( FD_ISSET(sockfd, &readset)) {
                if ( ( n = read(sockfd, recvline, MAXLINE-1)) == 0 ) {
                     if (stdin_eof == 1)
                         return;
                     else
                         ERROR_EXIT("Server terminated prematurely.", 1);
                }
                recvline[n] = '\0';
                fputs(recvline, stdout);
            }
```

```
            if ( FD_ISSET(fileno(stdin), &readset)) {
                if ( fgets(sendline, MAXLINE-1, stdin) == NULL ) {
                    stdin_eof = 1;
                    shutdown(sockfd, SHUT_WR);
                    FD_CLR(fileno(stdin), &readset);
                    continue;
                }
                write(sockfd, sendline, strlen(sendline));
            }
        }
    }
}
```

Comments.

- This client sends entire lines, one after the other, to the server. It appends a null character to each line it receives before printing it to standard output, even though in principle all lines received should be null-terminated, since they are identical to the null-terminated lines that it sent to the server, except for conversion of lowercase to uppercase letters in the line.

- The `shutdown(sockfd, SHUT_WR)` system call turns off writing to the socket. When the client detects the end-of-file on the standard input stream, it cannot close the socket completely, because if it did, it would not receive any lines sent to the server but not yet converted to uppercase. On the other hand, it has to send a notification to the server that there is no more input on the socket, so that the server's `read()` on the socket can return. The `shutdown()` accomplishes this; the server's read() returns and the socket stays open until the server closes its end of the socket. Without `shutdown()` there would be no way to achieve this. Its synopsis is:

  ```
  #include <sys/socket.h>

  int shutdown(int s, int how);
  ```

  Here, the integer `how` can be replaced by one of `SHUT_WR`, `SHUT_RD`, or `SHUT_RDWR`.

- Once the client detects the end-of-file, it also sets a flag for itself, `stdin_eof`, which it uses to decide whether to set a bit in the descriptor mask for the standard input. If end-of-file has been detected, it stops setting that bit; otherwise it sets it. In addition, when the `read()` on the socket returns 0 bytes, it uses this flag to distinguish between two cases: whether the server has stopped sending text because there is none left to send, or there was an error on the socket before the end-of-file condition occurred.

The server's main function is displayed below. Because the signal handling code is no different in this example than in `sockdemo1_server`, it is not included.

```
//sockdemo2_server.c
#include "sockdemo1.h"
#include "sys/wait.h"

#define LISTEN_QUEUE_SIZE   5
typedef void    Sigfunc(int);   /* for signal handlers */

Sigfunc*  Signal(int signo, Sigfunc *func);
void on_sigchld( int signo );
void convert(int sockfd);


int main(int argc, char **argv)
{
    int                 listenfd, connfd;
    pid_t               childpid;
    socklen_t           clilen;
    struct sockaddr_in  clientaddr, server_addr;

    if ( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
        ERROR_EXIT("socket call failed",1)
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family      = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port        = PORT;

    if ( bind(listenfd,SOCKADDR &server_addr,sizeof(server_addr)) == -1){
        ERROR_EXIT(" bind call failed",1)
    }
    if (listen(listenfd, LISTEN_QUEUE_SIZE ) == -1) {
        ERROR_EXIT(" listen call failed",1)
    }
    Signal(SIGCHLD, on_sigchld);

    while ( 1 ) {
        clilen = sizeof(clientaddr);
        if ((connfd = accept(listenfd,SOCKADDR &clientaddr,&clilen))<0) {
            if (errno == EINTR)
                continue;
            else
                ERROR_EXIT("accept error",1);
        }
        if ( (childpid = fork()) == 0) {
            close(listenfd);
```

```
            convert(connfd);
            exit(0);
        }
        close(connfd);
      }
}

void convert(int sockfd)
{
    ssize_t      n;
    int          i;
    char          line[MAXLINE];

    while ( 1 ) {
        if ( (n = read(sockfd, line, MAXLINE-1)) == 0)
            return;

    for ( i = 0; i < n; i++ )
        if ( islower(line[i]))
            line[i] = toupper(line[i]);
            write(sockfd, line, n);
    }
}
```

Comments.

- All of the logic is encapsulated in the `convert()` function, which the child executes. `convert()` reads the connected socket until it receives the end-of-file and then it terminates, which causes the child to exit in main.

9.8 Summary

Related processes can use unnamed pipes to exchange data. Unrelated processes running on the same host can use named pipes to exchange data. Unlike unnamed pipes, named pipes are entities in the file system. Both named and unnamed pipes are guaranteed by the kernel to be read and written atomically provide that the amount of data written is at most `PIPE_BUF` bytes. Sockets are a way for processes on the same host or different hosts to exchange data. Unlike pipes, sockets are bi-directional.

Servers can be iterative or concurrent. A concurrent server creates a child process to handle every distinct client. An iterative server handles each client within a single process, sharing its time among them. Concurrent servers provide more reliable response time to the clients.

When a process has to handle I/O from multiple file descriptors, it can multiplex the I/O by means of the `select()` system call. This is one alternative of many, but it provides a relatively simple solution. Other alternatives include asynchronous I/O and the `poll()` call.

Other methods of interprocess communication not discussed in this chapter are shared memory, memory-mapped files, and message queues.

References

[Haviland et al] Haviland, Keith, Dina Gray, & Ben Salama. Unix System Programming. Addison-Wesley. 1999.

[Stevens] Stevens, Richard W.. UNIX Network Programming, Volume 1. Prentice Hall. 1998.

Alphabetical Index

Chapter 10 Threads

Concepts Covered

Processes, threads,
multi-threading paradigms,
Pthreads, NPTL,
thread properties,
thread cancellation, detached threads,
mutexes, condition variables,
barrier synchronization, reduction algorithm
producer-consumer problem,
reader/writer locks,
thread scheduling, deadlock, starvation

10.1 Introduction

We saw in Chapter 8 that a process is associated with a set of resources including its memory segments (text, stack, initialized data, uninitialized data), environment variables and command line arguments, and various properties and data that are contained in kernel resources such as the process and user structures. A partial list of the kinds of information contained in these structures includes things such as the process's

- IDs such as process ID, process group ID, user ID, and group ID
- Hardware state
- Memory mappings, such as where process segments are located
- Flags such as set-uid, set-gid
- File descriptors
- Signal masks and dispositions
- Resource limits
- Inter-process communication tools such as message queues, pipes, semaphores, or shared memory.

A process is a fairly "heavy" object in the sense that when a process is created, all of these resources must be created for it. The `fork()` system call duplicates some, but not all, of the calling process's resources. Some of them are shared between the parent and child process.

Processes by default are limited in what they can share with each other because they do not share their memory spaces. Thus, for example, they do not in general share variables and other objects that they create in memory. Most operating systems provide an API for sharing memory though. For example, in Linux 2.4 and later, and glibc 2.2 and later, POSIX shared memory is available so that unrelated processes can communicate through shared memory objects. Solaris also supported shared memory, both natively and with support for the later POSIX standard. In addition, processes can share files and messages, and they can send each other signals to synchronize.

The biggest drawback to using processes as a means of multi-tasking is their consumption of system resources. This was the motivation for the invention of threads.

10.2 Thread Concepts

A *thread* is a flow of control (think sequence of instructions) that can be independently scheduled by the kernel. A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. When a program has multiple threads of control, more than one thing at a time can be done within a single process, with each thread handling a separate task. Some of the advantages of this are that

- Code to handle asynchronous events can be executed by a separate thread. Each thread can then handle its event using a synchronous programming model.
- Whereas multiple processes have to use mechanisms provided by the kernel to share memory and file descriptors, threads automatically have access to the same memory address space, which is faster and simpler.
- Even on a single processor machine, performance can be improved by putting calls to system functions with expected long waits in separate threads. This way, just the calling thread blocks, and not the whole process.
- Response time of interactive programs can be improved by splitting off threads to handle user input and output.

Threads share certain resources with the parent process and each other, and maintain private copies of other resources. The most important resources shared by the threads are the program's text, i.e., its executable code, and its global and heap memory. This implies that threads can communicate through the program's global variables, but it also implies that they have to synchronize their access to these shared resources. To make threads independently schedulable, at the very least they they must have their own stack and register values.

In UNIX, POSIX requires that each thread will have its own distinct

- thread ID
- stack and an alternate stack
- stack pointer and registers
- signal mask
- errno value
- scheduling properties
- thread specific data.

On the other hand, in addition to the text and data segments of the process, UNIX threads share

- file descriptors
- environment variables
- process ID

- parent process ID
- process group ID and session ID
- controlling terminal
- user and group IDs
- open file descriptors
- record locks
- signal dispositions
- file mode creation mask (the umask)
- current directory and root directory
- interval timers and POSIX timers
- nice value
- resource limits
- measurements of the consumption of CPU time and resources

To summarize, a thread

- is a single flow of control within a process and uses the process resources;
- duplicates only the resources it needs to be independently schedulable;
- can share the process resources with other threads within the process; and
- terminates if the parent process is terminated;

10.3 Programming Using Threads

Threads are suitable for certain types of parallel programming. In general, in order for a program to take advantage of multi-threading, it must be able to be organized into discrete, independent tasks which can execute concurrently. The first consideration when considering using multiple threads is how to decompose the program into such discrete, concurrent tasks. There are other considerations though. Among these are

- How can the load be balanced among the threads so that they no one thread becomes a bottleneck?
- How will threads communicate and synchronize to avoid race conditions?
- What type of data dependencies exist in the problem and how will these affect thread design?
- What data will be shared and what data will be private to the threads?

- How will I/O be handled? Will each thread perform its own I/O for example?

Each of these considerations is important, and to some extent each arises in most programming problems. Determining data dependencies, deciding which data should be shared and which should be private, and determining how to synchronize access to shared data are very critical aspects to the correctness of a solution. Load balancing and the handling of I/O usually affect performance but not correctness.

Knowing how to use a thread library is just the technical part of using threads. The much harder part is knowing how to write a parallel program. These notes are not intended to assist you in that task. Their purpose is just to provide the technical background, with pointers here and there. However, before continuing, we present a few common paradigms for organizing multi-threaded programs.

Thread Pool, or Boss/Worker Paradigm

In this approach, there is a single *boss* thread that dispatches threads to perform work. These threads are part of a worker thread pool which is usually pre-allocated before the boss begins dispatching threads.

Peer or WorkCrew Paradigm

In the WorkCrew model, tasks are assigned to a finite set of worker threads. Each worker can enqueue subtasks for concurrent evaluation by other workers as they become idle. The Peer model is similar to the boss/worker model except that once the worker pool has been created, the boss becomes the another thread in the thread pool, and is thus, a peer to the other threads.

Pipeline

Similar to how pipelining works in a processor, each thread is part of a long chain in a processing factory. Each thread works on data processed by the previous thread and hands it off to the next thread. You must be careful to equally distribute work and take extra steps to ensure non-blocking behavior in this thread model or you could experience pipeline "stalls."

10.4 Overview of the Pthread Library

In 1995 the Open Group defined a standard interface for UNIX threads (IEEE POSIX 1003.1c) which they named *Pthreads* (P for POSIX). This standard was supported on multiple platforms, including Solaris, Mac OS, FreeBSD, OpenBSD, and Linux. In 2005, a new implementation of the interface was developed by Ulrich Drepper and Ingo Molnar of Red Hat, Inc. called the *Native POSIX Thread Library* (*NPTL*), which was much faster than the original library, and has since replaced that library. The Open Group further revised the standard in 2008. We will limit our study of threads to the NPTL implementation of Pthreads.

The Pthreads library provides a very large number of primitives for the management and use of threads; there are 93 different functions defined in the 2008 POSIX standard. Some thread functions

are analogous to those of processes. The following table compares the basic process primitives to analogous Pthread primitives.

Process Primitive	**Thread Primitive**	**Description**
`fork()`	`pthread_create()`	Create a new flow of control with a function to execute
`exit()`	`pthread_exit()`	Exit from the calling flow of control
`waitpid()`	`pthread_join()`	Wait for a specific flow of control to exit and collect its status
`getpid()`	`pthread_self()`	Get the id of the calling flow of control
`abort()`	`pthread_cancel()`	Request abnormal termination of the calling flow of control

The Pthreads API can be categorized roughly by the following four groups

Thread management: This group contains functions that work directly on threads, such as creating, detaching, joining, and so on. This group also contains functions to set and query thread attributes.

Mutexes: This group contains functions for handling critical sections using mutual exclusion. Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables: This group contains functions that address communications between threads that share a mutex based upon programmer-specified conditions. These include functions to create, destroy, wait and signal based upon specified variable values, as well as functions to set and query condition variable attributes.

Synchronization: This group contains functions that manage read/write locks and barriers.

We will visit these groups in the order they are listed here, not covering any in great depth, but enough depth to write fairly robust programs.

10.5 Thread Management

10.5.1 Creating Threads

We will start with the `pthread_create()` function. The prototype is

```
int pthread_create  ( pthread_t            *thread,
                      const pthread_attr_t *attr,
                      void  *(*start_routine)(void *),
                      void                 *arg);
```

This function starts a new thread with thread ID `*thread` as part of the calling process. On successful creation of the new thread, `thread` contains its thread ID. Unlike `fork()`, this call passes the address of a function, `start_routine()`, to be executed by the new thread. This "start" function has exactly one argument, of type `void*`, and returns a `void*`. The fourth argument, `arg`, is the argument that will be passed to `start_routine()` in the thread.

The second argument is a pointer to a `pthread_attr_t` structure. This structure can be used to define attributes of the new thread. These attributes include properties such as its stack size, scheduling policy, and *joinability* (to be discussed below). If the program does not specifically set values for its members, default values are used instead. We will examine thread properties in more detail later.

Because `start_routine()` has just a single argument, if the function needs access to more than a simple variable, the program should declare a structure with all state that needs to be accessed within the thread, and pass a pointer to that structure. For example, if a set of threads is accessing a shared array and each thread will process a contiguous portion of that array, you might want to define a structure such as

```
typedef struct _task_data
{
    int first;    /* index of first element for task */
    int last;     /* index of last element for task */
    int *array;   /* pointer to start of array */
    int task_id;  /* id of thread */
} task_data;
```

and start each thread with the values of `first`, `last`, and `task_id` initialized. The array pointer may or may not be needed; if the array is a global variable, the threads will have access to it. If it is declared in the main program, then its address can be part of the structure. Suppose that the array is declared as a static local variable named `data_array` in the main program. Then a code fragment to initialize the thread data and create the threads could be

```
task_data  thread_data[NUM_THREADS];
for ( t = 0 ; t < NUM_THREADS; t++) {
    thread_data[t].first     = t*size;
    thread_data[t].last      = (t+1)*size -1;
    if ( thread_data[t].last > ARRAY_SIZE -1 )
        thread_data[t].last  = ARRAY_SIZE - 1;
    thread_data[t].array     = &data_array[0];
    thread_data[t].task_id   = t;

    if ( 0 != (rc = pthread_create(&threads[t], NULL, process_array,
                            (void *) &thread_data[t]) ) )  {
        printf("ERROR; %d return code from pthread_create()\n", rc);
        exit(-1);
    }
}
```

This would create `NUM_THREADS` many threads, each executing `process_array`(), each with its own structure containing parameters of its execution.

10.5.1.1 Design Decision Regarding Shared Data

The advantage of declaring the data array as a local variable in the main program is that it makes it easier to analyze and maintain the code when there are fewer global variables and side effects. Programs with functions that modify global variables are harder to analyze. On the other hand, making it a local in main and then having to add a pointer to that array in the thread data structure passed to each thread increases thread storage requirements and slows down the program. Each thread has an extra pointer in its stack when it executes, and each reference to the array requires two dereferences instead of one. Which is preferable? It depends what the overall project requirements are. If speed and memory are a concern, use a global and use good practices in documenting and accessing it. If not, use the static local.

10.5.2 Thread Identification

A thread can get its thread ID by calling `pthread_self()`, whose prototype is

```
pthread_t pthread_self(void);
```

This is the analog to `getpid()` for processes. This function is the only way that the thread can get its ID, because it is not provided to it by the creation call. It is entirely analogous to `fork()` in this respect.

A thread can check whether two thread IDs are equal by calling

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

This returns a non-zero if the two thread IDs are equal and zero if they are not.

10.5.3 Thread Termination

A thread can terminate itself by calling `pthread_exit()`:

```
void pthread_exit(void *retval);
```

This function kills the thread. The `pthread_exit()` function never returns. Analogous to the way that `exit()` returns a value to `wait()`, the return value may be examined from another thread in the same process if it calls `pthread_join()`[1]. The value pointed to by `retval` should not be located on the calling thread's stack, since the contents of that stack are undefined after the thread terminates. It can be a global variable or allocated on the heap. Therefore, if you want to use a locally-scoped variable for the return value, declare it as static within the thread.

It is a good idea for the main program to terminate itself by calling `pthread_exit()`, because if it has not waited for spawned threads and they are still running, if it calls `exit()`, they will be killed. If these threads should not be terminated, then calling `pthread_exit()` from `main()` will ensure that they continue to execute.

[1] Provided that the terminating thread is joinable.

10.5.4 Thread Joining and Joinability

When a thread is created, one of the attributes defined for it is whether it is *joinable* or *detached*. By default, created threads are joinable. If a thread is joinable, another thread can wait for its termination using the function `pthread_join()`. Only threads that are created as joinable can be joined.

Joining is a way for one thread to wait for another thread to terminate, in much the same way that the `wait()` system calls lets a process wait for a child process. When a parent process creates a thread, it may need to know when that thread has terminated before it can perform some task. Joining a thread, like waiting for a process, is a way to synchronize the performance of tasks.

However, joining is different from waiting in one respect: the thread that calls `pthread_join()` must specify the thread ID of the thread for which it waits, making it more like `waitpid()`. The prototype is

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The `pthread_join()` function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. If the target thread already terminated, `pthread_join()` returns successfully.

If `value_ptr` is not `NULL`, then the value passed to `pthread_exit()` by the terminating thread will be available in the location referenced by `value_ptr`, provided `pthread_join()` succeeds.

Some things that cause problems include:

- Multiple simultaneous calls to `pthread_join()` specifying the same target thread have undefined results.
- The behavior is undefined if the value specified by the thread argument to `pthread_join()` does not refer to a joinable thread.
- The behavior is undefined if the value specified by the thread argument to `pthread_join()` refers to the calling thread.
- Failing to join with a thread that is joinable produces a "zombie thread". Each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

The following listing shows a simple example that creates a single thread and waits for it using `pthread_join()`, collecting and printing its exit status.

Listing 10.1: Simple example of thread creation with join

```
int exitval;

void* hello_world( void * world)
{
    printf("Hello World from %s.\n", (char*) world);
    exitval = 2;
    pthread_exit((void*) exitval) ;
}
```

```
int main( int argc, char *argv[])
{
    pthread_t  child_thread;
    void  *status;
    char  *planet  = "Pluto";

    if ( 0 != pthread_create(&child_thread, NULL,
                             hello_world,( void*) planet) ) {
        perror("pthread_create");
        exit(-1);
    }
    pthread_join(child_thread, (void**) (&status));
    printf("Child exited with status %ld\n", (long) status);
    return 0;
}
```

Any thread in a process can join with any other thread. They are peers in this sense. The only obstacle is that to join a thread, it needs its thread ID.

10.5.5 Detached Threads

Because `pthread_join()` must be able to retrieve the status and thread ID of a terminated thread, this information must be stored someplace. In many Pthread implementations, it is stored in a structure that we will call a *Thread Control Block* (TCB). In these implementations, the entire TCB is kept around after the thread terminates, just because it is easier to do this. Therefore, until a thread has been joined, this TCB exists and uses memory. Failing to join a joinable thread turns these TCBs into waste memory.

Sometimes threads are created that do not need to be joined. Consider a process that spawns a thread for the sole purpose of writing output to a file. The process does not need to wait for this thread. When a thread is created that does not need to be joined, it can be created as a *detached thread*. When a detached thread terminates, no resources are saved; the system cleans up all resources related to the thread.

A thread can be created in a detached state, or it can be detached after it already exists. To create a thread in a detached state, you can use the `pthread_attr_setdetachstate()` function to modify the `pthread_attr_t` structure prior to creating the thread, as in:

```
pthread_t      tid;  /* thread ID        */
pthread_attr_t attr; /* thread attribute */

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

/* now create the thread */
pthread_create(&tid, &attr, start_routine, arg);
```

An existing thread can be detached using `pthread_detach()`:

```
int pthread_detach(pthread_t thread);
```

The function `pthread_detach()` can be called from any thread, in particular from within the thread itself! It would need to get its thread ID using `pthread_self()`, as in

```
pthread_detach(pthread_self());
```

Once a thread is detached, it cannot become joinable. It is an irreversible decision. The following listing shows how a main program can exit, using `pthread_exit()` to allow its detached child to run and produce output, even after `main()` has ended. The call to `usleep()` gives a bit of a delay to simulate computationally demanding output being produced by the child.

Listing 10.2: Example of detached child

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void *thread_routine(void * arg)
{
    int    i;
    int    bufsize = strlen(arg);
    int    fd = 1;

    printf("Child is running...\n");
    for (i = 0; i < bufsize; i++) {
              usleep(500000);
              write(fd, arg+i, 1);
    }
    printf("\nChild is now exiting.\n");
    return(NULL);
}

int main(int argc, char* argv[])
{
    char * buf = "abcdefghijklmnopqrstuvwxyz";
    pthread_t thread;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    if (pthread_create(&thread, NULL, thread_routine, (void *)(buf))) {
              fprintf(stderr, "error creating a new thread \n");
              exit(1);
    }

    printf("Main is now exiting.\n");
    pthread_exit(NULL);
}
```

10.5.6 Thread Cancellation

Threads can be *canceled* as well. Cancellation is roughly like killing a thread. When a thread is canceled, its resources are cleaned up and it is terminated. A thread can request that another thread be canceled by calling `pthread_cancel()`, the prototype for which is

```
int pthread_cancel(pthread_t thread);
```

This is just a request; it is not necessarily honored. When this is called, a cancellation request is sent to the thread given as the argument. Whether or not that thread is canceled depends upon the thread's cancelability state and type. A thread can enable or disable cancelability, and it can also specify whether its cancelability type is *asynchronous* or *deferred*. If a thread's cancelability type is asynchronous, then it will be canceled immediately upon receiving a cancellation request, assuming it has enabled its cancelability. On the other hand, if its cancelability is deferred, then cancellation requests are deferred until the thread enters a *cancellation point*. Certain functions are cancellation points. To be precise, if a thread is cancelable, and its type is deferred, and a cancellation request is pending for it, then if it calls a function that is a cancellation point, it will be terminated immediately. The list of cancellation point functions required by POSIX can be found on the man page for pthreads in section 7.

A thread's cancelability state is enabled by default and can be set by calling `pthread_setcancelstate()`:

```
int pthread_setcancelstate(int state, int *oldstate);
```

The two values are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`. The new state is passed as the first argument and a pointer to an integer to store the old state, or `NULL`, is the second argument. If a thread disables cancellation, then a cancellation request remains queued until it enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

A thread's cancellation type, which is deferred by default, can be set with `pthread_setcanceltype()` :

```
int pthread_setcanceltype(int type, int *oldtype);
```

To set the type to asynchronous, pass `PTHREAD_CANCEL_ASYNCHRONOUS` in the first argument. To make it deferred, pass `PTHREAD_CANCEL_DEFERRED`.

10.5.7 Thread Properties

10.5.7.1 Stack Size

The POSIX standard does not dictate the size of a thread's stack, which can vary from one implementation to another. Furthermore, with today's demanding problems, exceeding the default stack limit is not so unusual, and if it happens, the program will terminate, possibly with corrupted data.

Safe and portable programs do not depend upon the default stack limit, but instead, explicitly allocate enough stack for each thread by using the `pthread_attr_setstacksize()` function, whose prototype is

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

The first argument is the address of the threads attribute structure and the second is the size that you want to set for the stack. This function will fail if the attribute structure does not exist, or if the stack size is smaller than the allowed minimum (`PTHREAD_STACK_MIN`) or larger than the maximum allowed. See the man page for further caveats about its use.

To get the stack's current size, use

```
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
```

This retrieves the current size of the stack. It will fail of course if `attr` does not reference an existing structure.

The problem trying to use this function is that it must be passed the attributes structure of the thread. There is no POSIX function to retrieve the attribute structure of the calling thread, but there is a GNU extension, `pthread_getattr_np()`. If this extension is not used, the best that the calling thread can do is to get a copy of the attribute structure with which it was created, which may have different values than the one it it currently using. The following listing is of a program that prints the default stack size then sets the new stack size based on a command line argument, and from within the thread, displays the actual stack size it is using, using the GNU `pthread_getattr_np()` function. *To save space, some error checking has been removed.*

Listing 10.3: Setting a new stack size (with missing error checking)

```
#define _GNU_SOURCE     /* To get pthread_getattr_np() declaration */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

void *thread_start(void *arg)
{
   size_t          stack_size;
   pthread_attr_t  gattr;

   pthread_getattr_np       ( pthread_self(),  &gattr);
   pthread_attr_getstacksize( &gattr,          &stack_size);
   printf("Actual stack size is %ld\n", stack_size);
   pthread_exit(0);
}

int main(int argc, char *argv[])
{
    pthread_t       thr;
    pthread_attr_t  attr;
    int             retval;
    size_t          new_stack_size, stack_size;
    void            *sp;

    if ( argc < 2 ) {
        printf("usage: %s stacksize\n", argv[0] );
```

```
        exit(1);
    }

    new_stack_size = strtoul(argv[1], NULL, 0);

    retval = pthread_attr_init(&attr);
    if (retval) {
        exit(1);
    }
    pthread_attr_getstacksize (&attr, &stack_size);
    printf("Default stack size = %ld\n", stack_size);
    printf("New stack size will be %ld\n", new_stack_size);

    retval = pthread_attr_setstacksize(&attr, new_stack_size);
    if ( retval ) {
        exit(1);
    }

    retval = pthread_create(&thr, &attr, &thread_start, NULL);
    if ( retval ) {
        exit(1);
    }

    pthread_join(thr, NULL);
    return(0);
}
```

10.6 Mutexes

10.6.1 Introduction

When multiple threads share the same memory, the programmer must ensure that each thread sees a consistent view of its data. If each thread uses variables that no other threads read or modify, then there are no consistency problems with those variables. Similarly, if a variable is read-only, there is no consistency problem if multiple threads read its value at the same time. The problem occurs when one thread can modify a variable that other threads can read or modify. In this case the threads must be synchronized with respect to the shared variable. The segment of code in which this shared variable is accessed within a thread, whether for a read or a write, is called a *critical section*.

A simple example of a critical section occurs when each thread in a group of threads needs to increment some shared counter, after which it does some work that depends on the value of that counter. The main program would initialize the counter to zero, after which each thread would increment the counter and use it to access the array element indexed by that value. The following code typifies this scenario.

```
void * work_on_ticker( void * counter)
{
    int i;
    int  *ticker = (int*) counter;
```

```
    for ( i = 0; i < NUM_UPDATES; i++ ) {
        *ticker = *ticker + 1;
        /* use the ticker to do stuff here with A[*ticker] */
    }
    pthread_exit( NULL );
}
```

Without any synchronization to force the increment of `*ticker` to be executed in mutual exclusion, some threads may overwrite other threads' array data, and some array elements may remain unprocessed because the ticker skipped over them. You will probably not see this effect if this code is executed on a single-processor machine, as the threads will be time-sliced on the processor, and the likelihood of their being sliced in the middle of the update to the ticker is very small, but if you run this on a multi-processor machine, you will almost certainly see the effect.

A *mutex* is one of the provisions of Pthreads for providing mutual exclusive access to critical sections. A *mutex* is like a software version of lock. Its name derives from "mutual exclusion" because a mutex can only be held, or *owned*, by one thread at a time. Like a binary semaphore, the typical use of a mutex is to surround a critical section of code with a call to lock and then to unlock the mutex, as in

```
pthread_mutex_lock  ( &mutex );
/* critical section here */
pthread_mutex_unlock( &mutex );
```

Mutexes are a low-level form of critical section protection, providing the most rudimentary features. They were intended as the building blocks of higher-level synchronization methods. Nonetheless, they can be used in many cases to solve critical section problems. In the remainder of this section, we describe the fundamentals of using mutexes.

10.6.2 Creating and Initializing Mutexes

A mutex is a variable of type `pthread_mutex_t`. It must be initialized before it can be used. There are two ways to initialize a mutex:

1. Statically, when it is declared, using the `PTHREAD_MUTEX_INITIALIZER` macro, as in

   ```
   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
   ```

2. Dynamically, with the `pthread_mutex_init()` routine:

   ```
   int pthread_mutex_init(pthread_mutex_t *mutex,  pthread_mutexattr_t *attr);
   ```

   This function is given a pointer to a mutex and to a *mutex attribute structure*, and initializes the mutex to have the properties of that structure. If one is willing to accept the default mutex attributes, the `attr` argument may be `NULL`.

In both cases, the mutex is initially unlocked. The call

```
pthread_mutex_init(&mutex, NULL);
```

is equivalent to the static method except that no error-checking is done.

10.6.3 Locking a Mutex

To lock a mutex, one uses one of the functions

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

We will begin with `pthread_mutex_lock()`. The semantics of this function are a bit complex, in part because there are different types of mutexes. Here we describe the semantics of *normal* mutexes, which are the default type, `PTHREAD_MUTEX_NORMAL`.

If the mutex is not locked, the call returns with the mutex object referenced by mutex in the locked state with the calling thread as its *owner*. The return value will be 0. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked. If a thread tries to lock a mutex that it has already locked, it causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results. We will discuss the other types of mutexes later.

In short, if several threads try to lock a mutex only one thread will be successful. The other threads will be in a blocked state until the mutex is unlocked by its owner.

If a signal is delivered to a thread that is blocked on a mutex, when the thread returns from the signal handler, it resumes waiting for the mutex as if it had not been interrupted.

The `pthread_mutex_trylock()` function behaves the same as the `pthread_mutex_lock()` function except that it never blocks the calling thread. Specifically, if the mutex is unlocked, the calling thread acquires it and the function returns a 0, and if the mutex is already locked by any thread, the function returns the error value `EBUSY`.

10.6.4 Unlocking a Mutex

The call to unlock a mutex is

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The `pthread_mutex_unlock()` function will unlock a mutex if it is called by the owning thread. If a thread that does not own the mutex calls this function, it is an error. It is also an error to call this function if the mutex is not locked. If there are threads blocked on the mutex object referenced by mutex when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy determines which thread next acquires the mutex. If the mutex is a normal mutex that used the default initialization, there is no specific thread scheduling policy, and the underlying kernel scheduler makes the decision. The behavior of this function for non-normal mutexes is different.

10.6.5 Destroying a Mutex

When a mutex is no longer needed, it should be destroyed using

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The `pthread_mutex_destroy()` function destroys the mutex object referenced by mutex; the mutex object becomes uninitialized. The results of referencing the mutex object after it has been destroyed are undefined. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`.

10.6.6 Examples Using a Normal Mutex

Two examples will show how threads can use mutexes to protect their updates to a shared, global variable. The first example will demonstrate how multiple threads can increment a shared counter that serves as an index into a global array, so that no two threads access the same array element. Each thread will then modify that array element. In the second example, the update to the shared variable is on the back-end of the problem. Each thread is given an equal-size segment of two arrays, computes a function of this pair of segments, and adds the value of that function to a shared, global accumulator.

Example 1

Suppose that we want a function which, when given an integer `N` and an array `roots` of size `N`, stores the square roots of the first `N` non-negative integers into `roots`. A sequential version of this function would execute a loop of the form

```
for ( i = 0; i < N; i++ )
    roots[i] = sqrt(i);
```

To make this program run faster when there are multiple processors available, we distribute the work among multiple threads. Let P be the number of threads that will jointly solve this problem. Each thread will compute the square roots of a set of `N`/P integers. These integers are not necessarily consecutive. The idea is that each thread concurrently iterates a loop N times, incrementing a shared, global counter mutually exclusively in each iteration. In each iteration, the thread computes the square root of the current counter value and stores it in an array of roots at the position indexed by the counter value.

The program is in Listing 10.4. All of the multi-threading is opaque to the main program because it is encapsulated in a function. This way it can be ported easily to a different application.

To simplify the program, the array size and number of threads are hard-coded as macros in the program. This is easily changed.

Listing 10.4: A multi-threaded program to compute the first N square roots.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <pthread.h>
#include <errno.h>
#include <math.h>

#define  NUM_THREADS      20                    /* Number of threads */
#define  NUMS_PER_THREAD  50                    /* Number of roots per thread */
#define  SIZE   (NUM_THREADS*NUMS_PER_THREAD)   /* Total roots to compute */

/* Declare a structure to pass multiple variables to the threads in the
   pthread_create() function and for the thread routine to access in its single
   argument.
*/
typedef struct  _thread_data
{
    int     count;           /* shared counter, incremented by eacxh thread */
    int     size;            /* length of the roots array */
    int     nums_per_thread; /* number of roots computed by each thread */
    double* roots;           /* pointer to the roots array */

} thread_data;

pthread_mutex_t update_mutex;  /* Declare a global mutex */


/*****************************************************************************
                          Thread and Helper Functions
*****************************************************************************/

/** handle_error(num,mssge)
 *  A convenient error handling function
 *  Prints to standard error the system message associated with errno num
 *  as well as a custom message, and then exits the program with EXIT_FAILURE
 */
void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/** calc_square_roots()
 *  A thread routine that calculates the square roots of N  integers
 *  and stores them in an array. The integers are not necessarily consecutive;
 *  as it depends how the threads are scheduled.
 *  @param  [out] double data->roots[] is the array in which to store the roots
 *  @param  [inout] int data->count is the first integer whose root should be
 *                                  calculated
 *  This  increments data->count N times.
 *
 *  Loops to waste time a bit so that the threads may be scheduled out of order.
 */
void * calc_square_roots( void * data)
{
```

```
    int  i, j;
    int  temp;
    int  size;
    int  nums_to_compute;
    thread_data *t_data = (thread_data*) data;

    size            = t_data->size;
    nums_to_compute = t_data->nums_per_thread;

    for ( i = 0; i < nums_to_compute; i++ ) {
        pthread_mutex_lock (&update_mutex);   /* lock mutex   */
        temp = t_data->count;
        t_data->count = temp + 1;
        pthread_mutex_unlock (&update_mutex); /* unlock mutex */

        /* updating the array can be done outside of the CS since temp is
           a local variable to the thread. */
        t_data->roots[temp] = sqrt(temp);

        /* idle loop */
        for ( j = 0; j < 1000; j++ )
            ;
    }
    pthread_exit( NULL );
}

/** compute_roots()
 *  computes the square roots of the first num_threads*roots_per_thread many
 *  integers. It hides the fact that it uses multiple threads to do this.
 */
void compute_roots( double sqrts[],  int size, int num_threads )
{
    pthread_t    threads[num_threads];
    int          t;
    int          retval;
    static thread_data  t_data;

    t_data.count = 0;
    t_data.size  = size;
    t_data.nums_per_thread = size / num_threads;
    t_data.roots = &sqrts[0];

    /* Initialize the mutex */
    pthread_mutex_init(&update_mutex, NULL);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        retval = pthread_create(&threads[t], NULL, calc_square_roots,
                                (void *) &t_data);
        if ( retval )
            handle_error( retval, "pthread_create");
    }

    /* Join all threads and then print sum */
```

```
    for ( t = 0 ; t < num_threads; t++)
        pthread_join(threads[t], (void**) NULL);
}

/*****************************************************************************
                                    Main Program
*****************************************************************************/

int main( int argc, char *argv[])
{
    int     t;
    double roots[SIZE];

    memset((void*) &roots[0], 0, SIZE * sizeof(double));
    compute_roots(roots, SIZE, NUM_THREADS );

    for ( t = 0 ; t < SIZE; t++)
        printf("Square root of %5d is %6.3f\n", t, roots[t]);
    return 0;

}
```

A slightly different approach to this program is to allow each thread to compute as many roots as it can, as if the threads were in a race with each other. If the threads were scheduled on asymmetric processors, some being much faster than others, or if some threads had faster access to memory than others, so that they could do more work per unit time, then it would be advantageous to let these threads do more, rather than limiting them to a fixed number of roots to compute. This is the basis for the variation of `calc_square_roots()` from Listing 10.4 found in Listing 10.5.

The function in Listing 10.5 lets each thread iterate from 0 to `size` but it checks in each iteration whether the value of the counter has exceeded the array size, and if it has, that thread terminates. It has an extra feature that is used by the main program and requires a bit of extra code outside fo the function – it stores the id of the thread that computed the root in a global array that can be printed to see how uniformly the work was distributed.

Listing 10.5: A "greedy" thread function.

```
/*
   This function also stores the id of the thread that computed each root in a
   global array so that the main program can print these results. If it did not
   do this, there would be no need for the lines marked with /*****.

*/
void * calc_square_roots( void * data)
{
    int  i, j;
    int  temp;             /* local copy of counter */
    int  size;             /* local copy of size of roots array */
    int  nums_to_compute; /* local copy of number of roots to compute */
    thread_data *t_data = (thread_data*) data;
```

```
    int  my_id;           /****** unique id for this thread */

    /* Copy to local copies for faster access */
    size            = t_data->size;
    nums_to_compute = t_data->nums_per_thread;

    /* Each thread gets a unique thread_id by locking this mutex, capturing the
       current value of tid, assigning it to its own local variable and then
       incrementing it.
    */
    pthread_mutex_lock (&id_mutex);    /****** lock mutex   */
    my_id = tid;                       /****** copy current tid to local my_id */
    tid++;                             /****** increment tid for next thread */
    pthread_mutex_unlock (&id_mutex); /****** unlock mutex */

    i = 0;
    while ( i < size ) {
        pthread_mutex_lock (&update_mutex);    /* lock mutex   */
        temp = t_data->count;
        t_data->count = temp + 1;
        pthread_mutex_unlock (&update_mutex); /* unlock mutex */

        /* Check if the counter exceeds the roots array size */
        if ( temp >= size )
            break;

        /* updating the arrays can be done outside of the CS since temp and
           my_id are local variables to the thread. */
        t_data->roots[temp] = sqrt(temp);

        /* Store the id of the thread that just computed this root. */
        computed_by[temp] =  my_id;    /******  store the id */

        /* idle loop */
        for ( j = 0; j < 1000; j++ )
            ;
        i++;
    }
    pthread_exit( NULL );
}
```

Example 2

The second example, in Listing 10.6, computes the inner product of two vectors V and W by partitioning V and W into subvectors of equal sizes and giving the subproblems to separate threads. Assume for simplicity that V and W are each of length N and that the number of threads, P, divides N without remainder and let $s = N/P$. The actual code does not assume anything about N and

P. The main program creates P threads, with ids $0, 1, 2, ... P-1$. The thread with id k computes the inner product of $V[k \cdot s \cdots (k+1) \cdot s - 1]$ and $W[k \cdot s \cdots (k+1) \cdot s - 1]$ and stores the result in a temporary variable, temp_sum. It then locks a mutex and adds this partial sum to the global variable sum and unlocks the mutex afterward.

This example uses the technique of declaring the vectors and the sum as static locals in the main program.

Listing 10.6: Mutex example: Computing the inner product of two vectors.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libintl.h>
#include <locale.h>
#include <math.h>
#include <errno.h>

#define  NUM_THREADS     20

typedef struct _task_data
{
   int         first;
   int         last;
   double      *a;
   double      *b;
   double      *sum;
} task_data;


pthread_mutex_t mutexsum; /* Declare the mutex globally */

/*****************************************************************************
                          Thread and Helper Functions
*****************************************************************************/
void usage(char *s)
{
        char *p = strrchr(s, '/');
        fprintf(stderr,
                "usage: %s length datafile1 datafile2  \n", p ? p + 1 : s);
}

void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/**
   This function computes the inner product of the sub-vectors
   thread_data->a[first..last] and thread_data->b[first..last],
   adding that sum to thread_data->sum within the critical section
   protected by the shared mutex.
```

```
*/
void*  inner_product( void  *thread_data )
{
    task_data *t_data;
    int        k;
    double     temp_sum = 0;

    t_data    = (task_data*) thread_data;

    for ( k = t_data->first; k <= t_data->last; k++ )
        temp_sum += t_data->a[k] * t_data->b[k];

    pthread_mutex_lock (&mutexsum);
    *(t_data->sum) += temp_sum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
}

/*****************************************************************************
                                    Main Program
*****************************************************************************/

int main( int argc, char *argv[])
{
    static double  *a_vector;
    static double  *b_vector;
    FILE           *fp;
    float          x;
    int            num_threads = NUM_THREADS;
    int            length;
    int            segment_size;
    static double  total;
    int            k;
    int            retval;
    int            t;
    pthread_t      *threads;
    task_data      *thread_data;
    pthread_attr_t attr;


    if ( argc < 4 ) { /* Check usage */
        usage(argv[0]);
        exit(1);
    }

    /* Get command line args, no input validation here */
    length      = atoi(argv[1]);
    a_vector    = calloc( length, sizeof(double));
    b_vector    = calloc( length, sizeof(double));

    /* Zero the two vectors */
    memset(a_vector, 0, length*sizeof(double));
    memset(b_vector, 0, length*sizeof(double));
```

```
/* Open the first file, do check for failure and read the numbers
   from the file. Assume that it is in proper format
*/
if ( NULL == (fp = fopen(argv[2], "r")) )
    handle_error(errno, "fopen");
k = 0;
while (( fscanf(fp, " %f ", &x) > 0 ) && (k < length) )
    a_vector[k++] = x;
fclose(fp);

/* Open the second file, do check for failure and read the numbers
   from the file. Assume that it is in proper format
*/
if ( NULL == (fp = fopen(argv[3], "r")) )
    handle_error(errno, "fopen");
k = 0;
while (( fscanf(fp, " %f ", &x) > 0 ) && (k < length) )
    b_vector[k++] = x;
fclose(fp);


/* Allocate the array of threads and task_data structures*/
threads     = calloc( num_threads, sizeof(pthread_t));
thread_data = calloc( num_threads, sizeof(task_data));
if ( threads == NULL || thread_data == NULL  )
    exit(1);

/* Compute the size each thread will get */
segment_size = (int) ceil (length*1.0 / num_threads);

/* Initialize the mutex */
pthread_mutex_init(&mutexsum, NULL);

/* Get ready -- initialize the thread attributes */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

/* Initialize task_data for each thread and then create the thread */
for ( t = 0 ; t < num_threads; t++) {
    thread_data[t].first      = t*segment_size;
    thread_data[t].last       = (t+1)*segment_size -1;
    if ( thread_data[t].last > length -1 )
        thread_data[t].last   = length - 1;
    thread_data[t].a          = &a_vector[0];
    thread_data[t].b          = &b_vector[0];
    thread_data[t].sum        = &total;

    retval = pthread_create(&threads[t], &attr, inner_product,
                        (void *) &thread_data[t]);
    if ( retval )
        handle_error( retval, "pthread_create");
}
```

```
    /* Join all threads and print sum */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(threads[t], (void**) NULL);
    }

    printf("The array total is %8.2f\n", total);

    /* Free all memory allocated to program */
    free ( threads );
    free ( thread_data );
    free ( a_vector );
    free ( b_vector );

    return 0;
}
```

10.6.7 Other Types of Mutexes

The type of a mutex is determined by the mutex attribute structure used to initialize it. There are four possible mutex types:

`PTHREAD_MUTEX_NORMAL`

`PTHREAD_MUTEX_ERRORCHECK`

`PTHREAD_MUTEX_RECURSIVE`

`PTHREAD_MUTEX_DEFAULT`

The default type is always `PTHREAD_MUTEX_DEFAULT`, which is usually equal to `PTHREAD_MUTEX_NORMAL`. To set the type of a mutex, use

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

passing a pointer to the `mutexattr` structure and the type to which it should be set. Then you can use this `mutexattr` structure to initialize the mutex.

There is no function that, given a mutex, can determine the type of that mutex. The best one can do is to call

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                              int *restrict type);
```

which retrieves the mutex type from a `mutexattr` structure. But, since there is no function that retrieves the `mutexattr` structure of a mutex, if you need to retrieve the type of the mutex, you must access the `mutexattr` structure that was used to initialize the mutex to know the mutex type.

When a normal mutex is accessed incorrectly, undefined behavior or deadlock result, depending on how the erroneous access took place. A thread will deadlock if it attempts to re-lock a mutex that it already holds. But if the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking takes place instead of deadlock or undefined behavior. Specifically, if a thread attempts to re lock a

mutex that it has already locked, the `EDEADLK` error is returned, and if a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error is also returned.

Recursive mutexes, i.e., those of type `PTHREAD_MUTEX_RECURSIVE`, can be used when threads invoke recursive functions. Basically, the mutex maintains a counter. When a thread first acquires the lock, the counter is set to one. Unlike a normal mutex, when a recursive mutex is relocked, rather than deadlocking, the call succeeds and the counter is incremented. A thread can continue to re-lock the mutex, up to some system-defined number of times. Each call to unlock the mutex by that same thread decrements the counter. When the counter reaches zero, the mutex is unlocked and can be acquired by another thread. Until the counter is zero, all other threads attempting to acquire the lock will be blocked on calls to `pthread_mutex_lock()`. A thread attempting to unlock a recursive mutex that another thread has locked is returned an error. A thread attempting to unlock an unlocked recursive mutex also receives an error.

Listing 10.7 contains an example of a program with a recursive mutex. It does not do anything other than print some diagnostic messages.

Listing 10.7: A program that uses a recursive mutex.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define  NUM_THREADS     5  /* Fixed number of threads */

pthread_mutex_t    mutex;
int                counter = 0;

void bar(int tid);

void foo(int tid)
{
    pthread_mutex_lock(&mutex);
    printf("Thread %d: In foo(); mutex locked\n",  tid);
    counter++;
    printf("Thread %d: In foo(); counter = %d\n", tid, counter);
    bar(tid);
    pthread_mutex_unlock(&mutex);
    printf("Thread %d: In foo(); mutex unlocked\n", tid);
}

void bar(int tid)
{
    pthread_mutex_lock(&mutex);
    printf("Thread %d: In bar(); mutex locked\n", tid);
    counter = 2*counter;
    printf("Thread %d: In bar(); counter = %d\n", tid, counter);
    pthread_mutex_unlock(&mutex);
    printf("Thread %d: In bar(); mutex unlocked\n", tid);
}

void * thread_routine( void * data )
{
    int t =  (int) data;
```

```
    foo(t);
    pthread_exit(NULL);
}

/*****************************************************************************
                                    Main Program
*****************************************************************************/

int main( int argc, char *argv[])
{
    int         retval;
    int         t;
    pthread_t   threads[NUM_THREADS];
    pthread_mutexattr_t     attr;


    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&mutex, &attr);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < NUM_THREADS; t++) {
        if ( 0 != pthread_create(&threads[t], NULL, thread_routine,
                            (void *) t)  ) {
            perror("Creating thread");
            exit(EXIT_FAILURE);
        }
    }

    for ( t = 0 ; t < NUM_THREADS; t++)
        pthread_join(threads[t], (void**) NULL);

    return 0;
}
```

10.7 Condition Variables

Mutexes are not sufficient to solve all synchronization problems efficiently. One problem is that they do not provide a means for one thread to signal another[2]. Consider the classical *producer/consumer problem*. In this problem, there are one or more "producer" threads that produce data that they place into a shared, finite buffer, and one or more "consumer" threads that consume the data in that buffer. We think of the data as being consumed because once it is read, no other thread should be able to read it; it is discarded, like the data in a pipe or a socket.

Suppose that the data chunks are fixed size and that the buffer can store N chunks. A consumer thread needs to be able to retrieve a data chunk from the buffer as long as one available, but if the buffer is empty, it should wait until one becomes available. Although it is possible for a consumer to busy-wait in a loop, continuously checking whether the buffer is non-empty, this is an inefficient

[2]In case you are thinking that a call to `pthread_mutex_unlock()` can be used to signal another thread that is waiting on a mutex, recall that this is not the way that a mutex can be used. The specification states that if a thread tries to unlock a mutex that it has not locked, undefined behavior results.

solution that wastes CPU cycles. Therefore, a consumer should block itself if the buffer is empty. Similarly, a producer thread should be able to write a chunk into the buffer if it is not full but otherwise block until a consumer removes a chunk.

These two buffer-full and buffer-empty conditions require that consumers be able to signal producers and vice versa when the buffer changes state from empty to non-empty and full to non-empty. In short, this type of problem requires that threads have the ability to signal other threads when certain conditions hold.

Condition variables solve this problem. They allow threads to wait for certain conditions to occur and to signal other threads that are waiting for the same or other conditions. Consider the version of the producer/consumer problem with a single producer and a single consumer. The producer thread would need to execute something like the following pseudo-code:

1. generate data to store into the buffer
2. *try to lock a mutex*
3. *if the buffer is full*
4. *atomically release the mutex and wait for the condition "buffer is not full"*
5. *when the buffer is not full:*
6. *re-acquire the mutex lock*
7. *add the data to the buffer*
8. *unlock the mutex*
9. *signal the consumer that there is data in the buffer*

Steps 4, 5, and 9 involve condition variables. The above pseudo-code would become

```
generate data_chunk to store into the buffer;
pthread_mutex_lock(&buffer_mutex);
if ( buffer_is_full() ) {
     pthread_cond_wait(&buffer_has_space, &buffer_mutex);
}
add data chunk to buffer;
pthread_mutex_unlock(&region_mutex);
pthread_cond_signal(&data_is_available);
```

The logic of the above code is that

1. A producer first locks a mutex to access the shared buffer. It may get blocked at this point if the mutex is locked already, but eventually it acquires the lock and advances to the if-statement.
2. In the if-statement, it then tests whether the boolean predicate "buffer_is_full" is true.
3. If so, it blocks itself on a condition variable named `buffer_has_space`. Notice that the call to block on a condition variable has a second argument which is a mutex. This is important. Condition variables are only used in conjunction with mutexes. When the thread calls this function, the mutex lock is taken away from it, freeing the lock, and the thread instead gets blocked on the condition variable.

4. Now assume that when a consumer empties a slot in the buffer, it issues a signal on the condition variable `buffer_has_space`. When this happens, the producer is woken up and re-acquires the mutex in a single atomic step. In other words, the magic of the condition variable is that when a process is blocked on it and is later signaled, it is given back the lock that was taken away from it.

5. The producer thread next adds its data to the buffer, unlocks the mutex, and signals the condition variable `data_is_available`, which is a condition variable on which the consumer might be waiting in case it tried to get data from an empty buffer.

An important observation is that the thread waits on the condition variable `buffer_has_space` only within the true-branch of the if-statement. A thread should make the call to `pthread_cond_wait()` only when it has ascertained that the logical condition associated with the condition variable is false (so that it is guaranteed to wait.) It should never call this unconditionally. Put another way, *associated with each condition variable is a programmer-defined boolean predicate that should be evaluated to determine whether a thread should wait on that condition.*

We now turn to the programming details.

10.7.1 Creating and Destroying Condition Variables

A condition variable is a variable of type `pthread_cond_t`. Condition variable initialization is similar to mutex initialization. There are two ways to initialize a condition variable:

1. Statically, when it is declared, using the `PTHREAD_COND_INITIALIZER` macro, as in

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

2. Dynamically, with the `pthread_cond_init()` routine:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

   This function is given a pointer to a condition variable and to a condition attribute structure, and initializes the condition variable to have the properties of that structure. If the `attr` argument is `NULL`, the condition is given the default properties. Attempting to initialize an already initialized condition variable results in undefined behavior.

The call

```
pthread_cond_init(&cond, NULL);
```

is equivalent to the static method except that no error-checking is done.

On success, `pthread_cond_init()` returns zero.

Because the condition variable must be accessed by multiple threads, it should either be global or it should be passed by address into each thread's thread function. In either case, the main thread should create it.

To destroy the condition variable, use

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

The `pthread_cond_destroy()` function destroys the given condition variable `cond` after which it becomes, in effect, uninitialized. A thread can only destroy an initialized condition variable if no threads are currently blocked on it. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

10.7.2 Waiting on Conditions

There are two functions that a thread can call to wait on a condition, an untimed wait and a timed wait:

```
int pthread_cond_wait     (pthread_cond_t  *restrict cond,
                           pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t  *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
```

Before a thread calls either of these functions, it must first lock the `mutex` argument, otherwise the effect of the call is undefined. Calling either function causes the following two actions to take place atomically:

1. `mutex` is released, and
2. the thread is blocked on the condition variable `cond`.

In the case of the untimed `pthread_cond_wait()`, the calling thread remains blocked in this call until some other thread signals `cond` using either of the two signaling functions described in Section 10.7.3 below. The signal wakes up the blocked thread and the call returns with the value zero, with `mutex` locked and owned by the now-unblocked thread.

In the case of `pthread_cond_timedwait()`, the calling thread remains blocked in this call until either some other thread signals `cond` or the absolute time specified by `abstime` is passed. In either case the effect is the same as that of `pthread_cond_wait()`, but if the time specified by `abstime` is passed first, the call returns with the error `ETIMEDOUT`, otherwise it returns zero.

Condition variables hold no state; they have no record of how many signals have been received at any given time. Therefore, if a thread T_1 signals a condition `cond` before another thread T_2 issues a wait on `cond`, thread T_2 will still wait on `cond` because the signal will have been lost; it is not saved. Only a signal that arrives after a thread has called one of the wait functions can wake up that calling thread. This is why we need to clarify the sense in which `pthread_cond_wait()` is atomic.

When a thread T calls `pthread_cond_wait()`, the mutex is unlocked and then the thread is blocked on the condition variable. It is possible for another thread to acquire the mutex after thread T has released it, but before it is blocked. If a thread signals this condition variable after this mutex has been acquired by another thread, then thread T will respond to the signal as if it had taken place

after it had been blocked. This means that it will re-acquire the mutex as soon as it can and the call will return.

The fact that a thread returns from a wait on a condition variable does not imply anything about the boolean predicate associated with this condition variable. It might be true or false. This is because a thread can return from a call to either of these functions due to a *spurious wakeup*. A spurious wakeup might occur, for example, if a signal is delivered to the blocked thread. It can also occur under certain conditions when a multi-threaded program is running on a multiprocessor. Therefore, calls to wait on condition variables should be inside a loop, not in a simple if statement. For example, the above producer code should properly be written as

```
generate data_chunk to store into the buffer;
pthread_mutex_lock(&buffer_mutex);
while ( buffer_is_full() ) {
     pthread_cond_wait(&buffer_has_space, &buffer_mutex);
}
add data chunk to buffer;
pthread_mutex_unlock(&region_mutex);
pthread_cond_signal(&data_is_available);
```

It is in general safer to code with a loop rather than an if-statement, because if you made a logic error elsewhere in your code and it is possible that a thread can be signaled even though the associated predicate is not true, then the loop prevents the thread from being woken up erroneously.

10.7.3 Waking Threads Blocked on Conditions

A thread can send a signal on a condition variable in one of two ways:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Both of these functions unblock threads that are blocked on a condition variable. The difference is that `pthread_cond_signal()` unblocks (at least) one of the threads that are blocked on the condition variable whereas `pthread_cond_broadcast()` unblocks all threads blocked by the condition variable. Under normal circumstances, `pthread_cond_signal()` will unblock a single thread, but implementations of this function may inadvertently wake up more than one, if more than one are waiting. Both return zero on success or an error code on failure.

Other points to remember about these two functions include:

- When multiple threads blocked on a condition variable are all unblocked by a broadcast, the order in which they are unblocked depends upon the scheduling policy. As noted in Section 10.7.2 above, when they become unblocked, they re-acquire the mutex associated with the condition variable. Therefore, the order in which they re-acquire the mutex is dependent on the scheduling policy.
- Although any thread can call `pthread_cond_signal(&cond)` or `pthread_cond_broadcast(&cond)`, only a thread that has locked the mutex associated with the condition variable cond should make this call, otherwise the scheduling of threads will be unpredictable, even knowing the scheduling policy.

10.7.4 Condition Attributes

The only attributes that conditions have are the process-shared attribute and the clock attribute. These are advanced topics that are not covered here. There are several functions related to condition attributes, specifically the getting and setting of these properties, and they are described by the respective man pages:

```
int pthread_condattr_destroy   ( pthread_condattr_t *attr);
int pthread_condattr_init      ( pthread_condattr_t *attr);
int pthread_condattr_getclock  ( const pthread_condattr_t *restrict attr,
                                 clockid_t *restrict clock_id);
int pthread_condattr_setclock  ( pthread_condattr_t *attr,
                                 clockid_t clock_id);
int pthread_condattr_getpshared( const pthread_condattr_t *restrict attr,
                                 int *restrict pshared);
int pthread_condattr_setpshared( pthread_condattr_t *attr,
                                 int pshared);
```

10.7.5 Example

Listing 10.8 contains a multi-threaded solution to the single-producer/single-consumer problem that uses a mutex and two condition variables. For simplicity, it is designed to terminate after a fixed number of iterations of each thread. It sends output messages to a file named `prodcons_mssges` in the working directory. The buffer routines add a single integer and remove a single integer from a shared global buffer. The calls to these functions in the producer and consumer are within the region protected by the mutex `buffer_mutex`.

The consumer logic is a bit more complex because the producer may exit when the buffer is empty. Therefore, the consumer thread has to check whether the producer is still alive before it blocks itself on the condition data_available, otherwise it will hang forever without terminating, and so will main().

It is not enough for the producer to set the flag producer_exists to zero when it exits, because the consumer might check its value just prior to the producer's setting it to zero, and seeing producer_exists == 1, block itself on the data_available condition. That is why the producer executes the lines

```
pthread_mutex_lock(&buffer_mutex);
producer_exists = 0;
pthread_cond_signal(&data_available);
pthread_mutex_unlock(&buffer_mutex);
```

when it exits. It first locks the buffer_mutex. If the consumer holds the lock, it will block until the consumer releases the lock. This implies that either the consumer has just acquired the mutex and is about to block itself on the data_available condition or that it is getting data from the buffer and will unlock the mutex soon. In either case, the consumer will release the lock and the producer will set producer_exists to zero and then signal data_available. If the consumer was about to block itself on data_available, then the signal will wake it up, it will see that producer_exists is zero, and

it will exit. If it was getting data from the buffer and then released the mutex lock, after which the producer acquired it, then when it gets it again, producer_exists will be zero, and it will exit if the buffer is empty.

Listing 10.8: Single-producer/single-consumer multithreaded program.

```
#include <sys/time.h>
#include <sys/types.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>

/*****************************************************************************
                                Global, Shared Data
*****************************************************************************/

#define   NUM_ITERATIONS   500    /* number of loops each thread iterates */
#define   BUFFER_SIZE      20     /* size of buffer */

/* buffer_mutex controls buffer access */
pthread_mutex_t  buffer_mutex     = PTHREAD_MUTEX_INITIALIZER;

/*  space_available is a condition that is true when the buffer is not full */
pthread_cond_t   space_available  = PTHREAD_COND_INITIALIZER;

/*  data_available is a condition that is true when the buffer is not empty */
pthread_cond_t   data_available   = PTHREAD_COND_INITIALIZER;

int       producer_exists;   /* true when producer is still running */
FILE      *fp;               /* log file pointer for messages       */

/*****************************************************************************
                                Buffer Object
*****************************************************************************/

int buffer[BUFFER_SIZE]; /* the buffer of data -- just ints here */
int bufsize;             /* number of filled slots in buffer */

void add_buffer(int data)
{
    static  int  rear = 0;
    buffer[rear] = data;
    rear = (rear + 1) % BUFFER_SIZE;
    bufsize++;
}

int get_buffer()
{
    static  int  front = 0;
    int i;
    i = buffer[front];
    front= (front + 1) % BUFFER_SIZE;
    bufsize--;
    return i ;
```

```
}

/*****************************************************************************
                              Error Handling Function
*****************************************************************************/

void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/*****************************************************************************
                                 Thread Functions
*****************************************************************************/

void *producer( void * data)
{
    int i;
    for (i = 1; i <= NUM_ITERATIONS; i++ ) {
        pthread_mutex_lock(&buffer_mutex);
        while ( BUFFER_SIZE == bufsize ) {
            pthread_cond_wait(&space_available,&buffer_mutex);
        }
        add_buffer(i);
        fprintf(fp,"Producer added %d to buffer; buffer size = %d.\n",
                i, bufsize);
        pthread_cond_signal(&data_available);
        pthread_mutex_unlock(&buffer_mutex);
    }

    pthread_mutex_lock(&buffer_mutex);
    producer_exists = 0;
    pthread_cond_signal(&data_available);
    pthread_mutex_unlock(&buffer_mutex);

    pthread_exit(NULL);
}


void *consumer( void * data )
{
    int i;
    for (i = 1; i <= NUM_ITERATIONS; i++ ) {
        pthread_mutex_lock(&buffer_mutex);
        while ( 0 == bufsize ) {
            if ( producer_exists ) {
                pthread_cond_wait(&data_available,&buffer_mutex);
            }
            else {
                pthread_mutex_unlock(&buffer_mutex);
                pthread_exit(NULL);
            }
```

```
        }
        i = get_buffer();
        fprintf(fp,"Consumer got data element %d; buffer size = %d.\n",
                i, bufsize);
        pthread_cond_signal(&space_available);
        pthread_mutex_unlock(&buffer_mutex);
    }
    pthread_exit(NULL);
}

/*****************************************************************************
                                    Main Program
*****************************************************************************/

int main(int argc, char* argv[])
{
    pthread_t producer_thread;
    pthread_t consumer_thread;

    producer_exists = 1;
    bufsize = 0;

    if ( NULL == (fp = fopen("./prodcons_mssges", "w")) )
        handle_error(errno, "prodcons_mssges");

    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_create(&producer_thread, NULL, producer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    fclose(fp);
    return 0;
}
```

10.8 Barrier Synchronization

10.8.1 Motivation

Some types of parallel programs require that the individual threads or processes proceed in a lockstep manner, each performing a task in a given phase and then waiting for all other threads to complete their tasks before continuing to the next phase. This is typically due to mutual dependencies on the data written during the previous phase by the threads. Many simulations have this property. One simple example is a multithreaded version of Conway's *Game of Life*.

The *Game of Life* simulates the growth of a colony of organisms over time. Imagine a finite, two-dimensional grid in which each cell represents an organism. Time advances in discrete time steps, t_0, t_1, t_2, ad infinitum. Whether or not an organism survives in cell (i, j) at time t_{k+1} depends on how many organisms are living in the adjacent surrounding cells at time t_k. Whether or not an organism is born into an empty cell (i, j) is also determined by the state of the adjacent cells at the given time. The exact rules are not relevant.

A simple method of simulating the progression of states of the grid is to create a unique thread to simulate each individual cell, and to create two grids, A and B, of the same dimensions. The initial state of the population is assigned to grid A. At each time step t_k, the thread responsible for cell (i, j) would perform the following task:

1. For cell A[i,j], examine the states of each of its eight neighboring cells A[m,n] and set the value of B[i,j] accordingly.
2. When all other cells have finished their step 1, copy B[i,j] to A[i,j], and repeat steps 1 and 2.

Notice that this solution requires that each cell wait for all other cells to reach the same point in the code. This could be achieved with a combination of mutexes and condition variables. The main program would initialize the value of a counter variable, `count`, to zero. Assuming there are `N` threads, each would execute a loop of the form

```
loop forever {
    update cell (i,j);

    pthread_mutex_lock (&update_mutex);
    count++;
    if ( count < N )
        pthread_cond_wait(&all_threads_ready,&update_mutex);
    /* count reached N so all threads proceed */
    pthread_cond_broadcast( &all_threads_ready);
    count --;
    pthread_mutex_unlock (&update_mutex);
    pthread_mutex_lock (&count_mutex);
    if ( count > 0 )
        pthread_cond_wait(&all_threads_at_start,&count_mutex);
    pthread_cond_broadcast( &all_threads_at_start);
    pthread_mutex_unlock (&count_mutex);
}
```

After each thread updates its cell, it tries to acquire a mutex named `update_mutex`. The cell that acquires the mutex increments `count` and then waits on a condition variable named `all_threads_ready` associated with the predicate `count < N`. As it releases `update_mutex`, the next thread does the same, and so on until all but one thread has been blocked on the condition variable. Eventually the `N`*th* thread acquires the mutex, increments `count` and, finding `count == N`, issues a broadcast on `all_threads_ready`, unblocking all of the waiting threads, one by one.

One by one, each thread then decrements `count`. If each were allowed to cycle back to the top of the loop, this code would not work, because one thread could quickly speed around, increment `count` so that it equaled `N` again even though the others had not even started their updates. Instead, no thread is allowed to go back to the top of the loop until `count` reaches zero. This is achieved by using a second condition variable, `all_threads_at_start`. All threads will block on this condition except the one that sets the value of `count` to zero when it decrements it. When that happens, every thread is unblocked and they all start this cycle all over again.

Now as you can see, this adds so much serial code to the parallel algorithm that it defeats the purpose of using multiple threads in the first place. In addition, it ignores the possibility of spurious wake-ups

and would be even more complex if these were taken into account. Fortunately, there is a simpler solution; the Pthread library has a *barrier synchronization* primitive that solves this synchronization problem efficiently and elegantly.

A *barrier synchronization point* is an instruction in a program at which the executing thread must wait until all participating threads have reached that same point. If you have ever been in a guided group of people being taken on a tour of a facility or an institution of some kind, then you might have experienced this type of synchronization. The guide will wait for all members of the group to reach a certain point, and only then will he or she allow the group to move to the next set of locations.

10.8.2 PThreads Barriers

The Pthreads implementation of a barrier lets the programmer initialize the barrier to the number of threads that must reach the barrier in order for it to be opened. A barrier is declared as a variable of type `pthread_barrier_t`. The function to initialize a barrier is

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
       const pthread_barrierattr_t *restrict attr, unsigned count);
```

It is given the address of a barrier, the address of a barrier attribute structure, which may be `NULL` to use the default attributes, and a *positive* value `count`. The count argument specifies the number of threads that must reach the barrier before any of them successfully return from the call. If the function succeeds it returns zero.

A thread calls

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

to wait at the barrier given by the argument. When the required number of threads have called `pthread_barrier_wait()` specifying the barrier, the constant `PTHREAD_BARRIER_SERIAL_THREAD` is returned to exactly one unspecified thread and zero is returned to each of the remaining threads. At this point, the barrier is reset to the state it had as a result of the most recent `pthread_barrier_init()` function that referenced it. Some programs may not need to take advantage of the fact that a single thread received the value `PTHREAD_BARRIER_SERIAL_THREAD`, but others may find it useful, particularly if exactly one thread has to perform a task when the barrier has been reached. One can check for errors at the barrier with the code

```
retval = pthread_barrier_wait(&barrier);
if ( PTHREAD_BARRIER_SERIAL_THREAD != retval &&  0 != retval )
    pthread_exit((void*) 0);
```

which will force a thread to exit if it did not get one of the non-error values.

Finally, a barrier is destroyed using

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

which destroys the barrier and releases any resources used by it. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to `pthread_barrier_init()`. The results are undefined if `pthread_barrier_destroy()` is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

10.8.3 Example

Consider the problem of adding the elements of an array of N numbers, where N is extremely large. The serial algorithm would take $O(N)$ steps. Suppose that a processor has P subprocessors and that we want to use P threads to reduce the total running time of the problem. Assume for simplicity that N is a multiple of P. We can decompose the array into P segments of N/P elements each and let each thread sum its set of N/P numbers. But then how can we collect the partial sums calculated by the threads?

Let us create an array, `sums`, of length P. The partial sum computed by thread `k` is stored in `sums[k]`. To compute the sum of all numbers, we let the main program add the numbers in the sums array and store the result in sums[0]. In other words, we could execute a loop of the form

```
for ( i = 1; i < P; i++)
    sums[0] += sums[i];
```

This would run in time proportional to the number of threads. Alternatively, we could have each thread add its partial sum directly to a single accumulator, but we would need to serialize this by enclosing it in a critical section. The performance is the same, since there would still be P sequential additions.

Another solution is to use a *reduction algorithm* to add the partial sums. A *reduction algorithm* is like a divide-and-conquer solution. Each thread computes its partial sum and then waits at a barrier until all other threads have also computed their partial sums. At this point the algorithm proceeds in stages.

The set of thread ids is divided in half. Every thread in the lower half has a *mate* in the upper half, except possibly one odd thread. For example, if there are 100 threads, then thread 0 is mated to thread 50, thread 1 to thread 51, and so on, and thread 49 to thread 99. In each stage, each thread in the lower half of the set adds its mate's sum to its own. At the end of each stage, the upper half of threads is no longer needed, so the set is cut in half. The lower half becomes the new set and the process is repeated. For example, there would be 50 threads numbered 0 to 49, with threads 0 through 24 forming the lower half and threads 25 to 49 in the upper half. As this happens, the partial sums are being accumulated closer and closer to `sums[0]`.

Eventually the set becomes size 2, and thread 0 adds `sums[0]` and `sums[1]` into `sums[0]`, which is the sum of all array elements. This approach takes $O(\log(P))$ steps. The entire running time is thus $O((N/P) + \log(P))$.

Listing 10.9 contains the code.

Listing 10.9: Reduction algorithm with barrier synchronization.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libintl.h>
#include <locale.h>
#include <math.h>

/*****************************************************************************
                              Data Types and Constants
```

```
*****************************************************************************/

double       *sum;         /* array of partial sums of data           */
double       *array;       /* dynamically allocated array of data     */
int          num_threads; /* number of threads this program will use */
pthread_barrier_t barrier;

/*
   a task_data structure contains the data required for a thread to compute
   the sum of the segment of the array it has been delegated to total, storing
   the sum in its cell in an array of sums. The data array and the sum array
   are allocated on the heap. The threads get the starting addresses of each,
   and their task number and the first and last entries of their segments.
*/
typedef struct _task_data
{
    int first;        /* index of first element for task */
    int last;         /* index of last element for task */
    int task_id;      /* id of thread */
} task_data;


/*****************************************************************************
                          Thread and Helper Functions
*****************************************************************************/

/* Print usage statement */
void usage(char *s)
{
        char *p = strrchr(s, '/');
        fprintf(stderr,
                "usage: %s arraysize  numthreads \n", p ? p + 1 : s);
}

/**
   The thread routine.
*/
void  *add_array( void * thread_data )
{
    task_data *t_data;
    int   k;
    int   tid;
    int   half;
    int   retval;

    t_data  = (task_data*) thread_data;
    tid     = t_data->task_id;

    sum[tid] = 0;
    for ( k = t_data->first; k <= t_data->last; k++ )
        sum[tid] += array[k];

    half = num_threads;
    while ( half > 1 ) {
```

```
        retval = pthread_barrier_wait(&barrier);
        if ( PTHREAD_BARRIER_SERIAL_THREAD != retval &&
             0 != retval )
           pthread_exit((void*) 0);

        if ( half % 2 == 1 && tid == 0 )
            sum[0] = sum[0] + sum[half-1];
        half = half/2; // integer division
        if ( tid < half )
            sum[tid] = sum[tid] + sum[tid+half];
    }

    pthread_exit((void*) 0);

}

/*****************************************************************************
                                    Main Program
*****************************************************************************/
int main( int argc, char *argv[])
{
    int         array_size;
    int         size;
    int         k;
    int         retval;
    int         t;
    pthread_t   *threads;
    task_data   *thread_data;
    pthread_attr_t attr;

    /* Instead of assuming that the system creates threads as joinable by
       default, this sets them to be joinable explicitly.
    */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    if ( argc < 3 ) {
        usage(argv[0]);
        exit(1);
    }

    /* Get command line arguments, convert to ints, and compute size of each
       thread's segment of the array
    */
    array_size  = atoi(argv[1]);
    num_threads = atoi(argv[2]);
    size = (int) ceil(array_size*1.0/num_threads);

    /* Allocate the array of threads, task_data structures, data and sums */
    threads     = calloc( num_threads, sizeof(pthread_t));
    thread_data = calloc( num_threads, sizeof(task_data));
    array       = calloc( array_size,  sizeof(double));
    sum         = calloc( num_threads, sizeof(double));
```

```
    if ( threads == NULL || thread_data == NULL ||
         array == NULL || sum == NULL )
        exit(1);

    /* Synthesize array data here */
    for ( k = 0 ; k < array_size; k++ )
        array[k] = (double) k;


    /* Initialize a barrier with a count equal to the numebr of threads */
    pthread_barrier_init(&barrier, NULL, num_threads);

    /* Initialize task_data for each thread and then create the thread */
    for ( t = 0 ; t < num_threads; t++) {
        thread_data[t].first      = t*size;
        thread_data[t].last       = (t+1)*size -1;
        if ( thread_data[t].last > array_size -1 )
            thread_data[t].last   = array_size - 1;
        thread_data[t].task_id    = t;


        retval = pthread_create(&threads[t], &attr, add_array,
                                (void *) &thread_data[t]);
        if ( retval ) {
            printf("ERROR; return code from pthread_create() is %d\n", retval);
            exit(-1);
        }
    }

    /* Join all threads so that we can add up their partial sums */
    for ( t = 0 ; t < num_threads; t++) {
        pthread_join(threads[t], (void**) NULL);
    }

    pthread_barrier_destroy(&barrier);

    printf("The array total is %7.2f\n", sum[0]);

    /* Free all memory allocated to program */
    free ( threads );
    free ( thread_data );
    free ( array );
    free ( sum );

    return 0;
}
```

Although the solution in Listing 10.9 is asymptotically faster than the solution in which the threads add their partial sums to a running total in a critical section, it may not be faster in practice, because the final accumulation of partial sums must wait until all threads have calculated their partial sums. If the number of threads is very large, and there is one very slow thread, then the $\log(P)$ steps will be delayed until the slow thread completes. On the other hand, if the other solution is used, then all threads will have added their partial sums to the total while the slow thread was

still working, and when it finishes, a single addition will complete the task. The performance gain of this reduction algorithm depends upon the threads running on symmetric processors.

10.9 Reader/Writer Locks

10.9.1 Introduction

A mutex has the property that it has just two states, locked and unlocked, and only one thread can lock it at a time. For many problems this is fine, but for many others, it is not. Consider a problem in which one thread updates a database of some kind and multiple threads look up information in that database. For example, a web search engine might consist of thousands of "reading" threads that need to read the database of search data to deliver pages of search results to client browsers, and other "writing" threads that crawl the web and update the database with new data. When the database is not being updated, the reading threads should be allowed simultaneous access to the database, but when a writing thread is modifying the database, it needs to do so in mutual exclusion, at least on the parts of it that are changing.

To support this paradigm, POSIX provides *reader/writer locks*. Multiple readers can lock a reader/writer lock without blocking each other, but blocking writers from accessing it, and when a single writer acquires the lock, it obtains exclusive access to the resource; any thread, whether a reader or a writer, will be blocked if it attempts to acquire the lock while a writer holds the lock.

Clearly, reader/writer locks allow for a higher degree of parallelism than does a mutex. Unlike mutexes, they have three possible states: locked in read mode, locked in write mode, and unlocked. Multiple threads can hold a reader/writer lock in read mode, but only a single thread can hold a reader/writer lock in write mode.

Think of a reader/writer lock as the key to a large room. If the read/writer lock is not currently held by any thread and a reader acquires it, then it enters the room and leaves a guard at the door. If an arriving thread wants to write, the guard makes it wait on a line outside of the door until the reader leaves the room, or possibly later. All arriving writers will wait on this line while the reader is in the room. If an arriving thread wants to read, whether or not it is let into the room depends on how Pthreads has been configured.

Some systems support a Pthreads option known as the *Thread Execution Scheduling,* or *TES,* option. This option allows the programmer to control how threads are scheduled. If the system does not support this option, and a reader arrives at the door, and there are writers standing in line, it is up to the implementation as to whether the reader must stand at the end of the line, behind the waiting writer(s), or can be allowed to enter the room immediately. If *TES* is supported, then the decision is based on which scheduling policy is in force. If either FIFO, round-robin, or sporadic[3] scheduling is in force, then an arriving reader will stand in line behind all writers (and any readers who have set their priorities higher than the arriving reader's.)

These decisions about who must wait for whom when threads are blocked on a lock can lead to unfair scheduling and even starvation. A discussion of this topic is really outside of the scope of these notes, but you should at least have the intuition that, if the implementation gives arriving readers precedence over writers that are blocked when a reader has the lock, then *a steady stream of readers could prevent a writer from ever writing.* This is not good. Usually, a writer has something

[3]This is also an option to PThreads that may not be available in a given implementation.

important to do, updating information, and it should be given priority over readers. This is why the *TES* option allows this type of behavior, and why some implementations always give waiting writers priority over waiting readers. For this reason, it is also possible that a stream of writers will starve all of the readers, so if for some reason, there must be multiple writers, *the code itself must ensure that they do not starve the readers*, using mutexes and conditions to prevent this possibility.

10.9.2 Using Reader/Writer Locks

It is natural that, as a result of their increased complexity, there are more functions for locking and unlocking reader/writer locks than for simple mutexes. The prototypes for the functions in the API related to these locks, listed by category, are:

Initialization and destruction:

```
int   pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                  const pthread_rwlockattr_t *restrict attr);
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int   pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Locking for reading:

```
int   pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                  const struct timespec *restrict abstime);
```

Locking for writing:

```
int   pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int   pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                  const struct timespec *restrict abstime);
```

Unlocking:

```
int   pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Working with attributes:

```
int   pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int   pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int   pthread_rwlockattr_getpshared(const pthread_rwlockattr_t
                  *restrict attr, int *restrict pshared);
int   pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                   int pshared);
```

As with all of the other locks and synchronization objects described here so far, the first step is to initialize the reader/writer lock. This is done using either the function `pthread_rwlock_init()` or the initializer macro `PTHREAD_RWLOCK_INITIALIZER`, which is equivalent to using `pthread_rwlock_init()` with a `NULL` second argument. There are not many attributes that can be configured; the process-shared attribute is not required to be implemented by a POSIX-compliant system, and there are no others that can be modified. Therefore, it is fine to accept the defaults.

Notice that a thread wishing to use the lock for reading uses a different set of primitives than one that wants to write. For reading, a thread can use `pthread_rwlock_rdlock()`, which has the semantics described in the introduction above. If you do not want the thread to block in those cases where it might, use `pthread_rwlock_tryrdlock()`, which will return the error value `EBUSY` whenever it would block.

The `pthread_rwlock_timedrdlock()` function is like the `pthread_rwlock_rdlock()` function, except that, if the lock cannot be acquired without blocking, the wait is terminated when the specified timeout expires. The timeout expires when the *absolute time* specified by `abstime` passes, as measured by the real time clock (`CLOCK_REALTIME`) or if the absolute time specified by `abstime` has already been passed at the time of the call. Note that the time specification is not an interval, but what you might call "clock time", as the system perceives it. The `timespec` data type is defined in the `<time.h>` header file. The function does not fail if the lock can be acquired immediately, and the validity of the `abstime` parameter is not checked if the lock can be acquired immediately.

The same statements apply to the three functions for acquiring a writer lock, and so they are not repeated. As for unlocking, there is only one function to unlock. It does not matter whether the thread holds the lock for reading or writing – it calls `pthread_rwlock_unlock()` in either case.

10.9.3 Further Details

This section answers some more subtle, advanced questions about reader/writer locks.

- If the calling thread already holds a shared read lock on the reader/writer lock, another read lock can be successfully acquired by the calling thread. If more than one shared read lock is successfully acquired by a thread on a reader/writer lock, that thread is required to successfully call `pthread_rwlock_unlock()` a matching number of times.
- Some implementations of Pthreads will allow a thread that already holds an exclusive write lock on a reader/writer lock to acquire another write lock on that same lock. In these implementations, if more than one exclusive write lock is successfully acquired by a thread on a reader/writer lock, that thread is required to successfully call `pthread_rwlock_unlock()` a matching number of times. In other implementations, the attempt to acquire a second write lock will cause deadlock.
- If while either of `pthread_rwlock_wrlock()` or `pthread_rwlock_rdlock()` is waiting for the shared read lock, the reader/writer lock is destroyed, then the `EDESTROYED` error is returned.
- If a signal is delivered to the thread while it is waiting for the lock for either reading or writing, if a signal handler is registered for this signal, it runs, and the thread resumes waiting.
- If a thread terminates while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not succeed. In this case, the attempt to acquire the

lock does not return and will deadlock. If a thread terminates while holding a read lock, the system automatically releases the read lock.

- If a thread calls `pthread_rwlock_wrlock()` and currently holds a shared read lock on the reader/writer lock and no other threads are holding a shared read lock, the exclusive write request is granted. After the exclusive write lock request is granted, the calling thread holds both the shared read and the exclusive write lock for the specified reader/writer lock.
- In an implementation in which a thread can hold multiple read and write locks on the same reader/writer lock, if a thread calls `pthread_rwlock_unlock()` while holding one or more shared read locks and one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to `pthread_rwlock_unlock()` must be completed before all write locks are unlocked. At that time, subsequent calls to `pthread_rwlock_unlock()` will unlock the shared read locks.

10.9.4 Example

The program in Listing 10.10 demonstrates the use of reader/writer locks. It would be very simple if we did not attempt to prevent starvation, either of readers or writers. It uses barrier synchronization to ensure that no thread enters its main loop until all threads have at least been created. Without the barrier, the threads that are created first in the main program will always get the lock first, and if these are writers, the readers will starve.

If the number of writers is changed to be greater than one, they will starve the readers whenever the first writer grabs the lock. This is because writers are given priority over readers in the code below.

Listing 10.10: Reader/writer locks: A simple example.

```
#define  _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

/*****************************************************************************
                            Data Types and Constants
*****************************************************************************/

#define            NUM_READERS  10
#define            NUM_WRITERS  1
pthread_rwlock_t   rwlock;          /* the reader/writer lock */
pthread_barrier_t  barrier;         /* to try to improve fairness */

int           done;                 /* to terminate all threads */
int           num_threads_in_lock; /* for the monitor code */


/*****************************************************************************
                          Thread and Helper Functions
```

```
*****************************************************************************/
/** handle_error(num, mssge)
   Prints to standard error the system message associated with error number num
   as well as a custom message, and then exits the program with EXIT_FAILURE
*/
void handle_error(int num, char *mssge)
{
    errno = num;
    perror(mssge);
    exit(EXIT_FAILURE);
}

/** reader()
 *  A reader repeatedly gets the lock, sleeps a bit, and then releases the lock,
 *  until done becomes true.
 */
void *reader(void * data)
{
    int rc;
    int t = (int) data;

    /* Wait here until all threads are created */
    rc = pthread_barrier_wait(&barrier);
    if ( PTHREAD_BARRIER_SERIAL_THREAD != rc &&  0 != rc )
        handle_error( rc, "pthread_barrier_wait");

    /* repeat until user says to quit */
    while ( ! done ) {
        rc = pthread_rwlock_rdlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_rdlock");
        printf("Reader %d got the read lock\n", t);
        sleep(1);
        rc = pthread_rwlock_unlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_unlock");
        sleep(1);
    }
    pthread_exit(NULL);
}

/** writer()
 *  A writer does the same thing as a reader — it repeatedly gets the lock,
 *  sleeps a bit, and then releases the lock, until done becomes true.
 */
void *writer(void * data)
{
    int rc;
    int t = (int) data;

    /* Wait here until all threads are created */
    rc = pthread_barrier_wait(&barrier);
    if ( PTHREAD_BARRIER_SERIAL_THREAD != rc &&  0 != rc )
        handle_error( rc, "pthread_barrier_wait");

    /* repeat until user says to quit */
```

```
    while ( ! done ) {
        rc = pthread_rwlock_wrlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_wrlock");
        printf("Writer %d got the write lock\n", t);
        sleep(2);

        rc = pthread_rwlock_unlock(&rwlock);
        if ( rc ) handle_error( rc, "pthread_rwlock_unlock");
        sleep(2);
    }
    pthread_exit(NULL);
}

/*****************************************************************************
                                    Main Program
*****************************************************************************/

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_READERS+NUM_WRITERS];
    int retval;
    int t;
    unsigned int num_threads = NUM_READERS+NUM_WRITERS;

    done                = 0;
    printf("This program will start up a number of threads that will run \n"
           "until you enter a character. Type any character to quit\n");

    pthread_rwlockattr_t  rwlock_attributes;
    pthread_rwlockattr_init(&rwlock_attributes);
    /* The following non-portable function is a GNU extension that alters the
       thread priorities when readers and writers are both waiting on a rwlock,
       giving preference to writers.
    */
    pthread_rwlockattr_setkind_np(&rwlock_attributes,
           PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
    pthread_rwlock_init(&rwlock, &rwlock_attributes );

    /* Initialize a barrier with a count equal to the numebr of threads */
    retval = pthread_barrier_init(&barrier, NULL, num_threads);
    if ( retval ) handle_error( retval, "pthread_barrier_init");

    for ( t = 0 ; t < NUM_READERS; t++) {
        retval = pthread_create(&threads[t], NULL, reader, (void *)t);
        if ( retval ) handle_error( retval, "pthread_create");
    }

    for ( t = NUM_READERS ; t < NUM_READERS+NUM_WRITERS; t++) {
        retval = pthread_create(&threads[t], NULL, writer, (void *)t);
        if ( retval )  handle_error( retval, "pthread_create");
    }

    getchar();
    done = 1;
```

```
    for ( t = 0 ; t < NUM_READERS+NUM_WRITERS; t++)
        pthread_join(threads[t], NULL);

    return 0;
}
```

10.10 Other Topics Not Covered

Any serious multi-threaded program must deal with signals and their interactions with threads. The man pages for the various thread-related functions usually have a section on how signals interact with thoze functions. Spin locks are another synchronization primitive not discussed here; they have limited use. Real-time threads and thread scheduling, where supported, provide the means to control how threads are scheduled for more accurate performance control. Thread keys are a way to create thread-specific data that is visible to all threads in the process.