SYSTEM PROGRAMMING

From the book by STEWART WEISS

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.

File System: An Abstraction Supporting

- Create, Delete and Modify files;
- Organize files and directories;
- Control access to files and directories;
- Manage disk space.

File System (FS) Mounting

- In Microsoft DOS:
  - Each disk partition has a drive letter,
  - The file hierarchy on each separate drive or partition is separate from all others.
  - DOS has multiple trees whose roots are drive letters
- In UNIX:
  - There is a single file hierarchy.
  - It is a tree of directories as nodes and files as leaves.
  - An FS's root may be mounted to some directory of the single file hierarchy.
  - Then the FS becomes a subtree of the hierarchy.

File System Mounting

- The *mount* command without arguments displays a list of the FSs currently mounted:

```
/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)
```

Disk Partitions

Physical portions of the same disk divided into logical devices called Partitions which may allow:

- More control of **security**:
  - Different user groups on different partitions
  - Different mounting options (i.e., read only) on separate partitions
- 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 per partitions, not disks
- **Improved reliability**:
  - Damage could be restricted to a single partition rather than the entire disk
  - Redundancy could be built in
- **But,** once partitions created, their size can not be increased: Need to **reorganize whole the Disk!**

Defining and Creating File Systems

- In UNIX, disk partitions are not necessarily disjoint.
- Partitions often named with letters, i.e., a, b, c, etc.
- The "c" partition is the entire disk and does not have an FS. It is used to access the disk block by block.
- The "b" partition reserved as the swapping store and does not have an FS.
- The "a" partition is where the kernel installed and it is usually very small.

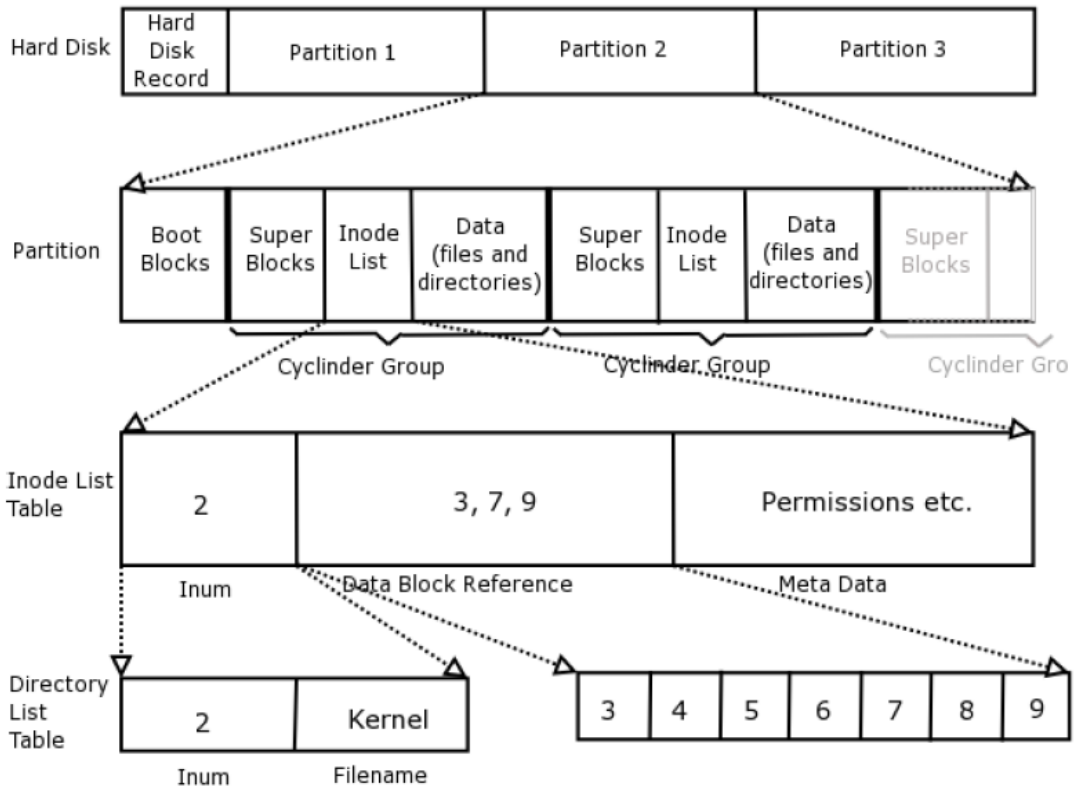
Defining and Creating File Systems

- To create files in a partition, an FS must be created first:
    - Partition divided into logical blocks of size 1024 - 4096 bytes each.
    - The block size is fixed at FS creation time.
    - Larger block sizes result in more wasted disk space.
    - Larger blocks are appropriate for FSs expecting large files.
    - Smaller block sizes result in more disk activity.

The Components of UNIX File System

- Every FS has (at least) one superblock located at the beginning of its allocated storage.
- The superblock contains information about how FS is configured, e.g., block size, block address range, and mount status.
- Copies of the superblock stored in several other places within a partition.
- Each FS in UNIX has at least one **table** that identifies the files in it.
- The entries of the table called ***i-nodes***, and their indices called ***i-numbers***.
- The i-nodes contain file attributes and a map indicating locations of the blocks of the file.

Figure 3.1: Unix File System Layout



- There are multiple superblocks and multiple i-node tables in a single file system, for performance reasons.
- The i-node in position 2 of the table usually points to the entry for the root directory file in the file system.

The **i-node** attributes

- The owner,
- The group,
- The permissions allowed on the file and the file type,
- The number of links to the file
- The time of last modication,
- 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,
- The id of the device on which the file resides.

How the Kernel Creates Files

The kernel;

- creates an i-node for the file, if possible.
- fills in the i-node with the file status.
- allocates data blocks for the file and stores the file data in these blocks.
- records the addresses of the data blocks in the i-node.
- creates a directory entry in the scratch directory with the i-node number and file name.

The ls Command

ls [options] FILE FILE ...

where FILE, FILE, . . . are filenames, whether they are regular files, special files, symbolic links, or directories.

- When the argument is a directory, ls displays its contents.
- When the argument is not a directory, ls displays its name.

The Directory Interface

From the kernel perspective, regular files and directories are just sequences of bytes except that directories:

- Are never empty:
  - Every directory has **.** and **..** referring to the directory itself and to the parent directory.
- They cannot be written to by unprivileged programs:
  - They can only be modified by very specific system calls.
- They have a specific structure:
  - A directory is a file that contains a collection of (name, i-number) pairs.

Reading Directories

$ man k directory | grep read

$man 3p readdir

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.

SEE ALSO
read (2), closedir (3), dirfd (3), ftw (3), opendir (3), rewinddir (3), scandir (3), seekdir (3), telldir (3), feature_test_macros (7)

The *dirent* structure

```
struct dirent {
  ino_t d_ino;               // inode number POSIX defined
  off_t d_off;               // offset to the next dirent
  unsigned short d_reclen;   // length of this record
  unsigned char d_type;      // type of file
  char d_name[256];          // filename POSIX defined
};
```

opendir

SYNOPSIS

#include <sys/type s.h>
#include <dirent.h>
DIR opendir(const char name);

DESCRIPTION

The *opendir()* function opens a directory stream corresponding 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.

```
#include <dirent.h>
long telldir(DIR *dirp); // the entry which is about to be read by a program
```

telldir() returns its index, which is an integer offset from the beginning of the directory stream.
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()*.

DIR is not a macro, but a typedef:

```
typedef struct __dirstream DIR;
```

```
/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
```

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_blk size; //  Preferred I/O block size
blkcnt_t st_blocks; // Number of 512 byte blocks allocated
```

The Three Special Bits

Consider the problem:

- A user should be able to change his/her password.
- Since all passwords stored in a single file, the user needs write permission to modify the file.
- If user given write permission on the password file, he/she can modify anyone else's password too, which is not acceptable.
- The password file should be owned by root, and no user (except the su) should have write permission to it.
- Then, what to do?

Solution on next slide!

The Three Special Bits: The Set-User-ID Bit

- A process has two associated user-ids: **real user-id** and **effective user-id**.
- The real user-id can never be changed. The effective user-id can be.
- Generally, when a user runs a command, the process executing the command has the same effective user-id of the user invoking the command.
- This is usually the real user-id of the user who indirectly ran the process.
- **If the setuid bit is set** on the exe file of the command, the effective user-id of the process executing it, is the user-id of the owner of the file.
- In *passwd* command case, root owns the /usr/bin/passwd program file thus no user can modify the password file, but root.
- But since its setuid bit turned on, passwd command runs with effective user-id of the root.

The Three Special Bits: The Set-User-ID Bit

- Consequently, *passwd* command CAN access the password file and change it, but the command checks the real user-id of the caller (by calling getuid()) first, and it limits access to 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.

The Three Special Bits: The Set-Group-ID Bit

- Set-Group-ID Bit sets the **effective-group-id** of the running program.
- If a program has the setgid bit on, it runs with the privileges of the owning group, not privileges of the running group.

- **Example**

- The write command (/usr/bin/write) lets users write to a terminal.
- But, how is it possible, one person writes on another's terminal?
- The write command needs write permission on the terminal on which it wants to write.
- Take a look at /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

The Three Special Bits: The Set-Group-ID Bit

- Notice that all terminals belong to the tty group and some of them have the group write bit set.
- This means 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**
- Now, observe that the write exe 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.

The Three Special Bits: The Sticky Bit

- Also called the save-text-image bit.
- Originally, UNIX was a pure swapping operating system.
- Processes swapped in and out of memory to maintain the multiprogramming level.
- The swapping store was a separate disk or a partition used for storing process images.
- Process images kept in contiguous bytes on the swapping store, making reads and writes faster.
- A program 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 **prevent**ed it from **being removed** from the **swapping store**.

The Three Special Bits: The Sticky Bit

- For directories, it is a different story.
- If a directory has sticky bit on:
  - the directory is readable and writable by everyone,
  - but no one can delete another person's files in that directory.
- This is how UNIX can implement directories such as /tmp, which is used to store temporary files.

The Special Bits and ls

- If the set-uid bit turned on, the permission of user prints as: *-rws------*
- If the set-gid bit turned on, the permission of group prints as: *----rws---*
- If the sticky bit turned on, the permission of other prints as: *-------rwt*

ls Implementation: Printing File Status

```
void print_file_status(char* dname, struct stat statbuf) {
  ssize_t count;
  char buf[NAME_MAX];

  printf("%10.10s", mode2str(statbuf.st_mode));       //print type, permissions
  printf("%3d", (int)statbuf.st_nlink);               // print number of links
  printf(" %-8.8s ", uid2name(statbuf.st_uid));       // print owner's name if it is found using getpwuid()
  printf(" %-8.8s ", gid2name(statbuf.st_gid));       // print group name if it is found using getgrgid()
  printf(" %8jd ", (intmax_t)statbuf.st_size);        // print sizeof file
  printf(" %.12s ", get_date_no_day(statbuf.st_mtime)); // print time of last modification
  printf(" %s ", dname);                              // print file name

  if (S_ISLNK(statbuf.st_mode)) {                     // if it is a link
    if (-1 == (count = readlink(dname, buf, NAME_MAX-1)))
      perror("print_file_status: ");
    else {
      buf[count] = '\0';
      printf("->%s ", buf);                           // print the linked file
    }
  }

  printf("\n ");
}
```

ls Implementation: Mode as Human Readable

```
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;
}
```

ls Implementation: Recent? Or Old?

```
char get_date_no_day(time_t timeval) {
  //# of secs in 182 days
  const int sixm onths = 15724800;

  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;
  }
}
```

ls Implementation: UID & GID Name

```c
//Given user-id, return user-name
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
    sprintf(numstr, "%d", uid);
    return numstr;
  } else
    return pw_ptr->pw_name;
}
```

```c
//Given group-id, return group-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;
}
```

ls Implementation: Main Function

```
void ls(char dirname[], int do_longlisting);
void print_file_status(char* dname, struct stat buf);
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;
}
```

ls Implementation: ls itself

```
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)
      continue; // skip dot and dot-dot entries
    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);
    }
  }
}
```

Modifying File Attributes

- We have seen **how to access** attributes stored in the i-nodes, but did not modify any of them.

- Now we investigate **how attributes can be modified** by user level programs.

Type of a File

Type of files determined at creation time and cannot be changed later. A file can be:

- A regular file,
- A directory,
- A device special file,
- A socket,
- A symbolic link,
- A named pipe.

- The *creat()* system call creates regular files.
- The *mkdir()* call makes a directory.
- The *mkdir* command creates a new directory.
- The *mknod()* is the system call that creates special files.
- The *mknod* command makes these at the user level.
- The *mkfifo()* system call creates FIFO special files.

Permission Bits and Special Bits

- Permission bits initialized when kernel creates files.
- The second argument of the *creat()* call initializes the file mode.
- However, this number modified by applying the process's *umask* before assigning to the file, using bitwise operation: *"mode = mode & ~umask"*
- Every user has a umask. The umask of the process is the umask of the effective user-id of the process. If the umask has value 022:
- *fd = creat("newfile", 0766);*

  would create "newfile" with permission 0744 (=766 & ~022).

Permission Bits and Special Bits

- A process inherits its umask value, but can change it by calling *umask()*:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

- The *umask()* system call (or umask shell command) changes the umask and returns the value of the previous mask. The mask parameter bitwise AND'ed with 0777 to strip out the file type and special bit values.
- The *chmod()* system call (or chmod shell command) changes a file's permission bits:

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

Number of Links to a File

- The name of a file is just a name stored in a directory.
- The total number of names of a file called its link count and stored in i-node.
- When a name deleted, the link count is decremented, and the file actually removed when it reaches zero.
- 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);
```

Owner and Group of a File

- The owner and group of a file are established when the file created 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.
- The owner and group of a file are changed only by the *chown()* and *chgrp()* system calls or their command equivalents.

Size of a File

- Size of a file set to zero by the *creat()* call and increases as data written to it using the *write()* system call.

Modication 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 timestamps are set by the kernel as the file is accessed and modified.
- Users have no control over *st_ctime*, but 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
} ;
```

Name of a File

- The *rename()* system call renames a file and returns 1 on failure, 0 on success.

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

- The behavior is complex, depending on:
  - whether oldname refers to a file, a directory, or a symbolic link,
  - whether newname already exists,
  - whether it is on the same file system.

Traversing the Tree, Up and Down

- **Traversing Up:** Both the *pwd* command and *getcwd()* system call travel from the current working directory up the tree so it can display the whole path from the root to the current working directory.

- **Traversing Down:** The *find* command and recursive versions of commands such as *ls, grep, chown, chmod,* and many more, start in the given directory and visit all files in the subtree rooted at that directory.

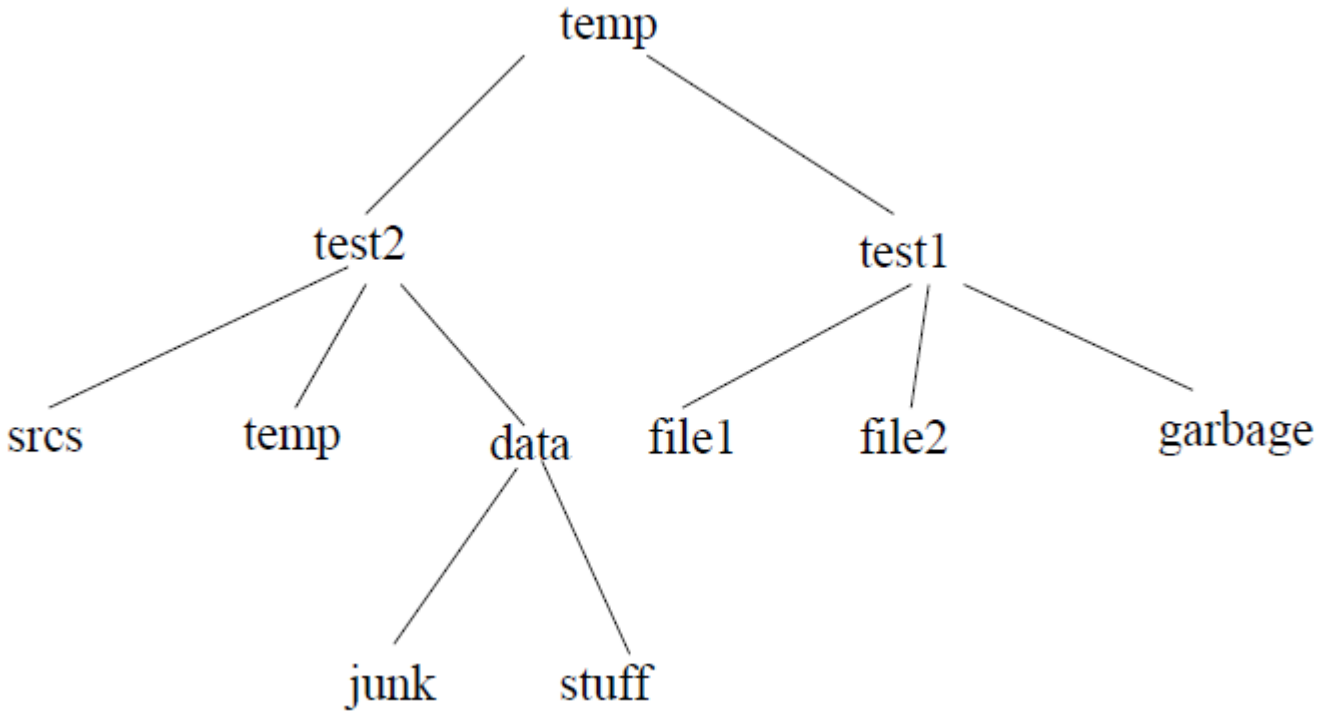
The pwd Command

Suppose we have a directory named *scratch*:

$ ls -iaR scratch

725 .
449 ..
753 temp
727 test1
728 test2
731 .
728 ..
733 junk
725 ..
731 data
748 srcs

732 stuff
727 .
725 ..
729 file1
730 file2
733 garbage
728 .
729 temp



How pwd Works

- The ".." in a directory always contains the i-number of the parent directory; but, the root of the file system has no parent, thus, in the root directory, "." and ".." have the same i-number.
- 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, using *stat()*.
    2. Change directory (*chdir()*) to the parent directory.
    3. Compare the i-number of the parent directory (now the current directory), to n. If they are equal, stop. Otherwise, find the name of the link with i-number n (*inum_to_name()*) and append it 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.

You can find code example for pwd.v1 in the Book Chapter 3.

Duplicate I-node Numbers and Cross-Device Links

- If another FS mounted in the tree, taking i-node numbers alone will cause errors.
- The kernel must deal with many files with possibly the same i-number, since i-numbers are unique only within a single FS.
- On all modern UNIX systems, the i-node contains a member that stores the name of the device on which it is located.
- Therefore, kernel can distinguish i-nodes by the name of the device they are located on.
- *stat()* system call can be used to read device_id.

You can find code example for pwd.v2 in the Book Chapter 3.

Symbolic Links

- FSs depend on the uniqueness of i-numbers
- However, a user may need to make links, even if they span FSs.
- Most UNIX systems provide files called **symbolic links** (or soft links).
- A symbolic link contains a reference to the name of the file which it links.
- The *symlink()* system call (and the command *ln s)* creates a symbolic link instead of a hard link.

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 *symlink()* creates symbolic links.
- The *readlink()* obtains name of the file a symbolic link is pointing.
- The *lstat()* obtains a stat structure for a file that is a link itself.

Tree Walks

- Four different ways to visit all nodes in a subtree:

- Custom recursive function,
- Custom non-recursive function,
- *nftw()* POSIX library function;
- *fts()* function in systems including the 4.4BSD API.

Tree Walks

- grep, chmod, chown, rm, cp, and chgrp use fts() to perform their recursive tree traversals.
- The GNU version of the ls command, uses internal stacks and queues to recurse the tree.
- The GNU find command uses mutually recursive functions whereas the versions on BSD systems use the fts() functions.

Thanks…