# SYSTEM PROGRAMMING

From the book by STEWART WEISS

# Chapter 04

## 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, tcush, tcdrain, ioctl,*

# Files and Disk Control: Open Files

- *open()* system call returns a file descriptor
- *fopen()* C standard I/O library call returns a FILE pointer.
- Either way, a scalar object (i.e., a small integer or a pointer) returned.
- It is associated with a kernel data structure that allows access to the file.
- Aside from the file pointer, the information it contains:
- whether the file is open for reading, writing or appending,
- whether the I/O is buffered or unbuffered,
- whether the access is exclusive or other processes also access the file.
- and some other information required by the kernel.

# Files and Disk Control: Open Files

- Many attributes of the connection can be changed by the process.
- Each process has a table called the open file table.
- 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.

# Files and Disk Control: Open Files

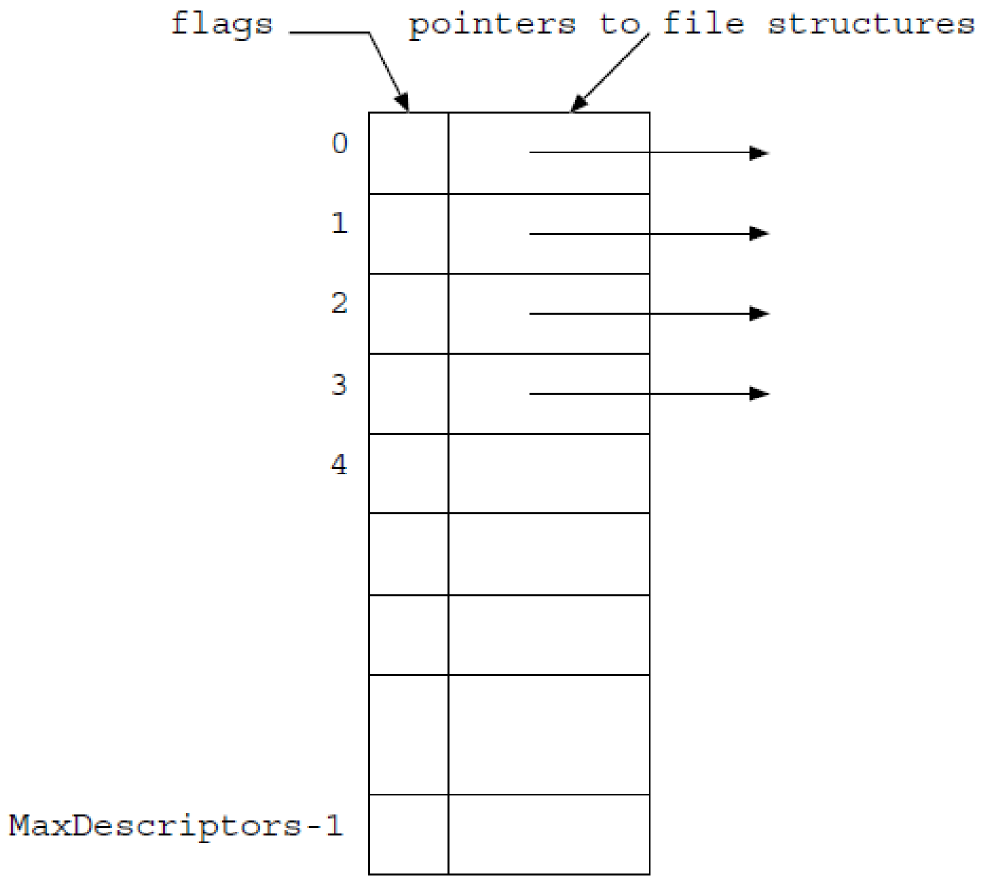
Recall that every process is given three file descriptors when it is created:

0: standard input

1: standard output

2: standard error.

These are the first three indices in this table, as shown in Figure 4.1.



# Using *fcntl()* to Control FD 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
- To modify the flags of an existing file structure:
    1. The process gets a copy of the current attributes of the connection from the kernel;
    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 copy.
- *fcntl()* a function that operates on open files and performs steps 1 and 3

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, … /* arg */);
```

# Parameters to *fcntl()*

- 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>*

- F_GETFL returns a copy of the set of flags;
- F_SETFL tells *fcntl()* to expect a third integer parameter that contains a new flag set to replace the current one.

- 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.
- The masks are defined in */usr/include/bits/fcntl.h*, which is included in the *<fcntl.h>* header file.

- The flags that can be changed after a file has already been opened are a subset of the file status flags. Some of them are:
- O_APPEND Append mode.
- 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_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.
- 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).

# Setting Flags Using *fcntl()*

```c
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;
```

```c
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;
```

# Controlling the Connection When Opening a 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);
```

# Possible Combinations of 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 |

# Device Files

- Every logical and physical devices associated with a device special file
- Logical devices are abstractions of real physical devices.
- Conventionally, all device files located in the */dev* directory.

# Accessing Devices Via Device Files

- To write a message to the terminal device /dev/pts/4, if permissions allow:

*$echo "Where are you?" > /dev/pts/4*

- *tty* displays the absolute pathname of the device file representing the terminal from which the command issued:

*$ tty*
*/dev/pts/4*

- The library function *ttyname()* returns pathname of the terminal device.
- The *ctermid()* standard I/O library function displays pathname of the controlling terminal.

# Accessing Devices Via Device Files

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

int main () {
  if (isatty(0))
    printf("%s\n", ttyname(0));
  else
    printf("not a terminal\n");
  return 0;
}
```

- This program outputs:

*$ mytty*
*/dev/pts/1*

- When input is redirected:

- *$ ls | mytty*
- *not a terminal*

# Device Drivers and the /dev Directory

- In the */dev* directory, ls –l outputs:

*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*

# Device Drivers and the /dev Directory

- In the */dev/pts* directory, ls –l outputs:

*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*

# Device Drivers and the /dev Directory

- The **c** indicates this is a character device
- **size** field consists of a pair of numbers.
- The first and second numbers are the major and minor device numbers.
- 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 the particular instance of this type of device represented by the file, or the action associated with this particular interface to the device.

# Pseudo-Terminals

- A terminal is a hardware device that emulates the old Teletype machines.
- Terminals connected to computers via RS-232 lines, into terminal multiplexers
- Computers had device drivers to communicate with multiplexed terminals.
- The terminal drivers had to control all aspects of the communication path, such as modem control, hardware flow control, echoing of characters, buffering of characters, and so on.

# Pseudo-Terminals

- A pseudo-terminal is a software-emulated terminal.
- A terminal window opened in a desktop environment such as Gnome/KDE, or an SSH client window are pseudo-terminals.
- 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.

# Character I/O Interfaces

- 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.

# Writing to a Device File

- As an exercise in writing to a device file, a simplified version of the write command coded.
- The write command writes messages to terminals.
- Please refer to section 4.3.8 of the book for the code example.

# Terminals and Terminal I/O

- Why we press ‘Enter’ key to send the typed characters to a program?
- Can a program suppress echoing of characters as they are typed?
- Can programs time-out while waiting for user input?
- Can programs override control sequences such as Ctrl-D and Ctrl-C? (‘vi’ and ‘emacs’ does!)
- Can a program get terminals’ row and column numbers dynamically and control how it wraps its output?
- Sometimes the ‘backspace’ key and sometimes the ‘delete’ key erases characters, and sometimes neither does. How?

# General Case: *copychars*

```
#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;
}
```

# The Problem

- In this program, although the main loop reads a single character and immediately writes that character, nothing gets written on the screen until ‘Enter’ key gets pressed.
- This is not due to C Library's streams doing buffering.
- The terminal is responsible for this.
- Somehow the characters typed are stored, but where, and how many can be stored before they are lost?

# The Solution

*$ stty -icanon; copychars*

*$ stty icanon*

- Once a character typed, it immediately gets echoed on the screen.
- The *stty* command allows us to control terminal characteristics.
- The commands above disabled buffering of input characters in the terminal.
- But, editing characters also seems to be disabled.

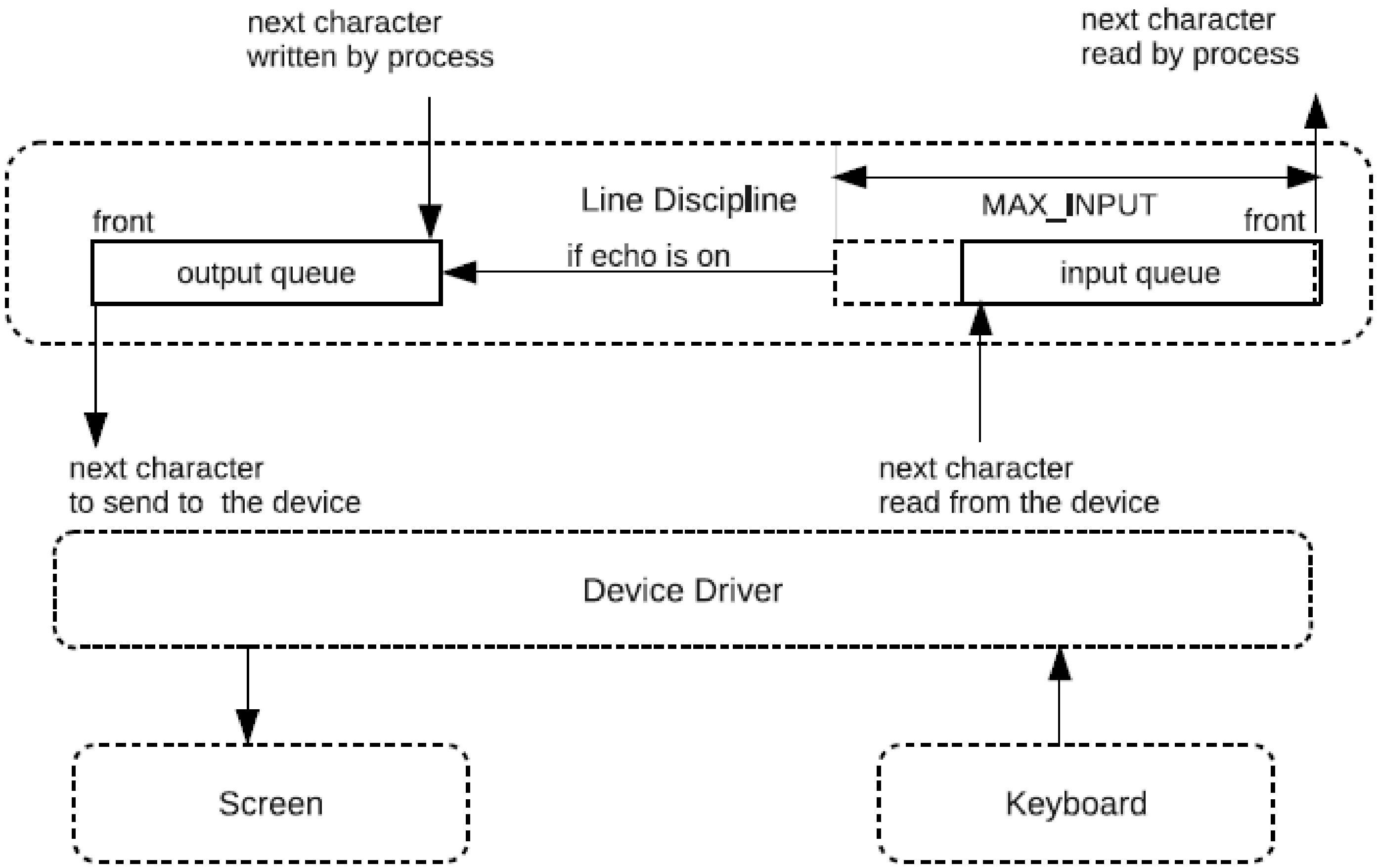
# Explanations

- The terminal driver pre-processes characters it receives (from the keyboard) and it sends (to the display device).
- By default, a terminal assembles the input into lines, processes special characters such as backspace, and delivers the input lines to the process.
- This mode of operation called canonical input mode.
- Terminals can be operated in various non-canonical input modes as well.
- In non-canonical modes, some part of this processing is turned off.
- Programs like ‘emacs’, ‘vi’, ‘less’ put the terminal into non-canonical mode.

# Terminal Devices: An Overview

- A terminal driver controls the behavior of a terminal device, and it consists of:
  - Terminal device driver.
  - Line discipline.
- The terminal device driver is a part of the kernel
- It transfers characters to and from the terminal device
- It talks directly with the hardware at one end, and the line discipline at the other.
- The line discipline does the processing of input and output.
- It maintains an input queue and an output queue for the terminal as illustrated in Figure 4.3.

# Figure 4.3: The Terminal Driver



# The Terminal Driver: Input /Output Queues

- When ‘echo’ is on, characters are copied from the input queue to the output queue.
- The size of the input queue is MAX_INPUT which is defined in <limits.h>.
- If the input queue fills, UNIX discards any extra characters.
- If the output queue fills, the kernel blocks the writing process until the queue has more room.

# Canonical Input Queue

- Another queue which is not shown in the Figure 4.3 is the canonical input queue.
- The canonical processing center is part of the line discipline.
- The line discipline uses an internal data structure to control the terminal.
- The kernel provides an interface to access this structure, and UNIX provides a command, stty, to access and modify attributes of the terminal stored in this structure.
- The name of the interface to this structure, in POSIX.1 compliant systems, is the termios struct.
- 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.

# The *stty* Command

*$ 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*

# The *stty* Command

*$ stty erase x*

*$ stty erase ^H*

*$ stty -echo*

*$ stty --file=/dev/pts/2 echo*

- To restore your settings:

*$ reset*

# The ‘*stty*’ Controls Terminal Settings

- **Special Characters** 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. Signal characters include Ctrl-C, the interrupt signal, and Ctrl-\, the quit signal.
- **Special Settings** control the terminal in general, e.g., its I/O speeds and dimensions, the rows, cols, min, and time values.
- **Input Settings** process characters coming from the terminal. Changing case, converting carriage returns to newlines, and ignoring various characters like breaks and carriage returns.
- **Output Settings** process characters sent to the terminal. Replacing tab characters by spaces, converting newlines to carriage returns, carriage returns to newlines, and changing case.
- **Control Settings** control character representation such as parity and stop bits, hardware flow control.
- **Local Settings** control how the driver stores and processes characters internally such as echo, processing erase and line-kill characters.
- **Combination Settings** define modes such as cooked mode or raw mode.

# Canonical vs Non-Canonical Mode

- In canonical mode, typed characters are processed and placed into the canonical input queue.

- To turn-off canonical mode:
- *$ stty -icanon*

- In non-canonical mode, they are delivered to the *read()* system call directly.

- To turn-on canonical mode:
- *$ stty icanon*

# Programming the Terminal Driver

- The *stty* command modify terminal settings from the shell
- To control the terminal settings in code, use the following POSIX compatible system calls:
- *tcgetattr()* and *tcsetattr()* gets and sets driver attributes.
- *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()* ushes 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 functions act on the line discipline; others act on the device driver settings.

- There is **an alternative** function:
- *ioctl()* can be used for controlling terminal settings, but it is not supported by the standard.
- The ioctl() function is necessary for controlling devices other than terminals.

# Modifying Terminal Attributes

- Retrieve the current settings to a *termios* structure, using *tcgetattr()*,
- Modify that structure locally,
- Write it back to the driver using the *tcsetattr()* call.

```
#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);
```

# The *termios* Structure

```
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
}
```

# FLAG MASKS

- The header file defines masks (see Figure 4.4) for each individual bits of *tcflag_t*.
- The *c_iflag* contains input processing flags.
- The *c_oflag* contains output processing flags.
- The *c_cflag* has control characteristics flags.
- The *c_lflag* has flags that define how characters are processed internally in the driver.
- The *c_cc* array stores control character assignments.
- This is where the map of erase key, backspace key, and so on, is stored.

| c_iflag | c_oflag | c_cflag | c_lflag |
|---|---|---|---|
| IGNBRK | OPOST | CSIZE | ISIG |
| BRKINT | ONLCR | CSTOPB | ICANON |
| IGNPAR | OLCUC | CREAD | ECHO |
| PARMRK | OCRNL | PARENB | ECHOE |
| INPCK | ONLRET | PARODD | ECHOK |
| ISTRIP | OFILL | HUPCL | ECHONL |
| INLCR | OFDEL | CLOCAL | NOFLSH |
| IGNCR | NLDLY | CRTSCTS | TOSTOP |
| ICRNL | CRDLY | CIBAUD | ECHOCTL |
| IUCLC | TABDLY | PAREXT | ECHOPRT |
| IXON | BSDLY | CBAUDEXT | ECHOKE |
| IXANY | FFDLY | | DEFECHO |
| IXOFF | VTDLY | | FLUSHO |
| IMAXBEL | | | PENDIN |
| IUTF8 | | | |

# Modify Single Bits of Flagsets

- MASK represents an arbitrary bit mask:

```
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:

```
flagset = flagset & ~ECHO;
```

# Turn echo off

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

int main (int argc, char* argv[]) {
  struct termios info, orig;
  char username[33];
  char passwd[33];
  FILE* fp;

  // get a FILE* to the control termina l -- 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;
}
```

# I/O Control Using *ioctl()*

- The *tcgetattr()* call accesses terminal driver attributes.
- Although most operations on devices can be achieved with the *tcgetattr()* & *tcsetattr()*, most devices also have some device-specific operations that do not fit into the general model.
- UNIX provides a more general-purpose device-control system call:
- *ioctl()* can be used to access and control any I/O device which has a device driver.

# I/O Control Using *ioctl()* *"Setecho" example*

```
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 ('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;
}
```

Thanks…