CH5: Connection Control Studying stty

Objectives

Ideas and Skills

- Similarities between files and devices
- Differences between files and devices
- Attributes of connections
- Race conditions and atomic operations
- Controlling device drivers
- Streams

System Calls and Functions

- fcntl, ioctl
- tcsetattr, tcgetattri

Commands

- stty
- write

Programming for Devices

- A computer has sources of data other than files and directories: modems, printers, scanners, mice, terminals, etc.
- Let's look at similarities and differences between files and devices.
- Let's see how to use those ideas to mange device connections.

Programming for Devices

- The chapter project is to write a version of the command stty.
- stty allows users to examine and modify the settings that control the connection to the keyboard and screen.

Devices are Just Like Files

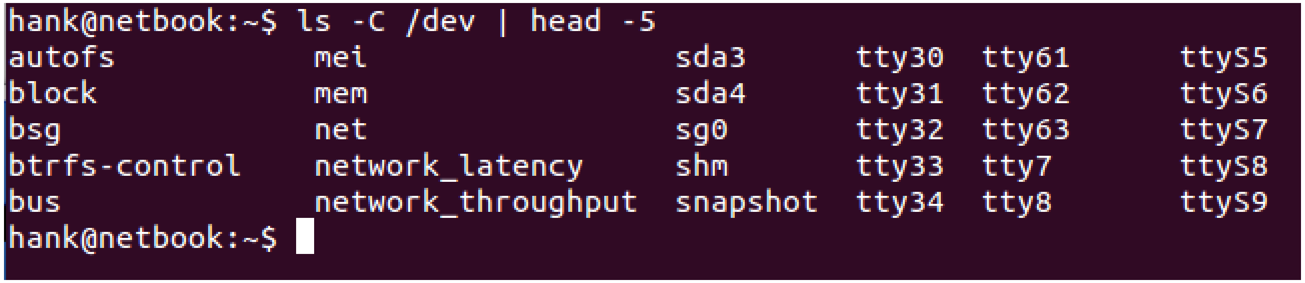
- To Unix, sound cards, terminals, mice, and disk files are the same sort of object.
- Every device is treated as a file on a Unix system.
- Devices have file names, inode numbers, owners, a set of permission bits, and a last-modified time, etc.

Devices are Just Like Files

- Much of what we've talked about regarding files applies to terminals and other devices.

Devices have file names

- Every device attached to a Unix machine has a file name.
- Files that represent devices are in the /dev directory.
- Here's a partial list of devices on my machine:



```
hank@netbook:~$ ls -C /dev | head -5
autofs           mei                 sda3      tty30  tty61      ttyS5
block            mem                 sda4      tty31  tty62      ttyS6
bsg              net                 sg0       tty32  tty63      ttyS7
btrfs-control    network_latency     shm       tty33  tty7       ttyS8
bus              network_throughput  snapshot  tty34  tty8       ttyS9
hank@netbook:~$ 
```

Devices have file names

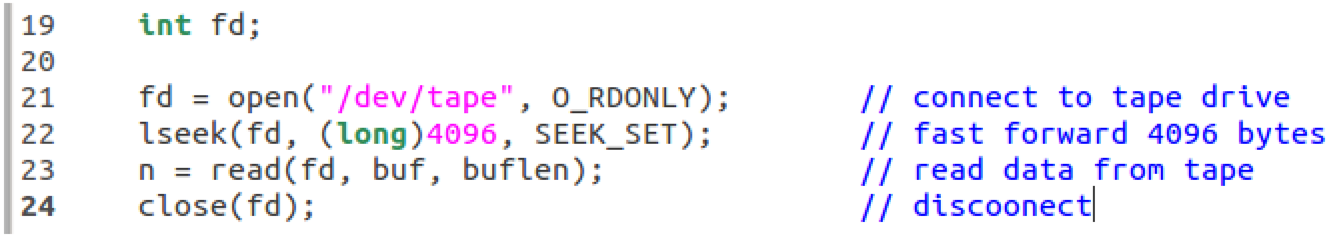
- There are several types of devices shown.
- sda* files are partitions on serial drives.
- tty* files are terminals.
- Programs get keystrokes by reading those files and send data to terminal screens by writing to those files.

Devices have file names

- A process can play a sound by writing bytes to the device file.
- A process can open /dev/mouse to read mouse clicks and changes in position.

Devices and System Calls

- Additionally, devices support all the file-related system calls: *open, read, write, lseek, close*, and, *stat*.
- For example, to read from a tape drive:



```
int fd;

fd = open("/dev/tape", O_RDONLY);       // connect to tape drive
lseek(fd, (long)4096, SEEK_SET);        // fast forward 4096 bytes
n = read(fd, buf, buflen);              // read data from tape
close(fd);                              // discoonect
```

- The same system calls for disk files work for devices. No other way to interact with them.

Some devices don't support all file operations

- This makes sense, when a mouse moves and clicks it sends bytes to the system.
- A program can read these bytes, but what does it mean to write to a mouse?
- It's nonsensical, so /dev/mouse doesn't support the write system call.

Example: Terminals Are Like Files

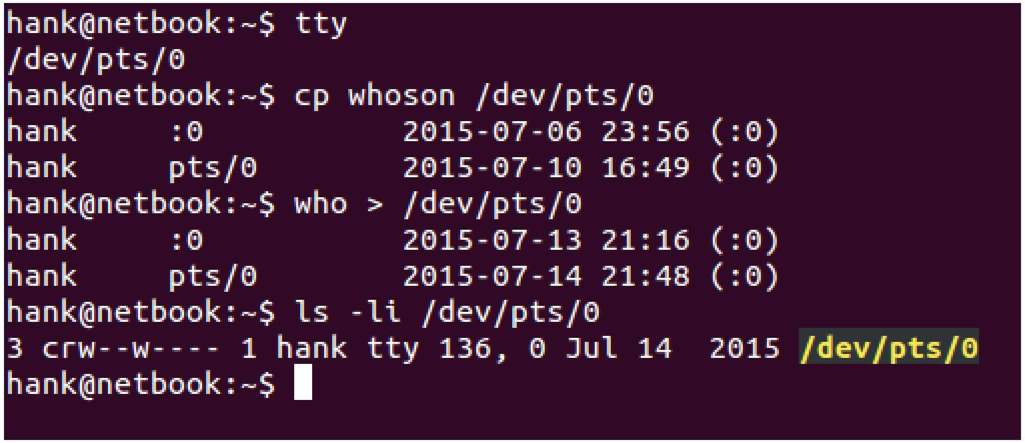
- Much user input comes from terminals.
- Files tty* represent terminals.
- A terminal is anything that acts like a classic keyboard and display unit.
- telnet or ssh connections behave like terminals

Example: Terminals Are Like Files

- The essential components of a terminal:
  --a source of character input from the user
  --a display unit for output to the user.
  - could be a monitor, generate braille, or even speech.
- The tty command tells us the name of the file that represents our terminal.

Example: Terminals Are Like Files

- Let's play with it...



```
hank@netbook:~$ tty
/dev/pts/0
hank@netbook:~$ cp whoson /dev/pts/0
hank     :0           2015-07-06 23:56 (:0)
hank     pts/0        2015-07-10 16:49 (:0)
hank@netbook:~$ who > /dev/pts/0
hank     :0           2015-07-13 21:16 (:0)
hank     pts/0        2015-07-14 21:48 (:0)
hank@netbook:~$ ls -li /dev/pts/0
3 crw--w---- 1 hank tty 136, 0 Jul 14  2015 /dev/pts/0
hank@netbook:~$ 
```

- tty says my terminal is attached to the file called /dev/pts/0.

Example: Terminals Are Like Files

- We can use any file commands and operations with that file: cp, output redirection with >, mv, ln, rm, cat, ls, etc.

Properties of Device Files

- Devices files have most disk file properties.
- The above ls output shows /dev/pts/o has inode 3, permission bits rw--w----, 1 link, owner hank, group tty, last modified Jul 14, 2015.
- The file type c indicates the file is really a device that transfers character by character.
- File size looks weird, there is an expression 136, 0 instead of number of bytes. What?

Device Files and File Size

- A device file is a connection, not a container like a disk file.
- The inode of a a device file doesn't store a file size and storage list, it stores a pointer.
- This pointer points to a subroutine in the kernel.
- Subroutines in the kernel that get data in and out of a device is called a *device driver*.

Device Files and File Size

- In the /dev/pts/0 example, 136 and 0 are called the *major number* and the *minor number* of the device.
- The major number specifies which subroutine handles the actual device.
- The minor number is passed to this subroutine.

Device Files and Permission Bits

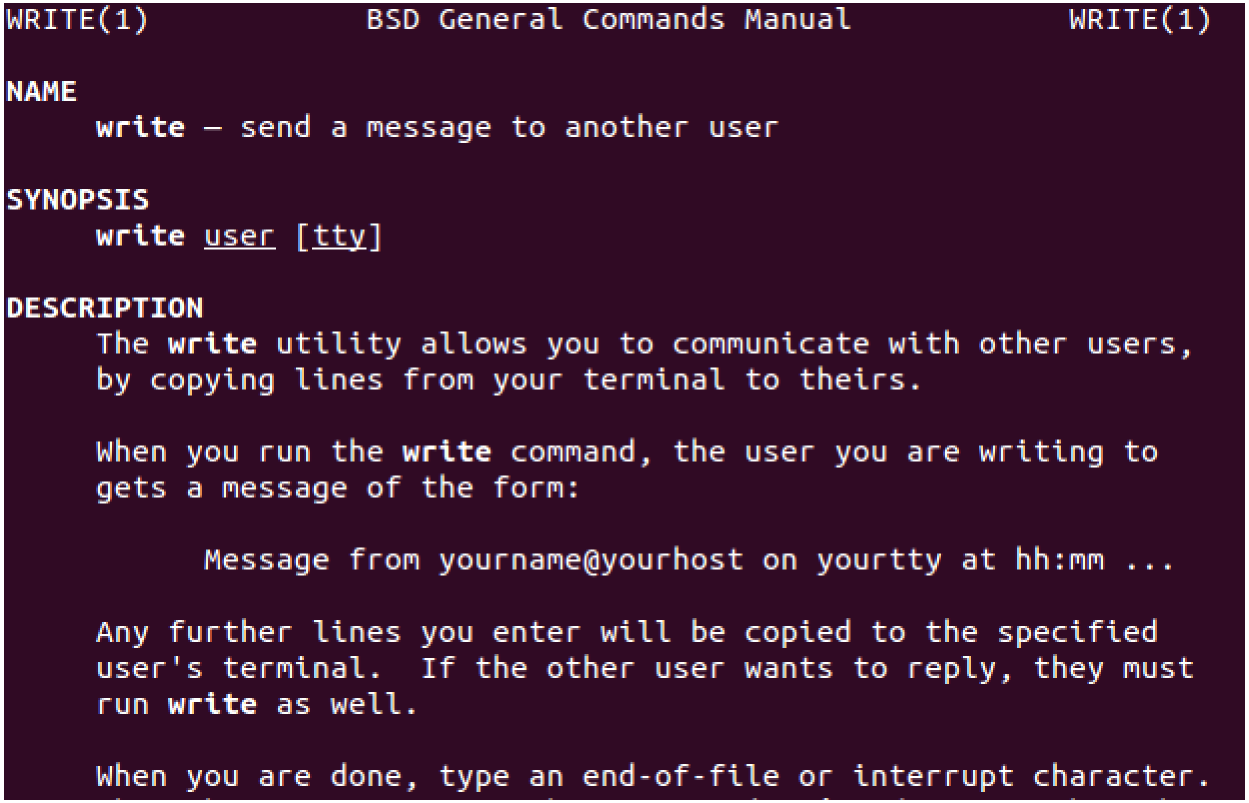
- Write permission means being able to write data to a device.
- In the above example, the owner of the file /dev/tty/0 is allowed to read and write from the terminal.
- The group is allowed to write to the terminal.
- Everyone else can't do anything.

Device Files and Permission Bits

- If users other than the owner of the file can read /dev/pts/0, other people can read characters typed at that keyboard. (keylogger anyone?)
- However, writing to other people's terminals is the purpose of the *write* command.

Writing write

- Before instant messaging and chat rooms, users chatted with friends at other terminals with the write command.



```
WRITE(1)               BSD General Commands Manual              WRITE(1)

NAME
     write — send a message to another user

SYNOPSIS
     write user [tty]

DESCRIPTION
     The write utility 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.
```

Writing write

- Let's look at the sample code, write0.c.
- It doesn't send the “Message from...” intro and requires the file name for the terminal, not the other person's user name.
- Take a look at the special features required to connect your keyboard to someone else's screen: there are none.
- It just writes lines from one file to another.

Device Files and Inodes

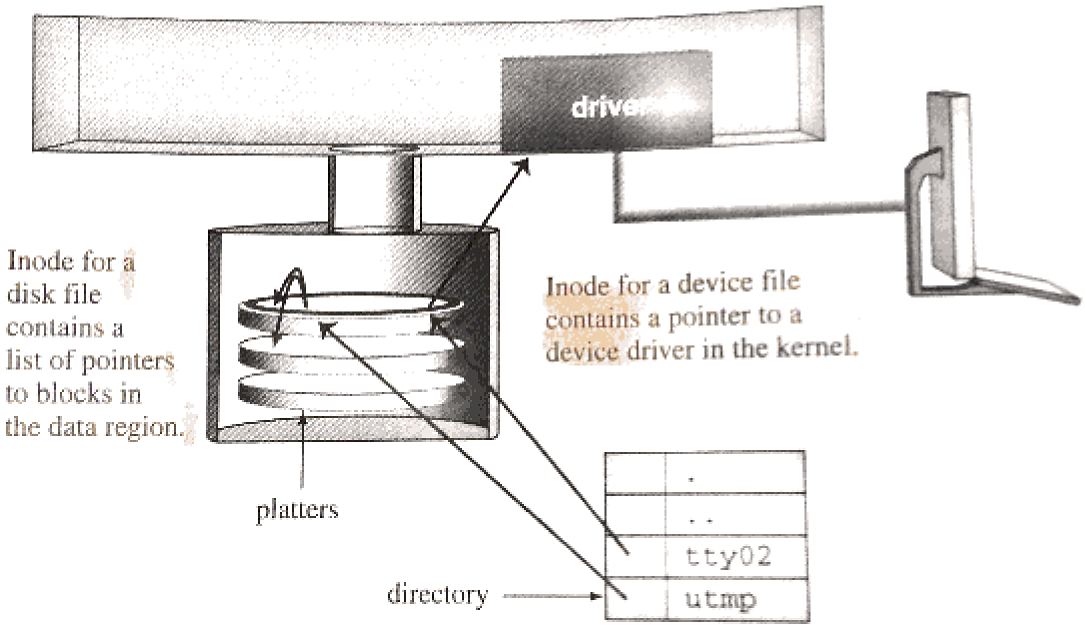

driver
Inode for a disk file contains a list of pointers to blocks in the data region.
Inode for a device file contains a pointer to a device driver in the kernel.
platters
directory
.
..
tty02
utmp

FIGURE 5.1

Inode points to data blocks or to driver code.

Device Files and Inodes

- Nothing in a directory tells you which names represent disk files and which names represent devices.
- The distinction is at the inode level.
- inodes can be a disk-file inode or a device-file inode.

Device Files and Inodes

- Disk-file inodes have pointers to data holding disk blocks.
- Device-file inodes have a pointer into a table of kernel subroutines.
- The *major number* tells where to find code that gets data from the device.

Devices Are Not Like Files

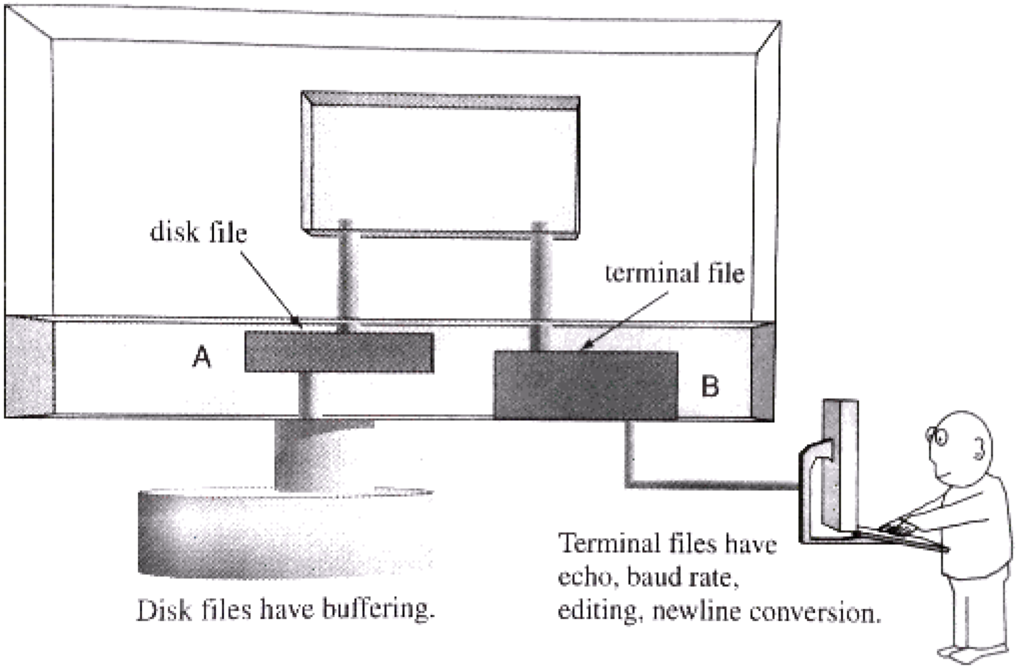


FIGURE 5.2

A process with two file descriptors.

Devices Are Not Like Files

- A to a disk file is different than a connection to a terminal.
- Connections to disk files involve kernel buffers, which allows data to be transferred in undetermined intervals.
- Connections to terminals are different: processes want data sent to terminals fast.

Devices Are Not Like Files

- Terminal connections also have properties.
- Serial connections have baud rate, parity, stop bits.
- Typed characters usually appear on the screen, but sometimes not.
- What if you type your password and the characters aren't echoed on the screen?

Devices Are Not Like Files

- Echoing of characters is not the keyboard, not the program, it's an attribute of the connection.
- Disk files don't have these attributes.

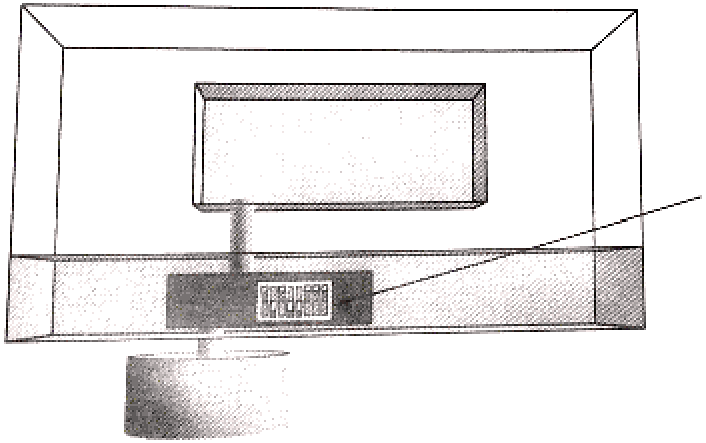
Attributes of Disk Connections

Attribute 1: Buffering

- A file descriptor can be depicted as two channels connected by a processing unit.
- The processing unit is kernel code and does buffering and other tasks.
- Inside the box are control variables that determine which steps the file descriptor performs.

Attribute 1: Buffering

- Here's what it looks like....



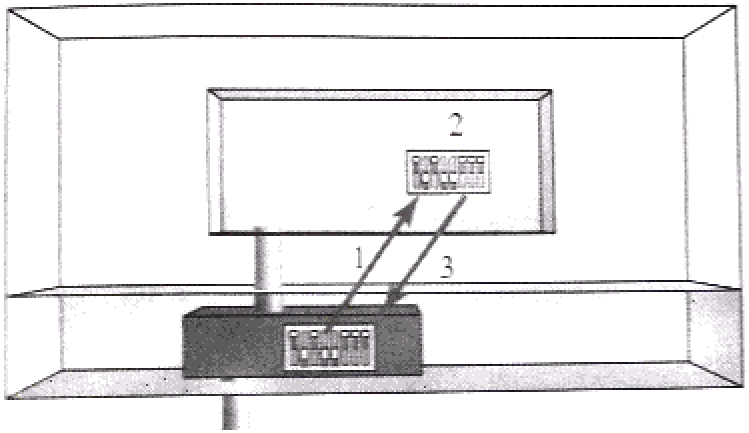
File descriptor settings control how the driver operates.

FIGURE 5.3

A processing unit in a data stream.

Attribute 1: Buffering

- Change the file descriptor action by changing those variables.
- For example, turn off disk buffering with this 3-step procedure:



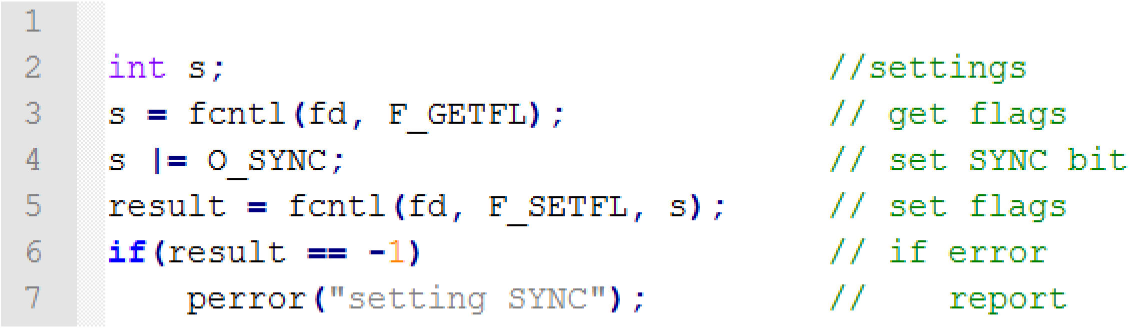
To change driver settings:
1. Get settings,
2. modify them
3. send them back.

FIGURE 5.4

Modifying the operation of a file descriptor.

Attribute 1: Buffering

- Here's the code for the three steps:



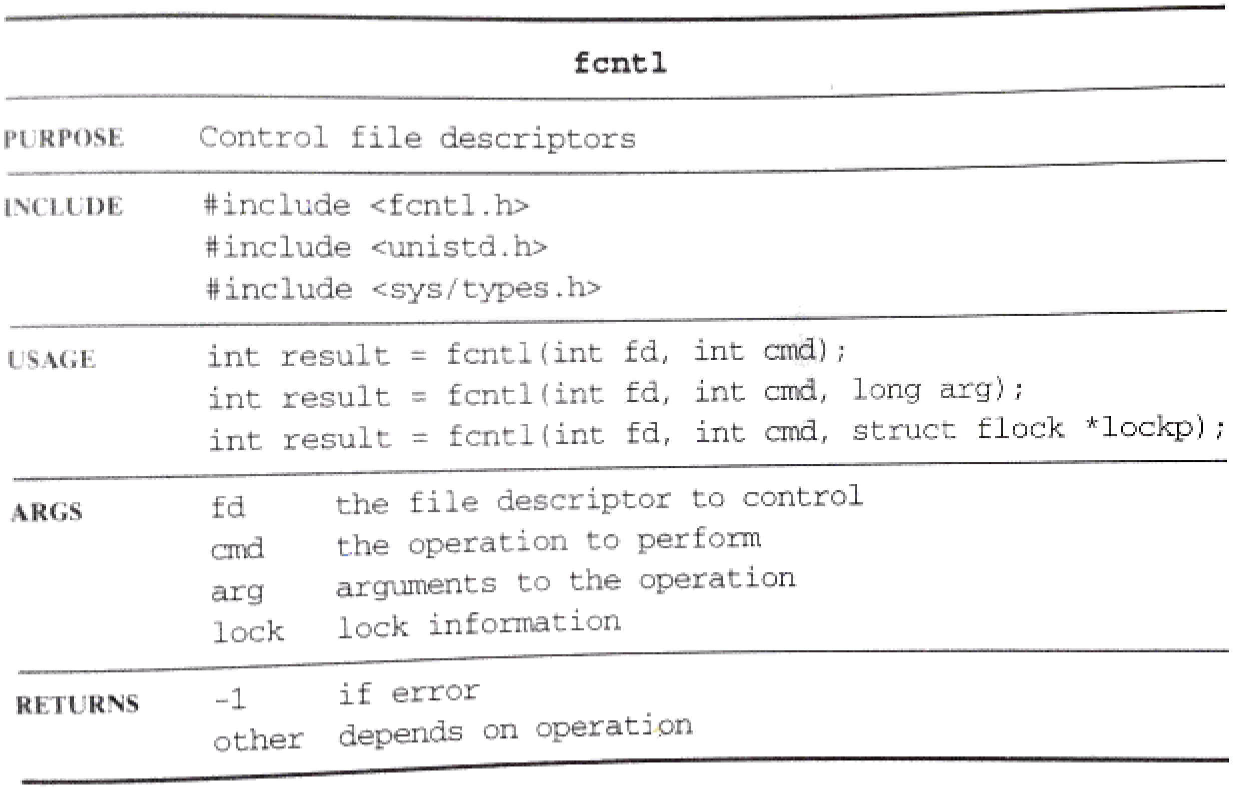
```

int s;                              //settings
s = fcntl(fd, F_GETFL);             // get flags
s |= O_SYNC;                        // set SYNC bit
result = fcntl(fd, F_SETFL, s);     // set flags
if(result == -1)                    // if error
    perror("setting SYNC");         //    report
```

- Attributes of a file descriptor are coded as bits in an integer.

Attribute 1: Buffering

- The *fcntl* system call lets you control a file descriptor by reading/writing that integer:



fcntl

PURPOSE	Control file descriptors	

INCLUDE	#include <fcntl.h>
	#include <unistd.h>
	#include <sys/types.h>	

USAGE	int result = fcntl(int fd, int cmd);
	int result = fcntl(int fd, int cmd, long arg);
	int result = fcntl(int fd, int cmd, struct flock *lockp);	

ARGS	fd	the file descriptor to control
	cmd	the operation to perform
	arg	arguments to the operation
	lock	lock information
RETURNS	-1	if error
	other	depends on operation

Attribute 1: Buffering

- *fcntl* performs operation *cmd* on the open file specified by *fd*.
- *arg* represents an argument used by *cmd*.
- In the sample code, *F_GETFL* operation gets the current set of bits (*flags*).
- *s* stores the *flagset*.
- *bitwise or* (|=) turns on the *O_SYNC* bit

Attribute 1: Buffering

- This bit tells kernel *write* calls should return only when the bytes are written to the actual hardware, rather than the default action of returning when bytes are copied to the kernel buffer.
- We then send the revised settings back to the kernel.
- Specify the *F_SETFL* operation and pass the revised settings as argument 3.

Attribute 1: Buffering

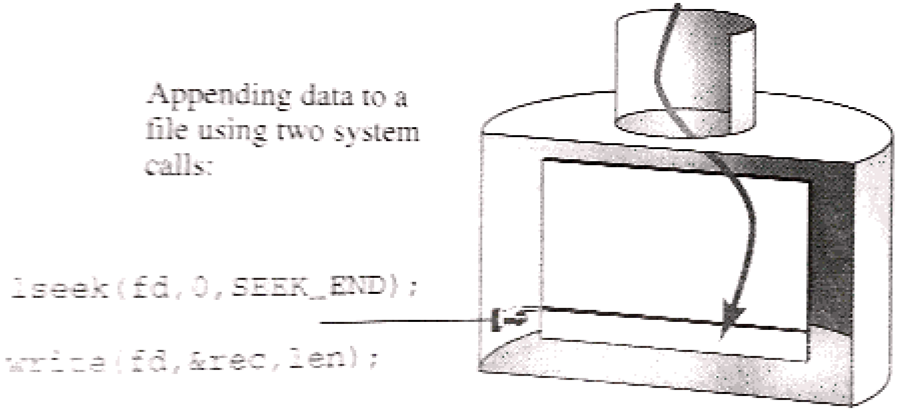
- This general procedure is how Unix reads and modifies data connection attributes.
- In summary: read settings, change settings, send settings back to kernel.

Attribute 2: Auto-append mode

- *Auto-append mode* is another file descriptor attribute.
- Useful for files written to by several processes at the same time.
- Consider the *wtmp* logfile.
- *wtmp* stores the login/logout history.
- When user logs in, the *login* program adds a record to the end of *wtmp*.

Attribute 2: Auto-Append Mode

- When a user logs out, the system adds a log-out record at the end of *wtmp*.
- ***Can't we just use lseek to add to the end?*** Consider...



```
lseek(fd,0,SEEK_END);
write(fd,&rec,len);
```

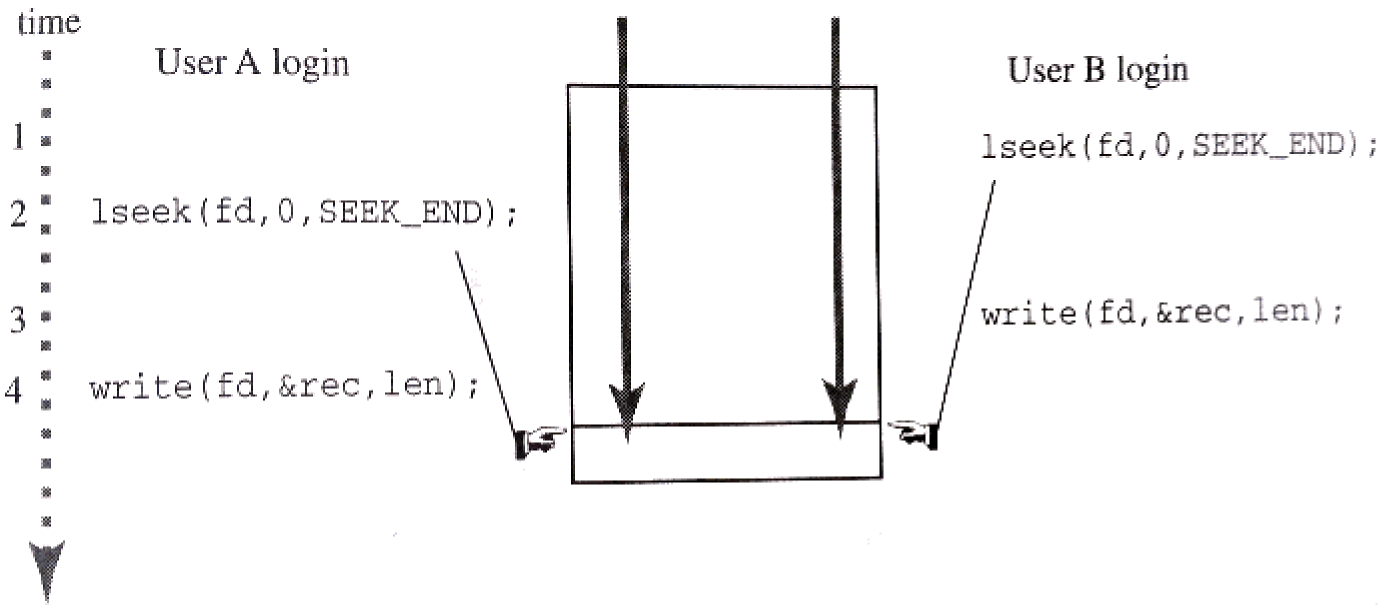
FIGURE 5.5

Appending with `lseek` and `write`.

Attribute 2: Auto-Append Mode

- *lseek* sets the current position to end, then *write* adds the record.
- ***What if two people log in at the same time?***

Attribute 2: Auto-Append Mode



```
lseek(fd,0,SEEK_END);
write(fd,&rec,len);
```

```
lseek(fd,0,SEEK_END);
write(fd,&rec,len);
```

FIGURE 5.6

Interleaved lseek and write = chaos.

Attribute 2: Auto-Append Mode

- *wtmp* is in the middle, time's arrow is on the left, and four time increments are shown
- Code for the login for user A is shown on the left, code for user B on the right.
- Now, Unix is a time-sharing system and this procedure requires two separate steps: *lseek* and *write*.

Attribute 2: Auto-Append Mode

- Watch the horror unfold in slow motion:

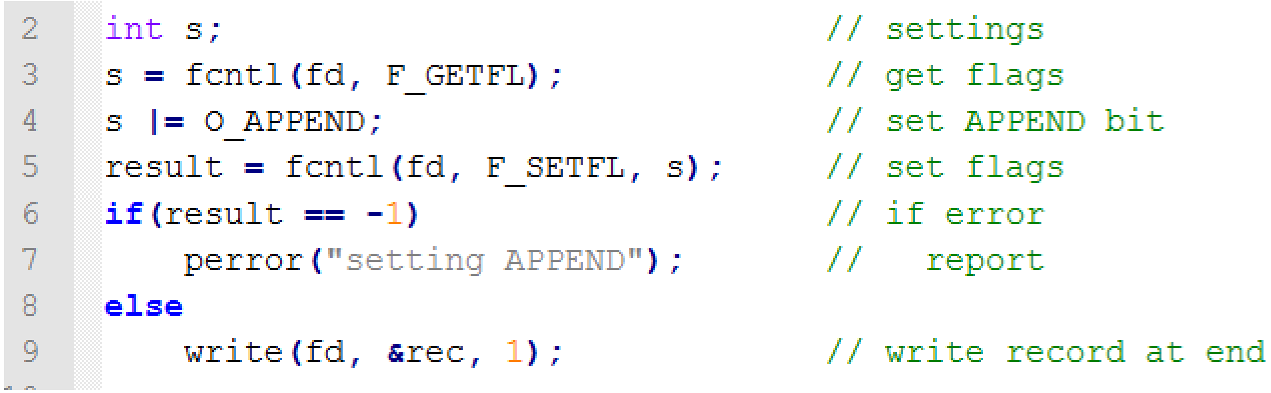
time 1: B's login process seeks to end of file
time 2: B's time slice ends, A's login process
seeks to end of file
time 3: A's slice ends, B's login process
writes to record
time 4: B's slice ends, A's login process
writes to record

Attribute 2: Auto-Append Mode

- So, B's login record is lost, overwritten by A's login process.
- This is known as a *race condition*.
- ***How can this be avoided?***
- Lots of ways, and race conditions are a critical problem.

Attribute 2: Auto-Append Mode

- For this case, the kernel provides auto-append mode.
- Setting the *O_APPEND* bit for a file descriptor, each call to write automatically includes lseek to the end of the file.



```
int s;                              // settings
s = fcntl(fd, F_GETFL);             // get flags
s |= O_APPEND;                      // set APPEND bit
result = fcntl(fd, F_SETFL, s);     // set flags
if(result == -1)                    // if error
    perror("setting APPEND");       //   report
else
    write(fd, &rec, 1);             // write record at end
```

Attribute 2: Auto-Append Mode

- ***Atomic operations***: this important term *race condition* is related to another important term, *atomic* *operation*.
- Calls to *lseek* and *write* are separate system calls: the kernel can interrupt the process between them.

Attribute 2: Auto-Append Mode

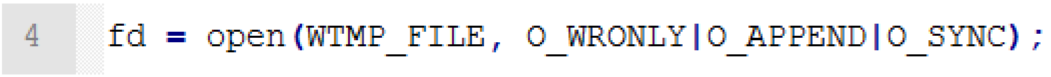
- With *O_APPEND*, the kernel combines *lseek* and *write* into an *atomic operation*.
- The two are joined into one indivisible unit.

Controlling File Descriptors with *open*

- There are several more attributes for a file descriptor than *O_SYNC* and *O_APPEND*.
- See man page for *fcntl* for a list.
- *fcntl* isn't the only way to set these attributes.
- They can be set when you open the file using the *open* system call.

Controlling File Descriptors with *open*

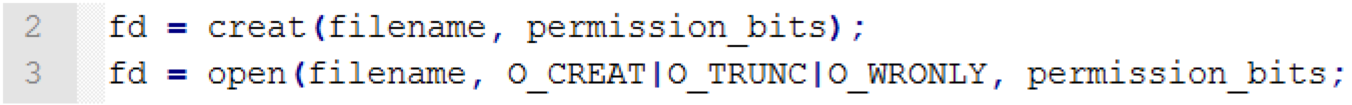
- An example:



```
fd = open(WTMP_FILE, O_WRONLY|O_APPEND|O_SYNC);
```

This opens the wtmp file for writing with *O_APPEND* and *O_SYNC* bits turned on.

- Another example:



```
fd = creat(filename, permission_bits);
fd = open(filename, O_CREAT|O_TRUNC|O_WRONLY, permission_bits;
```

These are the same.

Controlling File Descriptors with *open*

- Other flags *open* supports:

O_CREAT	Create the file if it doesn't exist
O_TRUNC	If file exist, truncate to length 0
O_EXCL	Intended to prevent two processes from creating the same file. If the named file already exists and O_CREAT is set, returns -1.

Summary of Disk Connections

- Kernel transfers data between disks and processes.
- Code in the kernel that does this has many options.
- A program can use *open* and *fcntl* to control the details of these transfers.

Summary of Disk Connections

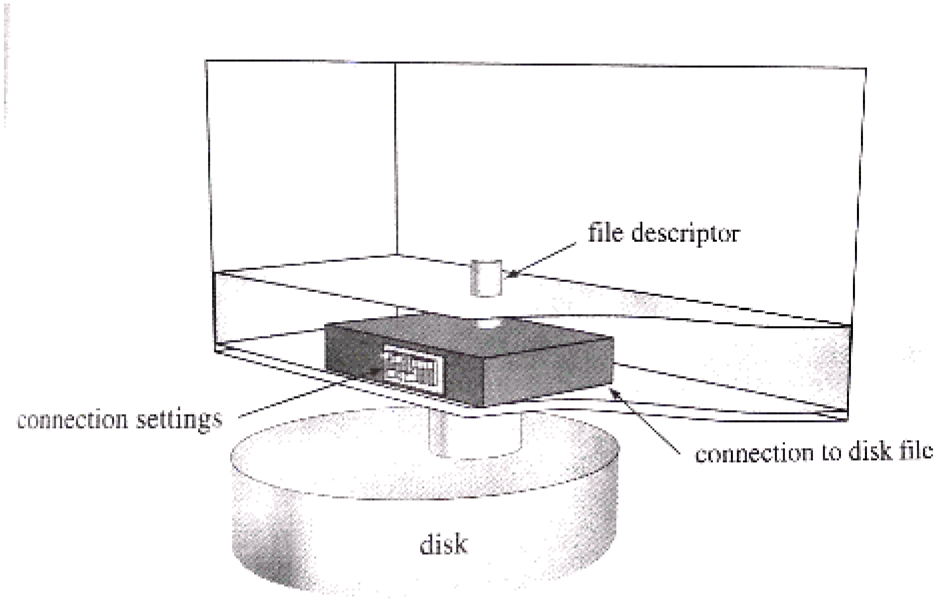


FIGURE 5.7

Connections to files have settings.

Attributes of Terminal Connections

Terminal I/O Not So Simple

- *open* creates a connection between a process and a terminal.
- Can use *getchar* and *putchar* to transfer bytes between device and process.

Terminal I/O Not So Simple

- The data stream abstraction:

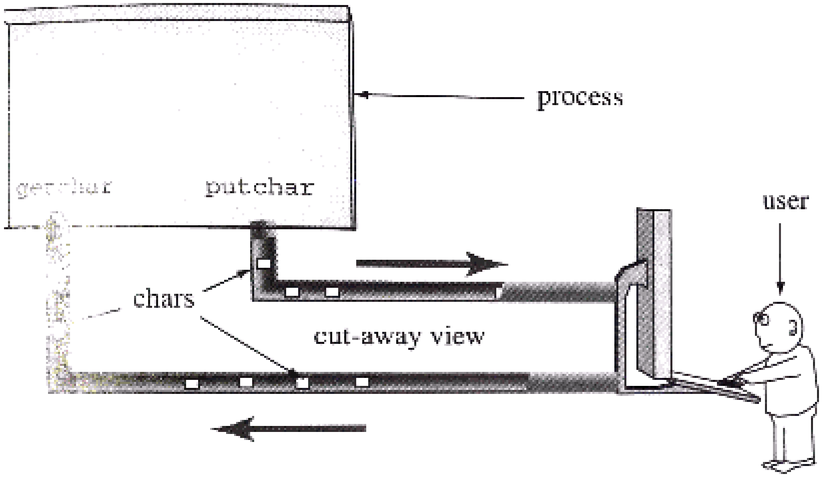
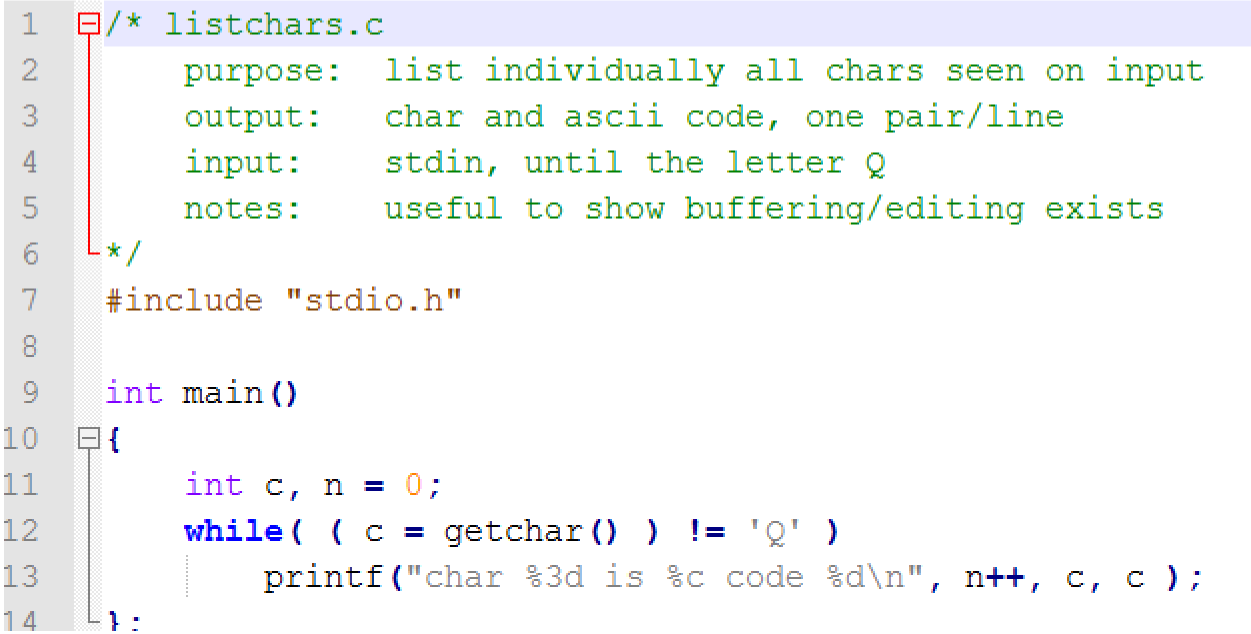


FIGURE 5.8

The illusion of a simple, direct connection.

Terminal I/O Not So Simple

- A simple experiment shows the model incomplete:

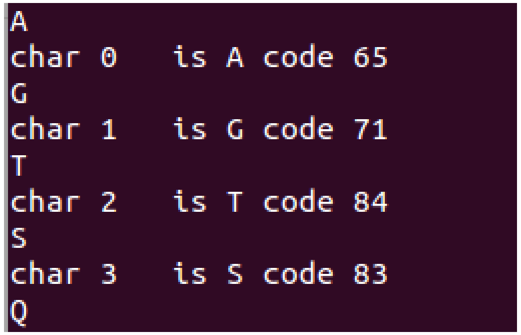


```c
/* listchars.c
    purpose:  list individually all chars seen on input
    output:   char and ascii code, one pair/line
    input:    stdin, until the letter Q
    notes:    useful to show buffering/editing exists
*/
#include "stdio.h"

int main()
{
    int c, n = 0;
    while( ( c = getchar() ) != 'Q' )
        printf("char %3d is %c code %d\n", n++, c, c );
};
```

Terminal I/O Not So Simple

- Processes characters one by one, reading a char then printing a count, the char itself, and its internal code. Compile and run, we get something like:



```
A
char 0   is A code 65
G
char 1   is G code 71
T
char 2   is T code 84
S
char 3   is S code 83
Q
```

Terminal I/O Not So Simple

- If character codes flowed directly from keyboard to *getchar,* we would see a response after each character.
- Instead, we only see the characters after hitting enter.
- Input appears buffered.

Terminal I/O Not So Simple

- Another thing, Return key usually sends ASCII 13, but we see ASCII 10.
- The *carriage return* character is replaced by the code for *line feed* or *newline.*
- One more thing, *listchars* sends a newline at end of each string.
- The code tells the cursor to go down one line.

Terminal I/O Not So Simple

- It doesn't tell it to go to the left margin, though.
- ASCII 13 tells the cursor to return to the left margin.
- This program shows there must be a processing layer somewhere in the middle of the file descriptor.

Terminal I/O Not So Simple

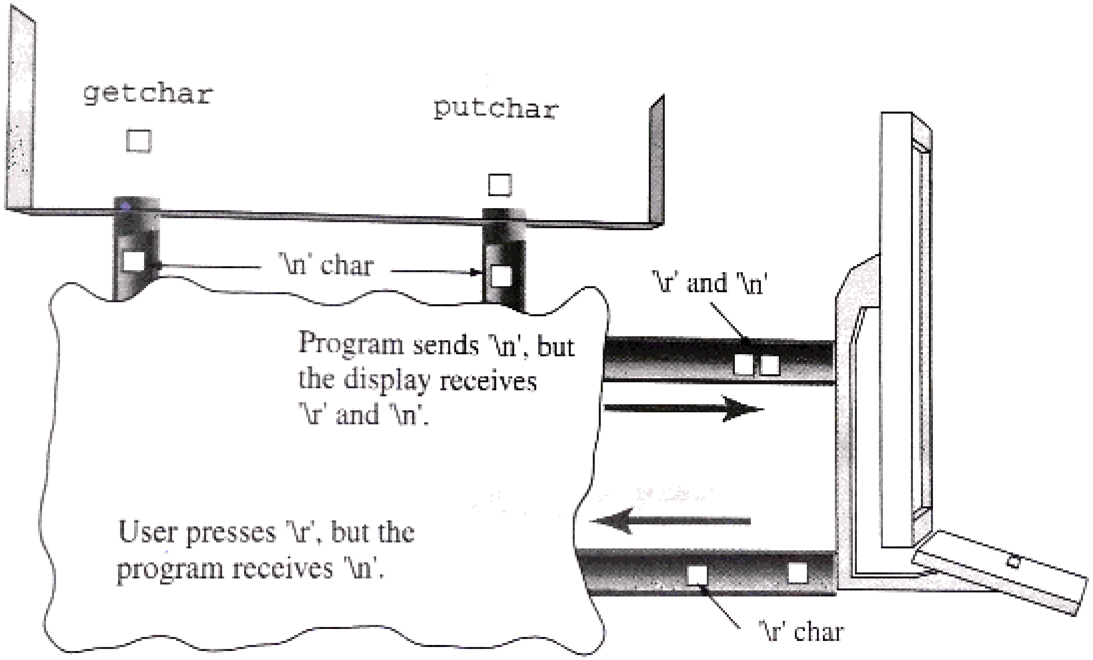

getchar
putchar
'\n' char
'\r' and '\n'
Program sends '\n', but the display receives '\r' and '\n'.
User presses '\r', but the program receives '\n'.
'\r' char

FIGURE 5.9

Kernel processes terminal data.

Terminal I/O Not So Simple

- This example shows 3 sorts of processing:
  1. The process receives no data until user hits Return.
  2. The user presses Return (ASCII 13), process sees newline (ASCII 10).
  3. Process sends newline, terminal receives Return-Newline pair

The Terminal Driver

- Here's what a connection between a terminal and a process looks like:

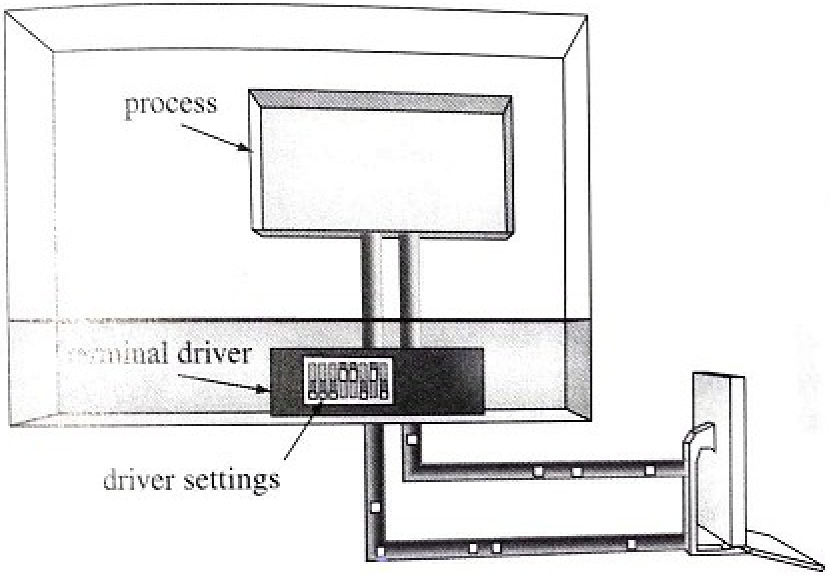


FIGURE 5.10

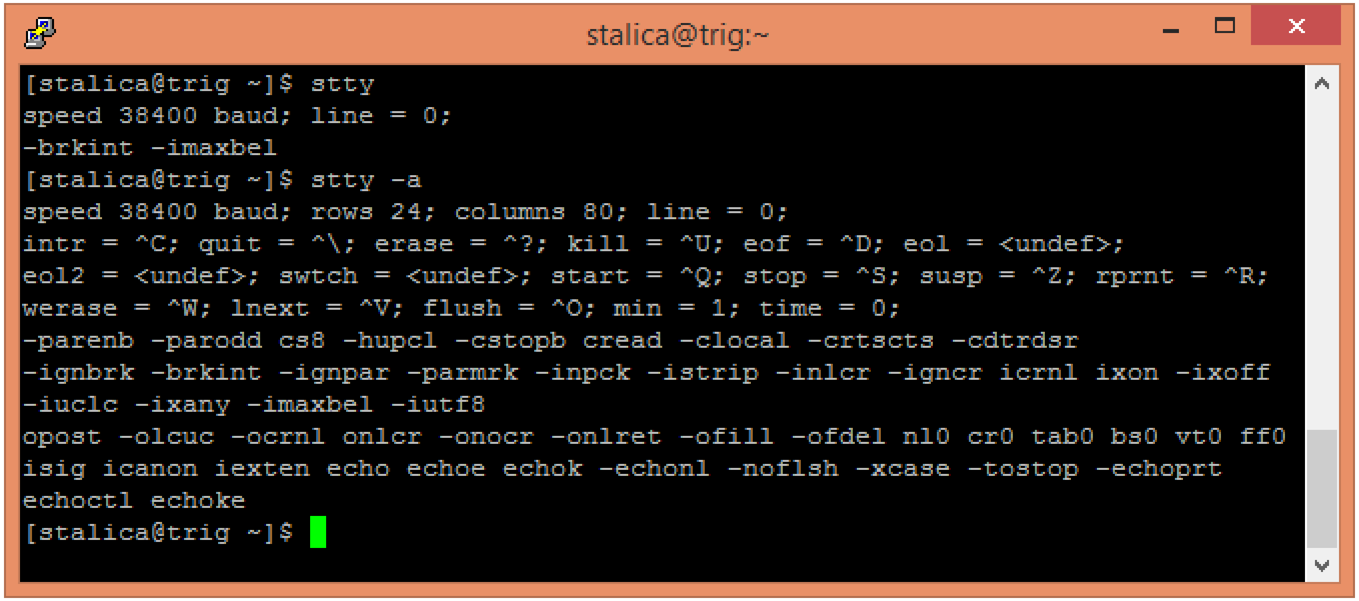
The terminal driver is part of the kernel.

The Terminal Driver

- The collection of kernel subroutines that process data moving between a process and the external device is the *terminal driver* aka *tty driver*.
- The driver has a lot of settings that control its operation.
- A process can read, modify, and reset these.

The *stty* command

- *stty* lets users read and change settings in the terminal driver.
- example



```
[stalica@trig ~]$ stty
speed 38400 baud; line = 0;
-brkint -imaxbel
[stalica@trig ~]$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts -cdtrdsr
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
[stalica@trig ~]$
```

The *stty* command

- The -all option lists many more settings.
- Some are variables with values, some are boolean.
- Some settings such as *icrnl, -olcuc,* and, *onlcr* are values that are on or off.
- *icrnl* means *Input: convert Carriage Return to New Line*

The *stty* command

- The – sign means turn off the operation:
- *-olcuc* means disable action for *Output: convert LowerCase to UpperCase*.

The *stty* command

- ***Using stty to Change Driver Settings***

  Some samples:

$ stty erase x	make 'X' erase key
$ stty -echo	type invisibly
$ stty erase @ echo	multiple requests

Programming the Terminal Driver: The Settings

- Four categories of tty driver operations:

Input What the driver does with chars coming from the terminal

Output What the driver does with chars going to the terminal.

Control how characters are represented : number of bits, parity, stop bits, etc

Local what the driver does while chars are inside the driver

Programming the Terminal Driver: The Settings

- Input processing could include converting cases, stripping off the high bit, converting carriage returns to new lines.
- Output processing could include replacing tabs with spaces, converting new lines to carriage returns, converting cases.
- Control settings include even parity, odd parity, and number of stop bits.

Programming the Terminal Driver: The Settings

- Local processing includes echoing keystrokes back to the user and buffering input.
- The driver also tracks keystrokes with special meanings.

  man for *stty* lists most of the settings and control characters.

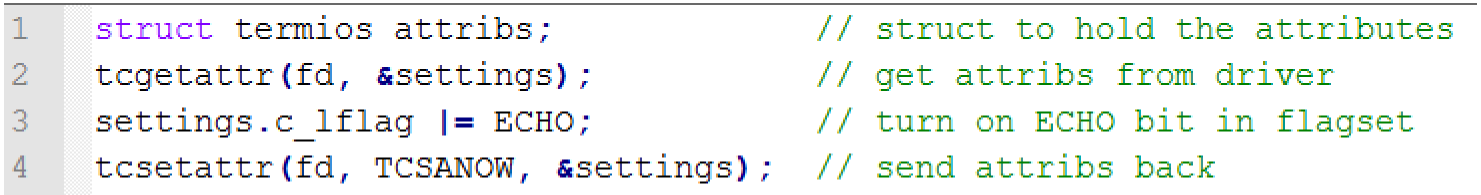
Programming the Terminal Driver: The Functions

- Changing settings similar to how you change settings for a disk file connection:

  1. Get the driver attributes
  2. Modify the attributes you need to change
  3. Send them back to the driver.

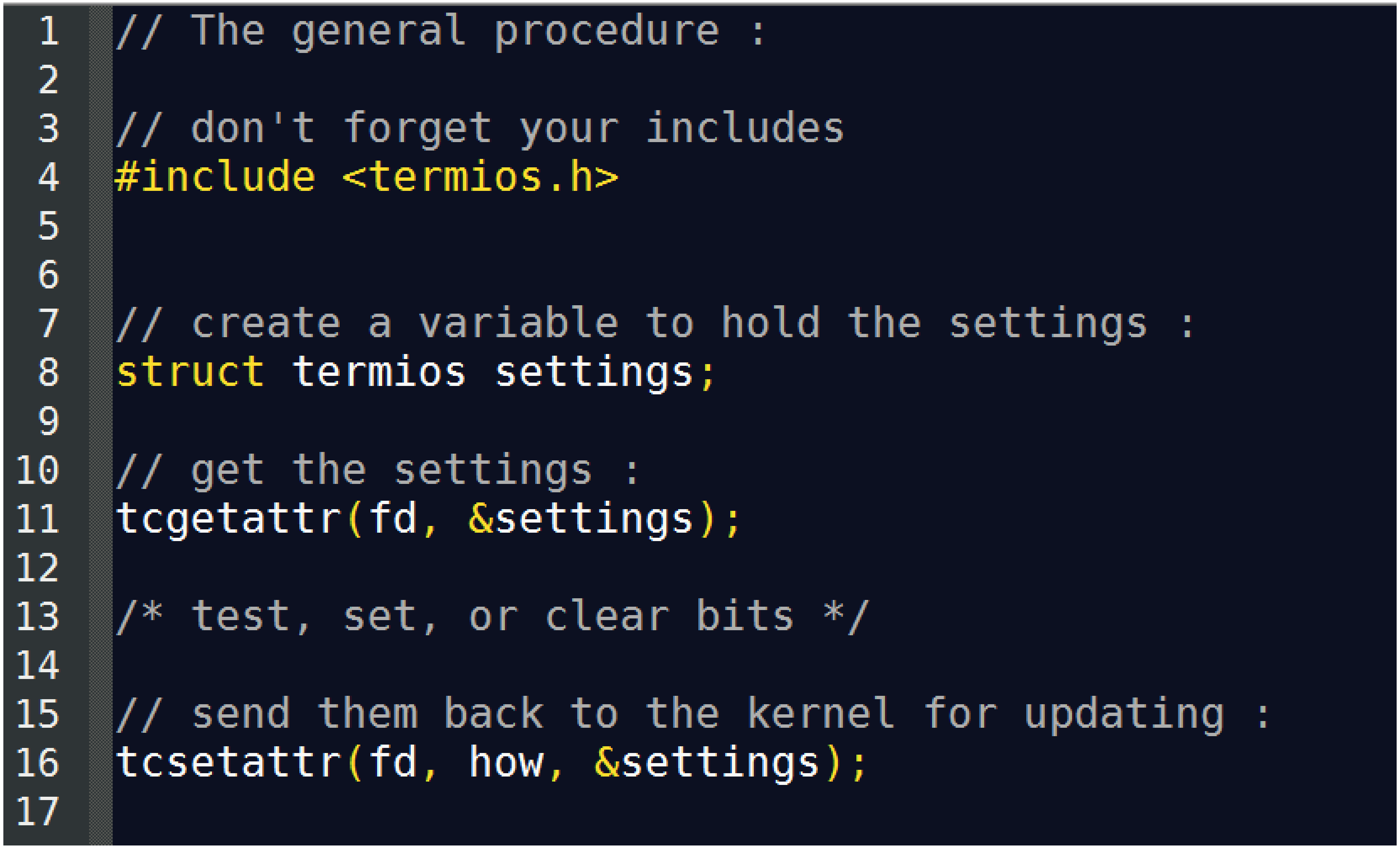
Programming the Terminal Driver: The Functions

- Sample code:



```
struct termios attribs;             // struct to hold the attributes
tcgetattr(fd, &settings);           // get attribs from driver
settings.c_lflag |= ECHO;           // turn on ECHO bit in flagset
tcsetattr(fd, TCSANOW, &settings);  // send attribs back
```

Programming the Terminal Driver: The Functions



```c
// The general procedure :

// don't forget your includes
#include <termios.h>


// create a variable to hold the settings :
struct termios settings;

// get the settings :
tcgetattr(fd, &settings);

/* test, set, or clear bits */

// send them back to the kernel for updating :
tcsetattr(fd, how, &settings);

```

Programming the Terminal Driver: The Functions

- The library functions *tcsetattr* and *tcgetattr* provide access to the terminal driver. Both transfer settings in a *struct termios*.

Programming the Terminal Driver: The Functions

tcgetattr

PURPOSE	Read attributes from tty driver	

INCLUDE	#include <termios.h>
	#include <unistd.h>	

USAGE	int result = tcgetattr(int fd, struct termios* info);	
ARGS	fd	file descriptor connected to a terminal
	info	pointer to a struct termios
RETURNS	-1	if error
	0	if success

Programming the Terminal Driver: The Functions

- *tcgetattr* copies current settings from the terminal driver associated to the file *fd* into struct pointed to by *info*.

Programming the Terminal Driver: The Functions

tcsetattr

PURPOSE	Set attributes in the tty driver	

INCLUDE	#include <termios.h>
	#include <unistd.h>	

USAGE	int result = tcsetattr(int fd, int when,struct termios* info);	
ARGS	fd	file descriptor connected to a terminal
	info	pointer to a struct termios
	when	when to change the settings
RETURNS	-1	if error
	0	if success

Programming the Terminal Driver: The Functions

- *tcsetattr* copies driver settings from the struct pointed to by *info* to the terminal driver associated to the open file *fd*.
- *when* tells tcsetattr when to update the driver settings.

Programming the Terminal Driver: The Functions

- Valid inputs for the *when* parameter :

TCSANOW Update settings immediately.

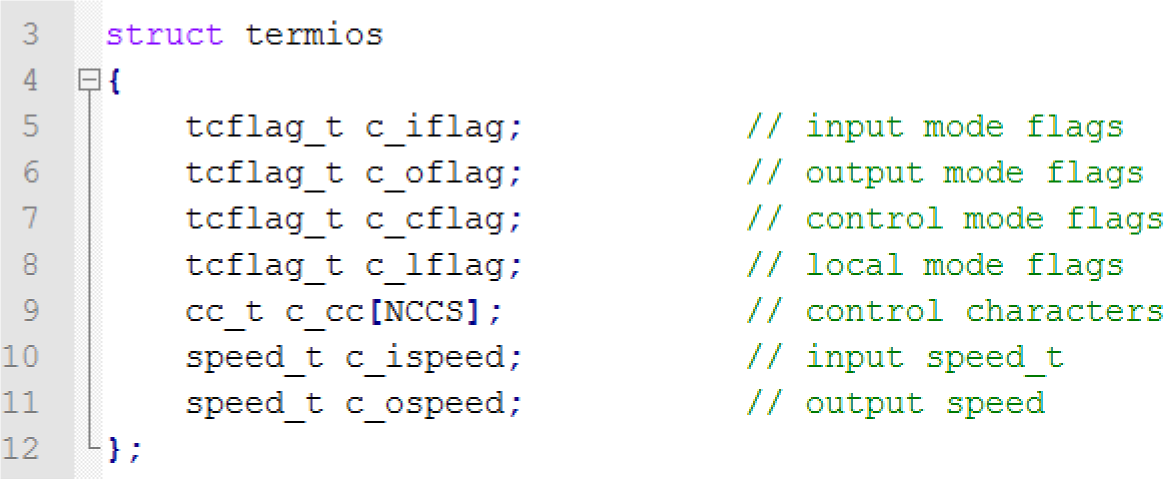
TCSADRAIN Wait until the output already queued in the driver has been transmitted to the terminal, then update the driver.

TCSAFLUSH

1. Wait until all output already queued in the driver has been transmitted.
2. Flush all queued input data.
3. Make the changes.

Programming the Terminal Driver: The Bits

- The *struct termios* data type contains several flagsets and an array of control characters. All Unix versions include



```
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_cc[NCCS];            // control characters
    speed_t c_ispeed;           // input speed_t
    speed_t c_ospeed;           // output speed
};
```

Programming the Terminal Driver: The Bits

- The individual bits in each flagset:

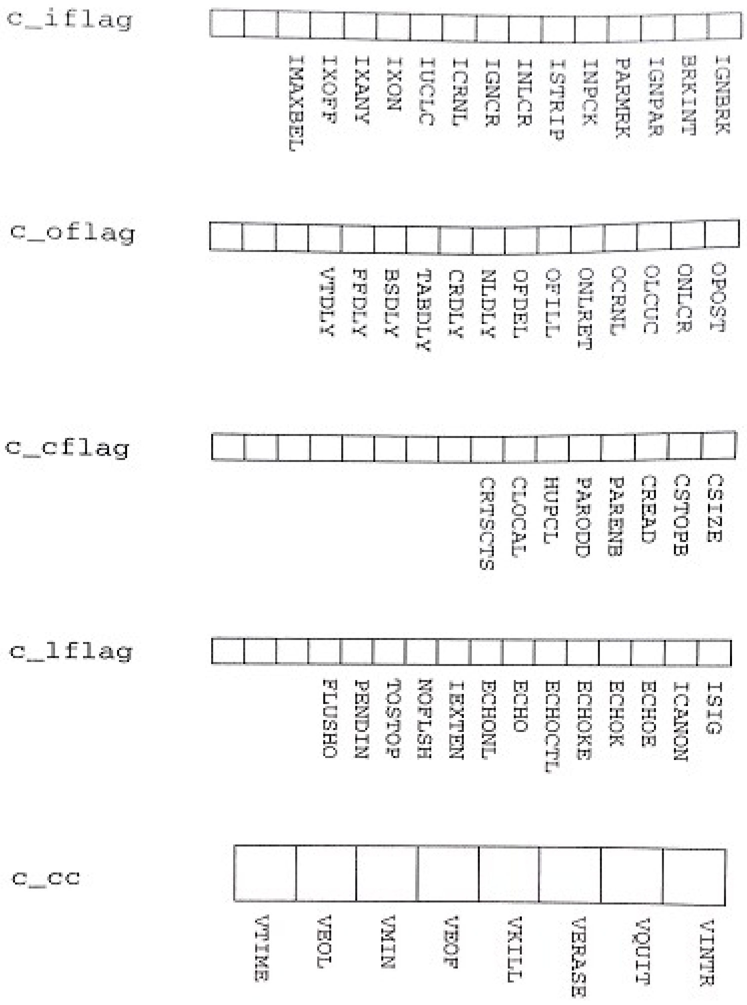


FIGURE 5.12

Bits and chars in

See

man 2 termios

for complete details.

Programming the Terminal Driver: The Bits

- The first four members depicted are flagsets.
- Each flagset contains bits for operations in that group.
- For example, the *c_iflag* member contains a bit for the value *INLCR*.
- *c_cflag* contains a bit for odd parity, *PARODD*.

Programming the Terminal Driver: The Bits

- These masks are all defined in *termios.h*.
- When reading values into a *struct termios*, all the values can be examined and modifed.
- The *c_cc* member is the array of *control characters*.
- Keystrokes that perform control functions are stored in here.

Programming the Terminal Driver: The Bits

- Each position in the array is defined by a constant in *termios.h*.
- For example, if a *struct termios* named attribs is defined, then

```
attribs.c_cc[VERASE] = '\b'
```

tells the driver to treat backspace as the erase character.

Programming the Terminal Driver: Bit Operations

- Each attribute is a bit in one of the flagsets.
- Masks for the attributes are defined in *termios.h*.
- To test the attribute, you mask the flagset with the mask for that bit.
- To enable, you turn on the bit; to disable, you turn it off.

Programming the Terminal Driver: Bit Operations

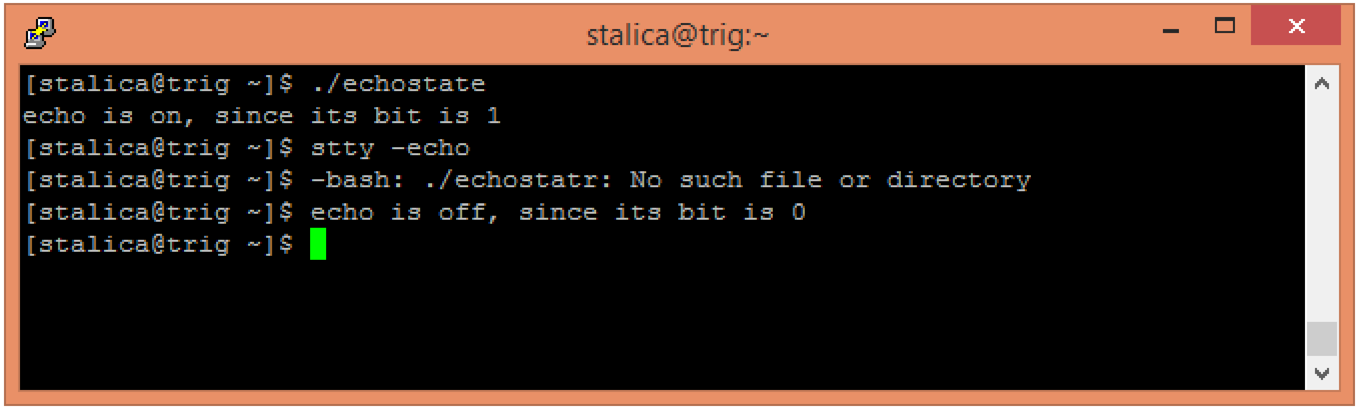
Action	**Code**
test a bit	if(flagset & MASK) . . .
set a bit	flagset |= MASK
clear a bit	Flagset &= ~MASK

Programming the Terminal Driver: Sample Programs

- Let's take a look at *echostate.c*, a program to show the state of the echo bit.
- This program reads terminal attributes for file descriptor 0
- This the file descriptor for standard input, usually attached to the keyboard.

Programming the Terminal Driver: Sample Programs

- Sample



```
[stalica@trig ~]$ ./echostate
echo is on, since its bit is 1
[stalica@trig ~]$ stty -echo
[stalica@trig ~]$ -bash: ./echostatr: No such file or directory
[stalica@trig ~]$ echo is off, since its bit is 0
[stalica@trig ~]$ 
```

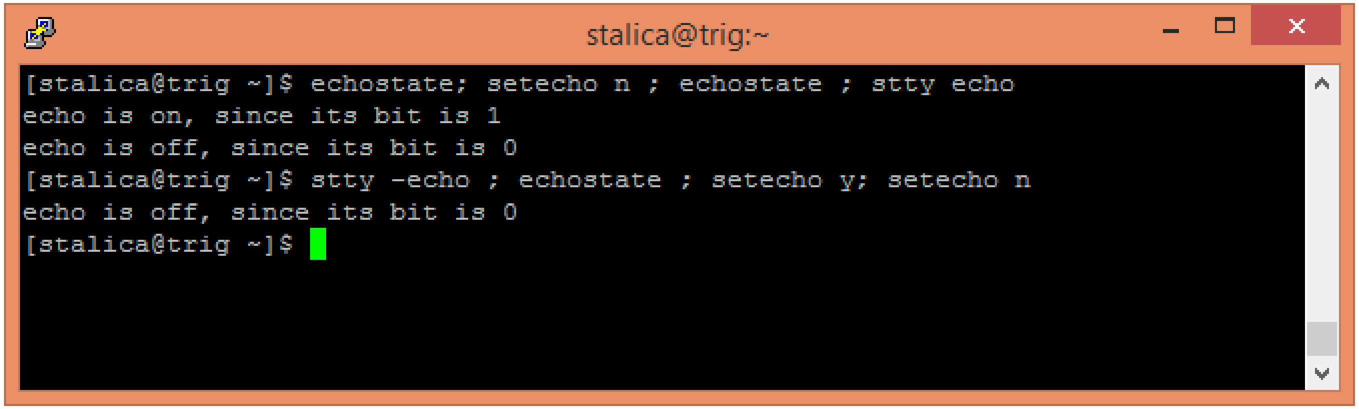
- This shows the command stty -echo disables keystroke echoing in the driver.
- Type two more commands, after that, but hey don't show on the screen, but the output does.

Programming the Terminal Driver: Sample Programs

- Let's look at another example program, *setecho.c*.
- This program turns keyboard echo on or off.
- If the command line argument begins with “y”, the echo flag for the terminal is turned on.
- Otherwise, it's turned off.

Programming the Terminal Driver: Sample Programs

- Sample Run:



```
[stalica@trig ~]$ echostate; setecho n ; echostate ; stty echo
echo is on, since its bit is 1
echo is off, since its bit is 0
[stalica@trig ~]$ stty -echo ; echostate ; setecho y; setecho n
echo is off, since its bit is 0
[stalica@trig ~]$
```

Programming the Terminal Driver: Sample Programs

- On the first line, we used setecho to turn off echoing.
- Then, we used stty to turn it back on.
- Driver and driver settings are stored in the kernel, not the process.
- One process can change driver settings, and a different one can read or change them.

Programming the Terminal Driver: Sample Programs

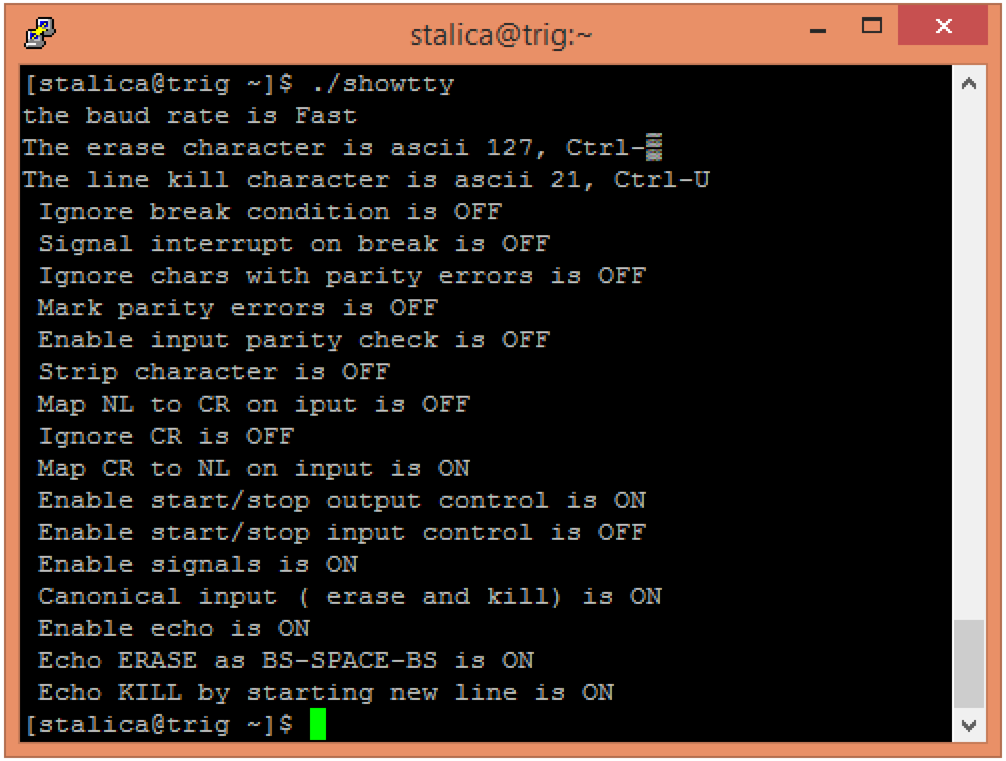
- One more example, let's look at *showtty.c*.
- We can leverage the previous program techniques to build a complete *stty* version.
- The *tty* driver contains 3 sorts of settings: special characters, numerical values, and bits.
- *showtty* contains functions to display each type.

Programming the Terminal Driver: Sample Programs

- *showtty* prints the current state, with some additional text, of attributes in the driver.
- This sample program uses a table of structs to simplify the code.
- A single function, *show_flagset* is passed an integer and a set of driver flags.

Programming the Terminal Driver: Sample Programs

- *show_flagset* loops through all the bits, testing and displaying their statuses.



```
[stalica@trig ~]$ ./showtty
the baud rate is Fast
The erase character is ascii 127, Ctrl-▒
The line kill character is ascii 21, Ctrl-U
 Ignore break condition is OFF
 Signal interrupt on break is OFF
 Ignore chars with parity errors is OFF
 Mark parity errors is OFF
 Enable input parity check is OFF
 Strip character is OFF
 Map NL to CR on iput is OFF
 Ignore CR is OFF
 Map CR to NL on input is ON
 Enable start/stop output control is ON
 Enable start/stop input control is OFF
 Enable signals is ON
 Canonical input ( erase and kill) is ON
 Enable echo is ON
 Echo ERASE as BS-SPACE-BS is ON
 Echo KILL by starting new line is ON
[stalica@trig ~]$
```

Summary of Terminal Connections

- Terminals are devices human beings use to communicate with processes.
- They have a keyboard a process reads characters from and a display the process sends characters to.
- Terminals are devices, so they appear as special files, usually in */dev*.

Summary of Terminal Connections

- Transfer and processing of data between the terminal and a process is handled by the terminal driver.
- The terminal driver is a part of the kernel.
- Kernel code provides buffering, editing, and data conversion.
- Programs can examine and modify the settings by calling *tcgetattr* and *tcsetattr*.

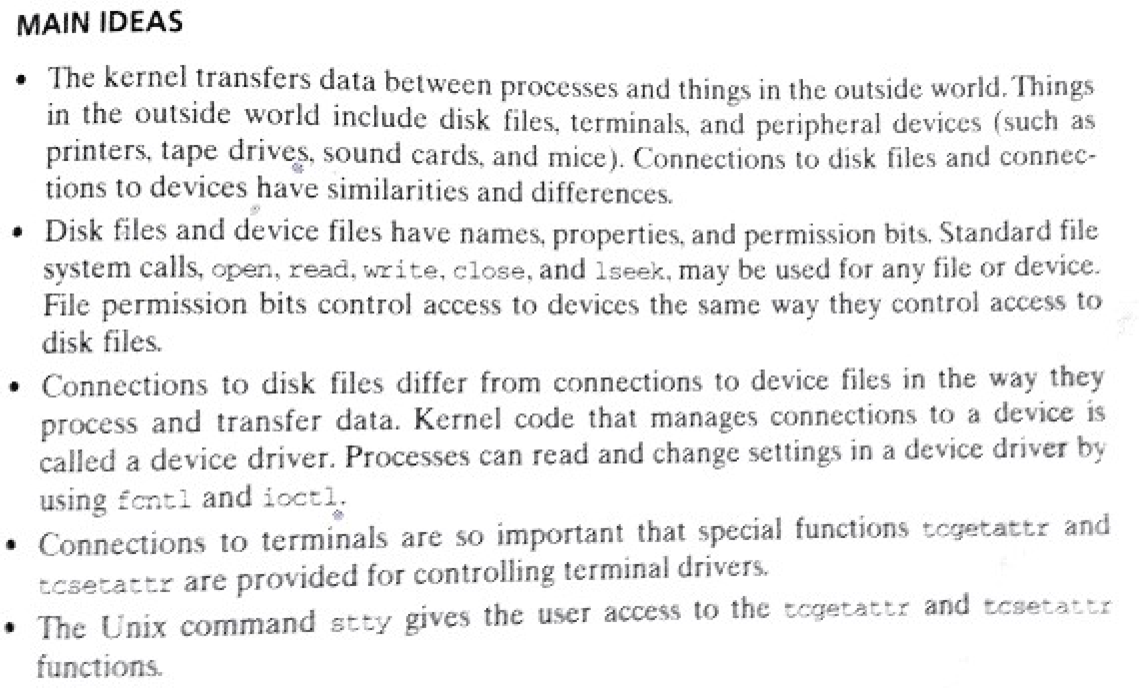
Programming Other Devices: *ioctl*

- Connections to other types of devices have other types of settings different than a terminal or disk file.
- Every device file will support the *ioctl* system call.
- See man 4 ioctl for an example of using ioctl for SCSI drives

Summary

Main Ideas

- summary



MAIN IDEAS

- The kernel transfers data between processes and things in the outside world. Things in the outside world include disk files, terminals, and peripheral devices (such as printers, tape drives, sound cards, and mice). Connections to disk files and connections to devices have similarities and differences.
- Disk files and device files have names, properties, and permission bits. Standard file system calls, `open`, `read`, `write`, `close`, and `lseek`, may be used for any file or device. File permission bits control access to devices the same way they control access to disk files.
- Connections to disk files differ from connections to device files in the way they process and transfer data. Kernel code that manages connections to a device is called a device driver. Processes can read and change settings in a device driver by using `fcntl` and `ioctl`.
- Connections to terminals are so important that special functions `tcgetattr` and `tcsetattr` are provided for controlling terminal drivers.
- The Unix command `stty` gives the user access to the `tcgetattr` and `tcsetattr` functions.