CH 6: Programming For Humans
Terminal Control and Signals

Objectives

Ideas and Skills

- Software tools vs. user programs
- Reading and changing settings of the terminal driver.
- Modes of the terminal driver
- Nonblocking input
- Timeouts on user input
- Introduction to signals: How Ctrl-C Works

System Calls

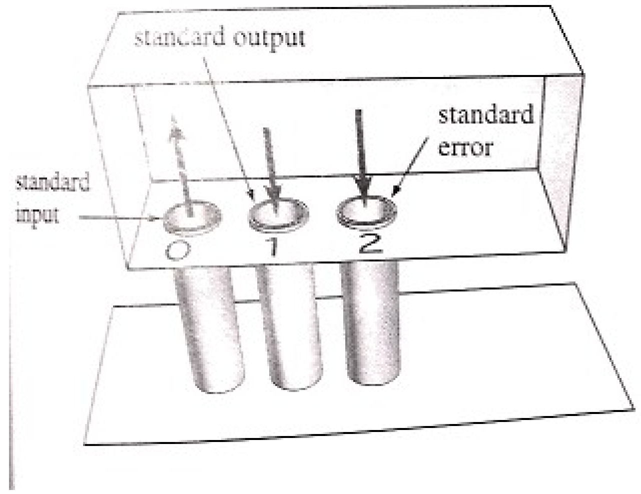
- fcntl
- signal

Software Tools vs Device-Specific Programs

Software tools: Read stdin or files, write to stdout

- Programs that see no difference between disk files and devices are called software tools.
- Example software tools: *who, ls, sort, uniq, grep, tr, du*.
- Software tools are based on the following model:

The three standard file descriptors



Fact: Most processes automatically have the first three file descriptors open. They do not need to call open() to make these connections.

FIGURE 6.1

The three standard file descriptors.

Software tools: Read stdin or files, write to stdout

- Software tools read bytes from standard input, perform processing, then writes a stream of bytes to standard output.
- A tool sends error messages, a stream of bytes, to standard error.
- We can connect these descriptors to files, terminals, mice, printers, pipes, etc.
- Tools don't care about sources and destinations.

Software tools: Read stdin or files, write to stdout

- Many tools also read from files named as arguments.
- Input and output can be easily attached to all sorts of connections:

```
$ sort > outputfile
$ sort x > /dev/lp
$ who | tr '[a-z]' '[A-Z]'
```

Device-Specific Programs: Control Device for Particular Application

- Other types of program are written to work with specific devices.
- Example, programs to control scanners, take digital pictures
- Generally, these programs primarily interact with terminals, designed for use by people.
- Terminal-oriented programs are referred to as *user programs*.

User Programs: A Common Type of Device-Specific Program

- Examples: vi, emacs, pine, more, lynx, hangman, robots, games
- These programs change settings in the terminal driver to control how input is handled and output is processed.

User Programs: A Common Type of Device-Specific Program

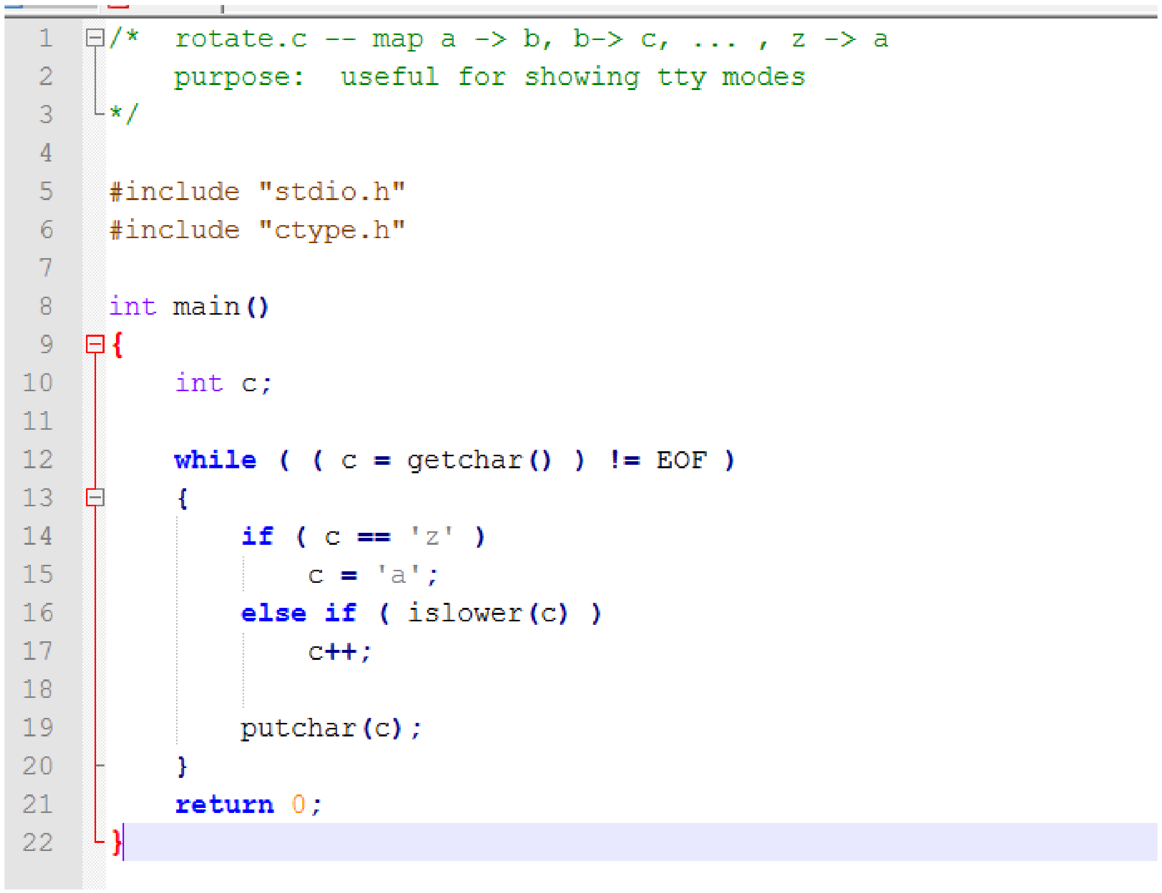
- Many settings, but common concerns of user programs:
  (a) immediate response to keys
  (b) limited input set
  (c) timeout on input
  (d) resistance to Ctrl-C
- Let's learn by writing a program to implement these features.

Modes of the Terminal Driver

rotate.c

- Let's look deeper into the terminal driver by experimenting with a translation program:

rotate.c



```
/*  rotate.c -- map a -> b, b-> c, ... , z -> a
    purpose:  useful for showing tty modes
*/

#include "stdio.h"
#include "ctype.h"

int main()
{
    int c;

    while ( ( c = getchar() ) != EOF )
    {
        if ( c == 'z' )
            c = 'a';
        else if ( islower(c) )
            c++;

        putchar(c);
    }
    return 0;
}
```

Canonical Mode: Buffering and Editing

- A run of the program could look like this:

```
$ gcc rotate.c -o rotate
$ ./rotate
$ abx←cd                        ( ← is backspace )
bcde                            ( output )
efgCtrl-C
$
```

Canonical Mode: Buffering and Editing

- Figure 6.2 shows the terminal, kernel, the rotate program, and the data stream:

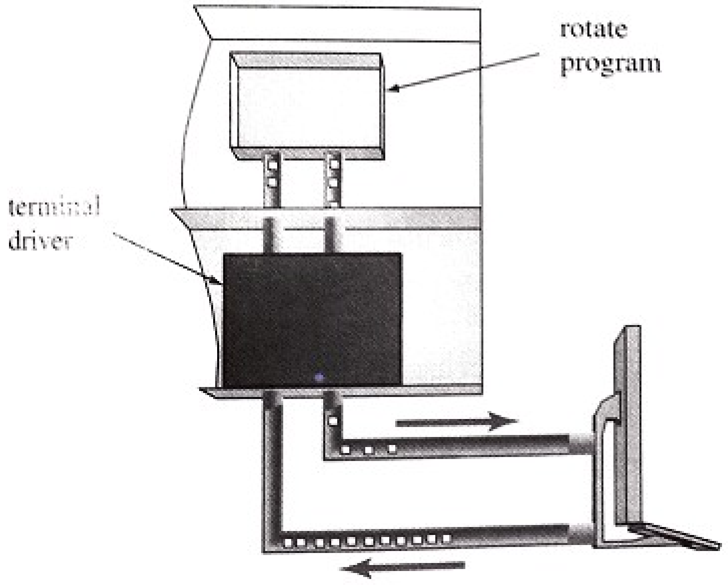


FIGURE 6.2

What you type and what the program gets.

Canonical Mode: Buffering and Editing

- This experiment tells us:
  (a) The 'x' key is never seen by the program, backspace erased it.
  (b) Characters appear on screen as you type them, but
  (c) Program doesn't receive input until you hit enter
  (d) *Ctrl-C* discards input and stops the program.

Canonical Mode: Buffering and Editing

- rotate.c does none of this. Buffering, echoing, editing, control key processing are done by the terminal driver.
- Figure 6.3 shows these operations as driver layers:

Canonical Mode: Buffering and Editing

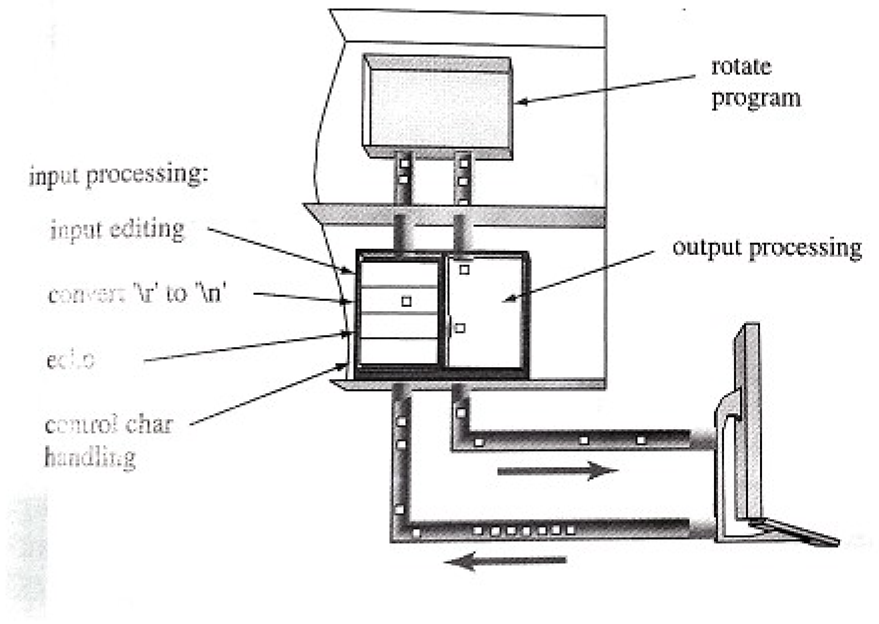


FIGURE 6.3

Processing layers in the terminal driver.

Canonical Mode: Buffering and Editing

- Buffering and editing comprise ***canonical processing***.
- When these features are 'on', the terminal connection is in ***canonical mode***.

Noncanonical Processing

- Let's do another experiment:
  $ stty -icanon ; ./rotate

  a**b**b**c**x**y**c**d**e**e**f**f**g**g**h

  $stty icanon

  Bold indicates output.

Noncanonical Processing

- *stty -icanon* turns off *canonical mode* processing.
- The slide doesn't give the 'full flavor' of noncanonical mode but shows how input processing changed.
- Specifically, *noncanonical mode* doesn't buffer.

Noncanonical Processing

- Press the letter 'a' and the driver skips the buffering layer and sends the character to *rotate*, which returns 'b'.
- Unbuffered mode can suck – when user hits backspace, driver can't do anything, character is already in user space.

Noncanonical Processing

One last experiment:

```
$ stty -icanon -echo ; ./rotate
bcy^?defgh
$ stty icanon echo  ( won't see this )
```

- Here, canonical mode *and* echo mode is turned off.
- Driver doesn't print out chars as we type them.

Noncanonical Processing

- Output comes only from *rotate*.
- When exiting, driver is still in no-echo, noncanonical mode and remains so until the settings are changed.
- The shell prints a prompt and waits for the next command line.
- Some shells reset the driver, some don't.
- If yours doesn't, you continue in no-echo, non-canonical mode.

Terminal Modes Summary

- When designing a Unix user program, you need to decide which terminal mode to use.
- *canonical mode*
  Also called *cooked mode*, is the mode users expect. The driver stores incoming chars in a buffer, sending them when driver reads the Enter key.

Terminal Modes Summary

- *canonical mode, cont*
  Buffering allows for basic editing functions, invoked when user presses the corresponding *erase key*. The keystrokes assigned to the editing functions are driver settings. Change them using *stty* or *tcsetattr*.

Terminal Modes Summary

- *noncanonical mode* aka *crmode*
  The mode where buffering, thus editing, is turned off. Specific character processing still done by the driver (i.e., Ctrl-C, etc), but editing keystrokes are turned off.

  Programs in *noncanonical mode* require editing functions if they want users to be able to edit data.

Terminal Modes Summary

- *non-anything mode* aka *raw mode*
  Each processing step is controlled by a separate bit. When all are turned off, the driver is in *raw mode*. This mode is essentially deprecated (no longer used).

Terminal Modes Summary

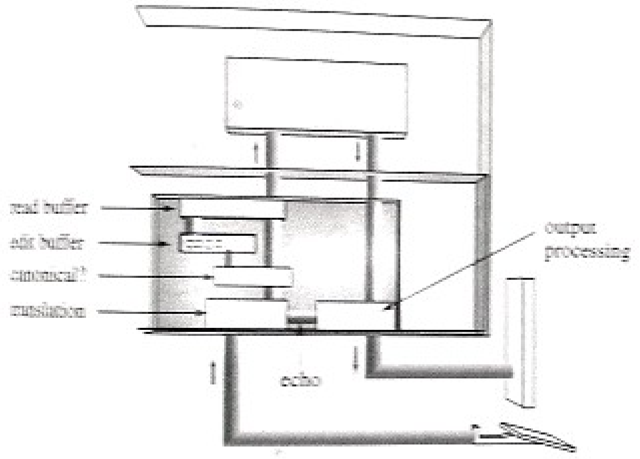
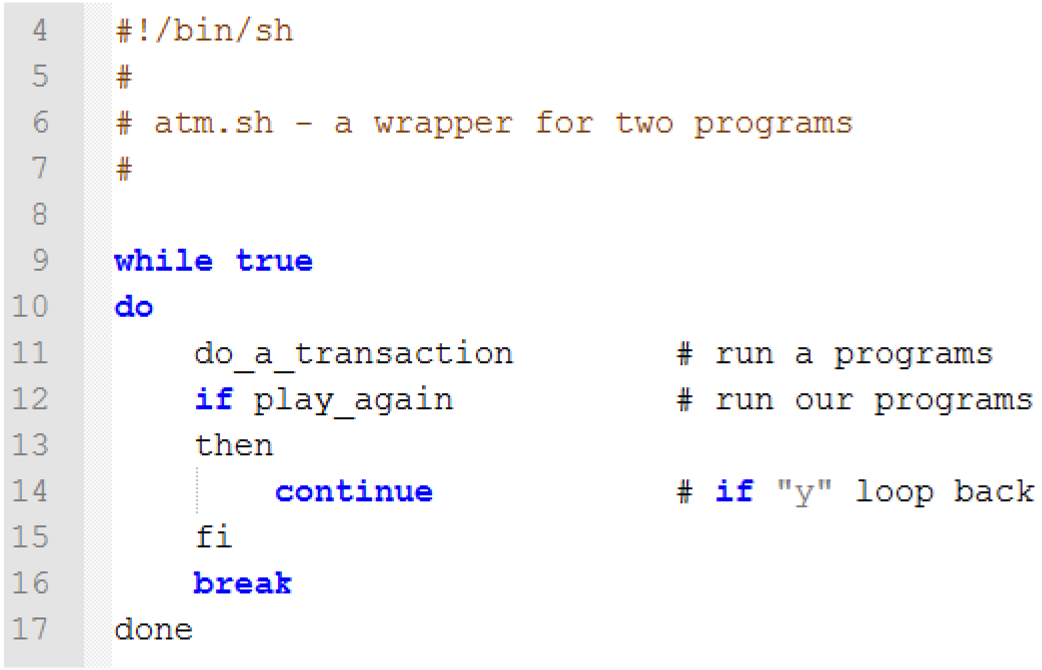
- The major parts of the terminal driver:
- 

FIGURE 6.4
Major components of the terminal driver.

Writing A User Program: play_again.c

Writing A User Program: play_again.c

- Consider the following shell script, modeling an atm machine, that processes y/n logic.



```
#!/bin/sh
#
# atm.sh - a wrapper for two programs
#

while true
do
    do_a_transaction        # run a programs
    if play_again           # run our programs
    then
        continue            # if "y" loop back
    fi
    break
done
```

Writing A User Program: play_again.c

- The first component, *do_a_transaction*, does the ATM work.
- The second component, *play_again* gets a yes/no answer from the user.
- Let's write *play_again*.

Writing A User Program: play_again.c

- Here's the logic:

```
prompt user with a question
accept input
if 'y', return 0
if 'n', return 1
```

Writing A User Program: play_again.c

- Let's look at play_again0.c
- The program prints a question and then loops.
- It reads user input until the user types 'y' or 'n' or 'Y' or 'n'.
- It has a couple problems, caused by running in canonical mode.

Writing A User Program: play_again.c

- Problems:
  1. user has to press Enter before *play_again0* can act.
  2. The program recieves an entire line of data when the user presses enter, so it reads the following:

```
$ play_again0
  Do you want another transaction(y/n) sure thing!
```

as a negative response.

Writing A User Program: play_again.c

- First improvement: turn off canonical input so program receives and processes chars as user types them.
- Let's examine *play_again1.c*, which now gives us an immediate response.
- It's going to put the terminal into character-by-character mode.

Writing A User Program: play_again.c

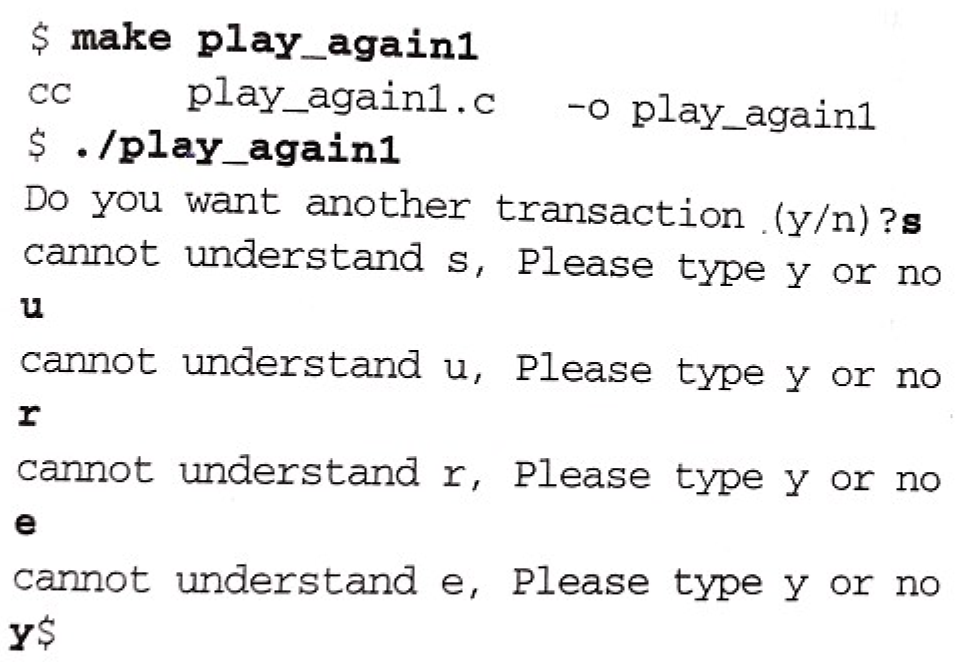
- Then, it's going to call a function to print a prompt and get a response.
- It finishes up by restoring the terminal to it's previous mode.
- Notice, the terminal driver is not set to canonical mode at the end, rather, the original settings are copied into *struct original_mode* which are restored later.

Writing A User Program: play_again.c

- Two steps involved in putting the terminal into character mode:
  1. Turn off the *ICANON* bit and
  2. assign the value 1 to the *VMIN* element in the control character array.

  (*VMIN* tells the driver how many characters to read at a time)

Writing A User Program: play_again.c



```
$ make play_again1
cc      play_again1.c   -o play_again1
$ ./play_again1
Do you want another transaction (y/n)?s
cannot understand s, Please type y or no
u
cannot understand u, Please type y or no
r
cannot understand r, Please type y or no
e
cannot understand e, Please type y or no
y$
```

Writing A User Program: play_again.c

- This fix works, but it still complains about every character.
- Let's fix that by turning off echo mode and throw away unacceptable characters.
- Let's examine *play_again2.c*, which ignores illegal keys.

Writing A User Program: play_again.c

- This version is different than the previous in a couple ways.

1. The function that sets the terminal driver mode turns off the echo bit.
2. The *get_response* function no longer reports illegal input errors.

Writing A User Program: play_again.c

- *play_again2.c* does what we want to, but what if this were used at a real ATM and a customer walked away without finishing?
- The next customer could hit y and get free monies.
- We need a timeout feature to be more secure.
- The next version will have it.

Nonblocking Input: *play_again3.c*

- Our solution will be to tell the driver not to wait for input.
- If there is no input, we sleep a few seconds then wait for some more input.
- After 3 tries, we give up.

Blocking vs Nonblocking Input

- When you call *getchar* or *read* to read data from a file descriptor, the call usually waits for input.
- When this happens, the program is said to be *blocked*.
- How can this be turned off?

Blocking vs Nonblocking Input

- Blocking is a property of any open file – not just terminal connections.
- So, we can use *fcntl* or *open* to enable *nonblocking input* for a file descriptor.
- *play_again3.c* will use *fcntl* to turn on the *O_NDELAY* flag for the file descriptor.

Blocking vs Nonblocking Input

- After turning off blocking, let's call read. What happens?
- If there is input, *read* gets it and returns the number of chars read.
- If there are no chars available, 0 is returned.
- If there is an error, then it returns -1.

Nonblocking Input: *play_again3.c*

- Let's look at *play_again3.c*
- It uses nonblocking mode for timeouts.
- The new features include using *fcntl* to turn on and off nonblocking mode, using *sleep* and *maxtries* counter in *get_response* function.

Nonblocking Input: *play_again3.c*

- *play_again3* is not ideal.
- In nonblocking mode, it sleeps for 2 seconds before calling *getchar* to give the user a chance at input.
- If the user types within a second, the program doesn't get the character until 2 seconds pass – can be confusing.

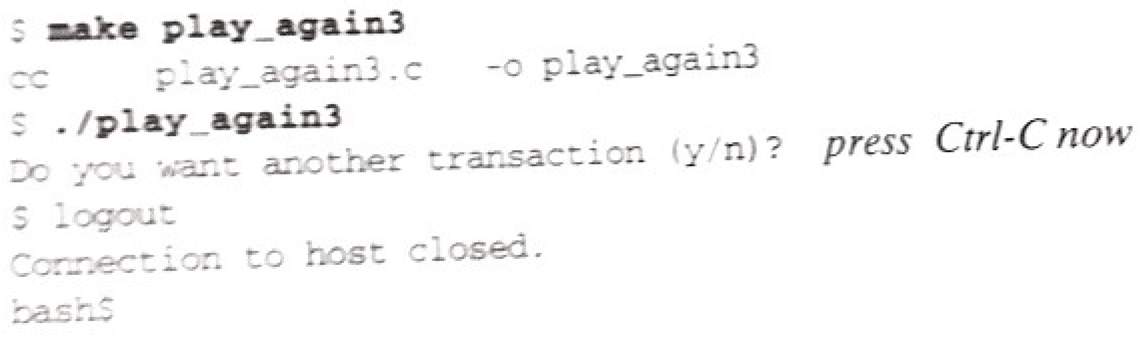
Nonblocking Input: *play_again3.c*

- Would be nice to make the program more responsive: reduce sleep time and compensate by increasing iterations.
- Another issue: notice *fflush* call? Without it, on some machines prompt doesn't appear until the program calls *getchar*. Why?
- The terminal driver buffers input on a line-by-line basis, but also buffers output. Gurantees sleep doesn't interfere.

Nonblocking Input: *play_again3.c*

- It's going to buffer that output until it gets a newline or until the program tries reading from the terminal.
- So, by postponing calling for input so the user has a chance to read the prompt, we have to have *fflush*.

A Big Problem

- *play_again3* ignores letters it dislikes, processes legal input, and exits if no legal input arrives during an interval.
- What if the user enters CTRL-C?
- 

```
$ make play_again3
cc      play_again3.c   -o play_again3
$ ./play_again3
Do you want another transaction (y/n)?  press Ctrl-C now
$ logout
Connection to host closed.
bash$
```

A Big Problem

- When CTRL-C is pressed, the program was killed, but also the entire session. How?
- Here's the steps: play_again3 has 3 steps: initialization, get input, restore settings:

A Big Problem

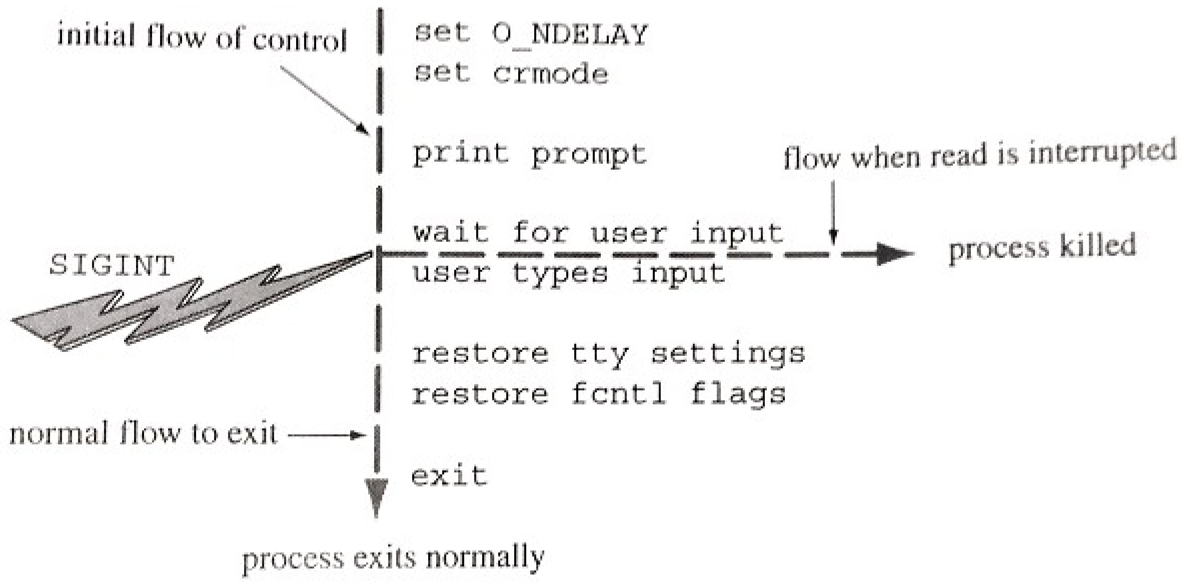
- 

FIGURE 6.5

Ctrl-C kills a program. It leaves terminal unrestored.

A Big Problem

- The initialization part sets nonblocking input.
- Program then enters main loop to prompt, sleep, and read input.
- We then kill the program in the middle with CTRL-C. What state is the driver in?

A Big Problem

- Well, the program quits immediately and doesn't execute the reset driver code.
- The terminal is still in nonblocking mode and the shell returns to print its prompt and get input from the user.
- Shell calls *read*, but since it's operating in nonblocking mode, returns 0.

A Big Problem

- In short, the program left the file descriptor and underlying driver with the wrong attributes.
- We will learn later how to protect our programs from CTRL-C.
- This won't happen with some shells, however, because the shells run in raw mode and reset things automatically.

SIGNALS

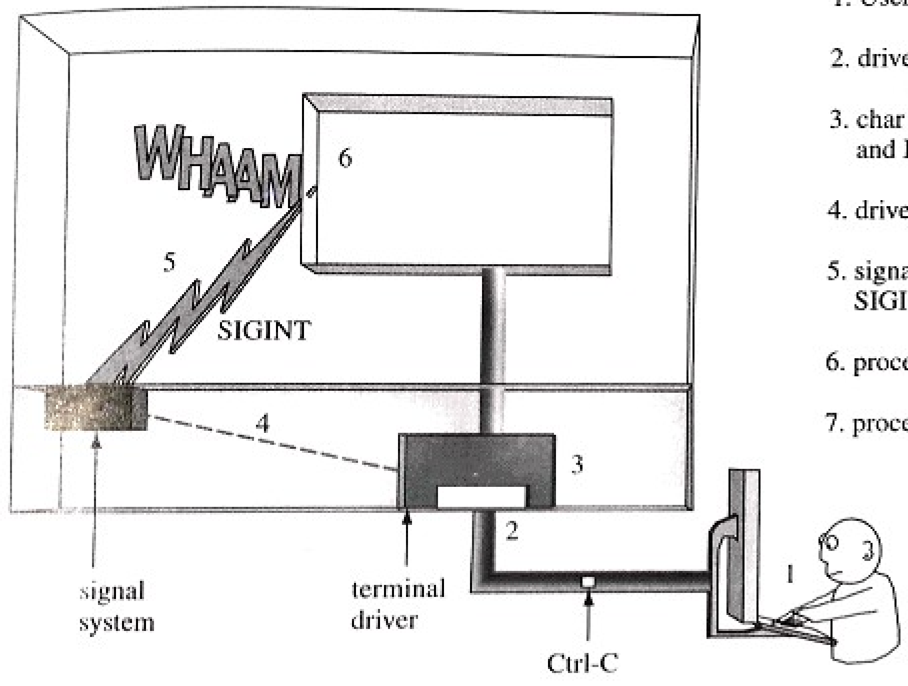
Intro

- CTRL-C interrupts the currently running program.
- The interruption is generated by the kernel mechanism known as a *signal*.

What Does CTRL-C Do?

- Press CTRL-C, and a program dies.
- How does this kill a process, take a look at this:

What Does CTRL-C Do?



1. User presses Ctrl-C
2. driver receives char
3. char matches VINTR and ISIG is on
4. driver calls signal system
5. signal system sends SIGINT to process
6. process receives SIGINT
7. process dies.

FIGURE 6.6
How Ctrl-C works.

What Is A Signal?

- A signal is a one-word message.
- When you press CTRL-C, you ask the kernel to send the *interrupt signal* to the current process.
- Each signal has a code.
- *interrupt* usually is code number 2.

Where do they come from?

- They come from the kernel, but requests for signals come from three sources.

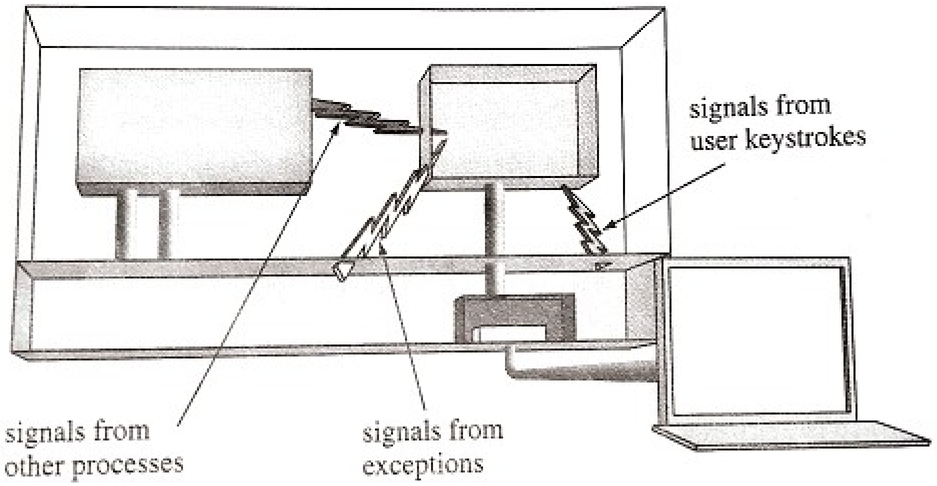


FIGURE 6.7
Three sources of signals.

Where do they come from?

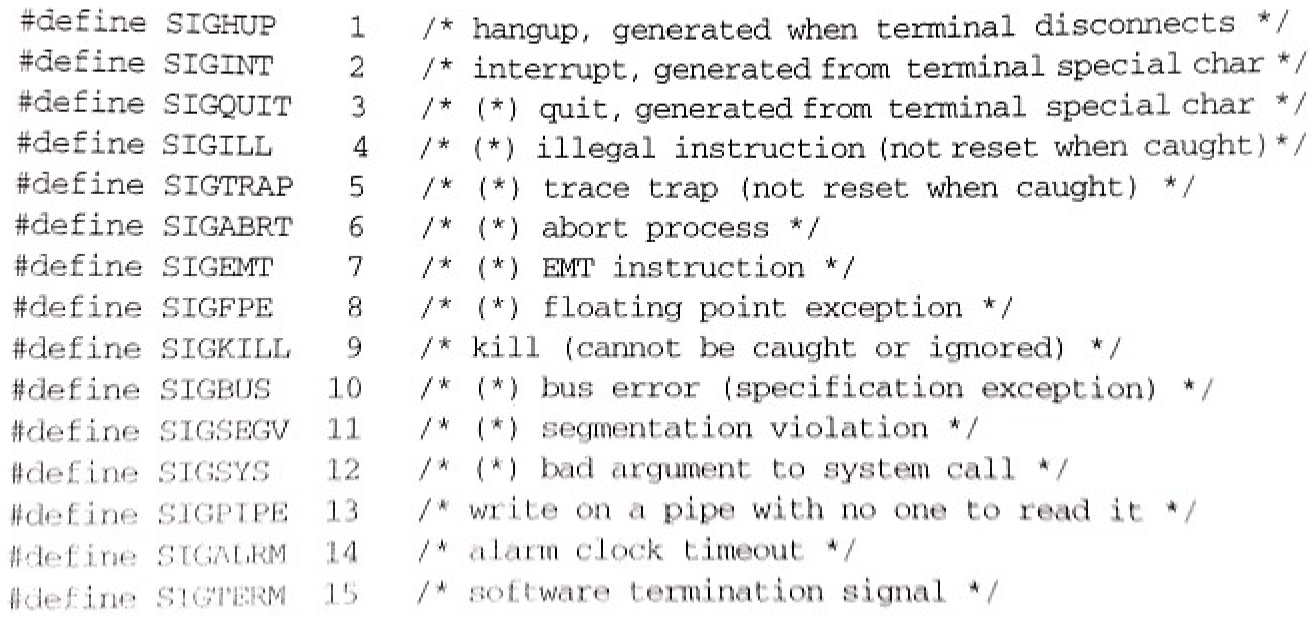
- users – can press CTRL-C, CTRL-/, or any other key assigned in the terminal driver to a signal control character
- kernel – sends a signal to a process when the process does something wrong. Also uses signals to notify processes of events.
- processes – can send a signal to another process by using the *kill* call. One way to communicate with other processes.

Where do they come from?

- Signals caused by something a process does (like dividing by 0) are called *synchronous signals*.
- Signals caused by events outside the process (like someone pressing the interrupt key) are called *asynchronous signals*.

Is there a list of signals?

- Yup, often found in *signal.h*.



```
#define SIGHUP    1   /* hangup, generated when terminal disconnects */
#define SIGINT    2   /* interrupt, generated from terminal special char */
#define SIGQUIT   3   /* (*) quit, generated from terminal special char */
#define SIGILL    4   /* (*) illegal instruction (not reset when caught)*/
#define SIGTRAP   5   /* (*) trace trap (not reset when caught) */
#define SIGABRT   6   /* (*) abort process */
#define SIGEMT    7   /* (*) EMT instruction */
#define SIGFPE    8   /* (*) floating point exception */
#define SIGKILL   9   /* kill (cannot be caught or ignored) */
#define SIGBUS   10   /* (*) bus error (specification exception) */
#define SIGSEGV  11   /* (*) segmentation violation */
#define SIGSYS   12   /* (*) bad argument to system call */
#define SIGPIPE  13   /* write on a pipe with no one to read it */
#define SIGALRM  14   /* alarm clock timeout */
#define SIGTERM  15   /* software termination signal */
```

What do Signals Do?

- Depends – many cause processes to die.
- However, processes can protect themselves from other processes murdering them.

What can a process do about a signal?

- A process doesn't have to die when receiving *SIGINT*.
- Processes can tell the kernel how it wants to respond, by using the *signal* system call.
- There are three choices a process can make:

What can a process do about a signal?

- *accept the default action (usually death)* man page lists default action for each signal. Default for *SIGINT* is death. A process can restore the default action:

  signal(SIGINT, SIG_DFL);

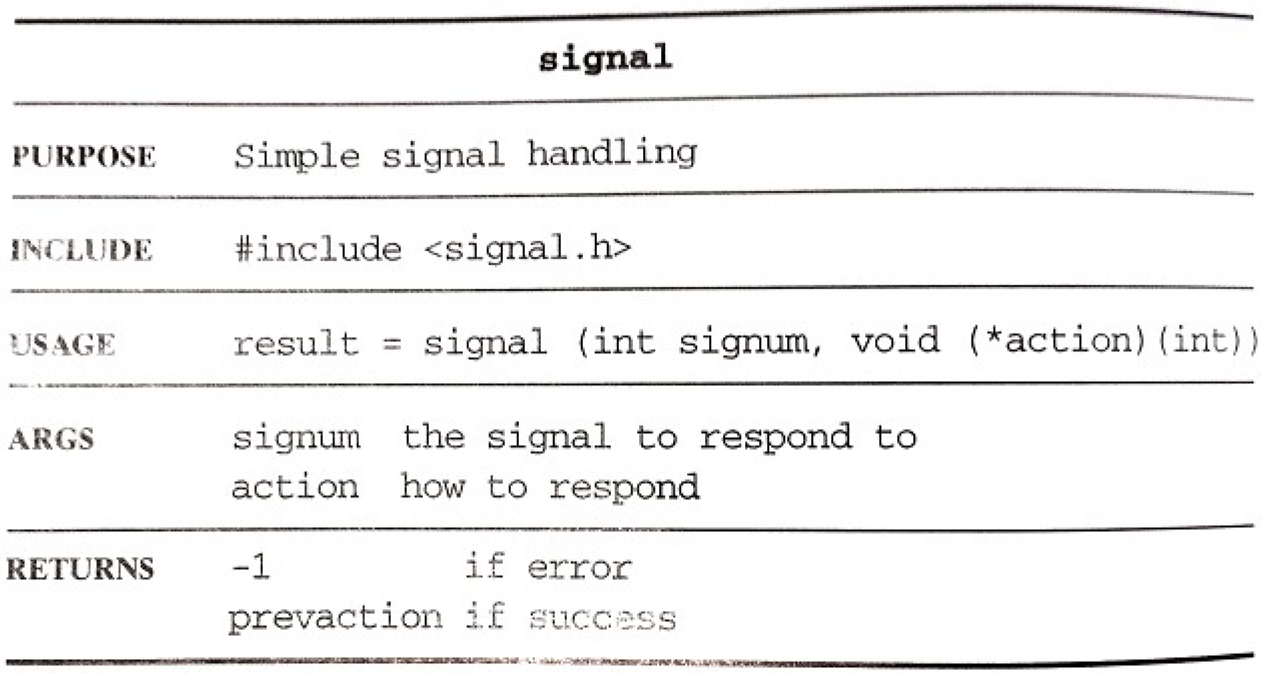
- *ignore the signal*

  signal(SIGINT, SIG_IGN);

What can a process do about a signal?

- *call a function*
  A program can tell the kernel which function to call if a particular signal arrives. This function is called a *signal handler*.
- To install a signal handler:
  signal(signum, functionname);

signal system call



signal

PURPOSE	Simple signal handling
INCLUDE	`#include <signal.h>`
USAGE	`result = signal (int signum, void (*action)(int))`
ARGS	`signum` the signal to respond to `action` how to respond
RETURNS	`-1` if error `prevaction` if success

signal system call

- *signal* call installs a new handler for the signal with number *signum*.
- The action may be the name of a function or one of these:

SIG_IGN	ignore the signal
SIG_DFL	reset signal to default

- returns the previous handler, a pointer to a function

Signal Handling Examples

- Example 1: Catching a signal: *sigdemo1.c*

Signal Handling Examples

- The main function has a call to *signal* and a loop.
- *sigdemo1.c* calls *signal* to install function *f* to handle *SIGINT*.
- If the process receives *SIGINT*, the kernel causes the program to call *f*.
- Program jumps to the function, executes the code, then returns – just like any other function call.

Signal Handling Examples

-

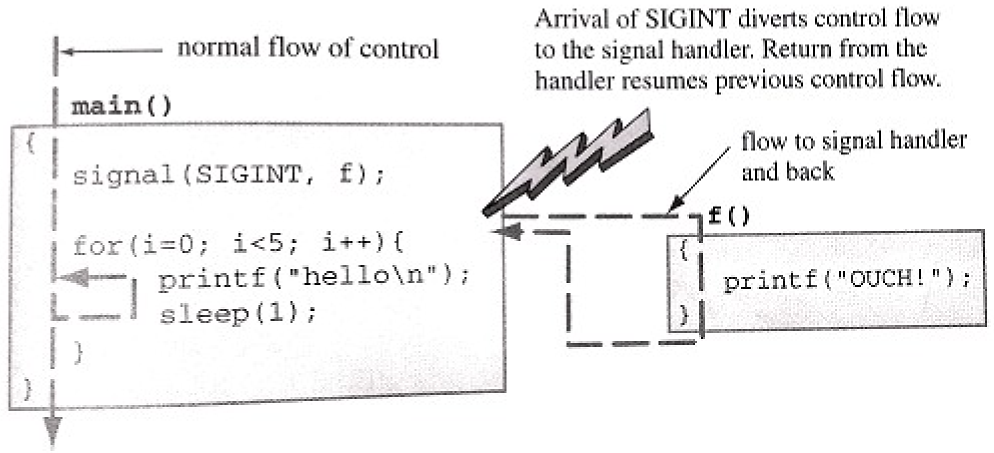

normal flow of control
Arrival of SIGINT diverts control flow to the signal handler. Return from the handler resumes previous control flow.
flow to signal handler and back
main()
{
signal(SIGINT, f);
for(i=0; i<5; i++){
printf("hello\n");
sleep(1);
}
}
f()
{
printf("OUCH!");
}

FIGURE 6.8

A signal causes a subroutine call.

Signal Handling Examples

- Two flows of control are shown: normal path into main, around the loop, and back from main, and the signal-induced path into f and back.

Signal Handling Examples

- Example 2: Ignoring a signal: *sigdemo2.c*
- *sigdemo2.c* uses *signal* to arrange to ignore the interrupt signal.
- Pressing CTRL-C has no effect.

Signal Handling Examples

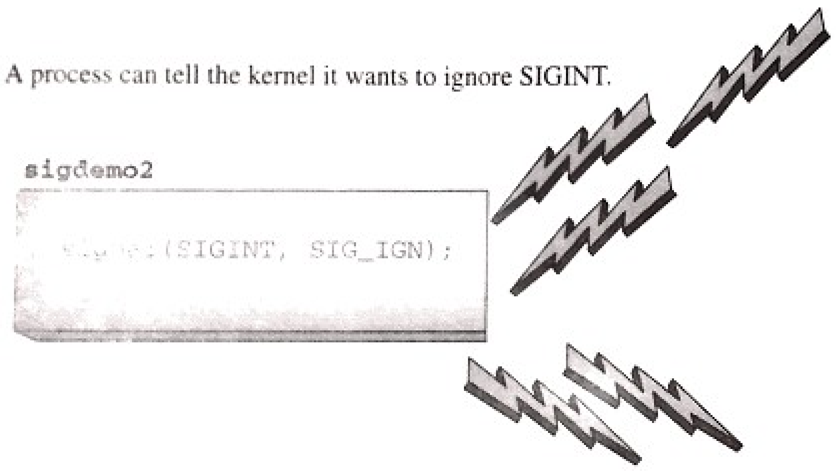
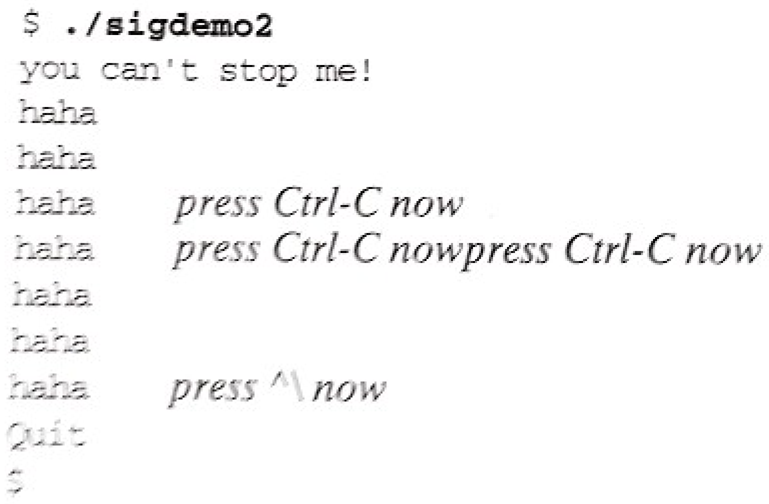


FIGURE 6.9

The effect of `signal(SIGINT, SIG_IGN)`.

Signal Handling Examples



```
$ ./sigdemo2
you can't stop me!
haha
haha
haha      press Ctrl-C now
haha      press Ctrl-C nowpress Ctrl-C now
haha
haha
haha      press ^\ now
Quit
$
```

Signal Handling Examples

- Ctrl-\ sends a different signal (quit) which isn't ignored.

Prepared for Signals: *play_again4.c*

- Let's update *play_again3.c* to handle signals.
- This version catches *SIGINT*, resets the driver, and returns the *no* code.
- Let's examine *play_again4.c*.

Processes Are Mortal

Processes Are Mortal

- Unix makes it impossible for a program to be immortal.
- Two signals can't be ignored or caught: SIGKILL and SIGSTOP
- Check the signals list in signals.h or read man pages for more:

  ```
  man signal
  man 7 signal
  ```

Programming for Devices

Programming for Devices

- We studied 3 aspects of writing terminal-controlling programs.
- First, we studied driver attributes and controlling connections.
- Then, we looked at application needs and adjusted the driver to meet those needs.
- Finally, we learned to handle signals – a type of interruption.

Programming for Devices

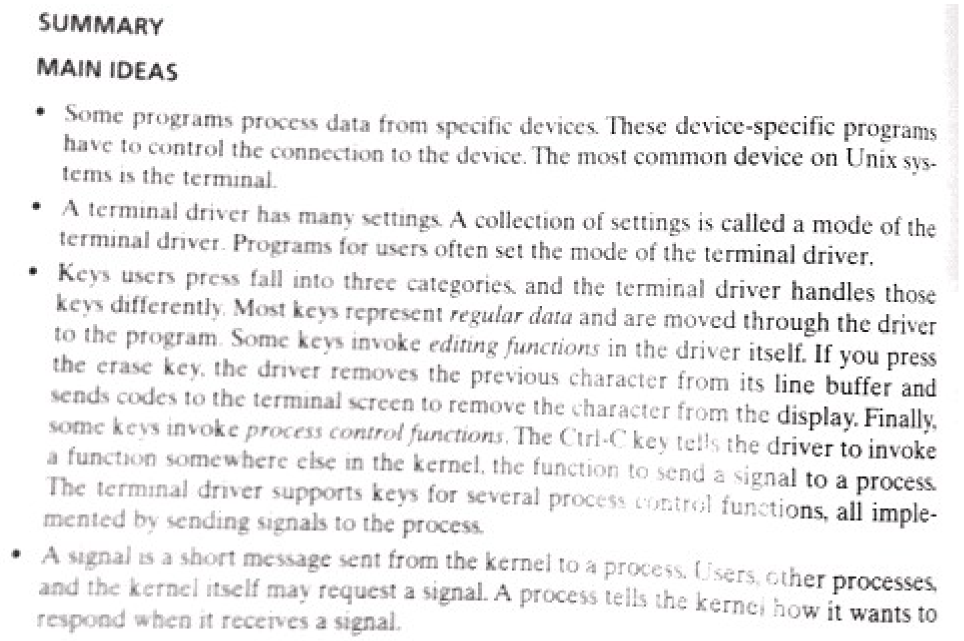
- These 3 ideas apply to every device.
- Consider sound cards.
- They have device driver settings you need to learn about.
- A program will need to perform in certain ways, so we adjust the driver to satisfy those requirements.

Programming for Devices

- Finally, device drivers often generate signals to announce errors or certain events.

Summary

Summary



SUMMARY

MAIN IDEAS

- Some programs process data from specific devices. These device-specific programs have to control the connection to the device. The most common device on Unix systems is the terminal.
- A terminal driver has many settings. A collection of settings is called a mode of the terminal driver. Programs for users often set the mode of the terminal driver.
- Keys users press fall into three categories, and the terminal driver handles those keys differently. Most keys represent *regular data* and are moved through the driver to the program. Some keys invoke *editing functions* in the driver itself. If you press the erase key, the driver removes the previous character from its line buffer and sends codes to the terminal screen to remove the character from the display. Finally, some keys invoke *process control functions*. The Ctrl-C key tells the driver to invoke a function somewhere else in the kernel, the function to send a signal to a process. The terminal driver supports keys for several process control functions, all implemented by sending signals to the process.
- A signal is a short message sent from the kernel to a process. Users, other processes, and the kernel itself may request a signal. A process tells the kernel how it wants to respond when it receives a signal.