Chapter 5 Interactive Programs and Signals

Concepts Covered

Software tools, daemons,
interactive programs
non-blocking reads, signals,
signal, sigaction, kill, fflush, raise
sigemptyset, sigfillset, sigaddset,
sigdelset,sigprocmask, sleep

5.1 The Different Types of UNIX Programs

Programs that run in a UNIX environment can be classified by their relationship to terminal devices and by their input/output streams. They generally fall into one of three categories: *software tools*, *daemons*, and *interactive user programs*.

5.1.1 Software Tools (Filters)

A *software tool* is a program that

- receives its input from either
  - one or more files given as command-line arguments, or
  - from standard input if no filename arguments are present,
- expects its input to be an unstructured stream of bytes, almost always treated as plain text, and
- puts its output, which is also a stream of bytes, usually plain text, on standard output.

Because software tools can read from standard input and write to standard output, they can be connected via shell pipes to form pipelines, like factory assembly lines. UNIX has many software tools, including `awk`, `cat`, `cut`, `du`, `fold`, `grep`, `od`, `sed`, `sort`, `tr`, and `uniq`.

5.1.2 Daemons

Another class of programs are device drivers, which are not even attached to a terminal. A program that is not attached to a terminal is called a *daemon* in UNIX. A commonly used, but inaccurate, definition of a daemon is that it is a "background" process. To be precise, it is a process that executes without an associated terminal or login shell, waiting for an event to occur. The event might be a user request for a service such as printing or connecting to the Internet, or a clock tick indicating that it is time to run. The word "daemon" is from Greek mythology, and refers to a lesser god who did helpful tasks for the people he or she protected. Daemons are like these lesser gods; they are created at boot time, and exist, in hiding, ready to provide services when called upon.

Because daemons must not be connected to a terminal, one of their first tasks is to close all open file descriptors (in particular, standard input, standard output and standard error). They usually make their working directory the root of the file system. They then take additional steps to break their association with any shell or terminal, among which are leaving their process group and registering their intent to ignore all incoming signals. The concept of process groups will be discussed in Chapter 8. Signals are covered later in this chapter.

Daemon process names typically end in 'd'. This is one way to identify a daemon process in the output of the `ps -ef` command; find the names ending in "d" as in `ftpd`, `httpd`, `lpd`, `sshd`, `syslogd`, and `telnetd`. Daemons will be covered thoroughly in Chapter 8.

5.1.3 Interactive User Programs

Another category of programs are those that are tied to the user terminal in an inextricable way, because they customize the terminal for their own use. Programs that interact with the user through the terminal, such as text editors (`vi`, `nano`, or `emacs`), pagers (`more` and `less`), terminal-based administrative tools such as `top`, games (`snake`, `worm`, and `chess`), and terminal-based mail clients (`pine` and `mailx`), are tightly coupled to the terminal and must control its settings and attributes. They cannot use the standard input and output streams for communicating with the user because these lack the types of controls that a terminal has. These types of programs usually need to control

- whether or not characters are echoed,
- the number of characters that are buffered, if any,
- the movement of the cursor on the screen,
- whether certain key presses should have their default meaning or have application-defined meaning,
- whether timeouts should occur on input,
- whether signals such as `Ctrl-C` should be ignored, queued, or handled immediately.

We already saw how to control the state of the terminal using `stty` at the command level and the `tcgetattr()` and `tcsetattr()` functions at the programming level. Here we will explore the various modes into which we can put the terminal for the benefit of creating interactive programs.

5.2 Designing Interactive User Programs

Most interactive user programs are event-driven or menu-driven, which means that they perform some short task and then wait for user input to do the next task. All window-based applications are event-driven; they are idle, often blocked on input, while they wait for mouse, keyboard, or other events to be delivered to them by the window manager.

Here we go through the steps that are necessary to design and develop an interactive, terminal-based application. We will begin by understanding the problem, and then we will go through successive stages of making a program more and more responsive to user inputs.

5.2.1 Two Different Paradigms

Consider the kind of terminal-based program in which the program repeatedly prompts the user for input and takes an action accordingly. One of the user responses such programs expect is some type of quit signal, which typically breaks the loop and ends the program. This can be modeled by a control structure such as the following:

```
while (true) {
    prompt the user to do something
    wait for the user to respond
    if user's response is to quit
        break the loop and quit
    handle the response
}
```

The input part of this loop usually results in the process's being blocked on input, but it does not have to be designed this way. It might look like

```
while (true) {
    prompt the user to do something
    if the user responded
        handle the response
    otherwise
        do other things
}
```

In this paradigm, the program checks whether there is input and if there is, it responds to it, and if not, it does something else. This approach requires the ability to check if there is input without blocking while waiting for it. In short, it requires a type of input operation known as non-blocking input, which will be discussed below.

Regardless of which input method is used, programs such as video games, ATM machines, and text editors respond immediately to user key presses, rather than waiting for the `Enter` key to be pressed. They run in non-canonical mode, so they do not buffer input characters. Usually, they do not echo the input characters when these characters behave like function keys[1]. Also, they usually ignore illegal key presses. Thus, one task in designing interactive programs is to determine how to control the state of the terminal in this way.

But this is not enough. There is a big difference between a video game and a text editor, having to do with their relationship to time. We can distinguish between two kinds of interactive programs: those whose state is independent of time, and those whose state depends upon time. Any program that animates, in any way, is time-dependent; its state changes as a function of time. Programs that terminate or advance to a different state if the user does not respond within a certain amount of time are also time-dependent, because their state changes as a function of time. Video games are time-dependent. In contrast, a text editor is usually time-independent; it has no time-outs and no animation of any kind.

Programming with time is more complex than programming in a time-independent way because it requires knowledge of several different parts of the kernel API. Before we tackle this problem, we will explore a simpler one, namely how to write a text editor.

[1] Think about `vi` for example, and how it behaves in Command mode; you type a 'j' and it moves the cursor without displaying the letter, or more, when you type a space character and it advances a screen's worth of lines.

5.2.2 A Simple Text Editor

Note. The code in this section is not yet working properly. It is in progress.

The `vi` text editor is a very complicated piece of software, but we can create an extremely stripped-down version of it and still learn quite a bit. We will call it `simplevi`. This simple editor will allow the user to insert characters anywhere in a text document, but it will not provide a means of deleting characters. This is just a minor extension to the program. Also, it will not open an existing text file, instead creating a new one each time. Adding a feature to open a file does not provide much more insight into the interactive design of the program, but it does make the program larger.

It would be much easier to write this program if we used the Curses library, but as we have not yet covered that API, we will do it the hard way, the way it was done before Curses existed. It will give you an appreciation of Curses when we get to it. Instead, we will use the ANSI esacpe sequences that we covered in Chapter 1.

The design challenge in writing this simple text editor is integrating the two major objects that the program must manage:

- A behind the scenes text buffer, and
- The visible screen.

The text that the user types must be stored in a buffer of some kind. There are many possible ways to organize this buffer, with time-space trade-offs associated to each. A reasonable solution would be to create an array of pointers to the individual lines of the buffer, each of which would be a dynamically allocated fixed-size array. The array of pointers could be replaced by a doubly-linked list of pointers with added programming complexity. The arrays holding the lines of text could be started smaller and reallocated to larger sizes as the lines grow in size. These details are not important to us now, as performance is less of a concern than understanding the principles. Therefore, we take an even simpler approach: the text buffer is just one large linear array of characters named `buffer`:

```
char buffer[BUFSIZ];   /* BUFSIZ is a system limit */
```

It would be very difficult to manage this buffer if we did not have a convenient means on knowing where each line began within the buffer. Therefore, we use a second object, an array with an entry for each text line, whose values are the lengths of those lines. We shall declare it as

```
int line_len[MAXLINES];
```

where `MAXLINES` is some predefined limit on the number of lines in any file that `simplevi` will create.

The screen is treated as nothing more than a display object, which is all that it really is. There are two types of actions that the program must perform relative to the screen:

- Moving the cursor around and keeping track of its position, and
- Changing the appearance of the screen by writing text onto it or removing text from it.

5.2.2.1 Managing the Cursor

The primary challenge in managing the cursor is that, at any given time, the program must be able to map the cursor position to a position in the text buffer, and vice versa. Therefore, we need two additional objects: a cursor, and a position in the buffer. A cursor will be a structure with a row and column index, representing a position in a two-dimensional array whose upper-left corner is position (1,1). The position in the buffer is an integer:

```
typedef struct _cursor
{
   int r;
   int c;
} Cursor;

Cursor  curs;
int     index_in_buf;
```

A text file is really a sequence of lines of text terminated by newline characters. Each newline character forces the next line to start at the left margin in the row below the last character of the current line. As characters are inserted into a line of text, that line changes its form, but not the others above or below it, although they may shift downward. Therefore, it is convenient to maintain two other pieces of information associated to the cursor's current position:

```
int     cur_line;           /* current text line, not screen line */
int     index_in_cur_line;  /* index in current line of cursor */
```

`cur_line` is the index of the text line within which the cursor is located. `index_in_cur_line` is the offset from the first character of that line to the location of the cursor. If the line is N characters long, `index_in_cur_line` can be a number from 0 to $N-1$. Given these two indices, we can compute the position in the buffer by calling

```
index_in_buf = buffer_index(index_in_cur_line, cur_line, line_len);
```

where `buffer_index()` is defined as

```
int buffer_index( int index_in_line, int cur_line, int linelen[] )
{
    int totalchars = 0;
    int i = 0;

    while (  i < cur_line ) {
        totalchars += linelen[i];
        i++;
    }
    totalchars += index_in_line;
    return totalchars;
}
```

The program will always make sure that it knows the current line and the current index in the line as the cursor moves around on the screen. In fact, the cursor movement operations will actually update these variables, and from those recalculate the cursor position. This needs explanation. Text lines may be longer than the screen width. When this happens, they wrap onto two or more lines. In `vi`, when the cursor is moved upward or downward, it "jumps" over the wrapped lines. In other words, it moves from one text line to another, not from one screen row to another. Our `simplevi` program emulates this behavior. Therefore, when the user presses an arrow key up or down, it will move to the preceding or following text line. To make this possible, it will increment or decrement the `cur_line` variable as needed, and possibly adjust the `index_in_cur_line` variable, as will be explained below. But this implies that it has to find the new position of the cursor on the screen. The function `get_screenpos()` does that. For example, calling

```
get_screenpos(index_in_cur_line, cur_line, line_len, cols, &curs );
```

will store into `curs` the screen coordinates for the text buffer position in the line whose index is `cur_line`, and whose offset in that line is `index_in_cur_line`. The `line_len[]` array and the `cols` variable are the array of line lengths and the width of the screen respectively. The function is defined as follows:

```
void get_screenpos( int index, int lineno, int linelength[], int numcols,
                    Cursor *curs )
{
    int total_lines_before = 0;
    int lines_in_current_textline = 0;
    int i;

    for ( i = 0; i < lineno; i++ ) {
        total_lines_before += (int) ceil((double)linelength[i]/numcols);
    }
    lines_in_current_textline = index/numcols;
    curs->r = total_lines_before + lines_in_current_textline;
    curs->c = index - lines_in_current_textline*numcols;
}
```

5.2.2.2 Writing to the Screen

Since all output operations must bypass the C standard I/O library, lest they fall victim to its internal buffering, the program will only use the kernel's `write()` system call for writing to the screen. A write to the terminal device will always place the bytes to be written at the current cursor position. After a write completes, the cursor is advanced to the right of the last character written, unless what was written was an ANSI escape sequence that does not actually write to the screen. This implies that sometimes we will have to save the current position of the cursor before the write operation in order to return to it.

5.2.2.3 Operations Supported by simplevi

The vi text editor is a three-state finite automaton, and so is simplevi. The three states are

- input mode

- command mode, and
- last_line mode

The program always starts in command mode. If the user enters the command 'i', `simplevi` enters input mode and remains there until the user presses the `Escape` key. If in command mode the user enters the ':' character, `simplevi` enters last_line mode. In last_line mode, only two commands are possible:

`q` which causes `simplevi` to quit, and

`w` which causes `simplevi` to write the current contents of the text buffer to a file whose name is `tempfile` and which is located in the process's current working directory.

These can be entered separately or in sequence or as the string "wq". In last_line mode, the cursor is on the last line of the screen (hence its name). The last line of the screen is reserved for last_line mode and for other messages that `simplevi` may write. It is thus off-limits to the cursor. The program must ensure that this line is treated as if it is not part of the screen when in input mode or command mode.

In command mode, the cursor can be moved using the arrow keys. As will be noted below, this is a problematic part of the program when the Curses library is not used, because it is very hardware dependent, and the program does not handle this dependence. It is designed under the assumption that the arrow keys generate a specific sequence of bytes. Nonetheless, the purpose is to show how to handle cursor movement. Command mode also allows the user to press `Ctrl-D`, `Ctrl-C`, and `Ctrl-H` and not have their default behavior be invoked. It does this by disabling the terminal's handling of keyboard generated signals.

5.2.2.4 Terminal Support Functions

Three functions are used for modifying or querying the terminal state, as shown in the following program listing.

```
void modify_termios(int fd, int echo, int canon )
{
    struct termios cur_tty;
    tcgetattr(fd, &cur_tty);

    if ( canon )
        cur_tty.c_lflag     |= ICANON;
    else
        cur_tty.c_lflag     &= ~ICANON;
    if ( echo )
        cur_tty.c_lflag     |= ECHO;
    else
        cur_tty.c_lflag     &= ~ECHO;
    cur_tty.c_lflag     &= ~ISIG;
    cur_tty.c_cc[VMIN]   = 1;
    cur_tty.c_cc[VTIME]  = 0;
    tcsetattr(fd, TCSADRAIN, &cur_tty);
}
```

```
void save_restore_tty(int fd, int action)
{
    static struct termios original_state;
    static int            retrieved = FALSE;

    if ( RETRIEVE == action ){
        retrieved = TRUE;
        tcgetattr(fd, &original_state);
    }
    else if (retrieved &&  RESTORE == action ) {
        tcsetattr(fd, TCSADRAIN, &original_state);
    }
    else
        fprintf(stderr, "Illegal action to save_restore_tty().\n");
}

void get_winsize(int fd, unsigned short *rows, unsigned short *cols )
{
    struct winsize size;

    if (ioctl(fd, TIOCGWINSZ,  &size) < 0) {
        perror("TIOCGWINSZ error");
        return;
    }
    *rows = size.ws_row;
    *cols = size.ws_col;
}
```

`modify_termios()` either enables or disables echo and canonical mode, and it disables keyboard signals. It also sets the `MIN` and `TIME` line discipline values to 1 and 0 respectively so that the program reads a single character at a time and does not time-out. `save_restore_tty()` can be used to save the current terminal state into a local static variable for later restoration. `get_winsize()` uses the `ioctl()` function to get the current window size when the program starts up. If the window is resized while the program is running, all bets are off – it does not handle resizing events.

5.2.2.5 Other Support Functions

These include one for showing the program state, specifically the cursor position, the line index and the offset within the line:

```
void show_cursor( Cursor cursor, int index_in_line, int line_number )
{
    char curs_str[80];

    sprintf(curs_str, "Cursor: [%d,%d]  line index: %d  offset in line: %d ",
                cursor.r+1, cursor.c+1, line_number, index_in_line  ) ;
    write(1,SAVE_CURSOR, strlen(SAVE_CURSOR));
    write(1,PARK, strlen(PARK));
    write(1, curs_str, strlen(curs_str));
    write(1,REST_CURSOR, strlen(REST_CURSOR));
}
```

This function uses several constant strings that contain ANSI escape sequences. The full set of these is displayed in the listing below:

```
const char CLEAR_DOWN[]   = "\033[0J";
const char CLEAR_RIGHT[]  = "\033[0K";
const char CURSOR_HOME[]  = "\033[1;1H";
const char CLEAR_SCREEN[] = "\033[2J";
const char CLEAR_LINE[]   = "\033[2K";
const char SAVE_CURSOR[]  = "\033[s";
const char REST_CURSOR[]  = "\033[u";
const char UP[]           = "\033[1A";
const char DOWN[]         = "\033[1B";
const char RIGHT[]        = "\033[1C";
const char LEFT[]         = "\033[1D";
```

The string PARK used in show_cursor() above is defined dynamically, because it depends upon the screen size. It is defined as follows:

```
get_winsize(STDIN_FILENO, &rows, &cols);
sprintf(PARK, "\033[%d;1H", rows);
```

The last two functions used by the main program are one for inserting characters into the buffer, and one for moving the cursor to an arbitrary position:

```
void insert( char buf[],
             int insertpos,
             int cur_length, char ch )
{
    int i;
    for ( i = cur_length; i > insertpos; i-- )
        buf[i] = buf[i-1];
    buf[insertpos] = ch;
}

void moveto(int line, int column )
{
    char seq_str[20];

    sprintf(seq_str, "\033[%d;%dH", line+1, column+1);
    write(1, seq_str, strlen(seq_str));
}
```

5.2.2.6 The Main Program

The main program begins by initializing all variables and setting up the terminal. It is fairly self-explanatory:

```
int     quit = 0;
int     in_input_mode    = 0;
int     in_lastline_mode = 0;
char    buffer[BUFSIZ];      /* text buffer */
char    statusstr[80];
```

```
int     line_len[MAXLINES];   /* lengths of text lines, including newline
                                 characters */
int     num_newlines = 0;     /* number of newline characters in buffer */
char    prompt = ':';         /* prompt character */
unsigned short rows, cols;    /* screen dimensions */
Cursor  curs;                 /* cursor position (0,0) is upper left */
int     index_in_buf;         /* index of cursor in text buffer */
int     buf_size;             /* total chars in buffer */
int     cur_line;             /* current text line, not screen line */
int     index_in_cur_line;    /* index in current line of cursor */
int     i;
int     fd;
char    c;

/* Check if either input or output is redirected and exit if so */
if ( !isatty(STDIN_FILENO) || !isatty(STDOUT_FILENO) ) {
    fprintf(stderr, "Not a terminal\n");
    exit(1);
}

/* Save the original tty state */
save_restore_tty(STDIN_FILENO, RETRIEVE);

/* Modify the terminal - turn off echo, keybd sigs, and canonical mode */
modify_termios( STDIN_FILENO, 0, 0);

/* Obtain terminal window size and create string to park cursor */
get_winsize(STDIN_FILENO, &rows, &cols);
sprintf(PARK, "\033[%d;1H", rows);

/* Clear the screen and put cursor in upper left corner */
write(1,CLEAR_SCREEN, strlen(CLEAR_SCREEN));
write(1,CURSOR_HOME, strlen(CURSOR_HOME));

/* Initialize the counters and other locators */
num_newlines   = 0;
cur_line       = 0;
line_len[0]    = 0;
curs.r         = 0;
curs.c         = 0;
buf_size       = 0;
index_in_cur_line = 0;
index_in_buf    = 0;
```

After this it enters a loop in which it waits for user input. the form of this loop, with the operative code omitted is as follows:

```
while ( !quit ) {
    if ( in_input_mode ) {
        if (  read(STDIN_FILENO, &c, 1) > 0  ) {
            if ( c != ESCAPE ) {
                /* insert typed char and echo it
                   updating screen as needed */
            }
            else { /* ESCAPE so exit */
```

```
                in_input_mode = 0;
            }
        }
    }
    else {
        if (  read(STDIN_FILENO, &c, 1) > 0  ) {
            switch ( c ) {
            case 'i':
                in_input_mode = 1;
                break;
            case ':':
                /* do last line mode */
                break;
            case '\003':
                /* do ctrl-c */
                break;
            case '\004':
                /* do ctrl-d */
                break;
            case '\010':
                /* do ctrl-h */
                break;
            case ESCAPE:
                read(STDIN_FILENO, &c, 1);
                if (  c  == 91 ) {
                    read(STDIN_FILENO, &c, 1);
                    switch ( c ) {
                    case KEY_UP:
                         /* do key up */
                         break;
                    case KEY_DOWN:
                        /* do key down */
                        break;
                    case KEY_RIGHT:
                        /* do key right */
                        break;
                    case KEY_LEFT:
                        /* do key left */
                        break;
                    }
                }
                break;
            }
        }
    }
}
/* cleanup code follows here */
```

Of course the real work is not displayed here yet. Handling the cursor movements and insertions is where the complexity enters the picture. Only selected parts of this are contained here. The actual program is in the demos directory.

5.2.3 Non-Blocking Input

Changing the state of the terminal itself is preferable to changing the attributes of the open file descriptor, because we can exercise more control over it. However, the downside of this is that all programs that use that terminal will be affected by the changes. Usually this is not a problem, since the program you are writing will be the only foreground process, and no other process will be reading from the terminal while it is running. However, it is worthwhile to understand how to exercise some limited control over input through the file descriptor, using the `O_NDELAY` flag (called `O_NONBLOCK` in POSIX).

The `O_NONBLOCK` flag controls whether reads and writes are blocking or non-blocking. When a read is blocking, the process that executes the read waits until input is available, and only then does it continue. This is the semantics that beginning programmers learn. This makes sense; after all, why would you ever want a program to continue past a read instruction if the read did not yield any data?

Exercise. Before reading further, try to answer the preceding question.

Non-blocking I/O is a property of open file connections, not of terminals or devices; when you open a file and get a file descriptor as the return value of the `open()` call, you can specify in the call that the file connection should be non-blocking. In other words, the property of being non-blocking is part of the process's connection to the file or the terminal, not the terminal itself. Two different processes can have the same file open for reading, one using blocking reads and the other, non-blocking.

Remember that when a connection is established between a process and an input source, a buffer is created that holds data from the source on its way to the process. Whether it is a disk file, a terminal, a pointing device, or an audio source, there is some temporary storage area used for buffering input. When a process opens a non-blocking connection to an input source, whether it is a file or a device, calls to read data from that source retrieve whatever data is in the buffer at the time of the call, up to the amount requested in the read request, and return immediately. If the buffer is empty, they return immediately with no data. To be clear, in the call

```
if ( (bytesRead = read(fd, dataToProcess, bytesToGet) ) > 0 )
    {/* statement block */}
```

if the connection is non-blocking and no data is in the buffer of the file descriptor `fd`, then the call returns immediately with `bytesRead` == 0 and the statement block will be skipped.

Similarly, a call to the C library function `getchar()` will return either the character in the front of the buffer or, if the buffer is empty, nothing at all. In either case, `getchar()` returns immediately, i.e., in

```
if ( ( c = getchar())  != EOF )  { /* statement block */ }
```

if there is no character in the buffer, `getchar()` will return `EOF`, which the program can use to decide how to proceed.

Non-blocking input should not be confused with *asynchronous input*. Asynchronous input occurs when the process makes a call to read data and returns immediately, without waiting for the data

to be ready. The `read()` call is executed by a separate process, and as soon as the data are available for it, that process performs the I/O and fills the buffer passed to it by the one calling `read()`. This is called asynchronous input because the caller does not synchronize with the process running the `read()` call; they proceed independently once the call is made. Asynchronous input is useful when you may not need the data right away.

Non-blocking input is useful when the lack of input itself is a significant condition to be identified by the program. For example, it might indicate that a user has left the terminal and is no longer responding, or that a connection to a remote host has been broken, or that a pipe is empty. It may also imply that the user is choosing to not supply input because supplying input may mean making something happen that the user does not want to happen, as in a video game. Very often the process requesting the input has other work to do and it can simply check later whether the input is available. For this reason, non-blocking reads are usually placed inside loops where the condition is tested and an appropriate action can be taken. In most programs that use non-blocking input, the state of the program is changing without the user's intervention. This might be because animation is taking place, or a computation is being performed, or something else entirely.

The following listing is of a program that uses non-blocking input and pretends to do a simple animation. It draws "dots" on the screen, nothing more. Because it uses the same `modify_termios()`, `save_restore_tty()` and `get_winsize()` functions from the `simplevi.c` program, their definitions are omitted. It uses a function to put a file descriptor into non-blocking mode, `set_non_block()`.

```
Listing nonblockdemo.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/stat.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

#define RETRIEVE        1                   /* action for set_tty */
#define RESTORE         2                   /* action for set_tty */

// Defined elsewhere:
void modify_termios(int fd, int echo, int canon );
void save_restore_tty(int fd, int action);
void  get_winsize( int fd, int *rows, int *cols );

void set_non_block(int fd )
{
    int flagset;

    flagset    = fcntl(fd, F_GETFL);
    flagset    |= O_NONBLOCK;
    fcntl(fd, F_SETFL, flagset);
}


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

```
{
    char     ch;          // stores user's char
    char   period = '.';
    size_t bytecount;
    int    count = 0;
    int    done  = 0;    /* to control when to stop loop */
    int    pause = 0;    /* to control pausing of output */
    char   PARK[20];     /* ANSI escape sequence for parking cursor */
    int    numrows;      /* number of rows in window */
    int    numcols;      /* number of columns in window */
    const  char CURSOR_HOME[]   = "\033[1;1H";
    const  char CLEAR_SCREEN[]  = "\033[2J";
    const  char SAVE_CURSOR[]   = "\033[s";
    const  char REST_CURSOR[]   = "\033[u";
    const  char MENU[] = "Type q to quit or p to pause or r to resume.";
    char   dots[20];


    /* Check whether input or output has been redirected */
    if ( !isatty(0) || !isatty(1) ) {
        fprintf(stderr,"Output has been redirected!\n");
        exit(EXIT_FAILURE);
    }

    /* Save the original tty state */
    save_restore_tty(STDIN_FILENO, RETRIEVE);

    /* Modify the terminal -
       turn off echo, keybd sigs, and canonical mode */
    modify_termios( STDIN_FILENO, 0, 0);

    /* Turn off blocking mode */
    set_non_block( STDIN_FILENO );

    /* Get the window's size */
    get_window_size(STDIN_FILENO, &numrows, &numcols);

    /* Create string to park cursor */
    sprintf(PARK, "\033[%d;1H", numrows+1);

    /* Clear the screen and put cursor in upper left corner */
    write(STDOUT_FILENO,CLEAR_SCREEN, strlen(CLEAR_SCREEN));
    write(STDOUT_FILENO,CURSOR_HOME, strlen(CURSOR_HOME));

    /* Start drawing. Stop when the screen is full */
    while ( !done  ) {
        if ( ! pause ) {
            count++;
            /* Is screen full except for bottom row? */
            if ( count > (numcols * (numrows-1)) ) {
                pause = 1;
                count--;
            }
            else
```

```
                write(STDOUT_FILENO, &period, 1);
        }
        usleep(10000);  /* delay a bit */
        sprintf(dots, "  dots:   %d ", count);

        /* Save the cursor, park it, write the menu prompt */
        write(STDOUT_FILENO,SAVE_CURSOR, strlen(SAVE_CURSOR));
        write(STDOUT_FILENO, PARK, strlen(PARK));
        write(STDOUT_FILENO, MENU, strlen(MENU) );
        write(STDOUT_FILENO, dots, strlen(dots));
        /* Do the read. If nothing was typed, do nothing */
        if ( (bytecount = read(STDIN_FILENO, &ch, 1) ) > 0 ) {
            if ( ch == 'q' )
                done = 1;
            else if ( ch == 'p' )
                pause = 1;
            else if ( ch == 'r' )
                pause = 0;
        }
        /* Restore the cursor so the next dot follows the previous */
        write(STDOUT_FILENO,REST_CURSOR, strlen(REST_CURSOR));
    }
    /* Cleanup -- flush queue, clear the screen, and restore terminal */
    tcflush(STDIN_FILENO,TCIFLUSH);
    write(1,CLEAR_SCREEN, strlen(CLEAR_SCREEN));
    write(1,CURSOR_HOME, strlen(CURSOR_HOME));
    save_restore_tty(STDIN_FILENO, RESTORE);
    return 0;
}
```

The program has a loop of the form

```
while ( !done ) {
    /* actual work here */
    if ( (bytecount = read(STDIN_FILENO, &ch, 1) ) > 0 ) {
        if ( ch == 'q' )
            done = 1;
        else if ( ch == 'p' )
            pause = 1;
        else if ( ch == 'r' )
            pause = 0;
        }
}
```

This is the second paradigm shown in Section 5.2.1. The user's input has one of three possible effects: (1) entering `'q'` terminates the loop, (2) entering `'p'` allows the loop to continue but stops output, giving the illusion that the program is paused, and (3) entering `'r'` resumes the output if it is paused and has no effect otherwise. This is a very inefficient method of pausing of course, because the program is gobbling up CPU cycles while it is pretending to do nothing. However, we do not know enough as yet to do otherwise.

The part of the loop with the comment labeled "actual work here" is the part in which it

1. increments the period count and checks how many periods have been written so far,
2. delays a bit,
3. if still room for another period, writes a period, and
4. moves the cursor to the bottom row, writing the prompt and a count of the periods.

This section of the code is a form of animation – it is changing the state of the screen as a function of time. In this case, the timing is achieved by pausing a constant amount of time between redraws. This is a simple, and inefficient, form of animation. Later we will see that there are better means of achieving this.

5.2.4 Allowing Time-Outs

Sometimes we would like to write a program that has time-outs: if the user does not respond within a certain amount of time, it will take this condition to be a significant event in itself. Many programs have some kind of time-out or delay feature like this, so that if the user does not respond within a certain amount of time, the program will either terminate or take some other default action.

We can add a time-out feature to a program by setting `MIN` to 0 and `TIME` to the number of deciseconds we would like it to wait before it decides that there is no input to wait for. For the sake of curiosity, though, we will design a program that will allow us to set the `MIN` and `TIME` terminal attributes to any values we choose, so that we can see how it behaves with different values.

5.2.5 A Test Program

We will build a program in which we can test the effects of changing both the terminal driver's attributes and the open connection's attributes. The main program will be a test driver that allows the user to control the state of the terminal and terminal connection by various command line options, and repeatedly runs a simple function, which we will call `get_response()`, that reads user input in the given state of the terminal and connection. The main program will have a few bells and whistles besides.

The program will have separate functions for controlling the state of the control terminal and for changing the attributes of the file connection to the terminal device. The main program will have a loop to allow us to experiment with the `get_response()` function until we are satisfied that we understand how it behaves under the given settings. There are several pieces to the program, which we present in a bottom-up approach.

First, we combine the `save_restore_tty()` and `set_non_block()` functions into a single function that saves and restores both the terminal settings and the file descriptor flags. It uses the same macros as before:

```
void save_restore_tty(int fd, int action, struct termios *copy)
{
    static struct termios original_state;
    static int            original_flags = -1;
    static int            retrieved = FALSE;
```

```
    if ( action == RETRIEVE ){
        retrieved = TRUE;
        tcgetattr(fd, &original_state);
        original_flags = fcntl(fd, F_GETFL);
        if ( copy != NULL )
            *copy = original_state;
    }
    else if (retrieved && action == RESTORE ) {
        tcsetattr(fd, TCSADRAIN, &original_state);
        fcntl( fd, F_SETFL, original_flags);
    }
    else
        fprintf(stderr, "Bad action to save_restore_tty().\n");
}
```

We will change our `modify_termios()`, function so that it can be given a structure whose members describe the terminal settings:

```
typedef struct tty_opts_tag {
    int min;     /* value to assign to MIN */
    int time;    /* value to assign to TIME */
    int echo;    /* value to assign to echo [0|1] */
    int canon;   /* value to assign to canon [0|1] */
} tty_opts;
```

`modify_termios()` will options in the `tty_opts` parameter that is passed to it to the given `termios` structure.

```
void modify_termios( struct termios *cur_tty, tty_opts ts )
{
    if ( ts.canon )
        cur_tty->c_lflag    |= ICANON;
    else
        cur_tty->c_lflag    &= ~ICANON;
    if ( ts.echo )
        cur_tty->c_lflag    |= ECHO;
    else
        cur_tty->c_lflag    &= ~ECHO;
    cur_tty->c_cc[VMIN]      =  ts.min;
    cur_tty->c_cc[VTIME]     =  ts.time;
}
```

The second, `apply_termios_settings()`, applies the values in the `termios` structure to the terminal line associated to the given file descriptor (which should be standard input.)

```
void apply_termios_settings( int fd, struct termios cur_tty )
{
```

```
    tcsetattr(fd, TCSANOW, &cur_tty);
}
```

The `set_non_block()` function is the same as the one we used above and is omitted.

The `get_response()` function will prompt the user to type a character and will return a value that indicates what the user typed. For simplicity, it will ask for yes or no answers. It prints a question on the screen and gives the user a chance to give a valid response. If the user types a valid response or `max_tries` attempts were made, it returns.

```
int get_response( FILE* fp, ui_params  uip )
{
    int input, n;
    unsigned char c;
    time_t time0, time_now;

    time(&time0);
    while ( TRUE ){
        printf( "%s (y/n)?", uip.prompt);
        fflush(stdout);
        if ( !uip.isblocking )
             sleep(uip.sleeptime);
        if ( (n = read(fileno(fp), &c, 1)) > 0 ) {
            tcflush(fileno(fp), TCIFLUSH);
            input = tolower(c);
            if ( input == 'y' || input == 'n' ) {
                return input;
            }
            else {
                printf("\nInvalid input: %c\n", input);
                continue;
            }
        }
        time(&time_now);
        printf("\nTimeout waiting for input: %d secs elapsed,"
               " %d timeouts left\n",
                (int)(time_now- time0), uip.maxtries);
        if ( uip.maxtries-- == 0 ) {
            printf("\nTime is up.\n");
            return 0;
        }
    }
}
```

Comments

- The `fflush()` call flushes the buffers associated with the file stream passed to it. The C Standard I/O Library provides buffered I/O for file streams. When a program is started,

by default, the streams `stdin`, `stdout`, and `stderr` are *line buffered*. This means that the characters are transmitted to the terminal only when a newline is placed onto the stream. Since functions such as `printf()`, `puts()`, and the others that act on `stdout`, act on file streams, they are line buffered. The preceding `printf()` call

```
printf("%s (y/n)?", uip.prompt);
```

sends a string to `stdout` without a terminated newline and therefore this string will not appear immediately. To force the characters to be delivered to the terminal device, we use `fflush(stdout)`, which empties the buffer. If we comment out the `fflush()` call, the prompt will not appear on the screen until after a `read()` runs.

Note that the buffering provided for streams is independent of the buffering done by the terminal within the line discipline. Even if you put the terminal into non-canonical mode, if you use the higher-level C library functions, C will continue to line buffer. You must use the lower-level file descriptor operations to avoid the buffering.

- The call to flush the terminal's input queue, `tcflush()`, is needed in case the program is run in canonical mode and input is buffered. In this case the user has to enter a newline before the terminal will deliver the characters to the `read()` call, and `get_response()` needs to remove that newline character, otherwise it will be used as the next input character when it is called again.

- `get_response()` calls `sleep()` to block itself for the number of seconds given by `uip.sleeptime`. The `sleep()` function's prototype is

  ```
  unsigned int sleep(unsigned int seconds);
  ```

  The process will sleep until either the given time has elapsed or it receives a signal that it does not ignore. (Signals will be covered soon.) This gives the user a chance to type a response in case non-blocking input is in effect. Without this delay, the `read()` call would return faster than the user could blink an eye. The delay is not needed when input is blocking. There are alternatives to `sleep()` with finer granularity, such as `usleep()` and `nanosleep()`.

- Lastly, `get_response()` computes the total time elapsed since the original prompt was displayed by calling the `time()` function initially and each time that the user fails to enter any character before time runs out, and computing the difference in seconds.

The main program, with includes and a few other utilities omitted, follows. This is `term_demo1.c`.

Listing: term_demo1.c

```
#define PROMPT              "Do you want to continue?"
#define MAX_TRIES           3          // max tries
#define SLEEPTIME_DEFAULT   2          // delay after prompt
#define BEEP                putchar('\a')    // alert user
#define RETRIEVE            1          // action for save_restore
#define RESTORE             2          // action for save_restore
#define FALSE               0
```

```
#define TRUE                1

int main(int argc, char* argv[])
{
    int             response;
    tty_opts        ttyopts = {1,0,1,1};
    ui_params       ui_parameters = {
                       SLEEPTIME_DEFAULT, TRUE,
                       MAX_TRIES, PROMPT
                    };
    int             fflags = 0;
    struct termios  ttyinfo;
    int             fd;
    FILE*           fp;

    get_options(&argc, &argv, &fflags, &ttyopts,
                       &(ui_parameters.sleeptime));

    if ( !isatty(0) || !isatty(1) )
        exit(1);
    fp = stdin;

    save_restore_tty(fileno(fp), RETRIEVE, &ttyinfo);
    modify_termios( &ttyinfo, ttyopts);
    apply_termios_settings( fileno(fp), ttyinfo);

    // If fflags != 0 then the O_NONBLOCK flag is set on fp
    if ( fflags ) {
        ui_parameters.isblocking = FALSE;
        set_non_blocking_mode(fileno(fp), fflags);
    }
    while ( TRUE ) {
        response = get_response(fp, ui_parameters );
        if ( response ) {
            printf("\nMain: Response from user = %c\n",
                    response );
            if ( 'n' == response )
                break;
            }
        else
            printf("\nMain: No response from user\n" );
    }
    save_restore(fileno(fp), RESTORE, NULL);
    tcflush(0,TCIFLUSH);
    return response;
}
```

Comments

1. One problem with the program as it stands is that it will not respond to any key presses until it wakes up from its sleep. You cannot kick it into waking up. It will sleep for the specified time, come hell or high water. This means that a user with a fast response time will be unhappy with this solution. It is a one-size-fits-all solution to making a program responsive to user response rates. It is, in essence, a polling solution because it just repeatedly checks in on the user. An alternative is to somehow make the program sleep until the user actually does something.

2. Another problem with the program is that if it is terminated abnormally, as when the user types `Ctrl-C`, it will not have a chance to restore the terminal to its original settings. If the program has turned off canonical input and echo, for example, and it is killed before reaching the instruction in the main program to restore the terminal, then when the shell resumes execution upon the program's untimely death, the terminal will still be in non-canonical mode with no echo. Fortunately for present-day UNIX programmers, modern shells such as `bash` and `tcsh` automatically reset the terminal when processes invoked from the shell are killed, so these users will not see this happen.

3. However, even with the shell's ability to reset the terminal, it is still not necessarily immune to the problem that occurs when the program turns on non-blocking I/O[2]. If the program turns on non-blocking reads and is subsequently killed, there is a good possibility that the shell will be killed too. This is because, when a program is invoked from the shell, it shares the file descriptors of its parent shell. In other words, file descriptor 0 in the program points to the same file structure in the kernel as it does in the shell. Suppose that the program turns on the `O_NONBLOCK` flag on the standard input connection. It is actually modifying the connection that its parent, i.e., the shell, uses as well. In fact, this standard input connection is shared by all related processes – siblings, cousins, and so on. Any process that is sharing this file connection can potentially make changes that affect all other processes that use this terminal. Once the changes are made, unless they are undone before the process terminates, the shell's connection has those changed properties.

   Now think about the shell for a moment. A shell is basically running a loop of the form

   ```
   while ( not end of input ) {
       display the shell prompt
       read the user's command line
       carry out the instruction
   }
   ```

   Thus, if the program turns on non-blocking reads and is killed before turning it off, when the shell resumes, it executes a read command. Usually the shell is in blocked input mode, so when it tries to read input but none is there, it enters a blocked state waiting for the user to press the Enter key, which sends input to the shell process. Since non-blocking input is still on, instead of waiting for the user's input, it receives an `EOF` from the function call that it uses to retrieve the input. This `EOF` may cause the shell itself to terminate because most shells are killed by the `EOF` character. You should set `ignoreeof` to prevent this. In `bash`, you add the line

[2]Some shells appear to have fixed this "bug" as well. If a spawned process leaves the `O_NONBLOCK` flag on standard input, they clear it.

```
set -o ignoreeof
```

in your `.bashrc` to do this. Anyway, the result is that your shell is killed, and if this is a login shell, you will be logged out. On my host machine, `bash` gets caught in a segmentation fault, which should not happen.

4. The problem with non-blocking reads causing the shell to exit can also be solved by opening a new file connection to the terminal instead of using standard input. In the demo program, explore the effect of replacing the line

```
fp = stdin;
```

by

```
fp = fopen(ctermid(NULL), "r");
```

5.3 Signals

Signals are, as Richard Stevens once put it, software interrupts. They are a mechanism for handling asynchronous events, such as when a user types Ctrl-C at a terminal. Most non-trivial applications need to handle signals. In this section we provide an overview of signals, including what they are, how they are generated, how they are named, and how processes can deal with them.

From a strictly technical point of view, a signal is a message that has a type but does not have content. Messages are usually defined to be containers for data. Signals are not containers. Signals outside of the computer are things like traffic lights, hand gestures that mean "please stop, taxi driver" or "give me the check", alarm clock rings, or warning lights like "you're about to run out of gas". They do not have contents. It is enough that they have identity, so a particular signal type has well-defined meaning. When a combatant raises a white flag, the enemy knows that this signal means "I give up." In UNIX, a signal is simply an integer with a mnemonic name. For example, `SIGINT` is the interrupt signal.

5.3.1 Typing Ctrl-C at a Terminal

When you type a Ctrl-C, the effect is to terminate the currently running process. Why? When you type the Ctrl-C, the character code for it is sent by hardware and then, within the kernel, to the terminal's device driver. The device driver checks the character code and sees that it matches the `INTR` code[3]. Since it knows this is a control character that is supposed to cause delivery of an interrupt signal, it checks whether the `isig` attribute is set for the terminal. If `isig` is set, then the corresponding signal must be delivered, which is `SIGINT`, so it calls the signal subsystem of the kernel to notify it to send the `SIGINT` signal to all processes whose control terminal[4] is the one that received the Ctrl-C. If any of these processes has not explicitly notified the kernel of how it wants to handle this signal, it will terminate upon receipt of the signal, because by default, processes are killed if they do not catch `SIGINT`. Soon we shall see that a process can declare what the disposition of a delivered signal should be while it is running.

[3] In SunOS it is `INTR`. On other systems, it might be `VINTR`.
[4] The operating system keeps track of which processes are attached to which terminals.

5.3.2 Sources of Signals

All signals are sent by the kernel to processes. There is no other way for a process to receive a signal; the kernel is like the central signal processing station inside the machine. The kernel will send a signal to one or more processes if it receives a request to do so. Requests can come from a few different types of sources.

The terminal A user can type a key combination that causes the terminal driver to ask the kernel to send a signal. This is an asynchronous signal, since it can arrive at a process at any time, independent of what the process might be doing. Examples include Ctrl-C, Ctrl-Z, Ctrl-S.

Hardware Hardware exceptions can generate signals. The kernel detects when the exception occurs and sends a signal to the offending process. These may be synchronous or asynchronous. Synchronous events are things such as floating-point exceptions, illegal instructions, addressing exceptions (such as attempts to access addresses outside of the process's address space), and other events generally caused by the process itself. They are synchronous because if the process is run again, they will occur again at the same point in the process's execution. Asynchronous events are things like power loss and terminal hang-ups.

Software Software conditions can generate signals when something noteworthy happens. This can happen when out-of-band data arrives over a network connection, or when a process writes to a pipe after the reader of the pipe has terminated, or when an alarm clock set by the process expires.

Processes Processes themselves can request the kernel to send signals to processes, even themselves. This is not so strange; an alarm clock is a way for you to send a signal to yourself in the future; you set the alarm and wait for it to signal you. Similarly, a process can ask the kernel to send a wake-up call to itself at some future time. A process can also ask the kernel to send a signal to other processes to which it has permission to send signals. For ordinary user processes, these include any processes with the same real or effective user-id.

5.3.3 Signal Types

UNIX systems define signals in the header file `<signal.h>`. More accurately, the header file `<signal.h>` includes the header file that contains the signal definitions. Different UNIX systems store this file in different places. In Linux, the file `<bits/signum.h>` is where the signals are defined. The kernel includes this header file, but user level program are supposed to include `<signal.h>`. The idea is to keep separate headers for the kernel and the user-level programs.

The exact set of signals varies from one system to another, but some of them are standard across all systems. Signal names are just names for small integers such as `SIGINT`, `SIGKILL`, `SIGHUP`, and `SIGCHLD`. All names begin with the prefix `SIG`. `SIGHUP` is the *hang-up* signal. It is sent to a process when its control terminal has been disconnected. `SIGCHLD` is the signal sent to a parent by the kernel when it detects that one of its child processes has terminated. There are typically about 30 to 35 different signals defined in any UNIX system. The list of signals has changed over the years. The first 30 signals listed below are found in Linux and Solaris 9 ; the last 4 only in Solaris 9. The constant `NSIG` is the the total number of signals defined. Since the signal numbers are allocated consecutively, `NSIG` is also one greater than the largest defined signal number.

Name	Value	Default	Event	Note	Category
SIGHUP	1	Exit	Hangup		Termination
SIGINT	2	Exit	Interrupt		Termination
SIGQUIT	3	Core	Quit		Termination
SIGILL	4	Core	Illegal Instruction		Program Error
SIGTRAP	5	Core	Trace or Breakpoint Trap		Program Error
SIGABRT	6	Core	Abort		Program Error
SIGEMT	7	Core	Emulation Trap		Program Error
SIGFPE	8	Core	Arithmetic Exception		Program Error
SIGKILL	9	Exit	Killed		Termination
SIGBUS	10	Core	Bus Error	1	Program Error
SIGSEGV	11	Core	Segmentation Fault		Program Error
SIGSYS	12	Core	Bad System Call	1	Program Error
SIGPIPE	13	Exit	Broken Pipe		Operation Error
SIGALRM	14	Exit	Alarm Clock		Alarm
SIGTERM	15	Exit	Terminated		Termination
SIGUSR1	16	Exit	User Signal 1	1	Miscellaneous
SIGUSR2	17	Exit	User Signal 2	1	Miscellaneous
SIGCHLD	18	Ignore	Child Status Changed	1	Job Control
SIGPWR	19	Ignore	Power Fail or Restart		Hardware
SIGURG	21	Ignore	Urgent Socket Condition		Asynchronous I/O
SIGPOLL	22	Exit	Pollable Event		Asynchronous I/O
SIGSTOP	23	Stop	Stopped (signal)	1	Job Control
SIGTSTP	24	Stop	Stopped (user)	1	Job Control
SIGCONT	25	Ignore	Continued	1	Job Control
SIGTTIN	26	Stop	Stopped (tty input)	1	Job Control
SIGTTOU	27	Stop	Stopped (tty output	1	Job Control
SIGVTALRM	28	Exit	Virtual Timer Expired	1	Alarm
SIGPROF	29	Exit	Profiling Timer Expired	1	Alarm
SIGXCPU	30	Core	CPU time limit exceeded	1	Operation Error
SIGXFSZ	31	Core	File size limit exceeded	1	Operation Error
SIGWINCH	20	Ignore	Window Size Change	2	Miscellaneous
SIGWAITING	32	Ignore	Concurrency signal	3	Miscellaneous
SIGLWP	33	Ignore	Inter-LWP signal	3	Miscellaneous
SIGFREEZE	34	Ignore	Check point Freeze	3	Miscellaneous
SIGTHAW	35	Ignore	Check point Thaw	3	Miscellaneous

Notes

1. In Linux the numerical value of the signal is architecture-dependent.
2. This is only found in Sun OS and BSD.
3. These are in Solaris 9.

The above list has four columns. The first is the mnemonic name for the signal, i.e., the name that can be used in a program. The second is the integer value, which you do not need to know. The third is the default action that happens to a process. For example, `SIGCHLD` is ignored by default,

`SIGSTOP` causes the program to stop by default, and `SIGINT` causes the process to terminate. The last column indicates the cause or condition that leads to this signal.

5.3.4 Sending Signals

In UNIX, the kernel can send a signal to a process when some hardware error condition arises. For example, if a program attempts to execute an illegal instruction, the kernel will receive the hardware notification and will send the `SIGILL` (illegal instruction signal) to the offending process. A process can also send a signal to one or more processes (or even itself) by using the `kill()` system call. The form of the call is

```
int kill(int processid, int signal);
```

The first parameter stores a means to specify the process id of the process to receive the signal. The second parameter is the kind of signal to send. In the simplest case,

```
kill(942, SIGTERM);
```

sends the `SIGTERM` signal to the process whose process-id is 942. A process cannot send a signal to another process if they do not share the same real or effective user-id1. If a process does not have permission to issue the kill call, `kill()` returns –1.

`processid` can be 0, -1, or a negative number, and it means something different in each case. If `processid` is 0, the signal will be sent to all processes in the same process group, whereas if it is –1, and the sender is not the superuser, it is sent to all processes for which it has permission to send signals, which are those processes with the same real or effective user-id. If `processid` $<$ -1, it is sent to all processes in the process group with id `-processid`.

A process can also send a signal to itself using

```
int raise( int signal);
```

which is equivalent to

```
kill(getpid(), signal);
```

The call to `raise()` will return only when the process has handled the signal.

5.3.5 Signal Generation and Delivery

UNIX systems generally distinguish between the generation of a signal and its delivery. According to the *Open Group Base Specification Issue 6* (IEEE Std 1003.1),

> "A signal is said to be ***'generated'*** for (or sent to) a process or thread when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, signals generated via the sigevent structure and terminal activity, as well as invocations of the kill() and sigqueue() functions. In some circumstances, the same event generates signals for multiple processes.
>
> "A signal is ***delivered*** to a process when the appropriate action for that process and signal is taken."

What this means is that delivery takes place when the process receives the signal and responds by either

- *ignoring* it,
- taking the *default* action, or
- executing a *signal handler* for it.

Signal handling is described below. The point to remember now is that from the moment that a signal is generated for a process until the moment that the signal is delivered, the signal is *pending*. Pending signals are managed by the operating system[5].

5.3.6 Signal Handling

A process does not have to accept the default action caused by a signal. It can choose to respond differently to all signals except for `SIGKILL` and `SIGSTOP`. These signals always terminate the process. To handle a signal, the programmer defines a function called a *signal handler*. The signal handler is executed when the signal is received, provided that it has been installed.

The program notifies the operating system that it has a handler for a specific signal by executing a system call to install the handler. The original system call for installing a signal handler was the `signal()` system call. The `signal()` system call was unreliable because it was possible to miss signals when using it. It was replaced by a reliable signal installing call named `sigaction()`. We will explore the `sigaction()` call later. For now, we start with the `signal()` system call, partly because it is an easier one to use, and partly so that you understand its weaknesses. Once you do, you should avoid using it.

The `signal()` call has the form

```
signal( signal_number, handler_action)
```

The first parameter is the number of a signal, but you should always use its mnemonic name such as `SIGINT` or `SIGQUIT`. The second parameter is one of the following:

`SIG_DFL` Take the default action, which is usually to terminate the process.

`SIG_IGN` Ignore the signal completely and continue.

user-defined function Address of a user-defined function

[5]This discussion of signals overlooks the complexity entailed because of threads and multi-threaded processes. Until we discuss threads in general, we have to overlook this topic. But you should bear in mind that the operating system has to make decisions when signals are generated as to whether they are to be sent to every thread in a process or just to a single thread in particular, and that certain signals must always be sent to one choice or the other.

Examples

The following program, `signal_demo1.c`, shows how signal handlers for Ctrl-C (`SIGINT`) and Ctrl-\ (`SIGQUIT`) are installed.

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

void catch1(int signum)
{
    printf("You can do better than that!\n");
}

void catch2(int signum)
{
    printf("I'm no quitter!\n");
}

int main()
{
    int i;

    signal( SIGINT, catch1 );
    signal( SIGQUIT,catch2 );
    for(i = 20; i > 0; i-- ) {
        printf("Try to kill me with ^C or ^\\.\n");
        sleep(1);
    }
    return 0;
}
```

The call `signal(signum, f)` installs `f()` as the signal handler for the signal `signum`. When `signal()` is executed, `f()` is installed. Until that point, `f()` is not installed. When you run `signal_demo1` and enter a Ctrl-C, the `SIGINT` signal is sent to the process executing `signal_demo1`; as a result, the handler `f()` runs, and when it terminates, the program resumes execution. In `signal_demo1.c`, the only action taken by either handler is to print a message on the screen, simply to show that the function was executed.

The next program, `signal_demo2.c`, is almost the same as `signal_demo1.c` with one exception: `SIGINT` and `SIGQUIT` are ignored by calling `signal()` with `SIG_IGN` as the second argument.

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

int main()
{
    signal( SIGINT, SIG_IGN );    // ignore Ctrl-C
```

```
    signal( SIGQUIT, SIG_IGN );  // ignore Ctrl-\

    for(i = 20; i > 0; i-- ) {
        printf("Try to kill me with ^C or ^\\.\n");
        sleep(1);
    }
    return 0;
}
```

5.3.7 Putting It Together

We revise `term_demo1.c` so that it handles Ctrl-C and Ctrl-\ interrupts (whether from the keyboard or sent via a `kill()` from another process.) This program is `term_demo2.c`. The listing below shows only the changed portion of the code.

```
....
#include        <signal.h>
....  (snip)

    if ( fflags ) {
        ui_parameters.isblocking = FALSE;
        set_nodelay(fileno(fp), fflags);
    }

    signal(SIGINT, interrupt_handler);
    signal(SIGQUIT, interrupt_handler);

    while ( TRUE ) {
        response = get_response(fp, ui_parameters );
.... (snip)

char * signame( int signo)
{
    static char  name[16];
    switch ( signo ) {
    case  SIGINT:
        strcpy(name, "SIGINT");
        break;
    case  SIGQUIT:
        strcpy(name, "SIGQUIT");
    }
    return name;
}

void interrupt_handler(int signum)
{
```

```
    printf("Exiting with signal %s\n", signame(signum));
    save_restore(fileno(stdin), RESTORE, NULL);
    exit(2);
}
```

The major changes are that the `signal()` function is used to install handlers for `SIGINT` and `SIGQUIT`. In this program they share the same handler, named `interrupt_handler()`. This handler prints a message on the standard output and then restores the `termios` structure's flags and the file status flags to what they were before the program was run.

5.3.8 Weaknesses of the Signal Mechanism

Signals in this form do not carry any information other than their particular values. Therefore, they are of limited use. They were never intended to be a robust form of communication, and they are still not completely reliable. The early form of signal handling using the signal() system call was very unreliable. While a process was in the midst of catching a signal, it was unable to detect the arrival of another signal of the same type; any new signals of that type were lost. This means that if two signals of the same type were sent in rapid succession to a process, the second might be lost. Later versions of the signal() function in BSD and in SVR corrected this problem in different ways, so that it now has at least two different behaviors. The modern version in Linux 2.6 combines the semantics of each. It is best to avoid the signal() call for that reason.

5.3.9 Signal Handling The "Right" Way

Signals are a primitive form of interprocess communication by today's standards, but at the time they were conceived, they provided a simple, efficient method of solving the most important interprocess communication problems. The `signal()` system call was early UNIX's method of defining and installing signal handlers. One problem with the `signal()` call is that it needs to be reset each time, like a mouse trap – once it catches a signal, arriving signals are missed. Another problem with `signal()` is that its behavior was left unspecified in the case when multiple signals arrived, and different implementations of UNIX provided different semantics to handle multiple signals.

5.3.10 Multiple Signals

Suppose that a signal handler is in the midst of handling a signal that has been delivered when a second signal is generated and is pending. There are a few possible ways to dispose of this new signal:

- Ignore it completely, effectively losing the new signal;
- Put it in a queue and handle it when the current signal has been handled completely, effectively blocking pending signals while handling the current one;
- Interrupt the processing of the current signal, handle the new signal, and return to the old signal when the new one has been handled, effectively treating the handler like an involuntary recursive function;

In any one UNIX system, you might have found one of these solutions employed rather than the others, without any consistency. The POSIX standard introduced a uniform solution to the problem in the `sigaction()` interface.

5.3.11 The sigaction() call

The `sigaction()` system call allows a process to install a signal handler and to specify how it will respond to multiple arriving signals. Its prototype is

```
#include <signal.h>
int (sigaction (int signum, const struct sigaction *act,
struct sigaction *oldaction);
```

where

`signum` is the value of the signal to be handled

`act` is a pointer to a `sigaction` structure that specifies the handler, masks, and flags for the signal

`oldact` is a pointer to a structure to hold the currently active `sigaction` data.

We will examine the `sigaction` structure first to see how flexible this interface is. Notice that the function name is the same as the name of the structure whose address is passed to it, like the `stat()` function and the `stat` structure.

5.3.12 The sigaction struct

The `sigaction` structure is defined in `<signal.h>`. The definition is unusual because it has two members (`sa_handler` and `sa_sigaction`) that are allowed to overlap in memory and must be used in mutual exclusion. The simplest way to present it is as if it were two different overloaded definitions of the same structure:

```
struct sigaction              // backward-compatible, old-style handler
{
    void (*sa_handler) (int); // the action to take
    sigset_t sa_mask;         // additional signals to block
                              // during handling of the signal
    int  sa_flags;            // flags that affect behavior
};
```

or

```
struct sigaction           // POSIX compliant, new-style handler
{
void (*sa_sigaction) (int, siginfo_t *, void *);
```

```
                                    // pointer to signal handler
    sigset_t sa_mask;               // additional signals to block
                                    // during handling of the signal
    int  sa_flags;                  // flags that affect behavior
    };
```

In other words, there are two different forms of the `sigaction` structure. The first one uses the old-style of handler, and the second uses the newer POSIX compliant method. The structs are otherwise identical.

Notes

In the old-style, the `sa_handler` member does not have to be a pointer to a function. It can also be one of the two flags `SIG_IGN` or `SIG_DFL`. If it is `SIG_IGN`, the signal is ignored; if `SIG_DFL`, then the default action is taken. If a pointer to a handler is supplied, that handler will be run. The handler must have a single integer argument.

In the new style, the `sa_handler` is replaced by a pointer to a function that has three parameters as follows:

- An `sa_mask`, which defines which signals should be blocked while the handler is processing the signal. By default, the signal that caused the handler to run will always be blocked. Adding signals to `SA_MASK` is a way to block other signals as well.
- The `sa_flags` is a flag set that can be used to control how subsequent signals of the same type as the one that caused the handler to run are handled. For example, if a handler is handling a `SIGINT` signal and another `SIGINT` arrives while the process is in the handler, the `sa_flags` will determine how to dispose of the second `SIGINT`. It has no effect on other arriving signals. The flags determine how arriving signals are handled after the handler has been invoked. All flags are independent and `sa_flags` is their bitwise-or. The most important flags are:

  `SA_RESETHAND` If this bit is set, the signal action is reset to `SIG_DFL`. This means that as soon as the signal is delivered, the default action will take place. This flag implies the `SA_NODEFER` flag; signals are not blocked, instead causing the process to take whatever is the default action for the type of signal. The intention is to make the handler behave like the old-style `signal()` handler, since any signal arriving after the first will cause the default behavior.

  `SA_NODEFER` If this bit is set, the kernel will not automatically block the signal while it is being caught. This means that an arriving signal will cause the handler itself to be interrupted and re-entered with the second signal. This is involuntary recursion.

  `SA_RESTART` If this bit is set, certain system calls that would otherwise be terminated if a signal were delivered during their execution, will be automatically restarted. This bit allows the BSD style handling.

  `SA_SIGINFO` If this bit is set, two additional arguments are passed to the signal-catching function. If the second argument is not `NULL`, it points to a `siginfo_t` structure containing the reason why the signal was generated; the third argument points to a `ucontext_t` structure containing the receiving process's context when the signal was delivered.

Note. If multiple instances of an individual signal are delivered while that signal is currently blocked, then only one instance is queued. For example, if you issue multiple `SIGINT` signals to a process, without setting the `SA_NODEFER` flag in the handler, then the first one will cause the handler to run, the second will be queued, but all those after that will be lost.

5.3.12.1 Example

The first example, sigaction_demo1.c, shows how the `SA_SIGINFO` flag can be used. A signal handler for `SIGINT` is installed with the `SA_SIGINFO` flag set. The program delays itself using `sleep(60)` so that the user has time to enter a Ctrl-C. When the Ctrl-C is entered and the program is delivered a `SIGINT` signal, the handler runs and accesses the `siginfo_t` structure to print the values of its members as a result of the signal.

Listing: sigaction_demo1.c

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <bits/siginfo.h>
#include <stdio.h>
#include <stdlib.h>

void sig_handler (int signo, siginfo_t *info,  void *context)
{
    printf ("Signal number: %d\n", info->si_signo);
    printf ("Error number:  %d\n", info->si_errno);
    printf ("PID of sender: %d\n", info->si_pid);
    printf ("UID of sender: %d\n", info->si_uid);
    exit(1);
}

int main (int argc, char* argv[])
{
   struct sigaction the_action;

   the_action.sa_flags = SA_SIGINFO;
   the_action.sa_sigaction = sig_handler;

   sigaction(SIGINT, &the_action, NULL);

   printf ("Type Ctrl-C within the next minute"
           "or send signal 2. \n");
   sleep(60);
   return 0;
}
```

5.3.12.2 Example

The following program will demonstrate the use of the `sigaction` structure for old-style handlers as well as the new-style handlers. The program begins by defining the `sigaction` structure that will be passed to the `sigaction` function. The handler function, `interrupt_handler`, is assigned to the `sa_handler` field. The `sa_flags` field is initialized with the bitwise-or of the flags referenced in the command-line. The `sa_mask` is built step by step. There are a few ways to do this. You can start with an empty set and add to it or start with a full set and remove from it. In this case, I am allowing the user to interactively add signal numbers to the blocked set, so I start with an empty set and add to it. The method of creating these sets is discussed below.

```
Listing sigaction_demo2.c
/*
 * Usage:
 *        sigaction_demo  [ reset | noblock | restart ]*
 *
 * i.e., 0 or more of the words reset, noblock, and restart may appear
 * on the command line, and multiple instances of the same word as the same
 * effect as a single instance.
 *
 * reset   turns on SA_RESETHAND
 * noblock turns on SA_NODEFER
  * restart turns on SA_RESTART
 *
 * NOTES
 * (1) If you supply the word "reset" on the command line it will set the
 *     handling to SIG_DFL for signals that arrive when the process is
 *     in the handler. If noblock is also set, the signal will have the
 *     default behavior immediately. If it is not set, the default will
 *     delay until after the handler exits. If noblock is set but reset is
 *     not, it will recursively enter the handler.
 * (2) The interrupt_handler purposely delays for a few seconds in order to
 *     give the user time to enter a few interrupts on the keyboard.
 * (3) interrupt_handler is the handler for both SIGINT and SIGQUIT, so if it
 *     is not reset, neither Ctrl-C nor Ctrl-\ will kill it .
 * (4) It will ask you to enter the numeric values of signals to block. If
 *     you don't give any, no signals are blocked.
 *
 */

#include   <stdio.h>
#include   <string.h>
#include   <signal.h>
#include   <time.h>
#include   <fcntl.h>

#define INPUTLEN        100

int main(int ac, char* av[])
{
    struct sigaction newhandler;        /* new settings         */
    sigset_t     blocked;               /* set of blocked sigs */
    void         inthandler();          /* the handler          */
```

```
    char          x[INPUTLEN];
    int           flags = 0;
    int           signo, n;
    char          s[] = "Entered text: ";
    int           s_len = strlen(s);

    while ( 1 < ac ) {
        if (0 ==  strncmp("reset", av[ac-1], strlen(av[ac-1])) )
            flags |= SA_RESETHAND;
        else if (0 ==  strncmp("noblock", av[ac-1], strlen(av[ac-1])) )
            flags |= SA_NODEFER;
        else if (0 ==  strncmp("restart", av[ac-1], strlen(av[ac-1])) )
            flags |= SA_RESTART;
        ac--;
    }
    /* load these two members first */
    newhandler.sa_sigaction = interrupt_handler;/* handler function    */
    newhandler.sa_flags = SA_SIGINFO | flags;   /* new style handler   */

    /* then build the list of blocked signals */
    sigemptyset(&blocked);                  /* clear all bits      */

    printf("Type the numeric value of a signal to block (0 to stop):");
    while ( 1 ) {
        scanf( "%d", &signo );
        if ( 0 == signo )
             break;
        sigaddset(&blocked, signo);         /* add signo to list */
        printf("next signal number (0 to stop): ");
    }
    newhandler.sa_mask = blocked;           /* store blockmask     */

    // install inthandler as the SIGINT handler
    if ( sigaction(SIGINT, &newhandler, NULL) == -1 )
        perror("sigaction");

    // if successful, install inthandler as the SIGQUIT handler also
    else if ( sigaction(SIGQUIT, &newhandler, NULL) == -1 )
        perror("sigaction");
    else
        while( 1 ) {
            x[0] = '\0';
            tcflush(0,TCIOFLUSH);
            printf("Enter text then <RET>: (quit to quit)\n");
            n = read(0, &x, INPUTLEN);
            if ( n == -1 && errno == EINTR ) {
                printf("read call was interrupted\n");
                x[n] = '\0';
                write(1, &x, n+1);
            }
            else if ( strncmp("quit", x, 4) == 0 )
                break;
            else {
                x[n] = '\0';
```

```
                write(1, &s, s_len);
                write(1, &x, n+1);
                printf("\n");
            }
        } //while
    return 0;
}

void interrupt_handler (int signo, siginfo_t *info, void *context)
{
    int      localid;     /* stores a number to uniquely identify signal */
    time_t   timenow;     /* current time -- used to generate id */
    static   int ticker = 0;  /* used for id also */
    struct tm *tp;

    time(&timenow);
    tp = localtime(&timenow);
    localid = 36000*tp->tm_hour + 600*tp->tm_min + 10*tp->tm_sec +
              ticker++ % 10;
    printf("Entered handler: sig = %d \tid = %d\n",
            info->si_signo, localid );
    sleep(3);
    printf("Leaving handler: sig = %d \tid = %d\n",
            info->si_signo, localid );
}
```

The while loop in the main program exists just to demonstrate that the system calls to perform I/O are restarted when the interrupts occur. You can run this program with any combination of `SA_RESTART`, `SA_RESETHAND`, and `SA_NODEFER` to see the combined effect of the flags. You can add any signal to the blocked set, but you will only be able to send `SIGINT` and `SIGQUIT` unless you open a separate window and use the `kill` command to send arbitrary signals to the process. You can try this – put signal 4 in the blocked set and issue `kill -4` with the process id from a second window while the process is handling a Ctrl-C. You will see that signal 4 is blocked until `interrupt_handler()` finishes.

The while loop is designed so that you do not have to kill this program to terminate it. If you type `"quit"` it will terminate.

5.3.13 Creating Signal Mask Sets

There are four functions that modify signal mask sets:

```
sigemptyset(sigset_t *setp);
sigfillset(sigset_t *setp);
sigaddset(sigset_t *setp, int signum);
sigdelset(sigset_t *setp, int signum);
```

The first two create empty and full mask sets respectively. The next two add or delete specific signals from the specified sets. You can either create an empty mask and add to it, or create a full mask and delete from it. If you plan on having more than half of the signals in it, then do the latter, otherwise do the former.

5.3.14 Blocking Signals Temporarily around Critical Sections

The `sigprocmask()` system call can be used to block or unblock signals sent to a process. This is useful if you need to temporarily turn off all signals in a small section of code. This is one way to create something like a critical section, in the sense that the process will not be interrupted in the middle of the code fragment. It does not prevent the kernel from preempting the process and letting another process run on the CPU, so it does not provide a means of protecting shared variables that might be accessed while the process is removed from the CPU, but it does allow the process to complete some critical sequence of statements without any signal handlers running in the interim, and without being terminated in the midst of it. The prototype is

```
int  sigprocmask( int how, const sigset_t *sigs, sigset_t *prev);
```

where `how` is one of `SIG_BLOCK`, `SIG_UNBLOCK`, or `SIG_SETMASK`. `SIG_BLOCK` will block the specified signal set, i.e., prevent those signals from reaching until the procmask is changed. `SIG_UNBLOCK` allows the signals in the set to be unblocked, and `SIG_SETMASK` is used to change the mask completely, i.e., assign a new mask to the procmask. The following program can be run to demonstrate how the blocking works. If you type Ctrl-C during the first loop, the process will continue to loop and it will exit before the second loop is executed. If you change the `SIG_BLOCK` to `SIG_UNBLOCK` then the Ctrl-C will kill the process when you type it.

```
Listing: procmask_demo.c
#include <signal.h>
#include <stdio.h>

int main()
{
    int i;
    sigset_t sigs, prevsigs;

    sigemptyset(&sigs);
    sigaddset(&sigs, SIGINT);
    sigprocmask(SIG_BLOCK, &sigs, &prevsigs);

    for ( i = 0; i < 5; i++) {
        printf("Waiting %d\n", i);
        sleep(1);
    }
    sigprocmask(SIG_SETMASK, &prevsigs, NULL);
    for ( i = 0; i < 5; i++) {
         printf("After %d\n", i);
        sleep(1);
    }
}
```

5.4 Summary

An understanding of terminals and signal handling is essential to being able to write UNIX System applications. You now have almost all of the tools at your disposal to write terminal-based interactive programs that can use the full terminal window and allow the user to interact with it at will. As you will discover in the next chapter, you are still missing a few more pieces. For one, we still need to manage timing more accurately and allow events to happen at specific times as controlled by the program. For another, we do not have the means of placing characters anywhere on the screen. This is what comes next.