

11

Processes and Signals

Processes and signals form a fundamental part of the Linux operating environment. They control almost all activities performed by Linux and all other UNIX-like computer systems. An understanding of how Linux and UNIX manage processes will hold any systems programmer, applications programmer, or system administrator in good stead.

In this chapter, you learn how processes are handled in the Linux environment and how to find out exactly what the computer is doing at any given time. You also see how to start and stop other processes from within your own programs, how to make processes send and receive messages, and how to avoid zombies. In particular, you learn about

- ❑ Process structure, type, and scheduling
- ❑ Starting new processes in different ways
- ❑ Parent, child, and zombie processes
- ❑ What signals are and how to use them

What Is a Process?

The UNIX standards, specifically IEEE Std 1003.1, 2004 Edition, defines a process as “an address space with one or more threads executing within that address space, and the required system resources for those threads.” We look at threads in Chapter 12. For now, we will regard a process as just a program that is running.

A multitasking operating system such as Linux lets many programs run at once. Each instance of a running program constitutes a process. This is especially evident with a windowing system such as the X Window System (often simply called X). Like Windows, X provides a graphical user interface that allows many applications to be run at once. Each application can display one or more windows.

As a multiuser system, Linux allows many users to access the system at the same time. Each user can run many programs, or even many instances of the same program, at the same time. The system itself runs other programs to manage system resources and control user access.

Chapter 11: Processes and Signals

As you saw in Chapter 4, a program — or process — that is running consists of program code, data, variables (occupying system memory), open files (file descriptors), and an environment. Typically, a Linux system will share code and system libraries among processes so that there's only one copy of the code in memory at any one time.

Process Structure

Let's have a look at how a couple of processes might be arranged within the operating system. If two users, `neil` and `rick`, both run the `grep` program at the same time to look for different strings in different files, the processes being used might look like Figure 11-1.

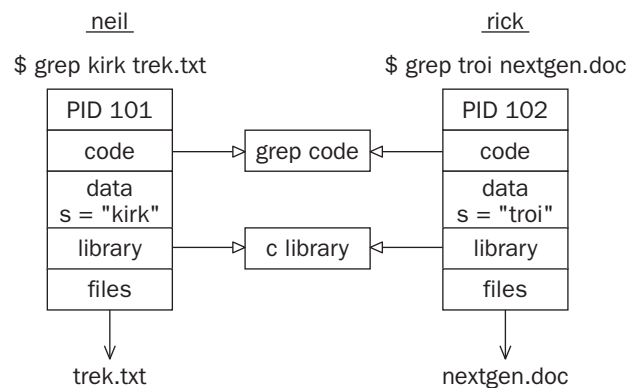


Figure 11-1

If you could run the `ps` command as in the following code quickly enough and before the searches had finished, the output might contain something like this:

```
$ ps -ef
UID    PID   PPID  C   STIME  TTY   TIME      CMD
rick   101    96    0   18:24  tty2   00:00:00  grep troi nextgen.doc
neil   102    92    0   18:24  tty4   00:00:00  grep kirk trek.txt
```

Each process is allocated a unique number, called a *process identifier* or *PID*. This is usually a positive integer between 2 and 32,768. When a process is started, the next unused number in sequence is chosen and the numbers restart at 2 so that they wrap around. The number 1 is typically reserved for the special `init` process, which manages other processes. We will come back to `init` shortly. Here you see that the two processes started by `neil` and `rick` have been allocated the identifiers 101 and 102.

The program code that will be executed by the `grep` command is stored in a disk file. Normally, a Linux process can't write to the memory area used to hold the program code, so the code is loaded into memory as read-only. You saw in Figure 11-1 that, although this area can't be written to, it can safely be shared.

The system libraries can also be shared. Thus, there need be only one copy of `printf`, for example, in memory, even if many running programs call it. This is a more sophisticated, but similar, scheme to the way dynamic link libraries (DLLs) work in Windows.

As you can see in the preceding diagram, an additional benefit is that the disk file containing the executable program `grep` is smaller because it doesn't contain shared library code. This might not seem like much saving for a single program, but extracting the common routines for (say) the standard C library saves a significant amount of space over a whole operating system.

Of course, not everything that a program needs to run can be shared. For example, the variables that it uses are distinct for each process. In this example, you see that the search string passed to the `grep` command appears as a variable, `s`, in the data space of each process. These are separate and usually can't be read by other processes. The files that are being used in the two `grep` commands are also different; the processes have their own set of file descriptors used for file access.

Additionally, a process has its own stack space, used for local variables in functions and for controlling function calls and returns. It also has its own environment space, containing environment variables that may be established solely for this process to use, as you saw with `putenv` and `getenv` in Chapter 4. A process must also maintain its own program counter, a record of where it has gotten to in its execution, which is the *execution thread*. In the next chapter you will see that when you use threads, processes can have more than one thread of execution.

On many Linux systems, and some UNIX systems, there is a special set of "files" in a directory called `/proc`. These are special in that rather than being true files they allow you to "look inside" processes while they are running as if they were files in directories. We took a brief look at the `/proc` file system back in Chapter 3.

Finally, because Linux, like UNIX, has a virtual memory system that pages code and data out to an area of the hard disk, many more processes can be managed than would fit into the physical memory.

The Process Table

The Linux *process table* is like a data structure describing all of the processes that are currently loaded with, for example, their PID, status, and command string, the sort of information output by `ps`. The operating system manages processes using their PIDs, and they are used as an index into the process table. The table is of limited size, so the number of processes a system will support is limited. Early UNIX systems were limited to 256 processes. More modern implementations have relaxed this restriction considerably and may be limited only by the memory available to construct a process table entry.

Viewing Processes

The `ps` command shows the processes you're running, the process another user is running, or all the processes on the system. Here is more sample output:

```
$ ps -ef
  UID      PID  PPID  C  STIME TTY          TIME CMD
  root      433    425  0  18:12 tty1          00:00:00 [bash]
  rick      445    426  0  18:12 tty2          00:00:00 -bash
  rick      456    427  0  18:12 tty3          00:00:00 [bash]
  root      467    433  0  18:12 tty1          00:00:00 sh /usr/X11R6/bin/startx
  root      474    467  0  18:12 tty1          00:00:00 xinit /etc/X11/xinit/xinitrc --
  root      478    474  0  18:12 tty1          00:00:00 /usr/bin/gnome-session
  root      487      1  0  18:12 tty1          00:00:00 gnome-smproxy --sm-client-id def
  root      493      1  0  18:12 tty1          00:00:01 [enlightenment]
  root      506      1  0  18:12 tty1          00:00:03 panel --sm-client-id default8
```

Chapter 11: Processes and Signals

```
root      508      1  0 18:12 tty1      00:00:00 xscreensaver -no-splash -timeout
root      510      1  0 18:12 tty1      00:00:01 gmc --sm-client-id default10
root      512      1  0 18:12 tty1      00:00:01 gnome-help-browser --sm-client-i
root      649     445  0 18:24 tty2      00:00:00 su
root      653     649  0 18:24 tty2      00:00:00 bash
neil      655     428  0 18:24 tty4      00:00:00 -bash
root      713      1  2 18:27 tty1      00:00:00 gnome-terminal
root      715     713  0 18:28 tty1      00:00:00 gnome-pty-helper
root      717     716 13 18:28 pts/0      00:00:01 emacs
root      718     653  0 18:28 tty2      00:00:00 ps -ef
```

This shows information about many processes, including the processes involved with the Emacs editor under X on a Linux system. For example, the `TTY` column shows which terminal the process was started from, `TIME` gives the CPU time used so far, and the `CMD` column shows the command used to start the process. Let's take a closer look at some of these.

```
neil      655     428  0 18:24 tty4      00:00:00 -bash
```

The initial login was performed on virtual console number 4. This is just the console on this machine. The shell program that is running is the Linux default, `bash`.

```
root      467     433  0 18:12 tty1      00:00:00 sh /usr/X11R6/bin/startx
```

The X Window System was started by the command `startx`. This is a shell script that starts the X server and runs some initial X programs.

```
root      717     716 13 18:28 pts/0      00:00:01 emacs
```

This process represents a window in X running Emacs. It was started by the window manager in response to a request for a new window. A new pseudo terminal, `pts/0`, has been assigned for the shell to read from and write to.

```
root      512      1  0 18:12 tty1      00:00:01 gnome-help-browser --sm-client-i
```

This is the GNOME help browser started by the window manager.

By default, the `ps` program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. You can use `ps` to see all such processes using the `-e` option and to get "full" information with `-f`.

The exact syntax for the `ps` command and the format of the output may vary slightly from system to system. The GNU version of `ps` used in Linux supports options taken from several previous implementations of `ps`, including those in BSD and AT&T variants of UNIX and adds more of its own. Refer to the manual for more details on the available options and output format of `ps`.

System Processes

Here are some of the processes running on another Linux system. The output has been abbreviated for clarity. In the following examples you will see how to view the status of a process. The `STAT` output from

Chapter 11: Processes and Signals

`ps` provides codes indicating the current status. Common codes are given in the following table. The meanings of some of these will become clearer later in this chapter. Others are beyond the scope of this book and can be safely ignored.

STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, “runnable,” that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or “zombie” process.
N	Low priority task, “nice.”
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

```
$ ps ax
  PID TTY          STAT TIME   COMMAND
    1 ?            Ss   0:03   init [5]
    2 ?            S    0:00   [migration/0]
    3 ?            SN   0:00   [ksoftirqd/0]
    4 ?            S<   0:05   [events/0]
    5 ?            S<   0:00   [khelper]
    6 ?            S<   0:00   [kthread]
   840 ?           S<   2:52   [kjournald]
   888 ?          S<S   0:03   /sbin/udevd --daemon
  3069 ?           Ss   0:00   /sbin/acpid
  3098 ?           Ss   0:11   /usr/sbin/hald --daemon=yes
  3099 ?           S    0:00   hald-runner
  8357 ?           Ss   0:03   /sbin/syslog-ng
  8677 ?           Ss   0:00   /opt/kde3/bin/kdm
  9119 ?           S    0:11   konsole [kdeinit]
  9120 pts/2        Ss   0:00   /bin/bash
  9151 ?           Ss   0:00   /usr/sbin/cupsd
  9457 ?           Ss   0:00   /usr/sbin/cron
  9479 ?           Ss   0:00   /usr/sbin/sshd -o PidFile=/var/run/sshd.init.pid
```

Chapter 11: Processes and Signals

```
9618 tty1      Ss+    0:00 /sbin/mingetty --noclear tty1
9619 tty2      Ss+    0:00 /sbin/mingetty tty2
9621 tty3      Ss+    0:00 /sbin/mingetty tty3
9622 tty4      Ss+    0:00 /sbin/mingetty tty4
9623 tty5      Ss+    0:00 /sbin/mingetty tty5
9638 tty6      Ss+    0:00 /sbin/mingetty tty6
10359 tty7     Ss+    10:05 /usr/bin/Xorg -br -nolisten tcp :0 vt7 -auth
10360 ?        S       0:00 -:0
10381 ?        Ss     0:00 /bin/sh /usr/bin/kde
10438 ?        Ss     0:00 /usr/bin/ssh-agent /bin/bash /etc/X11/xinit/xinitrc
10478 ?        S       0:00 start_kdeinit --new-startup +kcminit_startup
10479 ?        Ss     0:00 kdeinit Running...
10500 ?        S       0:53 kdesktop [kdeinit]
10502 ?        S       1:54 kicker [kdeinit]
10524 ?        Sl     0:47 beagled /usr/lib/beagle/BeagleDaemon.exe --bg
10530 ?        S       0:02 opensuseupdater
10539 ?        S       0:02 kpowersave [kdeinit]
10541 ?        S       0:03 klipper [kdeinit]
10555 ?        S       0:01 kio_uiserver [kdeinit]
10688 ?        S       0:53 konsole [kdeinit]
10689 pts/1     Ss+    0:07 /bin/bash
10784 ?        S       0:00 /opt/kde3/bin/kdesud
11052 ?        S       0:01 [pdflush]
19996 ?        SNl    0:20 beagled-helper /usr/lib/beagle/IndexHelper.exe
20254 ?        S       0:00 qmgr -l -t fifo -u
21192 ?        Ss     0:00 /usr/sbin/ntpd -p /var/run/ntp/ntpd.pid -u ntp -i /v
21198 ?        S       0:00 pickup -l -t fifo -u
21475 pts/2     R+     0:00 ps ax
```

Here you can see one very important process indeed.

```
1 ? Ss      0:03 init [5]
```

In general, each process is started by another process known as its *parent process*. A process so started is known as a *child process*. When Linux starts, it runs a single program, the prime ancestor and process number 1, `init`. This is, if you like, the operating system process manager and the grandparent of all processes. Other system processes you'll meet soon are started by `init` or by other processes started by `init`.

One such example is the login procedure. `init` starts the `getty` program once for each serial terminal or dial-in modem that you can use to log in. These are shown in the `ps` output like this:

```
9619 tty2      Ss+    0:00 /sbin/mingetty tty2
```

The `getty` processes wait for activity at the terminal, prompt the user with the familiar login prompt, and then pass control to the login program, which sets up the user environment and finally starts a shell. When the user shell exits, `init` starts another `getty` process.

You can see that the ability to start new processes and to wait for them to finish is fundamental to the system. You'll see later in this chapter how to perform the same tasks from within your own programs with the system calls `fork`, `exec`, and `wait`.

Process Scheduling

One further `ps` output example is the entry for the `ps` command itself:

```
21475 pts/2    R+      0:00 ps ax
```

This indicates that process 21475 is in a run state (R) and is executing the command `ps ax`. Thus the process is described in its own output! The status indicator shows only that the program is ready to run, not necessarily that it's actually running. On a single-processor computer, only one process can run at a time, while others wait their turn. These turns, known as time slices, are quite short and give the impression that programs are running at the same time. The `R+` just shows that the program is a foreground task not waiting for other processes to finish or waiting for input or output to complete. That is why you may see two such processes listed in `ps` output. (Another commonly seen process marked as running is the X display server.)

The Linux kernel uses a process scheduler to decide which process will receive the next time slice. It does this using the process priority (we discussed priorities back in Chapter 4). Processes with a high priority get to run more often, whereas others, such as low-priority background tasks, run less frequently. With Linux, processes can't overrun their allocated time slice. They are preemptively multitasked so that they are suspended and resumed without their cooperation. Older systems, such as Windows 3.x, generally require processes to yield explicitly so that others may resume.

In a multitasking system such as Linux where several programs are likely to be competing for the same resource, programs that perform short bursts of work and pause for input are considered better behaved than those that hog the processor by continually calculating some value or continually querying the system to see if new input is available. Well-behaved programs are termed *nice* programs, and in a sense this "niceness" can be measured. The operating system determines the priority of a process based on a "nice" value, which defaults to 0, and on the behavior of the program. Programs that run for long periods without pausing generally get lower priorities. Programs that pause while, for example, waiting for input, get rewarded. This helps keep a program that interacts with the user responsive; while it is waiting for some input from the user, the system increases its priority, so that when it's ready to resume, it has a high priority. You can set the process nice value using `nice` and adjust it using `renice`. The `nice` command increases the nice value of a process by 10, giving it a lower priority. You can view the nice values of active processes using the `-l` or `-f` (for long output) option to `ps`. The value you are interested in is shown in the `NI` (nice) column.

```
$ ps -l
 F S   UID     PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
000 S   500    1259    1254  0  75   0 -  710 wait4 pts/2    00:00:00 bash
000 S   500    1262    1251  0  75   0 -  714 wait4 pts/1    00:00:00 bash
000 S   500    1313    1262  0  75   0 - 2762 schedu pts/1    00:00:00 emacs
000 S   500    1362    1262  2  80   0 -  789 schedu pts/1    00:00:00 oclock
000 R   500    1363    1262  0  81   0 -  782 -      pts/1    00:00:00 ps
```

Here you can see that the `oclock` program is running (as process 1362) with a default nice value. If it had been started with the command

```
$ nice oclock &
```

Chapter 11: Processes and Signals

it would have been allocated a nice value of +10. If you adjust this value with the command

```
$ renice 10 1362
1362: old priority 0, new priority 10
```

the clock program will run less often. You can see the modified nice value with `ps` again:

```
$ ps -l
 F S      UID      PID  PPID  C  PRI   NI  ADDR  SZ  WCHAN  TTY          TIME CMD
000 S      500     1259   1254  0   75    0   -    710  wait4  pts/2      00:00:00 bash
000 S      500     1262   1251  0   75    0   -    714  wait4  pts/1      00:00:00 bash
000 S      500     1313   1262  0   75    0  2762  schedu  pts/1      00:00:00 emacs
000 S      500     1362   1262  0   90   10   -    789  schedu  pts/1      00:00:00 oclock
000 R      500     1365   1262  0   81    0   -    782  -      pts/1      00:00:00 ps
```

The status column now also contains `N` to indicate that the nice value has changed from the default.

```
$ ps x
  PID TTY          STAT       TIME COMMAND
 1362 pts/1        SN          0:00 oclock
```

The `PPID` field of `ps` output indicates the parent process ID, the PID of either the process that caused this process to start or, if that process is no longer running, `init` (PID 1).

The Linux scheduler decides which process it will allow to run on the basis of priority. Exact implementations vary, of course, but higher-priority processes run more often. In some cases, low-priority processes don't run at all if higher-priority processes are ready to run.

Starting New Processes

You can cause a program to run from inside another program and thereby create a new process by using the `system` library function.

```
#include <stdlib.h>

int system (const char *string);
```

The `system` function runs the command passed to it as a string and waits for it to complete. The command is executed as if the command

```
$ sh -c string
```

has been given to a shell. `system` returns 127 if a shell can't be started to run the command and -1 if another error occurs. Otherwise, `system` returns the exit code of the command.

Try It Out system

You can use `system` to write a program to run `ps`. Though this is not tremendously useful in and of itself, you'll see how to develop this technique in later examples. (We don't check that the `system` call actually worked for the sake of simplicity in the example.)

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

int main()
{
    printf("Running ps with system\n");
    system("ps ax");
    printf("Done.\n");
    exit(0);
}
```

When you compile and run this program, `system1.c`, you get something like the following:

```
$ ./system1
Running ps with system
  PID TTY          STAT TIME COMMAND
    1 ?           Ss    0:03 init [5]
...

1262 pts/1      Ss    0:00 /bin/bash
1273 pts/2      S      0:00 su -
1274 pts/2      S+    0:00 -bash
1463 pts/2      SN    0:00 oclock
1465 pts/1      S      0:01 emacs Makefile
1480 pts/1      S+    0:00 ./system1
1481 pts/1      R+    0:00 ps ax
Done.
```

Because the `system` function uses a shell to start the desired program, you could put it in the background by changing the function call in `system1.c` to the following:

```
system("ps ax &");
```

When you compile and run this version of the program, you get something like

```
$ ./system2
Running ps with system
  PID TTY          STAT TIME COMMAND
    1 ?           S      0:03 init [5]
...
Done.
$ 1274 pts/2      S+    0:00 -bash
1463 pts/1      SN    0:00 oclock
1465 pts/1      S      0:01 emacs Makefile
1484 pts/1      R      0:00 ps ax
```

Chapter 11: Processes and Signals

How It Works

In the first example, the program calls `system` with the string `"ps ax"`, which executes the `ps` program. The program returns from the call to `system` when the `ps` command has finished. The `system` function can be quite useful but is also limited. Because the program has to wait until the process started by the call to `system` finishes, you can't get on with other tasks.

In the second example, the call to `system` returns as soon as the shell command finishes. Because it's a request to run a program in the background, the shell returns as soon as the `ps` program is started, just as would happen if you had typed

```
$ ps ax &
```

at a shell prompt. The `system2` program then prints `Done.` and exits before the `ps` command has had a chance to finish all of its output. The `ps` output continues to produce output after `system2` exits and in this case does not include an entry for `system2`. This kind of process behavior can be quite confusing for users. To make good use of processes, you need finer control over their actions. Let's look at a lower-level interface to process creation, `exec`.

In general, using `system` is a far from ideal way to start other processes, because it invokes the desired program using a shell. This is both inefficient, because a shell is started before the program is started, and also quite dependent on the installation for the shell and environment that are used. In the next section, you see a much better way of invoking programs, which should almost always be used in preference to the `system` call.

Replacing a Process Image

There is a whole family of related functions grouped under the `exec` heading. They differ in the way that they start processes and present program arguments. An `exec` function replaces the current process with a new process specified by the `path` or `file` argument. You can use `exec` functions to "hand off" execution of your program to another. For example, you could check the user's credentials before starting another application that has a restricted usage policy. The `exec` functions are more efficient than `system` because the original program will no longer be running after the new one is started.

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ..., (char *)0);
int execlp(const char *file, const char *arg0, ..., (char *)0);
int execl_e(const char *path, const char *arg0, ..., (char *)0, char *const
envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

These functions belong to two types. `execl`, `execlp`, and `execl_e` take a variable number of arguments ending with a null pointer. `execv` and `execvp` have as their second argument an array of strings. In both cases, the new program starts with the given arguments appearing in the `argv` array passed to `main`.

Chapter 11: Processes and Signals

These functions are usually implemented using `execve`, though there is no requirement for it to be done this way.

The functions with names suffixed with a `p` differ in that they will search the `PATH` environment variable to find the new program executable file. If the executable isn't on the path, an absolute filename, including directories, will need to be passed to the function as a parameter.

The global variable `environ` is available to pass a value for the new program environment. Alternatively, an additional argument to the functions `execle` and `execve` is available for passing an array of strings to be used as the new program environment.

If you want to use an `exec` function to start the `ps` program, you can choose from among the six `exec` family functions, as shown in the calls in the code fragment that follows:

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
char *const ps_argv[] =
    {"ps", "ax", 0};

/* Example environment, not terribly useful */
char *const ps_envp[] =
    {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "ax", 0);           /* assumes ps is in /bin */
execlp("ps", "ps", "ax", 0);              /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "ax", 0, ps_envp); /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Try It Out **execlp**

Let's modify the example to use an `execlp` call:

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

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "ax", 0);
    printf("Done.\n");
    exit(0);
}
```

Chapter 11: Processes and Signals

When you run this program, `pexec.c`, you get the usual `ps` output, but no `Done.` message at all. Note also that there is no reference to a process called `pexec` in the output.

```
$ ./pexec
Running ps with execlp
  PID TTY          STAT TIME  COMMAND
    1 ?            S     0:03  init [5]
...
 1262 pts/1        Ss     0:00  /bin/bash
 1273 pts/2        S       0:00  su -
 1274 pts/2        S+     0:00  -bash
 1463 pts/1        SN     0:00  oclock
 1465 pts/1        S       0:01  emacs Makefile
 1514 pts/1        R+     0:00  ps ax
```

How It Works

The program prints its first message and then calls `execlp`, which searches the directories given by the `PATH` environment variable for a program called `ps`. It then executes this program in place of the `pexec` program, starting it as if you had given the shell command

```
$ ps ax
```

When `ps` finishes, you get a new shell prompt. You don't return to `pexec`, so the second message doesn't get printed. The PID of the new process is the same as the original, as are the parent PID and nice value. In effect, all that has happened is that the running program has started to execute new code from a new executable file specified in the call to `exec`.

There is a limit on the combined size of the argument list and environment for a process started by `exec` functions. This is given by `ARG_MAX` and on Linux systems is 128K bytes. Other systems may set a more reduced limit that can lead to problems. The POSIX specification indicates that `ARG_MAX` should be at least 4,096 bytes.

The `exec` functions generally don't return unless an error occurs, in which case the error variable `errno` is set and the `exec` function returns `-1`.

The new process started by `exec` inherits many features from the original. In particular, open file descriptors remain open in the new process unless their "close on exec flag" has been set (refer to the `fcntl` system call in Chapter 3 for more details). Any open directory streams in the original process are closed.

Duplicating a Process Image

To use processes to perform more than one function at a time, you can either use threads, covered in Chapter 12, or create an entirely separate process from within a program, as `init` does, rather than replace the current thread of execution, as in the `exec` case.

You can create a new process by calling `fork`. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process

is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. Combined with the `exec` functions, `fork` is all you need to create new processes.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

As you can see in Figure 11-2, the call to `fork` in the parent returns the PID of the new child process. The new process continues to execute just like the original, with the exception that in the child process the call to `fork` returns 0. This allows both the parent and child to determine which is which.

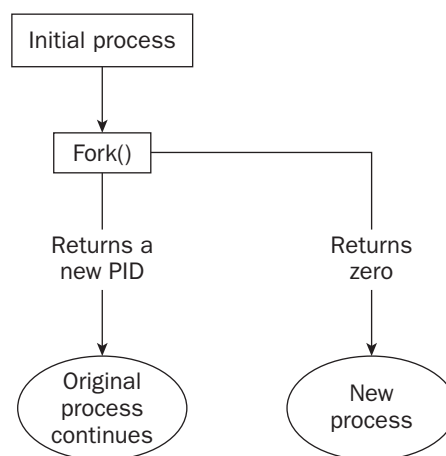


Figure 11-2

If `fork` fails, it returns `-1`. This is commonly due to a limit on the number of child processes that a parent may have (`CHILD_MAX`), in which case `errno` will be set to `EAGAIN`. If there is not enough space for an entry in the process table, or not enough virtual memory, the `errno` variable will be set to `ENOMEM`.

A typical code fragment using `fork` is

```
pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 :    /* Error */
    break;
case 0 :    /* We are child */
    break;
default :   /* We are parent */
    break;
}
```

Chapter 11: Processes and Signals

Try It Out **fork**

Let's look at a simple example, `fork1.c`:

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

int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            message = "This is the child";
            n = 5;
            break;
        default:
            message = "This is the parent";
            n = 3;
            break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```

This program runs as two processes. A child is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

```
$ ./fork1
fork program starting
This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

How It Works

When `fork` is called, this program divides into two separate processes. The parent process is identified by a nonzero return from `fork` and is used to set a number of messages to print, each separated by one second.

Waiting for a Process

When you start a child process with `fork`, it takes on a life of its own and runs independently. Sometimes, you would like to find out when a child process has finished. For example, in the previous program, the parent finishes ahead of the child and you get some messy output as the child continues to run. You can arrange for the parent process to wait until the child finishes before continuing by calling `wait`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

The `wait` system call causes a parent process to pause until one of its child processes is stopped. The call returns the PID of the child process. This will normally be a child process that has terminated. The status information allows the parent process to determine the exit status of the child process, that is, the value returned from `main` or passed to `exit`. If `stat_loc` is not a null pointer, the status information will be written to the location to which it points.

You can interpret the status information using macros defined in `sys/wait.h`, shown in the following table.

Macro	Definition
<code>WIFEXITED(stat_val)</code>	Nonzero if the child is terminated normally.
<code>WEXITSTATUS(stat_val)</code>	If <code>WIFEXITED</code> is nonzero, this returns child exit code.
<code>WIFSIGNALED(stat_val)</code>	Nonzero if the child is terminated on an uncaught signal.
<code>WTERMSIG(stat_val)</code>	If <code>WIFSIGNALED</code> is nonzero, this returns a signal number.
<code>WIFSTOPPED(stat_val)</code>	Nonzero if the child has stopped.
<code>WSTOPSIG(stat_val)</code>	If <code>WIFSTOPPED</code> is nonzero, this returns a signal number.

Try It Out wait

In this Try It Out, you modify the program slightly so you can wait for and examine the child process exit status. Call the new program `wait.c`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
```

Chapter 11: Processes and Signals

```
#include <stdlib.h>

int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {

    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
```

This section of the program waits for the child process to finish.

```
if (pid != 0) {
    int stat_val;
    pid_t child_pid;

    child_pid = wait(&stat_val);

    printf("Child has finished: PID = %d\n", child_pid);
    if(WIFEXITED(stat_val))
        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
    else
        printf("Child terminated abnormally\n");
}
exit(exit_code);
}
```

When you run this program, you see the parent wait for the child.

```
$ ./wait
fork program starting
```



```
This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 1582
Child exited with code 37
$
```

How It Works

The parent process, which got a nonzero return from the `fork` call, uses the `wait` system call to suspend its own execution until status information becomes available for a child process. This happens when the child calls `exit`; we gave it an exit code of 37. The parent then continues, determines that the child terminated normally by testing the return value of the `wait` call, and extracts the exit code from the status information.

Zombie Processes

Using `fork` to create processes can be very useful, but you must keep track of child processes. When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls `wait`. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls `wait`. It becomes what is known as *defunct*, or a *zombie process*.

You can see a zombie process being created if you change the number of messages in the `fork` example program. If the child prints fewer messages than the parent, it will finish first and will exist as a zombie until the parent has finished.

Try It Out Zombies

`fork2.c` is the same as `fork1.c`, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
case -1:
    perror("fork failed");
    exit(1);
case 0:
    message = "This is the child";
    n = 3;
    break;
default:
    message = "This is the parent";
    n = 5;
    break;
}
```

Chapter 11: Processes and Signals

How It Works

If you run the preceding program with `./fork2 &` and then call the `ps` program after the child has finished but before the parent has finished, you'll see a line such as this. (Some systems may say `<zombie>` rather than `<defunct>`.)

```
$ ps -al
```

	F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
004	S		0	1273	1259	0	75	0	-	589	wait4	pts/2	00:00:00	su
000	S		0	1274	1273	0	75	0	-	731	schedu	pts/2	00:00:00	bash
000	S		500	1463	1262	0	75	0	-	788	schedu	pts/1	00:00:00	oclock
000	S		500	1465	1262	0	75	0	-	2569	schedu	pts/1	00:00:01	emacs
000	S		500	1603	1262	0	75	0	-	313	schedu	pts/1	00:00:00	fork2
003	Z		500	1604	1603	0	75	0	-	0	do_exi	pts/1	00:00:00	fork2 <defunct>
000	R		500	1605	1262	0	81	0	-	781	-	pts/1	00:00:00	ps

If the parent then terminates abnormally, the child process automatically gets the process with PID 1 (`init`) as parent. The child process is now a zombie that is no longer running but has been inherited by `init` because of the abnormal termination of the parent process. The zombie will remain in the process table until collected by the `init` process. The bigger the table, the slower this procedure. You need to avoid zombie processes, because they consume resources until `init` cleans them up.

There's another system call that you can use to wait for child processes. It's called `waitpid`, and you can use it to wait for a specific process to terminate.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

The `pid` argument specifies the PID of a particular child process to wait for. If it's `-1`, `waitpid` will return information for any child process. Like `wait`, it will write status information to the location pointed to by `stat_loc`, if that is not a null pointer. The `options` argument allows you to modify the behavior of `waitpid`. The most useful option is `WNOHANG`, which prevents the call to `waitpid` from suspending execution of the caller. You can use it to find out whether any child processes have terminated and, if not, to continue. Other options are the same as for `wait`.

So, if you wanted to have a parent process regularly check whether a specific child process has terminated, you could use the call

```
waitpid(child_pid, (int *) 0, WNOHANG);
```

This will return zero if the child has not terminated or stopped, or `child_pid` if it has. `waitpid` will return `-1` on error and set `errno`. This can happen if there are no child processes (`errno` set to `ECHILD`), if the call is interrupted by a signal (`EINTR`), or if the option argument is invalid (`EINVAL`).

Input and Output Redirection

You can use your knowledge of processes to alter the behavior of programs by exploiting the fact that open file descriptors are preserved across calls to `fork` and `exec`. The next example involves a *filter program* — a program that reads from its standard input and writes to its standard output, performing some useful transformation as it does so.

Try It Out Redirection

Here's a very simple filter program, `upper.c`, that reads input and converts it to uppercase:

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

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

When you run the program, it does what you expect:

```
$ ./upper
hello THERE
HELLO THERE
^D
$
```

You can, of course, use it to convert a file to uppercase by using the shell redirection

```
$ cat file.txt
this is the file, file.txt, it is all lower case.
$ ./upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

What if you want to use this filter from within another program? This program, `useupper.c`, accepts a filename as an argument and will respond with an error if called incorrectly.

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

int main(int argc, char *argv[])
{
    char *filename;

    if (argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
```

Chapter 11: Processes and Signals

```
        exit(1);
    }

    filename = argv[1];
```

You reopen the standard input, again checking for any errors as you do so, and then use `execl` to call `upper`.

```
    if(!freopen(filename, "r", stdin)) {
        fprintf(stderr, "could not redirect stdin from file %s\n", filename);
        exit(2);
    }

    execl("./upper", "upper", 0);
```

Don't forget that `execl` replaces the current process; if there is no error, the remaining lines are not executed.

```
        perror("could not exec ./upper");
        exit(3);
    }
```

How It Works

When you run this program, you can give it a file to convert to uppercase. The job is done by the program `upper`, which doesn't handle filename arguments. Note that you don't require the source code for `upper`; you can run any executable program in this way:

```
$ ./useupper file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

The `useupper` program uses `freopen` to close the standard input and associate the file stream `stdin` with the file given as a program argument. It then calls `execl` to replace the running process code with that of the `upper` program. Because open file descriptors are preserved across the call to `execl`, the `upper` program runs exactly as it would have under the shell command:

```
$ ./upper < file.txt
```

Threads

Linux processes can cooperate, can send each other messages, and can interrupt one another. They can even arrange to share segments of memory between themselves, but they are essentially separate entities within the operating system. They do not readily share variables.

There is a class of process known as a *thread* that is available in many UNIX and Linux systems. Though threads can be difficult to program, they can be of great value in some applications, such as multithreaded database servers. Programming threads on Linux (and UNIX generally) is not as common as using multiple processes, because Linux processes are quite lightweight, and programming multiple cooperation processes is much easier than programming threads. Threads are covered in Chapter 12.

Signals

A *signal* is an event generated by the UNIX and Linux systems in response to some condition, upon receipt of which a process may in turn take some action. We use the term *raise* to indicate the generation of a signal, and the term *catch* to indicate the receipt of a signal. Signals are raised by some error conditions, such as memory segment violations, floating-point processor errors, or illegal instructions. They are generated by the shell and terminal handlers to cause interrupts and can also be explicitly sent from one process to another as a way of passing information or modifying behavior. In all these cases, the programming interface is the same. Signals can be raised, caught and acted upon, or (for some at least) ignored.

Signal names are defined by including the header file `signal.h`. They all begin with `SIG` and include those listed in the following table.

Signal Name	Description
<code>SIGABORT</code>	*Process abort
<code>SIGALRM</code>	Alarm clock
<code>SIGFPE</code>	*Floating-point exception
<code>SIGHUP</code>	Hangup
<code>SIGILL</code>	*Illegal instruction
<code>SIGINT</code>	Terminal interrupt
<code>SIGKILL</code>	Kill (can't be caught or ignored)
<code>SIGPIPE</code>	Write on a pipe with no reader
<code>SIGQUIT</code>	Terminal quit
<code>SIGSEGV</code>	*Invalid memory segment access
<code>SIGTERM</code>	Termination
<code>SIGUSR1</code>	User-defined signal 1
<code>SIGUSR2</code>	User-defined signal 2

*Implementation-dependent actions may also be taken.

If a process receives one of these signals without first arranging to catch it, the process will be terminated immediately. Usually, a core dump file is created. This file, called `core` and placed in the current directory, is an image of the process that can be useful in debugging.

Additional signals include those in the following table.

Chapter 11: Processes and Signals

Signal Name	Description
SIGCHLD	Child process has stopped or exited.
SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop executing. (Can't be caught or ignored.)
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process trying to read.
SIGTTOU	Background process trying to write.

SIGCHLD can be useful for managing child processes. It's ignored by default. The remaining signals cause the process receiving them to stop, except for SIGCONT, which causes the process to resume. They are used by shell programs for job control and are rarely used by user programs.

We'll look at the first group of signals in a little more detail later. For now, it's enough to know that if the shell and terminal driver are configured normally, typing the interrupt character (usually Ctrl+C) at the keyboard will result in the SIGINT signal being sent to the foreground process, that is, the program currently running. This will cause the program to terminate unless it has arranged to catch the signal.

If you want to send a signal to a process other than the current foreground task, use the `kill` command. This takes an optional signal number or name, and the PID (usually found using the `ps` command) to send the signal to. For example, to send a "hangup" signal to a shell running on a different terminal with PID 512, you would use the command

```
$ kill -HUP 512
```

A useful variant of the `kill` command is `killall`, which allows you to send a signal to all processes running a specified command. Not all versions of UNIX support it, though Linux generally does. This is useful when you do not know the PID, or when you want to send a signal to several different processes executing the same command. A common use is to tell the `inetd` program to reread its configuration options. To do this you can use the command

```
$ killall -HUP inetd
```

Programs can handle signals using the signal library function.

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

This rather complex declaration says that `signal` is a function that takes two parameters, `sig` and `func`. The signal to be caught or ignored is given as argument `sig`. The function to be called when the specified signal is received is given as `func`. This function must be one that takes a single `int` argument (the

signal received) and is of type `void`. The signal function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of these two special values:

<code>SIG_IGN</code>	Ignore the signal.
<code>SIG_DFL</code>	Restore default behavior.

An example should make things clear. In the following Try It Out, you write a program, `ctrlc.c`, that reacts to typing Ctrl+C by printing an appropriate message rather than terminating. Pressing Ctrl+C a second time will end the program.

Try It Out Signal Handling

The function `ouch` reacts to the signal that is passed in the parameter `sig`. This function will be called when a signal occurs. It prints a message and then resets the signal handling for `SIGINT` (by default, generated by typing Ctrl+C) back to the default behavior.

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
```

The main function has to intercept the `SIGINT` signal generated when you type Ctrl+C. For the rest of the time, it just sits in an infinite loop, printing a message once a second.

```
int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Typing Ctrl+C (shown as `^C` in the following output) for the first time causes the program to react and then continue. When you type Ctrl+C again, the program ends because the behavior of `SIGINT` has returned to the default behavior of causing the program to exit.

```
$ ./ctrlc1
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
```

Chapter 11: Processes and Signals

```
Hello World!  
Hello World!  
Hello World!  
Hello World!  
^C  
$
```

As you can see from this example, the signal handling function takes a single integer parameter, the signal number that caused the function to be called. This can be useful if the same function is used to handle more than one signal. Here you print out the value of `SIGINT`, which on this system happens to have the value 2. You shouldn't rely on traditional numeric values for signals; always use signal names in new programs.

It is not safe to call all functions, such as `printf`, from within a signal handler. A useful technique is to use a signal handler to set a flag and then check that flag from the main program and print a message if required. Toward the end of the chapter, you will find a list of calls that can safely be made inside signal handlers.

How It Works

The program arranges for the function `ouch` to be called when you give the `SIGINT` signal by typing `Ctrl+C`. After the interrupt function `ouch` has completed, the program carries on, but the signal action is restored to the default. (Different versions of UNIX, particularly those derived from Berkeley UNIX, have historically had subtly different signal behaviors. If you want the default action to a signal restored after it has occurred, it's always best to code it that way specifically.) When it receives a second `SIGINT` signal, the program takes the default action, which is to terminate the program.

If you wanted to retain the signal handler and continue to react to `Ctrl+C`, you would need to re-establish it by calling `signal` again. This leads to a short time when the signal is not handled, from the start of the interrupt function to just before the signal handler is re-established. It's possible for a second signal to be received in this time and terminate the program against your wishes.

We don't recommend that you use the signal interface for catching signals. We include it here because you will find it in many older programs. You'll see `sigaction`, a more cleanly defined and reliable interface later, which you should use in all new programs.

The `signal` function returns the previous value of the signal handler for the specified signal if there is one, or `SIG_ERR` otherwise, in which case, `errno` will be set to a positive value. `errno` will be set to `EINVAL` if an invalid signal is specified or an attempt is made to handle a signal that may not be caught or ignored, such as `SIGKILL`.

Sending Signals

A process may send a signal to another process, including itself, by calling `kill`. The call will fail if the program doesn't have permission to send the signal, often because the target process is owned by another user. This is the program equivalent of the shell command of the same name.

```
#include <sys/types.h>  
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```


The `kill` function sends the specified signal, `sig`, to the process whose identifier is given by `pid`. It returns 0 on success. To send a signal, the sending process must have permission to do so. Normally, this means that both processes must have the same user ID (that is, you can send a signal only to one of your own processes, although the superuser can send signals to any process).

`kill` will fail, return -1, and set `errno` if the signal given is not a valid one (`errno` set to `EINVAL`), if it doesn't have permission (`EPERM`), or if the specified process doesn't exist (`ESRCH`).

Signals provide you with a useful alarm clock facility. The alarm function call can be used by a process to schedule a `SIGALRM` signal at some time in the future.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

The alarm call schedules the delivery of a `SIGALRM` signal in `seconds` seconds. In fact, the alarm will be delivered shortly after that, due to processing delays and scheduling uncertainties. A value of 0 will cancel any outstanding alarm request. Calling `alarm` before the signal is received will cause the alarm to be rescheduled. Each process can have only one outstanding alarm. `alarm` returns the number of seconds left before any outstanding alarm call would be sent, or -1 if the call fails.

To see how `alarm` works, you can simulate its effect by using `fork`, `sleep`, and `signal`. A program could start a new process for the sole purpose of sending a signal at some time later.

Try It Out An Alarm Clock

In `alarm.c`, the first function, `ding`, simulates an alarm clock.

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

static int alarm_fired = 0;

void ding(int sig)
{
    alarm_fired = 1;
}
```

In `main`, you tell the child process to wait for five seconds before sending a `SIGALRM` signal to its parent.

```
int main()
{
    pid_t pid;

    printf("alarm application starting\n");

    pid = fork();
    switch(pid) {
        case -1:
```

Chapter 11: Processes and Signals

```
/* Failure */
perror("fork failed");
exit(1);
case 0:
    /* child */
    sleep(5);
    kill(getppid(), SIGALRM);
    exit(0);
}
```

The parent process arranges to catch `SIGALRM` with a call to `signal` and then waits for the inevitable.

```
/* if we get here we are the parent process */
printf("waiting for alarm to go off\n");
(void) signal(SIGALRM, ding);

pause();
if (alarm_fired)
    printf("Ding!\n");

printf("done\n");
exit(0);
}
```

When you run this program, it pauses for five seconds while it waits for the simulated alarm clock.

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
Ding!
done
$
```

This program introduces a new function, `pause`, which simply causes the program to suspend execution until a signal occurs. When it receives a signal, any established handler is run and execution continues as normal. It's declared as

```
#include <unistd.h>

int pause(void);
```

and returns `-1` (if the next received signal doesn't cause the program to terminate) with `errno` set to `EINTR` when interrupted by a signal. It is more common to use `sigsuspend` when waiting for signals, which we discuss a bit later in the chapter.

How It Works

The alarm clock simulation program starts a new process via `fork`. This child process sleeps for five seconds and then sends a `SIGALRM` to its parent. The parent arranges to catch `SIGALRM` and then pauses until a signal is received. You do not call `printf` in the signal handler directly; rather, you set a flag and then check the flag afterward.

Using signals and suspending execution is an important part of Linux programming. It means that a program need not necessarily run all the time. Rather than run in a loop continually checking whether an event has occurred, it can wait for an event to happen. This is especially important in a multiuser environment where processes share a single processor and this kind of busy wait has a large impact on system performance. A particular problem with signals is that you never know quite “What will happen if a signal occurs in the middle of a system call?” (The answer is a rather unsatisfactory “it depends.”) In general, you need to worry only about “slow” system calls, such as reading from a terminal, where the system call will return with an error if a signal occurs while it is waiting. If you start using signals in your programs, you need to be aware that some system calls could fail if a signal causes an error condition that you may not have considered before signal handling was added.

You must program your signals carefully, because there are a number of “race conditions” that can occur in programs that use them. For example, if you intend to call `pause` to wait for a signal and that signal occurs before the call to `pause`, your program may wait indefinitely for an event that won’t occur. These race conditions, critical timing problems, catch many a novice programmer. Always check signal code very carefully.

A Robust Signals Interface

We’ve covered raising and catching signals using `signal` and `friends` in some depth, because they are very common in older UNIX programs. However, the X/Open and UNIX specifications recommend a newer programming interface for signals that is more robust: `sigaction`.

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

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

The `sigaction` structure, used to define the actions to be taken on receipt of the signal specified by `sig`, is defined in `signal.h` and has at least the following members:

```
void (*) (int) sa_handler    /* function, SIG_DFL or SIG_IGN
sigset_t sa_mask             /* signals to block in sa_handler
int sa_flags                 /* signal action modifiers
```

The `sigaction` function sets the action associated with the signal `sig`. If `oact` is not null, `sigaction` writes the previous signal action to the location it refers to. If `act` is null, this is all `sigaction` does. If `act` isn’t null, the action for the specified signal is set.

As with `signal`, `sigaction` returns 0 if successful and -1 if not. The error variable `errno` will be set to `EINVAL` if the specified signal is invalid or if an attempt is made to catch or ignore a signal that can’t be caught or ignored.

Within the `sigaction` structure pointed to by the argument `act`, `sa_handler` is a pointer to a function called when signal `sig` is received. This is much like the function `func` you saw earlier passed to `signal`. You can use the special values `SIG_IGN` and `SIG_DFL` in the `sa_handler` field to indicate that the signal is to be ignored or the action is to be restored to its default, respectively.

Chapter 11: Processes and Signals

The `sa_mask` field specifies a set of signals to be added to the process's signal mask before the `sa_handler` function is called. These are the set of signals that are blocked and won't be delivered to the process. This prevents the case you saw earlier where a signal is received before its handler has run to completion. Using the `sa_mask` field can eliminate this race condition.

However, signals caught with handlers set by `sigaction` are by default not reset, and the `sa_flags` field must be set to contain the value `SA_RESETHAND` if you want to obtain the behavior you saw earlier with `signal`. Before we look in any more detail at `sigaction`, let's rewrite the program `ctrlc.c`, using `sigaction` instead of `signal`.

Try It Out `sigaction`

Make the changes that follow so that `SIGINT` is intercepted by `sigaction`. Call the new program `ctrlc2.c`.

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

When you run this version of the program, you always get a message when you type Ctrl+C because `SIGINT` is handled repeatedly by `sigaction`. To terminate the program, you have to type Ctrl+\, which generates the `SIGQUIT` signal by default.

```
$ ./ctrlc2
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
```

```
Hello World!
Hello World!
^\  
Quit
$
```

How It Works

The program calls `sigaction` instead of `signal` to set the signal handler for Ctrl+C (`SIGINT`) to the function `ouch`. It first has to set up a `sigaction` structure that contains the handler, a signal mask, and flags. In this case, you don't need any flags, and an empty signal mask is created with the new function, `sigemptyset`.

After running this program, you may find a core dump (in a file called `core`) has been created. You can safely delete it.

Signal Sets

The header file `signal.h` defines the type `sigset_t` and functions used to manipulate sets of signals. These sets are used in `sigaction` and other functions to modify process behavior on receipt of signals.

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

These functions perform the operations suggested by their names. `sigemptyset` initializes a signal set to be empty. `sigfillset` initializes a signal set to contain all defined signals. `sigaddset` and `sigdelset` add and delete a specified signal (`signo`) from a signal set. They all return 0 if successful and -1 with `errno` set on error. The only error defined is `EINVAL` if the specified signal is invalid.

The function `sigismember` determines whether the given signal is a member of a signal set. It returns 1 if the signal is a member of the set, 0 if it isn't, and -1 with `errno` set to `EINVAL` if the signal is invalid.

```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

The process signal mask is set or examined by calling the function `sigprocmask`. This signal mask is the set of signals that are currently blocked and will therefore not be received by the current process.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

`sigprocmask` can change the process signal mask in a number of ways according to the `how` argument. New values for the signal mask are passed in the argument `set` if it isn't null, and the previous signal mask will be written to the signal set `oset`.

Chapter 11: Processes and Signals

The `how` argument can be one of the following:

<code>SIG_BLOCK</code>	The signals in <code>set</code> are added to the signal mask.
<code>SIG_SETMASK</code>	The signal mask is set from <code>set</code> .
<code>SIG_UNBLOCK</code>	The signals in <code>set</code> are removed from the signal mask.

If the `set` argument is a null pointer, the value of `how` is not used and the only purpose of the call is to fetch the value of the current signal mask into `oset`.

If it completes successfully, `sigprocmask` returns 0, or it returns -1 if the `how` parameter is invalid, in which case `errno` will be set to `EINVAL`.

If a signal is blocked by a process, it won't be delivered, but will remain pending. A program can determine which of its blocked signals are pending by calling the function `sigpending`.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

This writes a set of signals that are blocked from delivery and are pending into the signal set pointed to by `set`. It returns 0 if successful, otherwise, -1 with `errno` set to indicate the error. This function can be useful when a program needs to handle signals and to control when the handling function is called.

A process can suspend execution until the delivery of one of a set of signals by calling `sigsuspend`. This is a more general form of the pause function you met earlier.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

The `sigsuspend` function replaces the process signal mask with the signal set given by `sigmask` and then suspends execution. It will resume after the execution of a signal handling function. If the received signal terminates the program, `sigsuspend` will never return. If a received signal doesn't terminate the program, `sigsuspend` returns -1 with `errno` set to `EINTR`.

sigaction Flags

The `sa_flags` field of the `sigaction` structure used in `sigaction` may contain the values shown in the following table to modify signal behavior:

<code>SA_NOCLDSTOP</code>	Don't generate <code>SIGCHLD</code> when child processes stop.
<code>SA_RESETHAND</code>	Reset signal action to <code>SIG_DFL</code> on receipt.
<code>SA_RESTART</code>	Restart interruptible functions rather than error with <code>EINTR</code> .
<code>SA_NODEFER</code>	Don't add the signal to the signal mask when caught.

Chapter 11: Processes and Signals

The `SA_RESETHAND` flag can be used to automatically clear a signal function when a signal is caught, as we saw before.

Many system calls that a program uses are interruptible; that is, when they receive a signal they will return with an error and `errno` will be set to `EINTR` to indicate that the function returned due to a signal. This behavior requires extra care by an application using signals. If `SA_RESTART` is set in the `sa_flags` field in a call to `sigaction`, a function that might otherwise be interrupted by a signal will instead be restarted once the signal handling function has been executed.

Ordinarily, when a signal handling function is being executed, the signal received is added to the process signal mask for the duration of the handling function. This prevents a subsequent occurrence of the same signal, causing the signal handling function to run again. If the function is not re-entrant, having it called by another occurrence of a signal before it finishes handling the first may cause problems. If, however, the `SA_NODEFER` flag is set, the signal mask is not altered when it receives this signal.

A signal handling function could be interrupted in the middle and called again by something else. When you come back to the first call, it's vital that it still operates correctly. It's not just recursive (calling itself), but re-entrant (can be entered and executed again without problems). Interrupt service routines in the kernel that deal with more than one device at a time need to be re-entrant, because a higher-priority interrupt might "get in" during the execution of the same code.

Functions that are safe to call inside a signal handler, those guaranteed by the X/Open specification either to be re-entrant or not to raise signals themselves, are listed in the following table.

All functions not listed in the following table should be considered unsafe with respect to signals.

<code>access</code>	<code>alarm</code>	<code>cfgetispeed</code>	<code>cfgetospeed</code>
<code>cfsetispeed</code>	<code>cfsetospeed</code>	<code>chdir</code>	<code>chmod</code>
<code>chown</code>	<code>close</code>	<code>creat</code>	<code>dup2</code>
<code>dup</code>	<code>execle</code>	<code>execve</code>	<code>_exit</code>
<code>fcntl</code>	<code>fork</code>	<code>fstat</code>	<code>getegid</code>
<code>geteuid</code>	<code>getgid</code>	<code>getgroups</code>	<code>getpgrp</code>
<code>getpid</code>	<code>getppid</code>	<code>getuid</code>	<code>kill</code>
<code>link</code>	<code>lseek</code>	<code>mkdir</code>	<code>mkfifo</code>
<code>open</code>	<code>pathconf</code>	<code>pause</code>	<code>pipe</code>
<code>read</code>	<code>rename</code>	<code>rmdir</code>	<code>setgid</code>
<code>setpgid</code>	<code>setsid</code>	<code>setuid</code>	<code>sigaction</code>
<code>sigaddset</code>	<code>sigdelset</code>	<code>sigemptyset</code>	<code>sigfillset</code>

Continued on next page

Chapter 11: Processes and Signals

sigismember	signal	sigpending	sigprocmask
sigsuspend	sleep	stat	sysconf
tcdrain	tcflow	tcflush	tcgetattr
tcgetpgrp	tcsendbreak	tcsetattr	tcsetpgrp
time	times	umask	uname
unlink	utime	wait	waitpid
write			

Common Signal Reference

In this section, we list the signals that Linux and UNIX programs typically need with their default behaviors.

The default action for the signals in the following table is abnormal termination of the process with all the consequences of `_exit` (which is like `exit` but performs no cleanup before returning to the kernel). However, the status is made available to `wait`, and `waitpid` indicates abnormal termination by the specified signal.

Signal Name	Description
SIGALRM	Generated by the timer set by the <code>alarm</code> function.
SIGHUP	Sent to the controlling process by a disconnecting terminal, or by the controlling process on termination to each foreground process.
SIGINT	Typically raised from the terminal by typing Ctrl+C or the configured interrupt character.
SIGKILL	Typically used from the shell to forcibly terminate an errant process, because this signal can't be caught or ignored.
SIGPIPE	Generated by an attempt to write to a pipe with no associated reader.
SIGTERM	Sent as a request for a process to finish. Used by UNIX when shutting down to request that system services stop. This is the default signal sent from the <code>kill</code> command.
SIGUSR1, SIGUSR2	May be used by processes to communicate with each other, possibly to cause them to report status information.

By default, the signals in the next table also cause abnormal termination. Additionally, implementation-dependent actions, such as creation of a core file, may occur.

Signal Name	Description
SIGFPE	Generated by a floating-point arithmetic exception.
SIGILL	An illegal instruction has been executed by the processor. Usually caused by a corrupt program or invalid shared memory module.
SIGQUIT	Typically raised from the terminal by typing Ctrl+\ or the configured quit character.
SIGSEGV	A segmentation violation, usually caused by reading or writing at an illegal location in memory either by exceeding array bounds or dereferencing an invalid pointer. Overwriting a local array variable and corrupting the stack can cause a SIGSEGV to be raised when a function returns to an illegal address.

A process is suspended by default on receipt of one of the signals in the following table.

Signal Name	Description
SIGSTOP	Stop executing (can't be caught or ignored).
SIGTSTP	Terminal stop signal, often raised by typing Ctrl+Z.
SIGTTIN, SIGTTOU	Used by the shell to indicate that background jobs have stopped because they need to read from the terminal or produce output.

SIGCONT restarts a stopped process and is ignored if received by a process that is not stopped. The SIGCHLD signal is ignored by default.

Signal Name	Description
SIGCONT	Continue executing, if stopped.
SIGCHLD	Raised when a child process stops or exits.

Summary

In this chapter, you have seen how processes are a fundamental part of the Linux operating system. You have learned how they can be started, terminated, and viewed, and how you can use them to solve programming problems. You've also taken a look at signals, events that can be used to control the actions of running programs. You have seen that all Linux processes, down to and including `init`, use the same set of system calls available to any programmer.

12

POSIX Threads

In Chapter 11, you saw how processes are handled in Linux (and indeed in UNIX). These multi-processing features have long been a feature of UNIX-like operating systems. Sometimes it may be very useful to make a single program do two things at once, or at least to appear to do so, or you might want two or more things to happen at the same time in a closely coupled way but consider the overhead of creating a new process with `fork` too great. For these occasions you can use threads, which allow a single process to multitask.

In this chapter, you look at

- ❑ Creating new threads within a process
- ❑ Synchronizing data access between threads in a single process
- ❑ Modifying the attributes of a thread
- ❑ Controlling one thread from another in the same process

What Is a Thread?

Multiple strands of execution in a single program are called *threads*. A more precise definition is that a thread is a sequence of control within a process. All the programs you have seen so far have executed as a single process, although, like many other operating systems, Linux is quite capable of running multiple processes simultaneously. Indeed, all processes have at least one thread of execution. All the processes that you have seen so far in this book have had just one thread of execution.

It's important to be clear about the difference between the `fork` system call and the creation of new threads. When a process executes a `fork` call, a new copy of the process is created with its own variables and its own PID. This new process is scheduled independently, and (in general) executes almost independently of the process that created it. When we create a new thread in a process, in contrast, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.