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.

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.

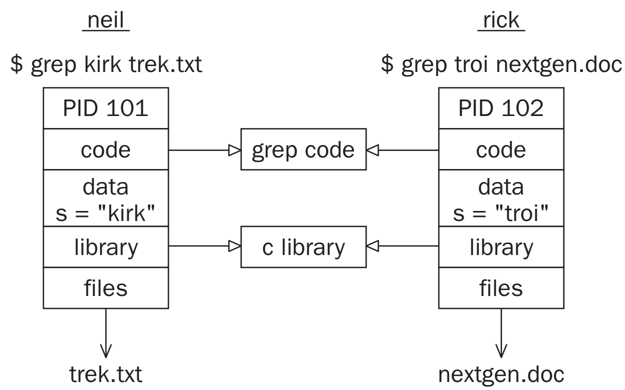

Figure 11-1

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
```

```
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 `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
```

```
 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 &
```

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
```

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 execle(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 `execle` 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`.

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);
}
```

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.

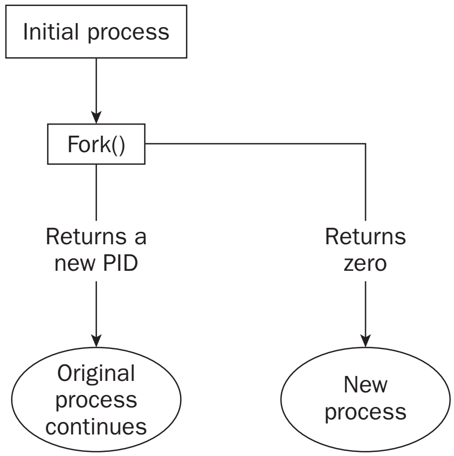

Figure 11-2

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;
}
```

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

```
#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;
}
```