3 Processes

A RUNNING INSTANCE OF A PROGRAM IS CALLED A *PROCESS*. If you have two terminal windows showing on your screen, then you are probably running the same terminal program twice—you have two terminal processes. Each terminal window is probably running a shell; each running shell is another process. When you invoke a command from a shell, the corresponding program is executed in a new process; the shell process resumes when that process completes.

Advanced programmers often use multiple cooperating processes in a single application to enable the application to do more than one thing at once, to increase application robustness, and to make use of already-existing programs.

Most of the process manipulation functions described in this chapter are similar to those on other UNIX systems. Most are declared in the header file `<unistd.h>`; check the man page for each function to be sure.

3.1 Looking at Processes

Even as you sit down at your computer, there are processes running. Every executing program uses one or more processes. Let's start by taking a look at the processes already on your computer.

3.1.1 Process IDs

Each process in a Linux system is identified by its unique *process ID*, sometimes referred to as *pid*. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

Every process also has a parent process (except the special `init` process, described in Section 3.4.3, "Zombie Processes"). Thus, you can think of the processes on a Linux system as arranged in a tree, with the `init` process at its root. The *parent process ID*, or *ppid*, is simply the process ID of the process's parent.

When referring to process IDs in a C or C++ program, always use the `pid_t` typedef, which is defined in `<sys/types.h>`. A program can obtain the process ID of the process it's running in with the `getpid()` system call, and it can obtain the process ID of its parent process with the `getppid()` system call. For instance, the program in Listing 3.1 prints its process ID and its parent's process ID.

Listing 3.1 **(*print-pid.c*) Printing the Process ID**

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

int main ()
{
  printf ("The process ID is %d\n", (int) getpid ());
  printf ("The parent process ID is %d\n", (int) getppid ());
  return 0;
}
```

Observe that if you invoke this program several times, a different process ID is reported because each invocation is in a new process. However, if you invoke it every time from the same shell, the parent process ID (that is, the process ID of the shell process) is the same.

3.1.2 Viewing Active Processes

The `ps` command displays the processes that are running on your system. The GNU/Linux version of `ps` has lots of options because it tries to be compatible with versions of `ps` on several other UNIX variants. These options control which processes are listed and what information about each is shown.

By default, invoking `ps` displays the processes controlled by the terminal or terminal window in which `ps` is invoked. For example:

```
% ps
  PID TTY          TIME CMD
21693 pts/8    00:00:00 bash
21694 pts/8    00:00:00 ps
```

This invocation of ps shows two processes. The first, bash, is the shell running on this terminal. The second is the running instance of the ps program itself. The first column, labeled PID, displays the process ID of each.

For a more detailed look at what's running on your GNU/Linux system, invoke this:

```
% ps -e -o pid,ppid,command
```

The -e option instructs ps to display all processes running on the system. The -o pid,ppid,command option tells ps what information to show about each process—in this case, the process ID, the parent process ID, and the command running in this process.

ps Output Formats

With the -o option to the ps command, you specify the information about processes that you want in the output as a comma-separated list. For example, ps -o pid,user,start_time,command displays the process ID, the name of the user owning the process, the wall clock time at which the process started, and the command running in the process. See the man page for ps for the full list of field codes. You can use the -f (full listing), -l (long listing), or -j (jobs listing) options instead to get three different preset listing formats.

Here are the first few lines and last few lines of output from this command on my system. You may see different output, depending on what's running on your system.

```
% ps -e -o pid,ppid,command
  PID  PPID COMMAND
    1     0 init [5]
    2     1 [kflushd]
    3     1 [kupdate]
...
21725 21693 xterm
21727 21725 bash
21728 21727 ps -e -o pid,ppid,command
```

Note that the parent process ID of the ps command, 21727, is the process ID of bash, the shell from which I invoked ps. The parent process ID of bash is in turn 21725, the process ID of the xterm program in which the shell is running.

3.1.3 Killing a Process

You can kill a running process with the kill command. Simply specify on the command line the process ID of the process to be killed.

The kill command works by sending the process a SIGTERM, or termination, signal.[1] This causes the process to terminate, unless the executing program explicitly handles or masks the SIGTERM signal. Signals are described in Section 3.3, "Signals."

1. You can also use the kill command to send other signals to a process. This is described in Section 3.4, "Process Termination."

3.2 Creating Processes

Two common techniques are used for creating a new process. The first is relatively simple but should be used sparingly because it is inefficient and has considerably security risks. The second technique is more complex but provides greater flexibility, speed, and security.

3.2.1 Using *system*

The `system` function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, `system` creates a subprocess running the standard Bourne shell (`/bin/sh`) and hands the command to that shell for execution. For example, this program in Listing 3.2 invokes the `ls` command to display the contents of the root directory, as if you typed `ls -l /` into a shell.

Listing 3.2 **(*system.c*) Using the *system* Call**

```
#include <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /");
  return return_value;
}
```

The `system` function returns the exit status of the shell command. If the shell itself cannot be run, `system` returns 127; if another error occurs, `system` returns –1.

Because the `system` function uses a shell to invoke your command, it's subject to the features, limitations, and security flaws of the system's shell. You can't rely on the availability of any particular version of the Bourne shell. On many UNIX systems, `/bin/sh` is a symbolic link to another shell. For instance, on most GNU/Linux systems, `/bin/sh` points to `bash` (the Bourne-Again SHell), and different GNU/Linux distributions use different versions of `bash`. Invoking a program with root privilege with the `system` function, for instance, can have different results on different GNU/Linux systems. Therefore, it's preferable to use the `fork` and `exec` method for creating processes.

3.2.2 Using *fork* and *exec*

The DOS and Windows API contains the `spawn` family of functions. These functions take as an argument the name of a program to run and create a new process instance of that program. Linux doesn't contain a single function that does all this in one step. Instead, Linux provides one function, `fork`, that makes a child process that is an exact

copy of its parent process. Linux provides another set of functions, the `exec` family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program. To spawn a new process, you first use `fork` to make a copy of the current process. Then you use `exec` to transform one of these processes into an instance of the program you want to spawn.

Calling *fork*

When a program calls `fork`, a duplicate process, called the *child process*, is created. The parent process continues executing the program from the point that `fork` was called. The child process, too, executes the same program from the same place.

So how do the two processes differ? First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID. One way for a program to distinguish whether it's in the parent process or the child process is to call `getpid`. However, the `fork` function provides different return values to the parent and child processes—one process "goes in" to the `fork` call, and two processes "come out," with different return values. The return value in the parent process is the process ID of the child. The return value in the child process is zero. Because no process ever has a process ID of zero, this makes it easy for the program whether it is now running as the parent or the child process.

Listing 3.3 is an example of using `fork` to duplicate a program's process. Note that the first block of the `if` statement is executed only in the parent process, while the `else` clause is executed in the child process.

Listing 3.3 **(*fork.c*) Using *fork* to Duplicate a Program's Process**

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

int main ()
{
  pid_t child_pid;

  printf ("the main program process ID is %d\n", (int) getpid ());

  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process ID is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int) getpid ());

  return 0;
}
```

Using the *exec* Family

The `exec` functions replace the program running in a process with another program. When a program calls an `exec` function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the `exec` call doesn't encounter an error.

Within the `exec` family, there are functions that vary slightly in their capabilities and how they are called.

- Functions that contain the letter *p* in their names (`execvp` and `execlp`) accept a program name and search for a program by that name in the current execution path; functions that don't contain the *p* must be given the full path of the program to be executed.
- Functions that contain the letter `v` in their names (`execv`, `execvp`, and `execve`) accept the argument list for the new program as a NULL-terminated array of pointers to strings. Functions that contain the letter *l* (`execl`, `execlp`, and `execle`) accept the argument list using the C language's varargs mechanism.
- Functions that contain the letter *e* in their names (`execve` and `execle`) accept an additional argument, an array of environment variables. The argument should be a NULL-terminated array of pointers to character strings. Each character string should be of the form "`VARIABLE=value`".

Because `exec` replaces the calling program with another one, it never returns unless an error occurs.

The argument list passed to the program is analogous to the command-line arguments that you specify to a program when you run it from the shell. They are available through the `argc` and `argv` parameters to `main`. Remember, when a program is invoked from the shell, the shell sets the first element of the argument list `argv[0]`) to the name of the program, the second element of the argument list (`argv[1]`) to the first command-line argument, and so on. When you use an `exec` function in your programs, you, too, should pass the name of the function as the first element of the argument list.

Using *fork* and *exec* Together

A common pattern to run a subprogram within a program is first to fork the process and then exec the subprogram. This allows the calling program to continue execution in the parent process while the calling program is replaced by the subprogram in the child process.

The program in Listing 3.4, like Listing 3.2, lists the contents of the root directory using the `ls` command. Unlike the previous example, though, it invokes the `ls` command directly, passing it the command-line arguments `-l` and `/` rather than invoking it through a shell.

Listing 3.4 (*fork-exec.c*) **Using *fork* and *exec* Together**

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

/* Spawn a child process running a new program. PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list. Returns the process ID of
   the spawned process.  */

int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}

int main ()
{
  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process ID.  */
  spawn ("ls", arg_list);

  printf ("done with main program\n");

  return 0;
}
```

3.2.3 Process Scheduling

Linux schedules the parent and child processes independently; there's no guarantee of which one will run first, or how long it will run before Linux interrupts it and lets the other process (or some other process on the system) run. In particular, none, part, or all of the `ls` command may run in the child process before the parent completes.[2] Linux promises that each process will run eventually—no process will be completely starved of execution resources.

You may specify that a process is less important—and should be given a lower priority—by assigning it a higher *niceness* value. By default, every process has a niceness of zero. A higher niceness value means that the process is given a lesser execution priority; conversely, a process with a lower (that is, negative) niceness gets more execution time.

To run a program with a nonzero niceness, use the `nice` command, specifying the niceness value with the `-n` option. For example, this is how you might invoke the command "`sort input.txt > output.txt`", a long sorting operation, with a reduced priority so that it doesn't slow down the system too much:

```
% nice -n 10 sort input.txt > output.txt
```

You can use the `renice` command to change the niceness of a running process from the command line.

To change the niceness of a running process programmatically, use the `nice` function. Its argument is an increment value, which is added to the niceness value of the process that calls it. Remember that a positive value raises the niceness value and thus reduces the process's execution priority.

Note that only a process with root privilege can run a process with a negative niceness value or reduce the niceness value of a running process. This means that you may specify negative values to the `nice` and `renice` commands only when logged in as root, and only a process running as root can pass a negative value to the `nice` function. This prevents ordinary users from grabbing execution priority away from others using the system.

3.3 Signals

Signals are mechanisms for communicating with and manipulating processes in Linux. The topic of signals is a large one; here we discuss some of the most important signals and techniques that are used for controlling processes.

A signal is a special message sent to a process. Signals are asynchronous; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code. There are several dozen different signals, each with a different meaning. Each signal type is specified by its signal number, but in programs, you usually refer to a signal by its name. In Linux, these are defined in `/usr/include/bits/signum.h`. (You shouldn't include this header file directly in your programs; instead, use `<signal.h>`.)

2. A method for serializing the two processes is presented in Section 3.4.1, "Waiting for Process Termination."

When a process receives a signal, it may do one of several things, depending on the signal's *disposition*. For each signal, there is a *default disposition*, which determines what happens to the process if the program does not specify some other behavior. For most signal types, a program may specify some other behavior—either to ignore the signal or to call a special *signal-handler* function to respond to the signal. If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes.

The Linux system sends signals to processes in response to specific conditions. For instance, `SIGBUS` (bus error), `SIGSEGV` (segmentation violation), and `SIGFPE` (floating point exception) may be sent to a process that attempts to perform an illegal operation. The default disposition for these signals it to terminate the process and produce a core file.

A process may also send a signal to another process. One common use of this mechanism is to end another process by sending it a `SIGTERM` or `SIGKILL` signal.[3] Another common use is to send a command to a running program. Two "user-defined" signals are reserved for this purpose: `SIGUSR1` and `SIGUSR2`. The `SIGHUP` signal is sometimes used for this purpose as well, commonly to wake up an idling program or cause a program to reread its configuration files.

The `sigaction` function can be used to set a signal disposition. The first parameter is the signal number. The next two parameters are pointers to `sigaction` structures; the first of these contains the desired disposition for that signal number, while the second receives the previous disposition. The most important field in the first or second `sigaction` structure is `sa_handler`. It can take one of three values:

- `SIG_DFL`, which specifies the default disposition for the signal.
- `SIG_IGN`, which specifies that the signal should be ignored.
- A pointer to a signal-handler function. The function should take one parameter, the signal number, and return `void`.

Because signals are asynchronous, the main program may be in a very fragile state when a signal is processed and thus while a signal handler function executes. Therefore, you should avoid performing any I/O operations or calling most library and system functions from signal handlers.

A signal handler should perform the minimum work necessary to respond to the signal, and then return control to the main program (or terminate the program). In most cases, this consists simply of recording the fact that a signal occurred. The main program then checks periodically whether a signal has occurred and reacts accordingly.

It is possible for a signal handler to be interrupted by the delivery of another signal. While this may sound like a rare occurrence, if it does occur, it will be very difficult to diagnose and debug the problem. (This is an example of a race condition, discussed in Chapter 4, "Threads," Section 4.4, "Synchronization and Critical Sections.") Therefore, you should be very careful about what your program does in a signal handler.

3. What's the difference? The `SIGTERM` signal asks a process to terminate; the process may ignore the request by masking or ignoring the signal. The `SIGKILL` signal always kills the process immediately because the process may not mask or ignore `SIGKILL`.

Even assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state. If you use a global variable to flag a signal from a signal-handler function, it should be of the special type `sig_atomic_t`. Linux guarantees that assignments to variables of this type are performed in a single instruction and therefore cannot be interrupted midway. In Linux, `sig_atomic_t` is an ordinary `int`; in fact, assignments to integer types the size of `int` or smaller, or to pointers, are atomic. If you want to write a program that's portable to any standard UNIX system, though, use `sig_atomic_t` for these global variables.

This program skeleton in Listing 3.5, for instance, uses a signal-handler function to count the number of times that the program receives `SIGUSR1`, one of the signals reserved for application use.

Listing 3.5 **(*sigusr1.c*) Using a Signal Handler**

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

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy stuff here.  */
  /* ...  */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

3.4 Process Termination

Normally, a process terminates in one of two ways. Either the executing program calls the `exit` function, or the program's `main` function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the `exit` function, or the value returned from `main`.

A process may also terminate abnormally, in response to a signal. For instance, the `SIGBUS`, `SIGSEGV`, and `SIGFPE` signals mentioned previously cause the process to terminate. Other signals are used to terminate a process explicitly. The `SIGINT` signal is sent to a process when the user attempts to end it by typing Ctrl+C in its terminal. The `SIGTERM` signal is sent by the `kill` command. The default disposition for both of these is to terminate the process. By calling the `abort` function, a process sends itself the `SIGABRT` signal, which terminates the process and produces a core file. The most powerful termination signal is `SIGKILL`, which ends a process immediately and cannot be blocked or handled by a program.

Any of these signals can be sent using the `kill` command by specifying an extra command-line flag; for instance, to end a troublesome process by sending it a `SIGKILL`, invoke the following, where `pid` is its process ID:

```
% kill -KILL pid
```

To send a signal from a program, use the `kill` function. The first parameter is the target process ID. The second parameter is the signal number; use `SIGTERM` to simulate the default behavior of the `kill` command. For instance, where `child pid` contains the process ID of the child process, you can use the `kill` function to terminate a child process from the parent by calling it like this:

```
kill (child_pid, SIGTERM);
```

Include the `<sys/types.h>` and `<signal.h>` headers if you use the `kill` function.

By convention, the exit code is used to indicate whether the program executed correctly. An exit code of zero indicates correct execution, while a nonzero exit code indicates that an error occurred. In the latter case, the particular value returned may give some indication of the nature of the error. It's a good idea to stick with this convention in your programs because other components of the GNU/Linux system assume this behavior. For instance, shells assume this convention when you connect multiple programs with the `&&` (logical and) and `||` (logical or) operators. Therefore, you should explicitly return zero from your `main` function, unless an error occurs.

With most shells, it's possible to obtain the exit code of the most recently executed program using the special `$?` variable. Here's an example in which the `ls` command is invoked twice and its exit code is displayed after each invocation. In the first case, `ls` executes correctly and returns the exit code zero. In the second case, `ls` encounters an error (because the filename specified on the command line does not exist) and thus returns a nonzero exit code.

```
% ls /
bin   coda  etc   lib         misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

Note that even though the parameter type of the `exit` function is `int` and the `main` function returns an `int`, Linux does not preserve the full 32 bits of the return code. In fact, you should use exit codes only between zero and 127. Exit codes above 128 have a special meaning—when a process is terminated by a signal, its exit code is 128 plus the signal number.

3.4.1 Waiting for Process Termination

If you typed in and ran the `fork` and `exec` example in Listing 3.4, you may have noticed that the output from the `ls` program often appears after the "main program" has already completed. That's because the child process, in which `ls` is run, is scheduled independently of the parent process. Because Linux is a multitasking operating system, both processes appear to execute simultaneously, and you can't predict whether the `ls` program will have a chance to run before or after the parent process runs.

In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed. This can be done with the `wait` family of system calls. These functions allow you to wait for a process to finish executing, and enable the parent process to retrieve information about its child's termination. There are four different system calls in the `wait` family; you can choose to get a little or a lot of information about the process that exited, and you can choose whether you care about which child process terminated.

3.4.2 The *wait* System Calls

The simplest such function is called simply `wait`. It blocks the calling process until one of its child processes exits (or an error occurs). It returns a status code via an integer pointer argument, from which you can extract information about how the child process exited. For instance, the `WEXITSTATUS` macro extracts the child process's exit code.

You can use the `WIFEXITED` macro to determine from a child process's exit status whether that process exited normally (via the `exit` function or returning from `main`) or died from an unhandled signal. In the latter case, use the `WTERMSIG` macro to extract from its exit status the signal number by which it died.

Here is the `main` function from the `fork` and `exec` example again. This time, the parent process calls `wait` to wait until the child process, in which the `ls` command executes, is finished.

```
int main ()
{
  int child_status;

  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process ID.  */
  spawn ("ls", arg_list);

  /* Wait for the child process to complete.  */
  wait (&child_status);
  if (WIFEXITED (child_status))
    printf ("the child process exited normally, with exit code %d\n",
            WEXITSTATUS (child_status));
  else
    printf ("the child process exited abnormally\n");

  return 0;
}
```

Several similar system calls are available in Linux, which are more flexible or provide more information about the exiting child process. The `waitpid` function can be used to wait for a specific child process to exit instead of any child process. The `wait3` function returns CPU usage statistics about the exiting child process, and the `wait4` function allows you to specify additional options about which processes to wait for.

3.4.3 Zombie Processes

If a child process terminates while its parent is calling a `wait` function, the child process vanishes and its termination status is passed to its parent via the `wait` call. But what happens when a child process terminates and the parent is not calling `wait`? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, is becomes a zombie process.

A *zombie process* is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The `wait` functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls `wait`. If the child process has not terminated at that point, the parent process will block in the `wait` call until the child process finishes. If the child process finishes before the parent process calls `wait`, the child process becomes a zombie. When the parent process calls `wait`, the zombie child's termination status is extracted, the child process is deleted, and the `wait` call returns immediately.

What happens if the parent does not clean up its children? They stay around in the system, as zombie processes. The program in Listing 3.6 forks a child process, which terminates immediately and then goes to sleep for a minute, without ever cleaning up the child process.

Listing 3.6 **(*zombie.c*) Making a Zombie Process**

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

int main ()
{
  pid_t child_pid;

  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

Try compiling this file to an executable named `make-zombie`. Run it, and while it's still running, list the processes on the system by invoking the following command in another window:

```
% ps -e -o pid,ppid,stat,cmd
```

This lists the process ID, parent process ID, process status, and process command line. Observe that, in addition to the parent `make-zombie` process, there is another `make-zombie` process listed. It's the child process; note that its parent process ID is the process ID of the main `make-zombie` process. The child process is marked as `<defunct>`, and its status code is Z, for zombie.

What happens when the main `make-zombie` program ends when the parent process exits, without ever calling `wait`? Does the zombie process stay around? No—try running `ps` again, and note that both of the `make-zombie` processes are gone. When a program exits, its children are inherited by a special process, the `init` program, which always runs with process ID of 1 (it's the first process started when Linux boots). The `init` process automatically cleans up any zombie child processes that it inherits.

3.4.4 Cleaning Up Children Asynchronously

If you're using a child process simply to `exec` another program, it's fine to call `wait` immediately in the parent process, which will block until the child process completes. But often, you'll want the parent process to continue running, as one or more children execute synchronously. How can you be sure that you clean up child processes that have completed so that you don't leave zombie processes, which consume system resources, lying around?

One approach would be for the parent process to call `wait3` or `wait4` periodically, to clean up zombie children. Calling `wait` for this purpose doesn't work well because, if no children have terminated, the call will block until one does. However, `wait3` and `wait4` take an additional flag parameter, to which you can pass the flag value `WNOHANG`. With this flag, the function runs in *nonblocking mode*—it will clean up a terminated child process if there is one, or simply return if there isn't. The return value of the call is the process ID of the terminated child in the former case, or zero in the latter case.

A more elegant solution is to notify the parent process when a child terminates. There are several ways to do this using the methods discussed in Chapter 5, "Interprocess Communication," but fortunately Linux does this for you, using signals. When a child process terminates, Linux sends the parent process the `SIGCHLD` signal. The default disposition of this signal is to do nothing, which is why you might not have noticed it before.

Thus, an easy way to clean up child processes is by handling `SIGCHLD`. Of course, when cleaning up the child process, it's important to store its termination status if this information is needed, because once the process is cleaned up using `wait`, that information is no longer available. Listing 3.7 is what it looks like for a program to use a `SIGCHLD` handler to clean up its child processes.

Listing 3.7 **(*sigchld.c*) Cleaning Up Children by Handling *SIGCHLD***

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

sig_atomic_t child_exit_status;

void clean_up_child_process (int signal_number)
{
  /* Clean up the child process.  */
  int status;
  wait (&status);
  /* Store its exit status in a global variable.  */
  child_exit_status = status;
}

int main ()
{
  /* Handle SIGCHLD by calling clean_up_child_process.  */
  struct sigaction sigchld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);

  /* Now do things, including forking a child process.  */
  /* ...  */

  return 0;
}
```

Note how the signal handler stores the child process's exit status in a global variable, from which the main program can access it. Because the variable is assigned in a signal handler, its type is `sig_atomic_t`.