CHAPTER 10
Signals

Signals are software interrupts that provide a mechanism for handling asynchronous events. These events can originate from outside the system, such as when the user generates the interrupt character by pressing Ctrl-C, or from activities within the program or kernel, such as when the process executes code that divides by zero. As a primitive form of interprocess communication (IPC), one process can also send a signal to another process.

The key point is not just that the events occur asynchronously—the user, for example, can press Ctrl-C at any point in the program's execution—but also that the program handles the signals asynchronously. The signal-handling functions are registered with the kernel, which invokes the functions asynchronously from the rest of the program when the signals are delivered.

Signals have been part of Unix since the early days. Over time, however, they have evolved, most noticeably in terms of reliability, as signals once could get lost, and in terms of functionality, as signals may now carry user-defined payloads. At first, different Unix systems made incompatible changes to signals. Thankfully, POSIX came to the rescue and standardized signal handling. This standard is what Linux provides and is what we'll discuss here.

In this chapter, we'll start with an overview of signals and a discussion of their use and misuse. We'll then cover the various Linux interfaces that manage and manipulate signals.

Most nontrivial applications interact with signals. Even if you deliberately design your application to not rely on signals for its communication needs—often a good idea!—you'll still be forced to work with signals in certain cases, such as when handling program termination.

Signal Concepts

Signals have a very precise lifecycle. First, a signal is *raised* (we sometimes also say it is *sent* or *generated*). The kernel then *stores* the signal until it is able to deliver it. Finally, once it is free to do so, the kernel *handles* the signal as appropriate. The kernel can perform one of three actions, depending on what the process asked it to do:

Ignore the signal
: No action is taken. There are two signals that cannot be ignored: `SIGKILL` and `SIGSTOP`. The reason for this is that the system administrator needs to be able to kill or stop processes, and it would be a circumvention of that right if a process could elect to ignore a `SIGKILL` (making it unkillable) or a `SIGSTOP` (making it unstoppable).

Catch and handle the signal
: The kernel will suspend execution of the process's current code path and jump to a previously registered function. The process will then execute this function. Once the process returns from this function, it will jump back to wherever it was when it caught the signal. `SIGINT` and `SIGTERM` are two commonly caught signals. Processes catch `SIGINT` to handle the user generating the interrupt character—for example, a terminal might catch this signal and return to the main prompt. Processes catch `SIGTERM` to perform necessary cleanup, such as disconnecting from the network or removing temporary files, before terminating. `SIGKILL` and `SIGSTOP` cannot be caught.

Perform the default action
: This action depends on the signal being sent. The default action is often to terminate the process. This is the case with `SIGKILL`, for instance. However, many signals are provided for specific purposes that concern programmers in particular situations, and these signals are ignored by default because many programs are not interested in them. We will look at the various signals and their default actions shortly.

Traditionally, when a signal was delivered, the function that handled the signal had no information about what had happened except for the fact that the particular signal had occurred. Nowadays, the kernel can provide a lot of context to programmers who wish to receive it. Signals, as we shall see, can even pass user-defined data.

Signal Identifiers

Every signal has a symbolic name that starts with the prefix *SIG*. For example, `SIGINT` is the signal sent when the user presses Ctrl-C, `SIGABRT` is the signal sent when the process calls the `abort()` function, and `SIGKILL` is the signal sent when a process is forcefully terminated.

These signals are all defined in a header file included from `<signal.h>`. The signals are simply preprocessor definitions that represent positive integers—that is, every signal is also associated with an integer identifier. The name-to-integer mapping for the signals is implementation-dependent and varies among Unix systems, although the first dozen or so signals are usually mapped the same way (`SIGKILL` is infamously *signal 9* for example). A portable program will always use a signal's human-readable name, and never its integer value.

The signal numbers start at 1 (generally `SIGHUP`) and proceed linearly upward. There are about 31 signals in total, but most programs deal regularly with only a handful of them. There is no signal with the value 0, which is a special value known as the *null signal*. There's really nothing important about the null signal—it doesn't deserve a special name—but some system calls (such as `kill()`) use a value of `0` as a special case.

You can generate a list of signals supported on your system with the command *kill -l*.

Signals Supported by Linux

Table 10-1 lists the signals that Linux supports.

Table 10-1. Signals

Signal	Description	Default action
`SIGABRT`	Sent by `abort()`	Terminate with core dump
`SIGALRM`	Sent by `alarm()`	Terminate
`SIGBUS`	Hardware or alignment error	Terminate with core dump
`SIGCHLD`	Child has terminated	Ignored
`SIGCONT`	Process has continued after being stopped	Ignored
`SIGFPE`	Arithmetic exception	Terminate with core dump
`SIGHUP`	Process's controlling terminal was closed (most frequently, the user logged out)	Terminate
`SIGILL`	Process tried to execute an illegal instruction	Terminate with core dump
`SIGINT`	User generated the interrupt character (Ctrl-C)	Terminate
`SIGIO`	Asynchronous I/O event	Terminate[a]
`SIGKILL`	Uncatchable process termination	Terminate
`SIGPIPE`	Process wrote to a pipe but there are no readers	Terminate
`SIGPROF`	Profiling timer expired	Terminate
`SIGPWR`	Power failure	Terminate
`SIGQUIT`	User generated the quit character (Ctrl-\)	Terminate with core dump

Signal	Description	Default action
SIGSEGV	Memory access violation	Terminate with core dump
SIGSTKFLT	Coprocessor stack fault	Terminate[b]
SIGSTOP	Suspends execution of the process	Stop
SIGSYS	Process tried to execute an invalid system call	Terminate with core dump
SIGTERM	Catchable process termination	Terminate
SIGTRAP	Break point encountered	Terminate with core dump
SIGTSTP	User generated the suspend character (Ctrl-Z)	Stop
SIGTTIN	Background process read from controlling terminal	Stop
SIGTTOU	Background process wrote to controlling terminal	Stop
SIGURG	Urgent I/O pending	Ignored
SIGUSR1	Process-defined signal	Terminate
SIGUSR2	Process-defined signal	Terminate
SIGVTALRM	Generated by `setitimer()` when called with the `ITIMER_VIRTUAL` flag	Terminate
SIGWINCH	Size of controlling terminal window changed	Ignored
SIGXCPU	Processor resource limits were exceeded	Terminate with core dump
SIGXFSZ	File resource limits were exceeded	Terminate with core dump

[a] The behavior on other Unix systems, such as BSD, is to ignore this signal.

[b] The Linux kernel no longer generates this signal; it remains only for backward compatibility.

Several other signal values exist, but Linux defines them to be equivalent to other values: `SIGINFO` is defined as `SIGPWR`,[1] `SIGIOT` is defined as `SIGABRT`, and `SIGPOLL` and `SIGLOST` are defined as `SIGIO`.

Now that we have a table for quick reference, let's go over each of the signals in detail:

`SIGABRT`
: The `abort()` function sends this signal to the process that invokes it. The process then terminates and generates a core file. In Linux, assertions such as `assert()` call `abort()` when the conditional fails.

`SIGALRM`
: The `alarm()` and `setitimer()` (with the `ITIMER_REAL` flag) functions send this signal to the process that invoked them when an alarm expires. Chapter 11 discusses these and related functions.

1. Only the Alpha architecture defines this signal. On all other machine architectures, this signal does not exist.

`SIGBUS`
: The kernel raises this signal when the process incurs a hardware fault other than memory protection, which generates a `SIGSEGV`. On traditional Unix systems, this signal represented various irrecoverable errors, such as unaligned memory access. The Linux kernel, however, fixes most of these errors automatically, without generating the signal. The kernel does raise this signal when a process improperly accesses a region of memory created via `mmap()` (see Chapter 9 for a discussion of memory mappings). Unless this signal is caught, the kernel will terminate the process and generate a core dump.

`SIGCHLD`
: Whenever a process terminates or stops, the kernel sends this signal to the process's parent. Because `SIGCHLD` is ignored by default, processes must explicitly catch and handle it if they are interested in the lives of their children. A handler for this signal generally calls `wait()`, discussed in Chapter 5, to determine the child's pid and exit code.

`SIGCONT`
: The kernel sends this signal to a process when the process is resumed after being stopped. By default, this signal is ignored, but processes can catch it if they want to perform an action after being continued. This signal is commonly used by terminals or editors that wish to refresh the screen.

`SIGFPE`
: Despite its name, this signal represents any arithmetic exception, and not solely those related to floating-point operations. Exceptions include overflows, underflows, and division by zero. The default action is to terminate the process and generate a core file, but processes may catch and handle this signal if they want. Note that the behavior of a process and the result of the offending operation are undefined if the process elects to continue running.

`SIGHUP`
: The kernel sends this signal to the session leader whenever the session's terminal disconnects. The kernel also sends this signal to each process in the foreground process group when the session leader terminates. The default action is to terminate, which makes sense—the signal suggests that the user has logged out. Daemon processes "overload" this signal with a mechanism to instruct them to reload their configuration files. Sending `SIGHUP` to Apache, for example, instructs it to reread *httpd.conf*. Using `SIGHUP` for this purpose is a common convention but not mandatory. The practice is safe because daemons do not have controlling terminals and thus should never normally receive this signal.

`SIGILL`
: The kernel sends this signal when a process attempts to execute an illegal machine instruction. The default action is to terminate the process and generate a core dump. Processes may elect to catch and handle `SIGILL`, but their behavior is undefined after its occurrence.

`SIGINT`
: This signal is sent to all processes in the foreground process group when the user enters the interrupt character (usually Ctrl-C). The default behavior is to terminate; however, processes can elect to catch and handle this signal and generally do so to clean up before terminating.

`SIGIO`
: This signal is sent when a BSD-style asynchronous I/O event is generated. This style of I/O is rarely used on Linux. (See Chapter 4 for a discussion of advanced I/O techniques that are common to Linux.)

`SIGKILL`
: This signal is sent from the `kill()` system call; it exists to provide system administrators with a surefire way of unconditionally killing a process. This signal cannot be caught or ignored, and its result is always to terminate the process.

`SIGPIPE`
: If a process writes to a pipe but the reader has terminated, the kernel raises this signal. The default action is to terminate the process, but this signal may be caught and handled.

`SIGPROF`
: The `setitimer()` function, when used with the `ITIMER_PROF` flag, generates this signal when a profiling timer expires. The default action is to terminate the process.

`SIGPWR`
: This signal is system-dependent. On Linux, it represents a low-battery condition (such as in an uninterruptible power supply, or UPS). A UPS monitoring daemon sends this signal to *init*, which then responds by cleaning up and shutting down the system—hopefully before the power goes out!

`SIGQUIT`
: The kernel raises this signal for all processes in the foreground process group when the user provides the terminal quit character (usually Ctrl-\). The default action is to terminate the processes and generate a core dump.

`SIGSEGV`
: This signal, whose name derives from *segmentation violation*, is sent to a process when it attempts an invalid memory access. This includes accessing unmapped memory, reading from memory that is not read-enabled, executing code in memory

that is not execute-enabled, or writing to memory that is not write-enabled. Processes may catch and handle this signal, but the default action is to terminate the process and generate a core dump.

`SIGSTOP`
: This signal is sent only by `kill()`. It unconditionally stops a process and cannot be caught or ignored.

`SIGSYS`
: The kernel sends this signal to a process when it attempts to invoke an invalid system call. This can happen if a binary is built on a newer version of the operating system (with newer versions of system calls) but then runs on an older version. Properly built binaries that make their system calls through *glibc* should never receive this signal. Instead, invalid system calls should return `-1` and set `errno` to `ENOSYS`.

`SIGTERM`
: This signal is sent only by `kill()`; it allows a user to gracefully terminate a process (the default action). Processes may elect to catch this signal and clean up before terminating, but it is considered rude to catch this signal and not terminate promptly.

`SIGTRAP`
: The kernel sends this signal to a process when it crosses a break point. Generally, debuggers catch this signal, and other processes ignore it.

`SIGTSTP`
: The kernel sends this signal to all processes in the foreground process group when the user provides the suspend character (usually Ctrl-Z).

`SIGTTIN`
: This signal is sent to a process that is in the background when it attempts to read from its controlling terminal. The default action is to stop the process.

`SIGTTOU`
: This signal is sent to a process that is in the background when it attempts to write to its controlling terminal. The default action is to stop the process.

`SIGURG`
: The kernel sends this signal to a process when out-of-band (OOB) data has arrived on a socket. Out-of-band data is beyond the scope of this book.

`SIGUSR1` *and* `SIGUSR2`
: These signals are available for user-defined purposes; the kernel never raises them. Processes may use `SIGUSR1` and `SIGUSR2` for whatever purpose they like. A common use is to instruct a daemon process to behave differently. The default action is to terminate the process.

`SIGVTALRM`
: The `setitimer()` function sends this signal when a timer created with the `ITIMER_VIRTUAL` flag expires. Chapter 11 discusses timers.

`SIGWINCH`
: The kernel raises this signal for all processes in the foreground process group when the size of their terminal window changes. By default, processes ignore this signal, but they may elect to catch and handle it if they are aware of their terminal's window size. A good example of a program that catches this signal is *top*—try resizing its window while it is running and watch how it responds.

`SIGXCPU`
: The kernel raises this signal when a process exceeds its soft processor limit. The kernel will continue to raise this signal once per second until the process exits or exceeds its hard processor limit. Once the hard limit is exceeded, the kernel sends the process a `SIGKILL`.

`SIGXFSZ`
: The kernel raises this signal when a process exceeds its file size limit. The default action is to terminate the process, but if this signal is caught or ignored, the system call that would have resulted in the file size limit being exceeded returns `-1` and sets `errno` to `EFBIG`.

Basic Signal Management

With the signals out of the way, we'll now turn to how you manage them from within your program. The simplest and oldest interface for signal management is the `signal()` function. Defined by the ISO C89 standard, which standardizes only the lowest common denominator of signal support, this system call is very basic. Linux offers substantially more control over signals via other interfaces, which we'll cover later in this chapter. Because `signal()` is the most basic and, thanks to its presence in ISO C, quite common, we'll cover it first:

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t handler);
```

A successful call to `signal()` removes the current action taken on receipt of the signal `signo` and instead handles the signal with the signal handler specified by `handler`. `signo` is one of the signal names discussed in the previous section, such as `SIGINT` or `SIGUSR1`. Recall that a process can catch neither `SIGKILL` nor `SIGSTOP`, so setting up a handler for either of these two signals makes no sense.

The handler function must return void, which makes sense because (unlike with normal functions) there is no standard place in the program for this function to return. The function takes one argument, an integer, which is the signal identifier (for example, SIGUSR2) of the signal being handled. This allows a single function to handle multiple signals. A prototype has the form:

```
void my_handler (int signo);
```

Linux uses a typedef, sighandler_t, to define this prototype. Other Unix systems directly use the function pointers; some systems have their own types, which may not be named sighandler_t. Programs seeking portability should not reference the type directly.

When it raises a signal to a process that has registered a signal handler, the kernel suspends execution of the program's regular instruction stream and calls the signal handler. The handler is passed the value of the signal, which is the signo originally provided to signal().

You may also use signal() to instruct the kernel to ignore a given signal for the current process or to reset the signal to the default behavior. This is done using special values for the handler parameter:

SIG_DFL
: Set the behavior of the signal given by signo to its default. For example, in the case of SIGPIPE, the process will terminate.

SIG_IGN
: Ignore the signal given by signo.

The signal() function returns the previous behavior of the signal, which could be a pointer to a signal handler, SIG_DFL, or SIG_IGN. On error, the function returns SIG_ERR. It does not set errno.

Waiting for a Signal, Any Signal

Useful for debugging and writing demonstrative code snippets, the POSIX-defined pause() system call puts a process to sleep until it receives a signal that either is handled or terminates the process:

```
#include <unistd.h>

int pause (void);
```

pause() returns only if a signal is received, in which case the signal is handled, and pause() returns –1 and sets errno to EINTR. If the kernel raises an ignored signal, the process does not wake up.

In the Linux kernel, `pause()` is one of the simplest system calls. It performs only two actions. First, it puts the process in the interruptible sleep state. Next, it calls `schedule()` to invoke the Linux process scheduler to find another process to run. As the process is not actually waiting for anything, the kernel will not wake it up unless it receives a signal. This whole ordeal consumes only two lines of C code.[2]

Examples

Let's look at a couple of simple examples. This first one registers a signal handler for `SIGINT` that simply prints a message and then terminates the program (as `SIGINT` would do anyway):

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

/* handler for SIGINT */
static void sigint_handler (int signo)
{
        /*
         * Technically, you shouldn't use printf() in a
         * signal handler, but it isn't the end of the
         * world. I'll discuss why in the section
         * "Reentrancy."
         */
        printf ("Caught SIGINT!\n");
        exit (EXIT_SUCCESS);
}

int main (void)
{
        /*
         * Register sigint_handler as our signal handler
         * for SIGINT.
         */
        if (signal (SIGINT, sigint_handler) == SIG_ERR) {
                fprintf (stderr, "Cannot handle SIGINT!\n");
                exit (EXIT_FAILURE);
        }

        for (;;)
                pause ();

        return 0;
}
```

2. Thus, `pause()` is only the second-simplest system call. The joint winners are `getpid()` and `gettid()`, each only one line.

In the following example, we register the same handler for `SIGTERM` and `SIGINT`. We also reset the behavior for `SIGPROF` to the default (which is to terminate the process) and ignore `SIGHUP` (which would otherwise terminate the process):

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

/* handler for SIGINT and SIGTERM */
static void signal_handler (int signo)
{
        if (signo == SIGINT)
                printf ("Caught SIGINT!\n");
        else if (signo == SIGTERM)
                printf ("Caught SIGTERM!\n");
        else {
                /* this should never happen */
                fprintf (stderr, "Unexpected signal!\n");
                exit (EXIT_FAILURE);
        }
        exit (EXIT_SUCCESS);
}

int main (void)
{
        /*
         * Register signal_handler as our signal handler
         * for SIGINT.
         */
        if (signal (SIGINT, signal_handler) == SIG_ERR) {
                fprintf (stderr, "Cannot handle SIGINT!\n");
                exit (EXIT_FAILURE);
        }

        /*
         * Register signal_handler as our signal handler
         * for SIGTERM.
         */
        if (signal (SIGTERM, signal_handler) == SIG_ERR) {
                fprintf (stderr, "Cannot handle SIGTERM!\n");
                exit (EXIT_FAILURE);
        }

        /* Reset SIGPROF's behavior to the default. */
        if (signal (SIGPROF, SIG_DFL) == SIG_ERR) {
                fprintf (stderr, "Cannot reset SIGPROF!\n");
                exit (EXIT_FAILURE);
        }

        /* Ignore SIGHUP. */
        if (signal (SIGHUP, SIG_IGN) == SIG_ERR) {
```

```
                fprintf (stderr, "Cannot ignore SIGHUP!\n");
                exit (EXIT_FAILURE);
        }

        for (;;)
                pause ();

        return 0;
}
```

Execution and Inheritance

On fork, the child process inherits the signal actions of its parent. That is, the child copies the registered actions (ignore, default, handle) for each signal from its parent. Pending signals are *not* inherited, which makes sense: the pending signal was sent to a specific pid, decidedly not the child.

When a process is created via one of the *exec* family of system calls, all signals are set to their default actions unless the parent process is ignoring them; in that case, the newly imaged process will also ignore those signals. Put another way, any signal caught by the process before *exec* is reset to the default action after *exec*, and all other signals remain the same. This makes sense because a freshly executed process does not share the address space of its parent, and thus any registered signal handlers may not exist. Pending signals are inherited. Table 10-2 summarizes the inheritance.

Table 10-2. Inherited signal behavior

Signal behavior	Across forks	Across execs
Ignored	Inherited	Inherited
Default	Inherited	Inherited
Handled	Inherited	Not inherited
Pending signals	Not inherited	Inherited

This behavior on process execution has one notable use: when the shell executes a process "in the background" (or when another background process executes another process), the newly executed process should ignore the interrupt and quit characters. Thus, before a shell executes a background process, it should set `SIGINT` and `SIGQUIT` to `SIG_IGN`. It is therefore common for programs that handle these signals to first check to make sure they are not ignored. For example:

```
/* handle SIGINT, but only if it isn't ignored */
if (signal (SIGINT, SIG_IGN) != SIG_IGN) {
        if (signal (SIGINT, sigint_handler) == SIG_ERR)
                fprintf (stderr, "Failed to handle SIGINT!\n");
}

/* handle SIGQUIT, but only if it isn't ignored */
```

```
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN) {
        if (signal (SIGQUIT, sigquit_handler) == SIG_ERR)
                fprintf (stderr, "Failed to handle SIGQUIT!\n");
}
```

The need to set a signal behavior to check the signal behavior highlights a deficiency in the `signal()` interface. Later, we will study a function that does not have this flaw.

Mapping Signal Numbers to Strings

In our examples thus far, we have hardcoded the names of the signals. But sometimes it is more convenient (or even a requirement) that you be able to convert a signal number to a string representation of its name. There are several ways to do this. One is to retrieve the string from a statically defined list:

```
extern const char * const sys_siglist[];
```

`sys_siglist` is an array of strings holding the names of the signals supported by the system, indexed by signal number.

An alternative is the BSD-defined `psignal()` interface, which is common enough that Linux supports it, too:

```
#include <signal.h>

void psignal (int signo, const char *msg);
```

A call to `psignal()` prints to `stderr` the string you supply as the `msg` argument, followed by a colon, a space, and the name of the signal given by `signo`. If `msg` is omitted, only the signal name is printed. If `signo` is invalid, the printed message will say so.

A better interface is `strsignal()`. It is not standardized, but Linux and many non-Linux systems support it:

```
#define _GNU_SOURCE
#include <string.h>

char * strsignal (int signo);
```

A call to `strsignal()` returns a pointer to a description of the signal given by `signo`. If `signo` is invalid, the returned description typically says so (some Unix systems that support this function return `NULL` instead). The returned string is valid only until the next invocation of `strsignal()`, so this function is not thread-safe.

Going with `sys_siglist` is usually your best bet. Using this approach, we could rewrite our earlier signal handler as follows:

```
static void signal_handler (int signo)
{
        printf ("Caught %s\n", sys_siglist[signo]);
}
```

Sending a Signal

The `kill()` system call, the basis of the common *kill* utility, sends a signal from one process to another:

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

int kill (pid_t pid, int signo);
```

In its normal use (i.e., if `pid` is greater than `0`), `kill()` sends the signal `signo` to the process identified by `pid`.

If `pid` is `0`, `signo` is sent to every process in the invoking process's process group.

If `pid` is `-1`, `signo` is sent to every process for which the invoking process has permission to send a signal, except itself and *init*. We will discuss the permissions regulating signal delivery in the next subsection.

If `pid` is less than `-1`, `signo` is sent to the process group `-pid`.

On success, `kill()` returns `0`. The call is considered a success so long as a single signal was sent. On failure (no signals sent), the call returns `-1` and sets `errno` to one of the following:

`EINVAL`
: The signal specified by `signo` is invalid.

`EPERM`
: The invoking process lacks sufficient permissions to send a signal to any of the requested processes.

`ESRCH`
: The process or process group denoted by `pid` does not exist or, in the case of a process, is a zombie.

Permissions

In order to send a signal to another process, the sending process needs proper permissions. A process with the `CAP_KILL` capability (usually one owned by root) can send a signal to any process. Without this capability, the sending process's effective or real user ID must be equal to the real or saved user ID of the receiving process. Put more simply, a user can send a signal only to a process that he or she owns.

Unix systems, including Linux, define an exception for `SIGCONT`: a process can send this signal to any other process in the same session. The user ID need not match.

If `signo` is `0` (the aforementioned null signal) the call does not send a signal, but it still performs error checking. This is useful to test whether a process has suitable permissions to send the provided process or processes a signal.

Examples

Here's how to send `SIGHUP` to the process with process ID 1722:

```
int ret;

ret = kill (1722, SIGHUP);
if (ret)
        perror ("kill");
```

This snippet is effectively the same as the following invocation of the *kill* utility:

```
$ kill -HUP 1722
```

To check that we have permission to send a signal to 1722 without actually sending any signal, we could do the following:

```
int ret;

ret = kill (1722, 0);
if (ret)
        ; /* we lack permission */
else
        ; /* we have permission */
```

Sending a Signal to Yourself

The `raise()` function is a simple way for a process to send a signal to itself:

```
#include <signal.h>

int raise (int signo);
```

This call:

```
raise (signo);
```

is equivalent to the following call:

```
kill (getpid (), signo);
```

The call returns `0` on success and a nonzero value on failure. It does not set `errno`.

Sending a Signal to an Entire Process Group

Another convenience function makes it easy to send a signal to all processes in a given process group in the event that negating the process group ID and using `kill()` is deemed too taxing:

```
#include <signal.h>

int killpg (int pgrp, int signo);
```

This call:

```
killpg (pgrp, signo);
```

is equivalent to the following call:

```
kill (-pgrp, signo);
```

This holds true even if `pgrp` is `0`, in which case `killpg( )` sends the signal `signo` to every process in the invoking process's group.

On success, `killpg()` returns `0`. On failure, it returns `-1` and sets `errno` to one of the following values:

`EINVAL`
: The signal specified by `signo` is invalid.

`EPERM`
: The invoking process lacks sufficient permissions to send a signal to any of the requested processes.

`ESRCH`
: The process group denoted by `pgrp` does not exist.

Reentrancy

When the kernel raises a signal, a process can be executing code anywhere. For example, it might be in the middle of an important operation that, if interrupted, would leave the process is an inconsistent state—say, with a data structure only half updated or a calculation only partially performed. The process might even be handling another signal.

Signal handlers cannot tell what code the process is executing when a signal hits; the handler can run in the middle of anything. It is thus very important that any signal handler your process installs be very careful about the actions it performs and the data it touches. Signal handlers must take care not to make assumptions about what the process was doing when it was interrupted. In particular, they must practice caution when modifying global (that is, shared) data. Indeed, it is a good idea for a signal handler never to touch global data; in an upcoming section, however, we will look at a way to temporarily block the delivery of signals as a way to allow safe manipulation of data shared by a signal handler and the rest of a process.

What about system calls and other library functions? What if your process is in the middle of writing to a file or allocating memory, and a signal handler writes to the same file or also invokes `malloc()`? Or what if a process is in the middle of a call to a function that uses a static buffer, such as `strsignal()`, when a signal is delivered?

Some functions are clearly not reentrant. If a program is in the middle of executing a nonreentrant function and a signal occurs and the signal handler then invokes that same nonreentrant function, chaos can ensue. A *reentrant function* is a function that is safe to call from within itself (or concurrently, from another thread in the same process). In order to qualify as reentrant, a function must not manipulate static data, must manipulate only stack-allocated data or data provided to it by the caller, and must not invoke any nonreentrant function.

Guaranteed-Reentrant Functions

When writing a signal handler, you have to assume that the interrupted process could be in the middle of a nonreentrant function (or anything else, for that matter). Thus, signal handlers must make use only of functions that are reentrant.

Various standards have decreed lists of functions that are *signal-safe*: reentrant and thus safe to use from within a signal handler. Most notably, POSIX.1-2003 and the Single UNIX Specification dictate a list of functions that are guaranteed to be reentrant and signal-safe on all compliant platforms. Table 10-3 lists the functions.

Table 10-3. Functions guaranteed to be safely reentrant for use in signals

`abort()`	`accept()`	`access()`
`aio_error()`	`aio_return()`	`aio_suspend()`
`alarm()`	`bind()`	`cfgetispeed()`
`cfgetospeed()`	`cfsetispeed()`	`cfsetospeed()`
`chdir()`	`chmod()`	`chown()`
`clock_gettime()`	`close()`	`connect()`
`creat()`	`dup()`	`dup2()`
`execle()`	`execve()`	`_Exit()`
`_exit()`	`fchmod()`	`fchown()`
`fcntl()`	`fdatasync()`	`fork()`
`fpathconf()`	`fstat()`	`fsync()`
`ftruncate()`	`getegid()`	`geteuid()`
`getgid()`	`getgroups()`	`getpeername()`
`getpgrp()`	`getpid()`	`getppid()`
`getsockname()`	`getsockopt()`	`getuid()`
`kill()`	`link()`	`listen()`
`lseek()`	`lstat()`	`mkdir()`
`mkfifo()`	`open()`	`pathconf()`
`pause()`	`pipe()`	`poll()`
`posix_trace_event()`	`pselect()`	`raise()`

`read()`	`readlink()`	`recv()`
`recvfrom()`	`recvmsg()`	`rename()`
`rmdir()`	`select()`	`sem_post()`
`send()`	`sendmsg()`	`sendto()`
`setgid()`	`setpgid()`	`setsid()`
`setsockopt()`	`setuid()`	`shutdown()`
`sigaction()`	`sigaddset()`	`sigdelset()`
`sigemptyset()`	`sigfillset()`	`sigismember()`
`signal()`	`sigpause()`	`sigpending()`
`sigprocmask()`	`sigqueue()`	`sigset()`
`sigsuspend()`	`sleep()`	`socket()`
`socketpair()`	`stat()`	`symlink()`
`sysconf()`	`tcdrain()`	`tcflow()`
`tcflush()`	`tcgetattr()`	`tcgetpgrp()`
`tcsendbreak()`	`tcsetattr()`	`tcsetpgrp()`
`time()`	`timer_getoverrun()`	`timer_gettime()`
`timer_settime()`	`times()`	`umask()`
`uname()`	`unlink()`	`utime()`
`wait()`	`waitpid()`	`write()`

Many more functions are safe, but Linux and other POSIX-compliant systems guarantee the reentrancy of only these functions.

Signal Sets

Several of the functions we will look at later in this chapter need to manipulate sets of signals, such as the set of signals blocked by a process or the set of signals pending to a process. The *signal set operations* manage these signal sets:

```
#include <signal.h>

int sigemptyset (sigset_t *set);

int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);

int sigdelset (sigset_t *set, int signo);

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

`sigemptyset()` initializes the signal set given by `set`, marking it empty (all signals excluded from the set). `sigfillset()` initializes the signal set given by `set`, marking it

full (all signals included in the set). Both functions return `0`. You should call one of these two functions on a signal set before further using the set.

`sigaddset()` adds `signo` to the signal set given by `set`, while `sigdelset()` removes `signo` from the signal set given by `set`. Both return `0` on success or `-1` on error, in which case `errno` is set to the error code `EINVAL`, signifying that `signo` is an invalid signal identifier.

`sigismember()` returns `1` if `signo` is in the signal set given by `set`, `0` if it is not, and `-1` on error. In the latter case, `errno` is again set to `EINVAL`, signifying that `signo` is invalid.

More Signal Set Functions

The preceding functions are all standardized by POSIX and found on any modern Unix system. Linux also provides several nonstandard functions:

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);

int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);

int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

`sigisemptyset()` returns `1` if the signal set given by `set` is empty and `0` otherwise.

`sigorset()` places the union (the binary OR) of the signal sets `left` and `right` in `dest`. `sigandset()` places the intersection (the binary AND) of the signal sets `left` and `right` in `dest`. Both return `0` on success and `-1` on error, setting `errno` to `EINVAL`.

These functions are useful, but programs desiring POSIX compliance should avoid them.

Blocking Signals

Earlier we discussed reentrancy and the issues raised by signal handlers running asynchronously, at any time. We discussed functions not to call from within a signal handler because they themselves are not reentrant.

But what if your program needs to share data between a signal handler and elsewhere in the program? What if there are portions of your program's execution during which you do not want any interruptions, including from signal handlers? We call such parts of a program *critical regions*, and we protect them by temporarily suspending the delivery of signals. We say that such signals are *blocked*. Any signals that are raised while blocked are not handled until they are unblocked. A process may block any number of signals; the set of signals blocked by a process is called its *signal mask*.

POSIX defines, and Linux implements, a function for managing a process's signal mask:

```
#include <signal.h>

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

The behavior of `sigprocmask()` depends on the value of `how`, which is one of the following flags:

`SIG_SETMASK`
: The signal mask for the invoking process is changed to `set`.

`SIG_BLOCK`
: The signals in `set` are added to the invoking process's signal mask. In other words, the signal mask is changed to the union (binary OR) of the current mask and `set`.

`SIG_UNBLOCK`
: The signals in `set` are removed from the invoking process's signal mask. In other words, the signal is changed to the intersection (binary AND) of the current mask, and the negation (binary NOT) of `set`. It is illegal to unblock a signal that is not blocked.

If `oldset` is not `NULL`, the function places the previous signal set in `oldset`.

If `set` is `NULL`, the function ignores `how` and does not change the signal mask, but it does place the signal mask in `oldset`. In other words, passing a null value as `set` is the way to retrieve the current signal mask.

On success, the call returns `0`. On failure, it returns `-1` and sets `errno` to either `EINVAL`, signifying that `how` was invalid, or `EFAULT`, signifying that `set` or `oldset` was an invalid pointer.

Blocking `SIGKILL` or `SIGSTOP` is not allowed. `sigprocmask()` silently ignores any attempt to add either signal to the signal mask.

Retrieving Pending Signals

When the kernel raises a blocked signal, it is not delivered. We call such signals *pending*. When a pending signal is unblocked, the kernel then passes it off to the process to handle.

POSIX defines a function to retrieve the set of pending signals:

```
#include <signal.h>

int sigpending (sigset_t *set);
```

A successful call to `sigpending()` places the set of pending signals in `set` and returns `0`. On failure, the call returns `-1` and sets `errno` to `EFAULT`, signifying that `set` is an invalid pointer.

Waiting for a Set of Signals

A third POSIX-defined function allows a process to temporarily change its signal mask and then wait until a signal is raised that either terminates or is handled by the process:

```
#include <signal.h>

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

If a signal terminates the process, `sigsuspend()` does not return. If a signal is raised and handled, `sigsuspend()` returns `-1` after the signal handler returns, setting `errno` to `EINTR`. If `set` is an invalid pointer, `errno` is set to `EFAULT`.

A common `sigsuspend()` usage scenario is to retrieve signals that might have arrived and been blocked during a critical region of program execution. The process first uses `sigprocmask()` to block a set of signals, saving the old mask in `oldset`. After exiting the critical region, the process then calls `sigsuspend()`, providing `oldset` for `set`.

Advanced Signal Management

691.170bThe `signal()` function that we studied at the beginning of this chapter is very basic. Because it is part of the standard C library and therefore has to reflect minimal assumptions about the capabilities of the operating system on which it runs, it can offer only a lowest common denominator to signal management. As an alternative, POSIX standardizes the `sigaction()` system call, which provides much greater signal management capabilities. Among other things, you can use it to block the reception of specified signals while your handler runs, and to retrieve a wide range of data about the system and process state at the moment a signal was raised:

```
#include <signal.h>

int sigaction (int signo,
               const struct sigaction *act,
               struct sigaction *oldact);
```

A call to `sigaction()` changes the behavior of the signal identified by `signo`, which can be any value except those associated with `SIGKILL` and `SIGSTOP`. If `act` is not `NULL`, the system call changes the current behavior of the signal as specified by `act`. If `oldact` is not `NULL`, the call stores the previous (or current, if `act` is `NULL`) behavior of the given signal there.

The `sigaction` structure allows for fine-grained control over signals. The header `<sys/signal.h>`, included from `<signal.h>`, defines the structure as follows:

```
struct sigaction {
        void (*sa_handler)(int);   /* signal handler or action */
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;          /* signals to block */
        int sa_flags;              /* flags */
        void (*sa_restorer)(void); /* obsolete and non-POSIX */
};
```

The `sa_handler` field dictates the action to take upon receiving the signal. As with `signal()`, this field may be `SIG_DFL`, signifying the default action, `SIG_IGN`, instructing the kernel to ignore the signal for the process, or a pointer to a signal-handling function. The function has the same prototype as a signal handler installed by `signal()`:

```
void my_handler (int signo);
```

If `SA_SIGINFO` is set in `sa_flags`, `sa_sigaction`, and not `sa_handler`, dictates the signal-handling function. This function's prototype is slightly different:

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

The function receives the signal number as its first parameter, a `siginfo_t` structure as its second parameter, and a `ucontext_t` structure (cast to a `void` pointer) as its third parameter. It has no return value. The `siginfo_t` structure provides an abundance of information to the signal handler; we will look at it shortly.

Note that on some machine architectures (and possibly other Unix systems), `sa_handler` and `sa_sigaction` are in a union, and you should not assign values to both fields.

The `sa_mask` field provides a set of signals that the system should block for the duration of the execution of the signal handler. This allows programmers to enforce proper protection from reentrancy among multiple signal handlers. The signal currently being handled is also blocked unless the `SA_NODEFER` flag is set in `sa_flags`. You cannot block `SIGKILL` or `SIGSTOP`; the call will silently ignore either in `sa_mask`.

The `sa_flags` field is a bitmask of zero, one, or more flags that change the handling of the signal given by `signo`. We already looked at the `SA_SIGINFO` and `SA_NODEFER` flags; other values for `sa_flags` include the following:

`SA_NOCLDSTOP`
: If `signo` is `SIGCHLD`, this flag instructs the system to not provide notification when a child process stops or resumes.

`SA_NOCLDWAIT`
: If `signo` is `SIGCHLD`, this flag enables *automatic child reaping*: children are not converted to zombies on termination, and the parent need not (and cannot) call `wait()` on them. See Chapter 5 for a lively discussion of children, zombies, and `wait()`.

`SA_NOMASK`
: This flag is an obsolete non-POSIX equivalent to `SA_NODEFER` (discussed earlier in this section). Use `SA_NODEFER` instead of this flag, but be prepared to see this value turn up in older code.

`SA_ONESHOT`
: This flag is an obsolete non-POSIX equivalent to `SA_RESETHAND` (discussed later in this list). Use `SA_RESETHAND` instead of this flag, but be prepared to see this value turn up in older code.

`SA_ONSTACK`
: This flag instructs the system to invoke the given signal handler on an *alternative signal stack*, as provided by `sigaltstack()`. If you do not provide an alternative stack, the default is used—that is, the system behaves as if you did not provide this flag. Alternative signal stacks are rare, although they are useful in some Pthreads applications with smaller thread stacks that might be overrun by some signal handler usage. We do not further discuss `sigaltstack()` in this book.

`SA_RESTART`
: This flag enables BSD-style restarting of system calls that are interrupted by signals.

`SA_RESETHAND`
: This flag enables "one-shot" mode. The behavior of the given signal is reset to the default once the signal handler returns.

The `sa_restorer` field is obsolete and no longer used in Linux. It is not part of POSIX, anyhow. Pretend that it is not there, and do not touch it.

`sigaction()` returns `0` on success. On failure, the call returns `-1` and sets `errno` to one of the following error codes:

`EFAULT`
: `act` or `oldact` is an invalid pointer.

`EINVAL`
: `signo` is an invalid signal, `SIGKILL`, or `SIGSTOP`.

The siginfo_t Structure

The `siginfo_t` structure is also defined in `<sys/signal.h>`, as follows:

```
typedef struct siginfo_t {
        int si_signo;      /* signal number */
        int si_errno;      /* errno value */
        int si_code;       /* signal code */
        pid_t si_pid;      /* sending process's PID */
        uid_t si_uid;      /* sending process's real UID */
        int si_status;     /* exit value or signal */
        clock_t si_utime;  /* user time consumed */
        clock_t si_stime;  /* system time consumed */
        sigval_t si_value; /* signal payload value */
        int si_int;        /* POSIX.1b signal */
        void *si_ptr;      /* POSIX.1b signal */
        void *si_addr;     /* memory location that caused fault */
        int si_band;       /* band event */
        int si_fd;         /* file descriptor */
};
```

This structure is rife with information passed to the signal handler (if you're using `sa_sigaction` in lieu of `sa_sighandler`). With modern computing, many consider the Unix signal model an awful method for performing IPC. Perhaps the problem is that these folks are stuck using `signal()` when they should be using `sigaction()` with `SA_SIGINFO`. The `siginfo_t` structure opens the door for wringing a lot more functionality out of signals.

There's a lot of interesting data in this structure, including information about the process that sent the signal and about the cause of the signal. Here is a detailed description of each of the fields:

`si_signo`
: The signal number of the signal in question. In your signal handler, the first argument provides this information as well (and avoids a pointer dereference).

`si_errno`
: If nonzero, the error code associated with this signal. This field is valid for all signals.

`si_code`
: An explanation of why and from where the process received the signal (for example, from `kill()`). We will go over the possible values in the following section. This field is valid for all signals.

`si_pid`
: For `SIGCHLD`, the pid of the process that terminated.

`si_uid`
: For `SIGCHLD`, the owning uid of the process that terminated.

`si_status`
: For `SIGCHLD`, the exit status of the process that terminated.

`si_utime`
: For `SIGCHLD`, the user time consumed by the process that terminated.

`si_stime`
: For `SIGCHLD`, the system time consumed by the process that terminated.

`si_value`
: A union of `si_int` and `si_ptr`.

`si_int`
: For signals sent via `sigqueue()` (see "Sending a Signal with a Payload" on page 361), the provided payload typed as an integer.

`si_ptr`
: For signals sent via `sigqueue()` (see "Sending a Signal with a Payload" on page 361), the provided payload typed as a `void` pointer.

`si_addr`
: For `SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV`, and `SIGTRAP`, this `void` pointer contains the address of the offending fault. For example, in the case of `SIGSEGV`, this field contains the address of the memory access violation (and is thus often `NULL`!).

`si_band`
: For `SIGPOLL`, out-of-band and priority information for the file descriptor listed in `si_fd`.

`si_fd`
: For `SIGPOLL`, the file descriptor for the file whose operation completed.

`si_value`, `si_int`, and `si_ptr` are particularly complex topics because a process can use them to pass arbitrary data to another process. Thus, you can use them to send either a simple integer or a pointer to a data structure (note that a pointer is not much help if the processes do not share an address space). These fields are discussed in the upcoming section "Sending a Signal with a Payload" on page 361.

POSIX guarantees that only the first three fields are valid for all signals. The other fields should be accessed only when handling the applicable signal. You should access the `si_fd` field, for example, only if the signal is `SIGPOLL`.

The Wonderful World of si_code

The `si_code` field indicates the cause of the signal. For user-sent signals, the field indicates how the signal was sent. For kernel-sent signals, the field indicates why the signal was sent.

The following `si_code` values are valid for any signal. They indicate how/why the signal was sent:

`SI_ASYNCIO`
: The signal was sent due to the completion of asynchronous I/O (see Chapter 5).

`SI_KERNEL`
: The signal was raised by the kernel.

`SI_MESGQ`
: The signal was sent due to a state change of a POSIX message queue (not covered in this book).

`SI_QUEUE`
: The signal was sent by `sigqueue()` (see the next section).

`SI_TIMER`
: The signal was sent due to the expiration of a POSIX timer (see Chapter 11).

`SI_TKILL`
: The signal was sent by `tkill()` or `tgkill()`. These system calls are used by threading libraries and are not covered in this book.

`SI_SIGIO`
: The signal was sent due to the queuing of `SIGIO`.

`SI_USER`
: The signal was sent by `kill()` or `raise()`.

The following `si_code` values are valid for `SIGBUS` only. They indicate the type of hardware error that occurred:

`BUS_ADRALN`
: The process incurred an alignment error (see Chapter 9 for a discussion of alignment).

`BUS_ADRERR`
: The process accessed an invalid physical address.

`BUS_OBJERR`
: The process caused some other form of hardware error.

For `SIGCHLD`, the following values identify what the child did to generate the signal sent to its parent:

`CLD_CONTINUED`
: The child was stopped but has resumed.

`CLD_DUMPED`
: The child terminated abnormally.

`CLD_EXITED`
: The child terminated normally via `exit()`.

`CLD_KILLED`
: The child was killed.

`CLD_STOPPED`
: The child stopped.

`CLD_TRAPPED`
: The child hit a trap.

The following values are valid for `SIGFPE` only. They explain the type of arithmetic error that occurred:

`FPE_FLTDIV`
: The process performed a floating-point operation that resulted in division by zero.

`FPE_FLTOVF`
: The process performed a floating-point operation that resulted in an overflow.

`FPE_FLTINV`
: The process performed an invalid floating-point operation.

`FPE_FLTRES`
: The process performed a floating-point operation that yielded an inexact or invalid result.

`FPE_FLTSUB`
: The process performed a floating-point operation that resulted in an out-of-range subscript.

`FPE_FLTUND`
: The process performed a floating-point operation that resulted in an underflow.

`FPE_INTDIV`
: The process performed an integer operation that resulted in division by zero.

`FPE_INTOVF`
: The process performed an integer operation that resulted in an overflow.

The following `si_code` values are valid for `SIGILL` only. They explain the nature of the illegal instruction execution:

`ILL_ILLADR`
: The process attempted to enter an illegal addressing mode.

`ILL_ILLOPC`
: The process attempted to execute an illegal opcode.

`ILL_ILLOPN`
: The process attempted to execute on an illegal operand.

`ILL_PRVOPC`
: The process attempted to execute a privileged opcode.

`ILL_PRVREG`
: The process attempted to execute on a privileged register.

`ILL_ILLTRP`
: The process attempted to enter an illegal trap.

For all of these values, `si_addr` points to the address of the offense.

For `SIGPOLL`, the following values identify the I/O event that generated the signal:

`POLL_ERR`
: An I/O error occurred.

`POLL_HUP`
: The device hung up or the socket disconnected.

`POLL_IN`
: The file has data available to read.

`POLL_MSG`
: A message is available.

`POLL_OUT`
: The file is capable of being written to.

`POLL_PRI`
: The file has high-priority data available to read.

The following codes are valid for `SIGSEGV`, describing the two types of invalid memory accesses:

`SEGV_ACCERR`
: The process accessed a valid region of memory in an invalid way—that is, the process violated memory-access permissions.

`SEGV_MAPERR`
: The process accessed an invalid region of memory.

For either of these values, `si_addr` contains the offending address.

For `SIGTRAP`, these two `si_code` values identify the type of trap hit:

`TRAP_BRKPT`
: The process hit a break point.

`TRAP_TRACE`
: The process hit a trace trap.

Note that `si_code` is a value field and not a bit field.

Sending a Signal with a Payload

As we saw in the previous section, signal handlers registered with the `SA_SIGINFO` flag are passed a `siginfo_t` parameter. This structure contains a field named `si_value`, which is an optional payload passed from the signal generator to the signal receiver.

The `sigqueue()` function, defined by POSIX, allows a process to send a signal with this payload:

```
#include <signal.h>

int sigqueue (pid_t pid,
              int signo,
              const union sigval value);
```

`sigqueue()` works similarly to `kill()`. On success, the signal identified by `signo` is queued to the process or process group identified by `pid`, and the function returns `0`. The signal's payload is given by `value`, which is a `union` of an integer and a `void` pointer:

```
union sigval {
        int sival_int;
        void *sival_ptr;
};
```

On failure, the call returns `-1` and sets `errno` to one of the following:

`EAGAIN`
: The invoking process has reached the limit on enqueued signals.

`EINVAL`
: The signal specified by `signo` is invalid.

`EPERM`
: The invoking process lacks sufficient permissions to send a signal to any of the requested processes. The permissions required to send a signal are the same as with `kill()` (see "Sending a Signal" on page 346).

`ESRCH`
: The process or process group denoted by `pid` does not exist or, in the case of a process, is a zombie.

As with `kill()`, you may pass the null signal (`0`) for `signo` to test permissions.

Signal Payload Example

This example sends the process with pid 1722 the `SIGUSR2` signal with a payload of an integer that has the value 404:

```
sigval value;
int ret;

value.sival_int = 404;

ret = sigqueue (1722, SIGUSR2, value);
if (ret)
        perror ("sigqueue");
```

If process 1722 handles `SIGUSR2` with an `SA_SIGINFO` handler, it will find `signo` set to `SIGUSR2`, `si->si_int` set to `404`, and `si->si_code` set to `SI_QUEUE`.

A Flaw in Unix?

Signals have a bad reputation among many Unix programmers. They are an old, antiquated mechanism for kernel-to-user communication and are, at best, a primitive form of IPC. In a world of multithreading programs and event loops, signals feel anachronistic. In a system that has so impressively weathered time and that finds itself sporting the original programming paradigm it introduced on day one, signals are a rare misstep. I would not take the challenge of rethinking signals lightly, but a more expressive, easily extensible, thread-safe, and file descriptor based solution seems a proper start.

Nonetheless, for better or worse, we are stuck with signals. They are the only way to receive many notifications (such as the notification of an illegal opcode execution) from the kernel. Additionally, signals are how Unix (and thus Linux) terminates processes and manages the parent/child relationship. Thus, programmers must understand them and use them.

One of the primary reasons for signals' derogation is that it is hard to write a proper signal handler that is safe from reentrancy concerns. If you keep your handlers simple, however, and use only the functions listed in Table 10-3 (if you use any!), they should be safe.

Another chink in signals' armor is that many programmers still use `signal()` and `kill()`, rather than `sigaction()` and `sigqueue()`, for signal management. As the last two sections have shown, signals are significantly more powerful and expressive when `SA_SIGINFO`-style signal handlers are used. Although I myself am no fan of signals, working around their flaws and using Linux's advanced signal interfaces eases much of the pain (if not the whining).