9

Signals and Timers

9.1 Signal Basics

A *signal* is a notification that an event has occurred, such as a user typing an interrupt (Ctrl-c, normally), a floating-point exception, or an alarm going off. Usually, a signal is delivered to a process or thread asynchronously and whatever the process or thread is doing is interrupted. The signal might immediately terminate the process, or, by prearrangement, a function designated to catch it might be executed.

9.1.1 Introduction to Signals

To show how a program handles a signal, here's a simple example of a program that displays a number once every three seconds, but when an interrupt signal occurs, it displays a message and terminates:

```
static void fcn(int signum),
{
    (void)write(STDOUT_FILENO, "Got signal\n", 11);
    _exit(EXIT_FAILURE);
}

int main(void)
{
    int i;
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = fcn;
    ec_neg1( sigaction(SIGINT, &act, NULL) )

    for (i = 1; ; i++) {
        sleep(3);
        printf("%d\n", i);
    }
```

```
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The call to `sigaction` installed a signal-handling function for the `SIGINT` signal. When I ran this program, I typed a Ctrl-c after a "2" appeared. That caused whatever the program was doing (perhaps sleeping, or inside `printf`, or just looping) to be interrupted and the function `fcn` to be called immediately. It displayed a message and terminated the program with a call to `_exit`. (See Section 9.1.7 for why I didn't use `exit`.)

The output on the screen was:

```
1
2
Got signal
```

If I hadn't installed the signal handler, typing Ctrl-c would have terminated the process right away. Technically, the reason is that the default action for the `SIGINT` signal is to terminate the process.

I also could have called `sigaction` to arrange for `SIGINT` signals to be ignored:

```
int main(void)
{
    int i;
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = SIG_IGN;
    ec_neg1( sigaction(SIGINT, &act, NULL) )

    for (i = 1; ; i++) {
        sleep(3);
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

This time it kept displaying numbers, and typing Ctrl-c had no effect. We finally terminated it by typing Ctrl-\, which generated a quit signal (`SIGQUIT`), whose default behavior, which we didn't modify, is to terminate the process.

Well, so much for the basics. Signals are actually pretty complicated because:

- There are lots of different signals, and sometimes the circumstances under which they're generated and what they mean are complicated.
- Arranging the appropriate action for a signal can be complicated.
- Handling a signal can be complicated.

I'll go through the issues step-by-step, and when you've finished this chapter all will be clear.

9.1.2 A Signal's Lifecycle

A signal is born when whatever event it's associated with occurs and *generates* it. Its life ends when it is *delivered,* which means that whatever *action* specified for it has been taken. There are three possible actions:

1. Default action (`SIG_DFL`): terminate, stop, or continue the process, or ignore the signal.

2. Ignore the signal (`SIG_IGN`).

3. Catch the signal by executing a signal handler when the signal is delivered.

Most signals can be generated *naturally* by whatever user or system event the signal is associated with. For example, dividing by zero generates a `SIGFPE` event naturally, and termination of a child generates a `SIGCHLD` event naturally. Alternatively, any signal can be generated *synthetically* by one of five system calls: `kill`, `killpg`, `pthread_kill`, `raise`, or `sigqueue`. The next section, which enumerates all the signals, indicates the natural cause of each signal that has one. Five signals, `SIGKILL`, `SIGSTOP`, `SIGTERM`, `SIGUSR1`, and `SIGUSR2` have no natural cause and exist only to be synthesized, usually with the `kill` system call.

Between generation and delivery a signal is *pending*. If another signal of the same type arrives while a signal is pending, whether multiple signals of the same type are delivered is implementation dependent and should not be assumed in portable programs. There's more on this subject in Section 9.5.3.

A thread (Section 5.17) can keep pending signals in the pending state by *blocking* them.[1] The set of all currently blocked signals is called the *signal mask*. There are various system calls, which I'll explain in Section 9.1.5, that can be used to create and manipulate masks and make a particular mask the effective signal mask for the thread.

A signal is either generated for a specific thread or for an entire process. In the latter case, if there's more than one thread that has it unblocked, which thread it's delivered to is undefined. Section 9.1.3 gives more information about which events are sent to a thread vs. a process and under what circumstances.[2]

The *action* for a signal is process-wide, even if there are multiple threads, although, as I said, the signal mask is per-thread.

Normally, when a signal handler is executed, the signal it's handling is temporarily added to that thread's signal mask so that a second signal of that type won't be delivered until the handler returns. That way you don't have to worry about a recursive call to a handler for the same signal. However, a recursive call is possible if you're using the same function for several different signal types.

9.1.3 Types of Signals

There are 28 signals defined in [SUS2002], and most implementations define a few more that you can look up but which I won't describe here. Also, there are additional signals that are part of the Realtime Signals Extension (Section 9.5). It's useful to consider the SUS signals as falling into a few groups. In the following list, letters in parentheses indicate the default action for the signal, which is explained below. The natural cause for each signal is indicated; the 5 synthetic-only signals (explained in the previous section) are indicated explicitly. Remember that all signals with a natural cause can be also generated synthetically.

- Detected errors:
  `SIGBUS` – access to undefined portion of a memory object (A)
  `SIGFPE` – erroneous arithmetic operation (A)
  `SIGILL` – illegal instruction (A)
  `SIGPIPE` – write on a pipe with no reader (T)

---

1. If there's only one thread in the process, perhaps because it's not doing any multithreading, then whatever I say about a thread applies to the process as a whole.

2. What I say about threads in this chapter applies only to implementations of POSIX Threads. Some implementations of "Linux Threads," widely used on Linux and FreeBSD systems, are not faithful POSIX Threads implementations, and signals don't work with them in the standard way. The newest version now being released, NPTL, works fine, however.

SIGSEGV – invalid memory reference (A)
SIGSYS – bad system call (A)
SIGXCPU – CPU-time limit exceeded (A)
SIGXFSZ – file-size limit exceeded (A)

- User/application-generated:
  SIGABRT – call to abort (A)
  SIGHUP – hangup (T)
  SIGINT – interrupt (from keyboard) (T)
  SIGKILL – kill; synthetic only (T)
  SIGQUIT – quit (from keyboard) (A)
  SIGTERM – termination; synthetic only (T)
  SIGUSR1 – user signal 1; synthetic only (T)
  SIGUSR2 – user signal 2; synthetic only (T)
- Job control:
  SIGCHLD – child process terminated or stopped (I)
  SIGCONT – continue executing (from keyboard) (C)
  SIGSTOP – stop executing; synthetic only (S)
  SIGTSTP – terminal stop signal (from keyboard) (S)
  SIGTTIN – background process attempting read (S)
  SIGTTOU – background process attempting write (S)
- Timer:
  SIGALRM – alarm clock expired (T)
  SIGVTALRM – virtual timer expired (T)
  SIGPROF – profiling timer expired (T)
- Miscellaneous events:
  SIGPOLL – pollable event (T)
  SIGTRAP – trace/breakpoint trap (A)
  SIGURG – out-of-band data available at socket (I)

Here's what the parenthesized letters (default actions) mean:

I signal is ignored

T termination

A same as T, but with additional implementation-defined actions, such as writing of a core-dump file

S stop

C continue after stop

Natural generation of a detected-error signal results from a program error. For `SIGBUS`, `SIGFPE`, `SIGILL`, and `SIGSEGV`, the exact cause of the error isn't standardized, but it's usually something detected by the hardware. Also, these four signals are subject to some special rules when they arise naturally:

- If they were set to be ignored by `sigaction`, their behavior is undefined.
- The result of a signal-catching function returning is undefined.
- The result of one of them occurring while blocked is undefined.

Another way to say this is that if the hardware-detected error is real, your program won't necessarily get past it. It's not safe to ignore it, to continue processing after a signal-handler returns, or to postpone the action by blocking it. You deal with it right away in a signal handler that must exit (or long-jump; see Section 9.6), rather than return, or the process is immediately terminated, which is the default action.

Two of the user/application-generated signals, `SIGINT` and `SIGQUIT`, are normally associated with keyboard control sequences, as explained in Section 4.5.7. `SIGHUP` normally results from hanging up a terminal device. `SIGABRT` is generated by the `abort` system call (Section 9.1.9), and `SIGTERM` is the default signal for the `kill` command—it's the primary way that arbitrary processes are terminated when, for example, the system administrator needs to shut down the system. `SIGUSR1` and `SIGUSR2` aren't used by any system call and are available for application use.

The job control signals were discussed in Section 4.3.

`SIGALRM` is discussed in Section 9.7.1. The other two timer signals are discussed in Section 9.7.4.

Among the miscellaneous-event signals, `SIGPOLL` can be used with STREAMS (Section 4.9); it's enabled with a call to `ioctl`. Normally, it's not generated, so you don't need to worry about it. `SIGURG` was explained in Section 8.7. `SIGTRAP` is used by debuggers.

When a signal is delivered, you can't tell whether it was generated naturally or synthetically. When one of the detected-error signals is generated naturally, it's sent to the offending thread; the other signals are sent to the process. A synthetically generated signal can be sent to the process or to a thread, depending on which system call was used (Section 9.1.9) to generate it.

Programs that need to clean up before terminating should arrange to catch signals SIGHUP, SIGINT, and SIGTERM. Until the program is solid, SIGQUIT should be left alone so there will be a way to terminate the program (with a core dump) from the keyboard. Arrangements for the other signals are made much less often; usually they are left to terminate the process. But a really polished program will want to catch everything it can, to clean up, possibly log the error, and print a nice error message. Psychologically, a message like "Internal error 53: contact customer support" is more acceptable than the message "Bus error-core dumped" from the shell. See Section 9.1.8 for more on this subject and an example program.

9.1.4 Interrupted System Calls

The delivery of a nonignored signal can cause a system call to be interrupted. If the action results in the termination of the process, either because that was the default action for the signal or because the signal-handler terminated the process, as in the examples we showed above, the interrupted system call is never resumed. If the action is to stop the process, execution picks up when the process is continued.

However, if the action is to catch the signal and the signal handler returns, the interrupted system is normally not restarted. Instead, it usually returns –1 with errno set to EINTR. In some cases that's exactly what you want; for example, you might deliberately set an alarm to generate an SIGALRM signal to interrupt a waiting read after, say, 10 seconds. But in other cases an interrupted system call causes problems because the algorithm doesn't allow for an interrupted call.

The simplest rule to follow is to never return from a signal handler unless you've carefully controlled the context in which the signal occurs. If that's not practical, you can set the SA_RESTART flag (Section 9.1.6) when you call sigaction for the signal so that system calls won't be interrupted—they will instead resume where they left off when the signal handler returns.

Only system calls that block can be interrupted. In this context, "blocked" means that the call is waiting for some event whose arrival can't be predicted, such as input from a terminal or socket, termination of a process, arrival of a message, posting of a semaphore, and so on. System calls that merely take some time, such as reading a file or creating a process, are not blocking; although there is a short delay, it's spent processing or waiting for a processor, not waiting for some unpredictable event.

Not every system call that blocks is interruptible. An example is `pthread_mutex_lock`, which continues to wait even if a signal arrives and its handler returns. The only way to know for sure that a system call is interruptible is to read its documentation, preferably in the SUS.

9.1.5 Managing the Signal Mask

As with an `fd_set` used by `select` (Section 4.2.3), signal masks have a collection of functions for manipulating the various bits:[3]

sigemptyset—initialize empty signal set

```
#include <signal.h>

int sigemptyset(
    sigset_t *set          /* signal set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigfillset—initialize full signal set

```
#include <signal.h>

int sigfillset(
    sigset_t *set          /* signal set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigaddset—add signal to signal set

```
#include <signal.h>

int sigaddset(
    sigset_t *set,         /* signal set */
    int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigdelset—delete signal from signal set

```
#include <signal.h>

int sigdelset(
    sigset_t *set,         /* signal set */
    int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

3. An alternative would be to just use an `unsigned long`, but at the time these functions were introduced `long`s were 32 bits on nearly all machines (`long long` hadn't yet been introduced), and 32 was considered to be too small.

sigismember—test for signal in signal set

```
#include <signal.h>

int sigismember(
    const sigset_t *set,        /* signal set */
    int signum                  /* signal */
);
/* Returns 1 if member, 0 if not, or -1 on error (sets errno) */
```

Given a `sigset_t`, you start with `sigemptyset` or `sigfillset`, and then you call `sigaddset` or `sigdelset` to add or delete members. You can call `sigismember` to test whether a signal is a member.

There's only one signal mask at a time for a thread, and you set it with `pthread_sigmask`:

pthread_sigmask—change thread's signal mask

```
int pthread_sigmask(
    int how,                    /* how signal mask is to be changed */
    const sigset_t *set,        /* input set */
    sigset_t *oset              /* previous signal mask */
);
/* Returns 0 on success, error number on failure (errno not set) */
```

How the signal mask is changed by the input `set` is determined by the `how` argument:

`SIG_BLOCK` New signal mask becomes union of current signal mask and `set`.

`SIG_SETMASK` New signal mask becomes `set`, entirely replacing current signal mask.

`SIG_UNBLOCK` New signal mask becomes union of current signal mask and complement of `set`.

In other words, `SIG_BLOCK` adds the signals in the `set` argument to the signal mask, `SIG_UNBLOCK` removes the signals in `set` from the signal mask, and `SIG_SETMASK` just sets the signal mask to `set`.

If `oset` isn't `NULL`, the previous signal mask is returned to what `oset` points to. Also, `set` can be `NULL`, in which case the signal mask isn't changed (regardless of `how`) but is returned through `oset` (if it isn't also `NULL`); this is a way of just getting the signal mask without changing it.

You can't block the SIGKILL or SIGSTOP signals, as they're always delivered to a process and always terminate or stop the process. (They can't be caught or ignored either.)

If there's only one thread in the process, you have the option of calling an older function (dating from before POSIX Threads were introduced) that works identically to pthread_sigmask, except that it uses errno instead of returning the error code:

sigprocmask—change thread's signal mask (single thread only)

```
#include <signal.h>

int sigprocmask(
     int how,               /* how signal mask is to be changed */
     const sigset_t *set,   /* input set */
     sigset_t *oset         /* previous signal mask */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Often, programs not using threads at all aren't linked with the "pthread" library, so pthread_sigmask isn't available and you have no choice but to use sigprocmask.[4]

You don't block a signal with the signal mask because you want your process to ignore it—that's what the SIG_IGN action is for. Instead, blocking is a temporary state that's used to protect some part of your code from the arrival of a signal. One example I already mentioned: When a signal handler is executing, the signal that caused it to be called is temporarily and automatically added to the signal mask and then removed when (and if) the function returns.[5]

Another example is when your application starts up, before it has a chance to set the action for all the signals it cares about. All it takes is a call to sigfillset and a call to pthread_sigmask (or sigprocmask) to get temporary relief.

There are other examples where signal masks are important throughout this chapter.

4. On my version of Solaris, pthread_sigmask was defined even without the pthread library, but it didn't do anything. I had to change it to sigprocmask.

5. If you jump out of a signal handler with longjmp, what happens to the signal mask is unspecified. Use siglongjmp instead (Section 9.6).

9.1.6 `sigaction` System Call

You determine the action to be taken when a signal is delivered with the `sigaction` system call, which you call for each signal whose action you want to set:

sigaction—set signal action

```
#include <signal.h>

int sigaction(
    int signum,                     /* signal */
    const struct sigaction *act,    /* new action */
    struct sigaction *oact          /* old action */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct sigaction—structure for sigaction

```
struct sigaction {
    void (*sa_handler)(int);   /* SIG_DFL, SIG_IGN, or function pointer */
    sigset_t sa_mask;          /* additional signals to be blocked */
    int sa_flags;              /* flags */
    void (*sa_sigaction)(int, siginfo_t *, void *); /* Realtime Signal
                                                       handler */
};
```

The argument `act` points to a structure that specifies the new action for the whole process—signal actions are not kept for individual threads. If `oact` is non-`NULL`, the old action is returned to where it points. If all you want is the old action, you can set `act` to `NULL`, in which case the action isn't changed. In most of our examples we've set `oact` to `NULL`, but Section 9.7.2 has an example where it's used to save the old action so it can be restored.

You can't change the action for `SIGKILL`, which always terminates the process (not just a thread), or `SIGSTOP`, which always stops a process (not just a thread).

In the `sigaction` structure, `sa_handler` specifies the action with one of these values:

`SIG_DFL` The signal gets its default action, which depends on the signal, as described in Section 9.1.3, but is always ignore, terminate, stop, or continue. Again, these always apply to the entire process, never to just an individual thread.

`SIG_IGN` Ignore the signal, so that delivery has no effect. There's also a side-effect when the `SIGCHLD` signal is set to `SIG_IGN`, which is the same as the effect of the `SA_NOCLDWAIT` flag; see below. Confusingly, setting SIGCHLD to SIG_DFL does *not* cause this side-effect, even though the default for this signal is to ignore it.

function A pointer to a signal-handling function; the signal is said to be *caught*.

A signal-handler looks like the one in the example in Section 9.1.1:

```
static void fcn(int signum)
{
    (void)write(STDOUT_FILENO, "Got signal\n", 11);
    _exit(EXIT_FAILURE);
}
```

The function can be static or not, as long as it has the right prototype. When the signal is delivered, the function is called with the argument set to the number of that signal (e.g., SIGINT, SIGUSR1). The different signals and their macro names are described in Section 9.1.3. You can have a separate function for each signal number, one function for them all, or anything in-between. There's more about what you can do inside a signal handler in Section 9.1.7.

If the Realtime Signals option is supported, you can set the SA_SIGINFO flag (see below) and then use the sa_sigaction member for the signal handler instead of the sa_handler member, which provides a lot more information to the signal handler. This feature is discussed in Section 9.5. An implementation might use the same storage for sa_sigaction and sa_handler, so make sure you set only one of them. Don't set one and then zero the other.

As I mentioned, the signal that caused the handler to be called is blocked while the handler is executing, but you can arrange to block additional signals by specifying them in the sa_mask argument, which you set up with the signal-mask manipulation functions in Section 9.1.8.

When a thread receives a signal that is caught, the signal handler executes within that thread, and that thread's signal mask is what's temporarily modified during the execution of the signal handler. Recall that there are two ways a caught signal can be delivered to a thread: It is delivered to the process, and a thread that has it unblocked is chosen in an unspecified way, or it is delivered to a specific thread.

The following is a list of the portable flags for the sa_flags member. Note that the first two apply to the SIGCHLD signal only.

SA_NOCLDSTOP Don't generate a SIGCHLD signal when a child stops or continues.

SA_NOCLDWAIT	Don't transform a terminated child process into a zombie. Explained in Section 5.9. Explicitly setting SIGCHLD signals to SIG_IGN has the identical effect.
SA_NODEFER	Don't add the signal to the signal mask on entry to the signal handler unless it is explicitly included in the sa_mask member. This flag is only present so that the obsolete signal function (Section 9.4) can be implemented.
SA_ONSTACK	Deliver the signal on the alternate signal stack if one has been declared with sigaltstack. See Section 9.3.
SA_RESETHAND	Reset the signal's action to SIG_DFL and clear the SA_SIGINFO flag when a signal handler is entered. Ineffective for SIGILL and SIGTRAP signals. Also forces the SA_NODEFER behavior and, like that flag, is only present to allow the implementation of signal.
SA_RESTART	Don't allow the signal to interrupt a system call; see Section 9.1.4. There's an example in Section 9.7.4.
SA_SIGINFO	Use the sa_sigaction member instead of the sa_handler member; see Section 9.5.

To summarize the flags:

- SA_NOCLDSTOP and SA_NOCLDWAIT are for the SIGCHLD signal only.
- SA_NODEFER and SA_RESETHAND are compatible with an obsolete and unreliable signal mechanism that you should never use (unless you're doing Exercise 9.4).
- SA_ONSTACK is for very specialized uses.
- SA_SIGINFO is for use with the Realtime Signals option.
- SA_RESTART is occasionally useful.

To summarize the actions and their effect on threads:

- The action for a signal is always on a process-wide basis and is either catch, ignore, terminate, stop, or continue.
- Signal masks are on a per-thread basis.
- You can set catch and ignore explicitly with sigaction; you get ignore, terminate, stop, or continue if the action is SIG_DFL, but you don't get to choose which of the four it is (see Section 9.1.3).
- Ignore, terminate, stop, and continue always apply to the entire process.

- A caught signal executes a signal handler in only one thread. If the signal is delivered to the process (not targeted to a specific thread) and more than one thread has it unblocked, it is delivered to a thread chosen essentially at random.

As an example, here's a function to ignore `SIGINT` and `SIGQUIT` signals. It's called from the shell in Section 6.4:

```
static struct sigaction entry_int, entry_quit;

static bool ignore_sig(void)
{
    static bool first = true;
    struct sigaction act_ignore;

    memset(&act_ignore, 0, sizeof(act_ignore));
    act_ignore.sa_handler = SIG_IGN;
    if (first) {
        first = false;
        ec_neg1( sigaction(SIGINT, &act_ignore, &entry_int) )
        ec_neg1( sigaction(SIGQUIT, &act_ignore, &entry_quit) )
    }
    else {
        ec_neg1( sigaction(SIGINT, &act_ignore, NULL) )
        ec_neg1( sigaction(SIGQUIT, &act_ignore, NULL) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Note the static `bool` variable, `first`, to ensure that we capture the original action only the first time `sigaction` is called for each signal.

Here's a companion function to restore the actions to what they were before `ignore_sig` was called:

```
static bool entry_sig(void)
{
    ec_neg1( sigaction(SIGINT, &entry_int, NULL) )
    ec_neg1( sigaction(SIGQUIT, &entry_quit, NULL) )
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

9.1.7 Signal Handlers

After a signal handler has been installed for a signal by a call to `sigaction`, that signal handler is called when the caught signal is delivered. Unless the `SA_NODEFER` has been set, which it rarely is, the caught signal is blocked while the signal handler is executing. In addition, any signals set in the `sa_mask` member of the `sigaction` structure are also blocked. When the signal handler returns, the original mask is restored, even if the temporary mask was modified explicitly (by a call to `pthread_sigmask` or `sigprocmask`) while the signal handler was executing.

OK, so you've caught a signal. What do you do about it? Well, that depends on what kind of signal it is and why it was generated:

- The signal may have been generated by the kernel because it detected an error. Examples would be `SIGFPE` (arithmetic error) or `SIGPIPE` (write on a pipe with no reader). You probably want to display or log an error message and terminate the thread or process. Returning from the handler may be a bad idea because for some signals the state of the computation may be uncertain. And, for hardware-generated errors, like `SIGFPE`, the process may be terminated if you return, as I explained in Section 9.1.3.
- The signal may have been generated by something the user did, such as typing Ctrl-c, which normally generates a `SIGINT` signal. You may want to terminate the program after cleaning up, or you may want to stop a computation, such as a database search, and go back to the user prompt. These are only examples—whatever you do is highly application dependent.
- The signal may be part of your application's design. An example would be sending a process a `SIGUSR1` signal to indicate that a data file is ready for processing.
- A timer may have expired.

Whatever you do, there are always two things to think about:

1. What to do inside the signal handler to change the state of the application so it's known that the signal occurred.
2. Where to go from the signal handler. The choices are returning from the handler, terminating the program, doing a global jump to another part of the program, or generating another signal.

Inside the signal handler, you're restricted as to what system calls or standard functions you can call because the signal may have occurred in a place that can't safely be re-entered. In fact, the SUS (Version 3) defines only 116 so-called async-signal-safe functions, which are listed in Table 9.1.

Table 9.1 Async-Signal-Safe Functions

accept	getppid	sigdelset
access	getsockname	sigemptyset
aio_error	getsockopt	sigfillset
aio_return	getuid	sigismember
aio_suspend	kill	signal
alarm	link	sigpause
bind	listen	sigpending
cfgetispeed	lseek	sigprocmask
cfgetospeed	lstat	sigqueue
cfsetispeed	mkdir	sigset
cfsetospeed	mkfifo	sigsuspend
chdir	open	sleep
chmod	pathconf	socket
chown	pause	socketpair
clock_gettime	pipe	stat
close	poll	symlink
connect	posix_trace_event	sysconf
creat	pselect	tcdrain
dup	raise	tcflow
dup2	read	tcflush
execle	readlink	tcgetattr
execve	recv	tcgetpgrp
_exit/_Exit	recvfrom	tcsendbreak
fchmod	recvmsg	tcsetattr
fchown	rename	tcsetpgrp
fcntl	rmdir	time
fdatasync	select	timer_getoverrun
fork	sem_post	timer_gettime
fpathconf	send	timer_settime
fstat	sendmsg	times
fsync	sendto	umask
ftruncate	setgid	uname
getegid	setpgid	unlink
geteuid	setsid	utime
getgid	setsockopt	wait
getgroups	setuid	waitpid
getpeername	shutdown	write
getpgrp	sigaction	
getpid	sigaddset	

It's not generally safe to call a higher-level function, such as one in a library or even one in your own application, since you can't in general be sure what it's doing, especially after it's evolved over time. You can't even call `printf`, which is why in the example that started this chapter we used `write` instead.

There's a worse restriction: You can't safely refer to a global variable either unless it's of type `volatile sig_atomic_t`.

Technically, it is OK to call an unsafe (nonasync-signal-safe) function or refer to unsafe storage from within a handler if you know that an unsafe function was not interrupted, but since you would know that only under unusual circumstances, it's unwise to program that way. Better to restrict yourself to the list in the table and to modifying globals of type `volatile sig_atomic_t`.

This whole situation seems very risky and it is. Here are some recommendations to keep yourself sane and your programs reliable:

- Signal handlers that display an error (using `write` or other safe functions) and then `_exit` are OK.
- Setting a flag of type `volatile sig_atomic_t` and returning is OK if the `SA_RESTART` flag has been set for the signal.
- Avoiding signal handlers altogether is the best choice. Instead, use threads and `sigwait` (Section 9.2).

Actually, if you've read this far, you've probably already decided to adopt the last recommendation! But, even without signal handlers, signals are still useful because there are two entirely safe ways to handle them without a signal handler: With `sigsuspend` (Section 9.2.3) and with an even better choice, `sigwait` (Section 9.2.2).

To close out this section, here's an example that shows a completely legal signal handler that records what signal arrives and then returns. Even though `SA_RESTART` is specified, `sleep` is still interrupted because it's not affected by the flag.

```
static volatile sig_atomic_t gotsig = -1;

static void handler(int signum)
{
    gotsig = signum;
}
```

```
int main(void)
{
    struct sigaction act;
    time_t start, stop;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    act.sa_flags = SA_RESTART;
    ec_neg1( sigaction(SIGINT, &act, NULL) )
    printf("Type Ctrl-c in the next 10 secs.\n");
    ec_neg1( start = time(NULL) )
    sleep(20);
    ec_neg1( stop = time(NULL) )
    printf("Slept for %ld secs\n", (long)(stop - start));
    if (gotsig > 0)
        printf("Got signal number %ld\n", (long)gotsig);
    else
        printf("Did not get signal\n");
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output I got; I typed Ctrl-c after I saw the first line:

```
Type Ctrl-c in the next 10 secs.
Slept for 4 secs
Got signal number 2
```

9.1.8 Minimal Defensive Signal Handling

Even if you decide to avoid signal handlers, you usually still need at least a minimal level of signal handling to prevent your application from being terminated accidentally by the user. Also, it's not very professional to just have your program terminated abnormally if there's a bug that causes, say, invalid memory to be referenced (a "segmentation violation"). It's better to catch the signal, log the problem, and inform the user with something better than the simple "Program aborted – segmentation violation," or whatever the shell decides to display.

So most applications should do at least this much:

- Block all signals as soon as your program begins, like this:

```
sigset_t set;

ec_neg1( sigfillset(&set) )
ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
```

(Use `pthread_sigmask` if you have multiple threads.)

- Set all the keyboard-generated signals you don't want to catch to be ignored, such as `SIGINT`.
- Catch `SIGTERM` and arrange to cleanup and terminate when it arrives, as it's the standard way that system administrators shut down processes.
- Catch all the error-generated signals and arrange to display and/or log the error and terminate when one arrives.
- Ignore `SIGPIPE` so that `write` will return an error if it is writing to an empty pipe. This is more convenient than receiving a signal.
- Unblock all signals with calls to `sigemptyset` and `sigprocmask` (or `pthread_sigmask`).

Here's a function you can call at the start of your application to minimally handle signals:

```
static bool handle_signals(void)
{
    sigset_t set;
    struct sigaction act;

    ec_neg1( sigfillset(&set) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    memset(&act, 0, sizeof(act));
    ec_neg1( sigfillset(&act.sa_mask) )
    act.sa_handler = SIG_IGN;
    ec_neg1( sigaction(SIGHUP, &act, NULL) )
    ec_neg1( sigaction(SIGINT, &act, NULL) )
    ec_neg1( sigaction(SIGQUIT, &act, NULL) )
    ec_neg1( sigaction(SIGPIPE, &act, NULL) )
    act.sa_handler = handler;
    ec_neg1( sigaction(SIGTERM, &act, NULL) )
    ec_neg1( sigaction(SIGBUS, &act, NULL) )
    ec_neg1( sigaction(SIGFPE, &act, NULL) )
    ec_neg1( sigaction(SIGILL, &act, NULL) )
    ec_neg1( sigaction(SIGSEGV, &act, NULL) )
    ec_neg1( sigaction(SIGSYS, &act, NULL) )
    ec_neg1( sigaction(SIGXCPU, &act, NULL) )
    ec_neg1( sigaction(SIGXFSZ, &act, NULL) )
    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's the actual handler and two supporting functions it calls:

```
static void handler(int signum)
{
    int i;
    struct {
        int signum;
        char *msg;
    } sigmsg[] = {
        { SIGTERM, "Termination signal" },
        { SIGBUS, "Access to undefined portion of a memory object" },
        { SIGFPE, "Erroneous arithmetic operation" },
        { SIGILL, "Illegal instruction" },
        { SIGSEGV, "Invalid memory reference" },
        { SIGSYS, "Bad system call" },
        { SIGXCPU, "CPU-time limit exceeded" },
        { SIGXFSZ, "File-size limit exceeded" },
        { 0, NULL}
    };

    clean_up();
    for (i = 0; sigmsg[i].signum > 0; i++)
        if (sigmsg[i].signum == signum) {
            (void)write(STDERR_FILENO, sigmsg[i].msg,
              strlen_safe(sigmsg[i].msg));
            (void)write(STDERR_FILENO, "\n", 1);
            break;
        }
    _exit(EXIT_FAILURE);
}

static void clean_up(void)
{
    /*
        Clean-up code goes here --
        must be async-signal-safe.
    */
}

static size_t strlen_safe(const char *s)
{
    size_t n = 0;
    while (*s++ != '\0')
        n++;
    return n;
}
```

The idea is that you replace the guts of `clean_up` with code appropriate for your application. I coded my own version of `strlen` because, silly as it sounds, the standard `strlen` isn't in the list of async-signal-safe functions. For the same reason I used `write` in the handler instead of `fprintf`. Note also the use of `_exit` instead of `exit`—as explained in Section 5.7, the underscored version skips calling `atexit` and flushing of standard C I/O buffers and is the only way to do an async-signal-safe normal exit.

9.1.9 Generating a Signal Synthetically

As I said in Section 9.1.3, each signal has a natural cause and can also be generated explicitly by a call to `kill`, `killpg`, `pthread_kill`, `abort`, `raise`, or `sigqueue` (Section 9.5.4):

kill—generate signal for processes

```
#include <signal.h>

int kill(
    pid_t pid,             /* process ID or other specification */
    int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

killpg—generate signal for process group

```
#include <signal.h>

int killpg(
    pid_t pgrp,            /* process-group ID */
    int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

pthread_kill—generate signal for thread

```
#include <signal.h>

int pthread_kill(
    pthread_t thread_id,   /* thread ID */
    int signum             /* signal */
);
/* Returns 0 on success, error number on failure (errno not set) */
```

abort—generate SIGABRT

```
#include <stdlib.h>

void abort(void);
/* Does not return */
```

raise—generate signal for thread

```
#include <signal.h>

int raise(
    int signum      /* signal */
);
/* Returns 0 on success or non-zero on error (sets errno) */
```

The misnamed `kill` system call sends any signal, not just `SIGKILL`, to one or more processes for which it has permission to send signals. It has permission if it's run by the superuser or if the real or effective user ID of the sending process matches the real or saved set-user-ID of the receiving process. Which processes the signal gets delivered to depends on the `pid` argument:

- \>0 the process whose process ID is `pid`
- 0 the process group whose process-group ID is the same as that of the sending process
- <–1 the process group whose process-group ID is the same as the absolute value of `pid`
- –1 all processes for which the sender has permission, except for an implementation defined set of system processes

If `signum` is 0, `kill` just tests its `pid` argument for validity. It's a way to tell whether a process or process-group is alive. If the sending process doesn't have permission to send a signal, the call will fail, but the `errno` value will indicate whether the process or process-group is alive: It will be `ESRCH` if `pid` doesn't exist, and `EPERM` if it does but the sender has no permission.

`killpg`, a totally unnecessary system call, generates a signal only for the process group whose process-group ID is `pgrp`, so it's identical to:

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

`pthread_kill` is like `kill`, except it sends the signal to only thread `thread_id`, which must be in the same process as the sending thread. It doesn't have the broadcast ability that `kill` has for processes. Be careful with `pthread_kill`—remember that signals that terminate, stop, or continue always affect the whole process. So, `pthread_kill` works the way you want it to only for caught signals. If you execute

```
pthread_kill(tid, SIGKILL);
```

the process will be terminated, not just thread `tid`. (You use `pthread_cancel` to terminate just one thread.)

`abort` is almost like `kill` with an argument of `SIGABRT`, but unless that signal is caught and the signal handler does not return (e.g., calls `siglongjmp` or `_exit`), the process is terminated anyway as if `SIGABRT` had its default behavior; `abort` *never* returns. For example, if `SIGABRT` is set to `SIG_IGN` by `sigaction`, `abort` terminates a process, but

```
kill(getpid(), SIGABRT)
```

has no effect.

`raise` is actually a Standard C function. It sends a signal to the thread that executes it. That is, it's the same as:

```
pthread_kill(pthread_self(), signum);
```

However, `raise` is always present even if the POSIX Threads option is not supported, in which case it's the same as:

```
kill(getpid(), signum);
```

9.1.10 Effect of `fork`, `pthread_create`, and `exec` on Signals

There are three properties of a thread that are set for a new process, thread, or program by a `fork`, `pthread_create`, or `exec`:

1. **Signal actions:** After a `fork`, the child inherits all signal actions. After an `exec`, signals set to `SIG_DFL` remain that way; signals set to `SIG_IGN` remain that way, except for `SIGCHLD`, which may be set to `SIG_IGN` or `SIG_DFL`, as the implementation chooses; caught signals are set to `SIG_DFL`. As all actions are process-wide, `pthread_create` has no effect.
2. **Signal mask:** Inherited from the `fork`ing thread after a `fork`; stays the same as the `exec`ing thread after an `exec`; copied to the new thread from the creating thread after a `pthread_create`.
3. **Pending signals:** Cleared after a `fork`; same as the `exec`ing thread after an `exec`; cleared after a `pthread_create`.

The simple way to remember these nine rules (three properties times three system calls) is this: The bias is to copy or inherit properties unchanged, so there are only three exceptions to remember, the first two of which make perfect sense:

- A caught signal has to be changed to `SIG_DFL` if the signal handler disappears, which it does on an `exec`.
- Pending signals are per-process or per-thread, so they're cleared when there's a new process or new thread.
- The standards makers were more concerned with not breaking existing implementations than in forcing portability, so they waffled on `SIGCHLD`.

9.2 Waiting for a Signal

This section describes system calls that allow a process to wait for the delivery of a signal.

9.2.1 `pause` System Call

We've encountered lots of system calls that block waiting for an event before they complete some activity. For example, when reading a terminal, `read` normally waits for a full line to be typed. `pause` is pure wait: It doesn't do anything, and it's not waiting for anything in particular.

pause—wait for signal

```
#include <unistd.h>

int pause(void);
/* Returns -1 on error (sets errno) */
```

Since a delivered signal interrupts most system calls that are blocked, we might as well say that `pause` waits for a caught signal. If the signal-catching function returns, `pause` returns with `errno` set to `EINTR`, but since that's the only way `pause` ever returns there's no point testing for it.

Usually, you'll use a more sophisticated call like `sigwait` instead of `pause`.

9.2.2 `sigwait` System Call

Unlike `pause`, `sigwait` lets you choose what you want to wait for. You don't need a signal handler, and when it returns you're told what signal arrived:

sigwait—wait for signal

```
#include <signal.h>

int sigwait(
     const sigset_t *set,       /* signals to wait for */
     int *signum                /* signal that was accepted */
);
/* Returns 0 on success or error number on error */
```

The argument `set` is a set of signals (see Section 9.1.5) that `sigwait` is to wait for. When one becomes pending, its number is returned through the `signum` argument and `sigwait` returns. If one or more signals are already pending when `sigwait` is called, one is chosen in an undefined way and immediately returned. The technical term for a signal returned by `sigwait` is "accepted"; it is not "delivered."

When you use `sigwait`, you want a signal to stay pending until `sigwait` returns it; you don't want a signal to ever be delivered. So, you block the signals that `sigwait` is to wait for and leave them blocked. (The lifecycle of a signal—generated-to-pending-to-delivered—was described in Section 9.1.2.)

If more than one thread is in `sigwait` waiting for the same signal sent to a process, only one thread gets it, and the choice is made in an undefined way. If a signal is sent to a specific thread, only that thread's `sigwait` (if it has one) can return it.

Typically, you use `sigwait` for one of two purposes:

- When the thread can't proceed until some event occurs that's associated with a signal. For example, one thread might send `SIGUSR1` to another thread when a message has arrived. Usually, though, it's better to use a condition variable (Section 5.17.4) for this purpose. There's an example later in this section.
- When one thread is designated to handle signals. That is, instead of signal handling function, a waiting thread is used. All threads have the signals to be waited for blocked, and one thread executes a `sigwait`. But this only works for signals sent to a process—if a signal is sent to a thread, only that thread's `sigwait` can return it.

For signals sent to a process, it's much better to use `sigwait` instead of a signal handler because none of the restrictions for signal handlers apply. When `sigwait` returns, you're free to call any system call or function or do anything else that you can do in a thread.

In Section 9.1.8 we showed a function, `handle_signals`, that arranges to catch all the detected-error signals (e.g., `SIGFPE`, `SIGSEGV`) so it can call a cleanup function and display a nice message before exiting. Let's recode this function to use `sigwait` in a thread instead of a signal handler:

```
static bool handle_signals(void) /* do not use -- see below */
{
    sigset_t *set;
    struct sigaction act;
    pthread_t tid;

    ec_null( set = malloc(sizeof(*set)) )
    ec_neg1( sigfillset(set) )
    ec_rv( pthread_sigmask(SIG_SETMASK, set, NULL) )
    memset(&act, 0, sizeof(act));
    act.sa_handler = SIG_IGN;
    ec_neg1( sigaction(SIGHUP, &act, NULL) )
    ec_neg1( sigaction(SIGINT, &act, NULL) )
    ec_neg1( sigaction(SIGQUIT, &act, NULL) )
    ec_neg1( sigaction(SIGPIPE, &act, NULL) )
    ec_neg1( sigemptyset(set) )
    ec_neg1( sigaddset(set, SIGTERM) )
    ec_neg1( sigaddset(set, SIGBUS) )
    ec_neg1( sigaddset(set, SIGFPE) )
    ec_neg1( sigaddset(set, SIGILL) )
    ec_neg1( sigaddset(set, SIGSEGV) )
    ec_neg1( sigaddset(set, SIGSYS) )
    ec_neg1( sigaddset(set, SIGXCPU) )
    ec_neg1( sigaddset(set, SIGXFSZ) )
    ec_rv( pthread_sigmask(SIG_SETMASK, set, NULL) )
    ec_rv( pthread_create(&tid, NULL, sig_thread, set) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void *sig_thread(void *arg)
{
    int signum;
    int i;
    struct {
        int signum;
        char *msg;
    } sigmsg[] = {
        { SIGTERM, "Termination signal" },
        { SIGBUS, "Access to undefined portion of a memory object" },
```

```
        { SIGFPE, "Erroneous arithmetic operation" },
        { SIGILL, "Illegal instruction" },
        { SIGSEGV, "Invalid memory reference" },
        { SIGSYS, "Bad system call" },
        { SIGXCPU, "CPU-time limit exceeded" },
        { SIGXFSZ, "File-size limit exceeded" },
        { 0, NULL}
    };

    while (true) {
        ec_rv( sigwait((sigset_t *)arg, &signum) )
        clean_up();
        for (i = 0; sigmsg[i].signum > 0; i++)
            if (sigmsg[i].signum == signum) {
                fprintf(stderr, "%s\n", sigmsg[i].msg);
                break;
            }
        _exit(EXIT_FAILURE);
    }
    return (void *)true; /* never get here */

EC_CLEANUP_BGN
    EC_FLUSH("sig_thread")
    return (void *)false;
EC_CLEANUP_END
}

static void clean_up(void)
{
    /*
        Clean-up code goes here --
        need not be async-signal-safe.
    */
}
```

The advantage of this version over the one in Section 9.1.8 is that, when a signal is returned by `sigwait`, we're in a thread, not a signal handler, and we're free to use any system calls or functions we like—we're not restricted to the async-signal-safe list. Note that the comment in the `clean_up` function has been changed accordingly.

The disadvantage of this version is that it doesn't work! There are two reasons why, both serious:

- If some other thread gets a `SIGSYS`, for example, that signal will be sent to that thread, not to the process, and the `sigwait` in the `sig_thread` function won't return with it. Indeed, since that signal is blocked in all threads, it will just stay pending forever. So, the original version, using signal handlers, is the one you should use for the detected-error signals.

- If one of the hardware-detected signals, SIGBUS, SIGFPE, SIGILL, and SIGSEGV, occurs naturally while blocked, the result is undefined (see Section 9.1.3). Most likely the process will be immediately terminated, and sigwait will never get a chance to return it. This wasn't a problem with the signal-handler version because, after the initial setup, those four signals were unblocked.

This is not to say that sigwait isn't useful. Most signals, including all but the detected-error group in Section 9.1.3, are sent to the process when they are generated naturally (i.e., not by pthread_kill), so a thread waiting in sigwait works perfectly well and is a much better choice than a signal handler.

9.2.3 **sigsuspend** System Call

sigsuspend is an older, nonmultithreading system call that also waits for a signal. Before we get to its details, let's explore the problem it solves. Back in Chapter 8 when we kept running examples that forked to create processes that connected to the parent's socket, it was important for the parent to get the socket bound before the child connected, and we used the crude technique of having the children sleep for a few seconds to give the parent a chance to get ahead. Not only is sleeping unreliable, because there's no guarantee that a few seconds is enough, but it's also inefficient because a few seconds may be much too long. To recap, here's the same problem in a simpler example:

```
void try1(void)
{
    if (fork() == 0) {
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    return;
}
```

This function displayed

```
child
parent
```

but we want the parent to execute first and then tell the child when to proceed. Our first attempt to synchronize them has the parent sending a `SIGUSR1` signal to the child, who catches it and sets a variable:

```
static volatile sig_atomic_t got_sig;
static void handler(int signum)
{
    if (signum == SIGUSR1)
        got_sig = 1;
}

void try2(void)
{
    pid_t pid;

    got_sig = 0;
    ec_neg1( pid = fork() )
    if (pid == 0) {
        struct sigaction act;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_neg1( sigaction(SIGUSR1, &act, NULL) )
        while (got_sig == 0)
            if (pause() == -1 && errno != EINTR)
                EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try2")
EC_CLEANUP_END
}
```

The handler is safe—it just sets a variable of the approved type. The child tests the variable in a loop, pausing to await the arrival of a signal. (`pause`, which we'll get to in Section 9.2.1, blocks until a signal arrives.) Now the sequence is what we want:

```
parent
child
```

But there are two problems:

- If `SIGUSR1` is delivered to the child before it has a chance to install the handler, it will terminate the child process. A potential solution is to install the handler before the `fork`, so the child will inherit it, but that only works in a parent-child situation. We'd like a solution that works for arbitrary processes that need to synchronize.
- If `SIGUSR1` is delivered between the test in the `while` statement and the call to `pause`, `pause` will wait forever since the signal that's supposed to wake it up arrived before it even got to sleep.

We can try to fix things by blocking the `SIGUSR1` signal until we're ready for it:

```
void try3(void)
{
    sigset_t set;
    pid_t pid;

    got_sig = 0;
    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGUSR1) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_neg1( pid = fork() )
    if (pid == 0) {
        struct sigaction act;
        sigset_t suspendset;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_neg1( sigaction(SIGUSR1, &act, NULL) )
        ec_neg1( sigfillset(&suspendset) )
        ec_neg1( sigdelset(&suspendset, SIGUSR1) )
        ec_neg1( sigprocmask(SIG_SETMASK, &suspendset, NULL) )
        while (got_sig == 0)
            if (pause() == -1 && errno != EINTR)
                EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try3")
EC_CLEANUP_END
}
```

This totally fixes problem #1. Because it's blocked, SIGUSR1 can't arrive until after the handler is installed. But problem #2 is still with us, and there's nothing we can do to fix it. Although the gap is small, it's still possible for the signal to arrive between the test and the pause.

What we want is a way to keep the signal blocked until the pause begins. Or, to say it another way, we want unblocking and pausing to be atomic. That's exactly what sigsuspend does:

sigsuspend—change signal mask and wait for signal

```
#include <signal.h>

int sigsuspend(
    const sigset_t *sigmask       /* temporary signal mask */
);
/* Returns -1 on error, always (sets errno) */
```

sigsuspend temporarily replaces the thread's signal mask with sigmask and then waits until an unblocked signal is delivered whose action is termination or being caught. If it's termination, the process (not just the thread, remember) is terminated, and sigsuspend doesn't return. If it's caught and the signal handler returns, the previous signal mask is restored and sigsuspend returns with an error. Usually, the error is only EINTR; that is, a return of –1 with an errno or EINTR is normal for an interrupted system call (same as pause, as shown in the examples above).

In essentially all cases, the mask passed to sigsuspend has the effect of unblocking one or more signals that were blocked prior to the call, although that isn't actually a requirement. It's just that anything else doesn't make much sense.

OK, perfect. We're now set to fix our synchronizing problem for good by simply replacing pause with sigsuspend. We don't need the while loop anymore because unblocking the signal and suspending are now atomic.

```
void try4(void)
{
    sigset_t set;
    pid_t pid;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGUSR1) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_neg1( pid = fork() )
```

```
    if (pid == 0) {
        struct sigaction act;
        sigset_t suspendset;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_neg1( sigaction(SIGUSR1, &act, NULL) )
        ec_neg1( sigfillset(&suspendset) )
        ec_neg1( sigdelset(&suspendset, SIGUSR1) )
        if (sigsuspend(&suspendset) == -1 && errno != EINTR)
            EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try4")
EC_CLEANUP_END
}
```

We could still use the same handler function, which sets the `got_sig` variable; however, we don't need the variable anymore, and an empty handler will work fine:

```
static void handler(int signum)
{
}
```

Also, the test of the return code from `sigsuspend` against –1 isn't really needed since it always returns –1. But doing that is weird and would confuse readers who don't happen to know or remember all the details of `sigsuspend`.

Note that we are blocking `SIGUSR1` before the `fork` so that the child inherits the signal mask. In a situation where the two processes to be synchronized are unrelated, the process that calls `sigsuspend` simply does its own blocking. Actually, this is just a special case of the general recommendation to start every application with all signals blocked, as I said in Section 9.1.8.

Note that, unlike `sigwait`, you almost always use `sigsuspend` along with a signal handler, but usually the handler doesn't do anything. It's only present so that the delivered signal will interrupt `sigsuspend`.

Another big difference between `sigsuspend` and `sigwait` is that with `sigwait` the signal you're waiting for stays blocked. In fact, it is never delivered—rather, `sigwait` accepts it: removes it from the collection of pending signals and returns it. So the race condition between unblocking the signal and waiting for it to be delivered, which `sigsuspend` eliminates, doesn't exist when you're using `sigwait` because you never unblock the signal. Our synchronization code thus becomes even simpler:

```
void try5(void)
{
    sigset_t set;
    pid_t pid;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGUSR1) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_neg1( pid = fork() )
    if (pid == 0) {
        int signum;

        ec_rv( sigwait(&set, &signum) )
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try5")
EC_CLEANUP_END
}
```

We should mention that synchronizing the parent and child can also be done with a pipe, skipping the complexities of signals altogether:

```
void try6(void)
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    ec_neg1( pid = fork() )
    if (pid == 0) {
        char c;

        ec_neg1( close(pfd[1]) )
        ec_neg1( read(pfd[0], &c, 1) )
        ec_neg1( close(pfd[0]) )
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
```

```
    printf("parent\n");
    ec_neg1( close(pfd[0]) )
    ec_neg1( close(pfd[1]) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try6")
EC_CLEANUP_END
}
```

Here the child blocks in `read` until the parent closes the writing end of the pipe, resulting in the child getting a zero return.

9.3 Miscellaneous Signal System Calls

You can find out whether a signal is pending with `sigpending`, which returns a set of the pending signals:

sigpending—examine pending signals

```
#include <signal.h>

int sigpending(
     sigset_t *set             /* returned set of pending signals */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

You can test which signals are pending by using `sigismember` (Section 9.1.5) on the returned set. Of course, it may not still be pending by the time you test it unless it's blocked.

Recall from Section 9.1.6 that the `SA_ONSTACK` flag arranges for a signal's handler to execute on an alternate stack. You manage the stack with `sigaltstack`:

sigaltstack—set and get alternate stack context

```
#include <signal.h>

int sigaltstack(
     const stack_t *stack,     /* new stack */
     stack_t *ostack           /* old stack */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

See your system's documentation or the SUS for details of this system call.

Also, recall that the `SA_RESTART` flag prevents a signal from interrupting a function. You can turn the flag on or off, without calling `sigaction`, with `siginterrupt`:

siginterrupt—set or clear SA_RESTART flag

```
#include <signal.h>

int siginterrupt(
     int signum,            /* signal */
     int flag               /* non-zero to clear, zero to set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9.4 Deprecated Signal System Calls

The system calls in this section are standardized, but they don't add any functionality to the system calls already presented and aren't worth spending your time on, except for your time on Exercises 9.4 and 9.5.

The classic way to set the action for a signal was with the `signal` system call:

signal—set signal action

```
#include <signal.h>

void (*signal(
     int signum,            /* signal */
     void (*act)(int)       /* new action */
))(int);
/* Returns old action or SIG_ERR on error (sets errno) */
```

The strange declaration means that `signal` returns an action that could be a pointer to a void function taking an integer argument (the signal number).

There are two problems when you catch a signal with `signal`:

- Upon delivery, the action is set to its default. You need to call `signal` again if you still want to catch it.
- The delivered signal isn't blocked, so a second arrival can terminate the process.

The way to get around these problems is to use `sigaction` (Section 9.1.6) and forget about `signal`.

There is a group of five system calls that provide for simplified signal handling, but you should avoid using them, as they don't do anything that the primary functions (e.g., `sigaction`) don't do better. I'll briefly describe them anyway.

You can set a `sigaction`-style action for a signal without using a structure with `sigset`:

sigset—set signal action

```
#include <signal.h>

void (*sigset(
    int signum,            /* signal */
    void (*act)(int)       /* new action */
))(int);
/* Returns old action or SIG_ERR on error (sets errno) */
```

`sigset` is as simple to call as `signal`, but it has the behavior of `sigaction`, in that a delivered signal is masked while the handler is executing, and the action is not changed. In addition, it takes a new action, `SIG_HOLD`, which just adds the signal to the signal mask. Unlike `sigaction`, if you call `sigset` without the `SIG_HOLD` action, it also unblocks it (removes it from the signal mask) as a byproduct.

The chief reason for avoiding `sigset` and the other related functions that follow is that it's hard enough mastering what `sigaction` does without also trying to learn a somewhat different combination of features. *Less is more!* Another reason for avoiding them is that they're not defined in a multithreading environment. Imagine the problems if you've used them in a single-threading program and then multithreading is added later!

To unblock a signal directly (remove it from the signal mask), you can call `sigrelse`:

sigrelse—unblock signal

```
#include <signal.h>

int sigrelse(
    int signum,            /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

There's a simplified version of `sigsuspend`, `sigpause`, which removes a signal from the signal mask, pauses until a signal arrives (any unblocked signal), and then restores the mask:

sigpause—change signal mask and wait for signal

```
#include <signal.h>

int sigpause(
    int signum,          /* signal */
);
/* Returns -1 on error, always (sets errno) */
```

Finally, here are two redundant system calls, as they do exactly what `sigset` does with the `SIG_HOLD` and `SIG_IGN` actions:

sighold—block signal

```
#include <signal.h>

int sighold(
    int signum,          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigignore—ignore signal

```
#include <signal.h>

int sigignore(
    int signum,          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9.5 Realtime Signals Extension (RTS)

The so-called POSIX.4 (real-time) standard includes a new signal mechanism, RTS, that improves on the signal-related system calls described earlier in this chapter by:

- Increasing the number of signals for application use beyond just `SIGUSR1` and `SIGUSR2`
- Allowing for queuing of signals, so that a signal that's generated when a signal of the same type is pending isn't lost
- Specifying the order of delivery of signals
- Including additional information with a signal, such as who it came from and why, and possibly some application data

The new features add on to the traditional features so you can still use the classic 28 signals (Section 9.1.3), with the same handlers and other actions, and with the same synthesizing calls, such as `kill`. The standard doesn't say that you can use the new RTS features (e.g., queuing, passing a value) with the old signals, although on many systems you can, so avoid doing that if you want to be portable.

There are several feature-test macros (Section 1.5.4) that indicate whether these features are available. The principal one is `_POSIX_REALTIME_SIGNALS`.

The subsections in this section discuss most of the RTS features in detail. At various times, it will be convenient to refer to a group of functions that use the RTS signal mechanism to send signals. They're spelled out in the next section as those associated with codes `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCIO`, and `SI_MESGQ`. I call the group the *RTS-generation* functions (my name, not one used elsewhere).

There are a few more RTS features described in Sections 9.7.5 and 9.7.6.

9.5.1 RTS Signal Handlers

You may want to review the discussion of `sigaction` in Section 9.1.6 before continuing.

If you set the `SA_SIGINFO` flag in the structure passed to `sigaction`, you use the `sa_sigaction` member instead of the `sa_handler` member to hold the pointer to the signal-handling function.[6] Instead of just a signal-number argument, it has this more elaborate prototype:

```
void rts_handler(int signum, siginfo_t *info, void *context);
```

The `info` argument points to information about the signal in a `siginfo_t` structure, which we first used with `waitid` in Section 5.8:

siginfo_t—structure for sigaction

```
typedef struct {
    int si_signo;            /* signal number */
    int si_errno;            /* errno value associated with signal */
    int si_code;             /* signal code (see below) */
    pid_t si_pid;            /* sending process ID */
    uid_t si_uid;            /* real user ID of sending process */
    void *si_addr;           /* address of faulting instruction */
    int si_status;           /* exit value or signal */
    long si_band;            /* band event for SIGPOLL */
    union sigval si_value;   /* signal value */
} siginfo_t;
```

6. The SUS is unclear about whether you can use `SIG_IGN` or `SIG_DFL` with the `sa_sigaction` member; to be safe, use them only with `sa_handler`.

union sigval—union for signal values[7]

```
union sigval {
     int sival_int;         /* integer signal value */
     void *sival_ptr;       /* pointer signal value */
};
```

Member `si_signo` is just a repeat of the `signum` argument to the handler.

The use of `si_errno` is up to the implementation; it may contain an error code that indicates the cause of the signal.

Member `si_code` indicates the reason for the signal. Usually, if it's 0 or negative, the signal is from a process; the process ID is in `si_pid` and the real user ID is in `si_uid`. The `si_code` member is one of:

`SI_USER` sent by `kill` or, at the discretion of the implementation, by one of the other system calls for synthesizing a signal (e.g., `raise`)

`SI_QUEUE` sent by `sigqueue` (Section 9.5.4)

`SI_TIMER` expiration of timer set by `timer_settime` (Section 9.7.6)

`SI_ASYNCIO` completion of asynchronous I/O (Section 3.9)

`SI_MESGQ` arrival of message (Section 7.7)

For the last four, the RTS-generation functions, there's also a value that can be sent with the signal that's accessible through the `si_value` member.

If `si_code` is positive, the signal came from the kernel and the code depends on the signal. For example, if the signal is `SIGFPE`, the code is `FPE_INTDIV` for integer divide by zero, `FPE_INTOVF` for integer overflow, and `FPE_FLTDIV` for floating-point divide by zero. For the full list of the `SIGFPE` codes and the codes for other signals, see [SUS2002] or your system's documentation. Also, Section 5.8 lists the codes for `SIGCHLD`.

How the other members are used depends on the signal. Those for `SIGCHLD` were described in Section 5.8. For `SIGILL` and `SIGSEGV`, the `si_addr` member gives the actual machine address that caused the problem. For `SIGPOLL` (used with STREAMS), member `si_band` indicates the band event.

7. FreeBSD (and perhaps other systems) defines the members as `sigval_int` and `sigval_ptr`, but it doesn't claim to support RTS. Perhaps that's OK.

Here's a program that displays some of the additional information that's available in a signal handler when you set the SA_SIGINFO flag:

```
int main(void)
{
    struct sigaction act;
    union sigval val;

    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = handler;
    ec_neg1( sigaction(SIGUSR1, &act, NULL) )
    ec_neg1( sigaction(SIGRTMIN, &act, NULL) )
    ec_neg1( kill(getpid(), SIGUSR1) )
    val.sival_int = 1234;
    ec_neg1( sigqueue(getpid(), SIGRTMIN, val) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void handler(int signum, siginfo_t *info, void *context)
{
    printf("signal number: %d\n", info->si_signo);
    printf("sending process ID: %ld\n", (long)info->si_pid);
    printf("real user ID of sending process: %ld\n", (long)info->si_uid);
    switch (info->si_code) {
    case SI_USER:
        printf("Signal from user\n");
        break;
    case SI_QUEUE:
        printf("Signal from sigqueue; value = %d\n",
          info->si_value.sival_int);
        break;
    case SI_TIMER:
        printf("Signal from timer expiration; value = %d\n",
          info->si_value.sival_int);
        break;
    case SI_ASYNCIO:
        printf("Signal from asynchronous I/O completion; value = %d\n",
          info->si_value.sival_int);
        break;
    case SI_MESGQ:
        printf("Signal from message arrival; value = %d\n",
          info->si_value.sival_int);
        break;
```

```
    default:
        printf("Other signal\n");
    }
}
```

Compare this code to, say, the first example in this chapter and you'll see the differences in setting up for the call to `sigaction`: the `SA_SIGINFO` flag is set, and the member for the handler is `sa_sigaction` instead of `sa_handler`. I'll introduce the signal `SIGRTMIN` in the next section; for now just pretend it's `SIGUSR2` if that helps you understand the code.

It's legal for us to call `printf` in the handler, even though that function is not async-signal-safe, because the signal was generated by a system call that is async-signal-safe: `kill` or `sigqueue` (which we'll get to shortly, in Section 9.5.4).

Here's the output I got:

```
signal number: 10
sending process ID: 29501
real user ID of sending process: 500
Signal from user
signal number: 32
sending process ID: 29501
real user ID of sending process: 500
Signal from sigqueue; value = 1234
```

The third argument to the signal handler, `context`, is a pointer to an object of type `ucontext_t` that describes the receiving process's context at the time it was interrupted. It's a pointer to a `void` rather than a `ucontext_t` because at the time this prototype was introduced the new type wasn't standardized. This pointer is not always implemented and rarely is used. For more information, see [SUS2002] or your system's documentation.

9.5.2 RTS Signals

RTS introduced a group of new signals whose numbers range from `SIGRTMIN` to `SIGRTMAX`, with at least `RTSIG_MAX` signals available. You can get the actual number at runtime with `sysconf` (Section 1.5.5), but it's always at least 8. (That's in fact what it was on my version of Solaris; on Linux it was 32.)

For the RTS-generation functions (defined at the start of Section 9.5), you get to specify what signal you want, and you should choose one of the RTS signals; whether you can use one of the classic signals is implementation-dependent.

There aren't any symbols for the RTS signals other than for the first and last, so you refer to them as SIGRTMIN, SIGRTMIN + 1, and so on, up to SIGRTMAX. Of course, you'll want to define your own macros, like this:

```
#define DB_IO_DONE SIGRTMIN + 4
```

Alas, the problem of two libraries both using the same signals wasn't solved by the POSIX.4 group, so be careful.

SIGRTMIN and SIGRTMAX aren't necessarily constant integer expressions, so you can't portably use them as case labels or in preprocessor expressions (e.g., in #ifs).

If more than one RTS signal (SIGRTMIN through SIGRTMAX) is pending, the lowest-numbered signal is delivered first. So, you can think of them as being in priority order, and you may want to take that into account as you assign them to different application purposes. The default action for all RTS signals is termination.

9.5.3 Queued Signals

Normally, you can't count on what happens when a signal is generated while a signal of the same type is pending. If the implementation isn't equipped to queue signals, keeping only a flag for each signal, say, then it will just forget about the second signal. You must never use signals for counting purposes—to keep track of the number of times SIGUSR1 was sent, for example.

With RTS, however, an RTS signal (SIGRTMIN through SIGRTMAX) sent by an RTS-generation function (defined at the start of Section 9.5) is queued if the SA_SIGINFO flag was set for that signal by sigaction. Whether queuing also works for classic signals is implementation dependent, so don't count on it. Many systems queue the RTS signals even if you don't set the SA_SIGINFO flag, which is perfectly legal, but you should set the flag anyway to be portable.

If you want queuing but don't want a signal handler, because you're going to use, say, sigwait (Section 9.2.2), it would seem that you could set the sa_sigaction member of the sigaction structure to SIG_DFL, but the SUS isn't clear on this point. To be safe, code an empty handler and set the action to point to it.

The maximum queue length for a process is at least 32; you can find out the actual number with sysconf (Section 1.5.5).

9.5.4 `sigqueue`

sigqueue—generate signal for process

```
#include <signal.h>

int sigqueue(
    pid_t pid,                  /* process ID */
    int signum,                 /* signal */
    const union sigval value    /* value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

We've already seen `sigqueue` in use, in the example in Section 9.5.1. The first two arguments are like those for `kill`, except `pid` can only be a process ID—the broadcast capabilities of `kill` (e.g., sending to all processes) aren't there. In addition, you can pass a value that the signal handler can receive if the `SA_SIGINFO` flag was set and it uses the three-argument form described in Section 9.5.1. You decide whether you want to use the `int` or pointer members of the union; the handler has to know which to use because no information about which you used is passed along.

If you're sending the signal to another process, you probably can't use a pointer because it won't mean anything to the other process. The only way it would be valid would be if the two processes shared memory that was located at the same address and the pointer pointed into that shared memory.

As I said in the previous section, if the `SA_SIGINFO` flag was set and there's already a `signum` signal pending for the receiving process, the new signal is queued.

You can count on queuing and passing a value only for RTS Signals (Section 9.5.2).

You can only queue so many signals, as explained in the previous section. If the new signal can't be queued, `sigqueue` returns –1 with `errno` set to `EAGAIN`. There's another example using `sigqueue` in the next section.

9.5.5 `sigwaitinfo` and `sigtimedwait`

The `sigwait` system call in Section 9.2.2 isn't part of RTS, but it works just fine on RTS signals, even with queuing. It accepts a pending signal and returns it; if there are still signals of that type pending, they stayed queued. As with signal handlers, if more than one RTS signal (SIGRTMIN through SIGRTMAX) is pending, the highest priority (lowest numbered) signal is accepted first.

But the problem with sigwait is that you can't get the siginfo_t stuff, including the value, which is perhaps the most important RTS feature. So, there's a slightly enhanced system call named sigwaitinfo:

sigwaitinfo—wait for signal

```
#include <signal.h>

int sigwaitinfo(
    const sigset_t *set,       /* signals to wait for */
    siginfo_t *info            /* returned info */
);
/* Returns signal number or -1 on error (sets errno) */
```

If info is NULL, you don't get the information back, and sigwaitinfo is exactly like sigwait except that the signal number is returned as the function value instead of through an argument. If info is non-NULL, it receives a structure just as for an SA_SIGINFO-style signal handler, as described in Section 9.5.1. As with sigwait, make sure the signals in set are blocked; otherwise it's unpredictable what will happen when one is delivered.

Here's an example using queued signals. First, these are definitions for the two RTS signals we'll use:

```
#define MYSIG_COUNT SIGRTMIN
#define MYSIG_STOP SIGRTMIN + 1
```

Here's the function for the thread that waits for a signal, displays its value as a string if it's MYSIG_COUNT, and returns (terminating the thread) if it's MYSIG_STOP:

```
static void *sig_thread(void *arg)
{
    int signum;
    siginfo_t info;

    do {
        signum = sigwaitinfo((sigset_t *)arg, &info);
        if (signum == MYSIG_COUNT)
            printf("Got MYSIG_COUNT; value: %s\n",
              (char *)info.si_value.sival_ptr);
```

```
        else if (signum == MYSIG_STOP) {
            printf("Got MYSIG_STOP; terminating thread\n");
            return (void *)true;
        }
        else
            printf("Got %d\n", signum);
    } while (signum != -1 || errno == EINTR);
    EC_FAIL

EC_CLEANUP_BGN
    EC_FLUSH("sig_thread")
    return (void *)false;
EC_CLEANUP_END
}
```

Note that we couldn't use a `switch` statement because `MYSIG_COUNT` and `MYSIG_STOP` are potentially nonconstant integer expressions, as noted in Section 9.5.2.

Now here's the `main` function:

```
int main(void)
{
    sigset_t set;
    struct sigaction act;
    union sigval value;
    pthread_t tid;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, MYSIG_COUNT) )
    ec_neg1( sigaddset(&set, MYSIG_STOP) )
    ec_rv( pthread_sigmask(SIG_SETMASK, &set, NULL) )
    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = dummy_handler;
    ec_neg1( sigaction(MYSIG_COUNT, &act, NULL) )
    ec_neg1( sigaction(MYSIG_STOP, &act, NULL) )
    value.sival_ptr = "One";
    ec_neg1( sigqueue(getpid(), MYSIG_COUNT, value) )
    value.sival_ptr = "Two";
    ec_neg1( sigqueue(getpid(), MYSIG_COUNT, value) )
    value.sival_ptr = "Three";
    ec_neg1( sigqueue(getpid(), MYSIG_COUNT, value) )
    value.sival_ptr = NULL;
    ec_neg1( sigqueue(getpid(), MYSIG_STOP, value) )
    ec_rv( pthread_create(&tid, NULL, sig_thread, &set) )
    ec_rv( pthread_join(tid, NULL) )
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void dummy_handler(int signum, siginfo_t *info, void *context)
{
}
```

Note that four signals are generated before the thread is even created, to demonstrate that they will be queued. It's guaranteed that the `MYSIG_COUNT` signals will be accepted before `MYSIG_STOP` because they have a lower number. Also note that the signals are blocked and stay blocked. The dummy handler is needed because we must set `SA_SIGINFO` to turn on queuing. It so happens that Solaris queues the RTS signals no matter what, but you can't count on that in a portable program.

Here's the output, proving that the signals were queued:

```
Got MYSIG_COUNT; value: One
Got MYSIG_COUNT; value: Two
Got MYSIG_COUNT; value: Three
Got MYSIG_STOP; terminating thread
```

If you only want to wait for a limited time, you can use `sigtimedwait`, which adds a time-out argument to `sigwaitinfo`:

sigtimedwait—wait for signal

```
#include <signal.h>

int sigtimedwait(
     const sigset_t *set,          /* signals to wait for */
     siginfo_t *info,              /* returned info */
     const struct timespec *ts     /* max. time to wait */
);
/* Returns signal number or -1 on error (sets errno) */
```

`sigtimedwait` waits for at most the number of nanoseconds specified by `ts` for a signal in `set` to become pending and be returned. If the time period expires, it returns –1 with `errno` set to `EAGAIN`. You can't have a `timeout` of `NULL`. If both `timespec` members are zero and no signal in `set` is pending, `sigtimedwait` returns immediately, but this is not really a special case—it has timed-out.

9.5.6 `sigevent` Structure and `SIGEV_THREAD`

The RTS-generation functions other than `sigqueue` are described elsewhere, as listed in Section 9.5.1, but they all use a `sigevent` structure to specify what signal and value to send. I'll explain just the `sigevent` structure here.

struct sigevent—structure for RTS-generation functions

```
struct sigevent {
    int sigev_notify;             /* notification type (see below) */
    int sigev_signo;              /* signal number */
    union sigval sigev_value;     /* signal value */
    void (*sigev_notify_function)(union sigval); /* thread function */
    pthread_attr_t *sigev_notify_attributes;     /* thread attributes */
};
```

The signal to be sent (`SIGRTMIN`, say) goes into `sigev_signo`, and the value to go along with it goes into `sigev_value`. As with the other RTS features, it's guaranteed to work only for RTS signals. The `sigev_notify` member specifies how the signal is to be sent:

`SIGEV_SIGNAL`	Generate a signal, as though `sigqueue` had been called
`SIGEV_THREAD`	Start a thread, as though `pthread_create` had been called
`SIGEV_NONE`	Don't provide any notification, which usually doesn't make much sense

`SIGEV_THREAD` is the most interesting. It causes a new thread to be started when the event occurs, every time it occurs, and the function pointed to by `sigev_notify_function` becomes the start function for the thread. Normally, start functions have a pointer-to-`void` argument; in this case it's a pointer-to-`union-sigval`, which is more or less the same, since one of the union members is a `void` pointer. You can set the attributes for the thread with the `sigev_notify_attributes` member, but you can't make the thread joinable. If the member is `NULL`, the thread is detached by default, which is the opposite of what `pthread_create` does.

There's a lot of overhead in creating a thread, so clearly you don't want to specify `SIGEV_THREAD` unless the event is fairly rare and/or its generation represents a large amount of work relative to the cost of starting a new thread. For example, starting a new thread every time input arrives is probably overdoing it.

In case you're confused, creating a thread when a signal would otherwise be generated is completely different from having a thread waiting in `sigwait` (Section 9.2.2). All they have in common is that they both use threads.

9.6 Global Jumps

Normally, a function returns to its caller only by executing a return statement or, if it's a `void` function, by flowing off the end of its outer block, which is the same thing. But it's possible to jump to an arbitrary, but preplanned, point in the program with the Standard C functions `setjmp` and `longjmp`:

setjmp—set jump point

```
#include <setjmp.h>

int setjmp(
     jmp_buf loc_info       /* saved location information */
);
/* Returns 0 if called directly, non-zero if from longjmp (no error
   return) */
```

longjmp—jump to jump point

```
#include <setjmp.h>

void longjmp(
     jmp_buf loc_info,      /* saved location information */
     int val                /* value for setjmp to return */
);
```

You mark the location you want to jump to—the label, so to speak—with a call to `setjmp` that saves whatever location information it needs (e.g., current stack pointer, machine address) in its argument, which doesn't look like a pointer that can receive information, but is. Then, from within any function, no matter how deeply nested, you execute `longjmp` with that same argument and the effect is as if `setjmp` had returned `val`, which can't be zero; if it is, `setjmp` returns 1.

As calling the functions in which `longjmp` appears may have pushed data onto the stack, `longjmp` automatically pops it all off. The function containing the `setjmp` is not "called," and there is no recursion upon the return, although there may have been on the way to calling `longjmp`.

Here's an example. If you've never seen `setjmp` and `longjmp` before, you'll think it's pretty weird:

```
static jmp_buf loc_info;

static void fcn2(void)
{
    printf("In fcn2\n");
    longjmp(loc_info, 1234);
    printf("Leaving fcn2\n");
}

static void fcn1(void)
{
    printf("In fcn1\n");
    fcn2();
    printf("Leaving fcn1\n");
}

int main(void)
{
    int rtn;

    rtn = setjmp(loc_info);
    printf("setjmp returned %d\n", rtn);
    if (rtn == 0)
        fcn1();
    printf("Exiting\n");
    exit(EXIT_SUCCESS);
}
```

And here's the output:

```
setjmp returned 0
In fcn1
In fcn2
setjmp returned 1234
Exiting
```

So you can see that `setjmp` was called once but returned twice and that the jump went straight from the middle of `fcn2` back to the middle of `main`; the two "Leaving" lines weren't printed.

If you don't completely understand, don't worry about it because I'm now going to say this: *Don't ever use `longjmp`*. Here's why:

- While the stack is cleaned up, little else (e.g., allocated memory, open files) is.
- There are some restrictions about where you can use `setjmp` and `longjmp` that will get you into trouble if you forget them.

- Some people (including me) don't even like `goto`s *within* a function. They like `longjmp` much less. It makes programs very difficult to understand and maintain.
- The main use for `longjmp` is from within a signal handler, but it's not async-signal-safe, so you can't use it there unless you can guarantee that the signal handler was invoked by an async-signal-safe function, such as a `kill` executed from within the same process (Section 9.1.7).

If you think you need `longjmp`, it's probably because you haven't designed your functions well enough to allow them to return to their caller, perhaps with an indication of an error or other exceptional condition. Work harder and you'll come up with something.

`longjmp` may or may not restore the signal mask. If you want to force it to be restored, there are variants of `setjmp`/`longjmp` that are identical, except that they can save and restore the signal mask:

sigsetjmp—set jump point

```
#include <setjmp.h>

int sigsetjmp(
    sigjmp_buf loc_info,       /* saved location information */
    int savemask               /* non-zero to save signal mask */
);
/* Returns 0 if called directly, non-zero if from siglongjmp (no error
return) */
```

siglongjmp—jump to jump point, restore signal mask if saved

```
#include <setjmp.h>

void siglongjmp(
    sigjmp_buf loc_info,       /* saved location information */
    int val                    /* value for sigsetjmp to return */
);
```

The signal mask that gets restored is the one that was in effect when `sigsetjmp` was called. If `siglongjmp` is called from within a signal handler, which is what it was designed for, this may be different from what the signal handler would restore were it allowed to return normally.

Since `siglongjmp` is no more async-signal-safe than is `longjmp`, it can't usually be used in a signal handler, and its added ability to restore the signal mask is therefore almost useless.

If you don't want the signal mask to be saved, and for some reason you don't want to just set the `savemask` argument of `sigsetjmp` to zero, you can use yet another pair:

_setjmp—set jump point

```
#include <setjmp.h>

int _setjmp(
     jmp_buf loc_info       /* saved location information */
);
/* Returns 0 if called directly, non-zero if from longjmp (no error
   return) */
```

_longjmp—jump to jump point without restoring signal mask

```
#include <setjmp.h>

void _longjmp(
     jmp_buf loc_info,      /* saved location information */
     int val                /* value for setjmp to return */
);
```

Of course, `_longjmp` isn't async-signal-safe either. Actually, these two underscored functions are just leftovers from an early standard.

9.7 Clocks and Timers

This section describes various system clocks and timers that use those clocks to generate a signal, after a preset interval.

9.7.1 `alarm` System Call

alarm—schedule an alarm signal

```
#include <unistd.h>

unsigned alarm(
     unsigned secs      /* seconds until signal */
);
/* Returns seconds left on previous alarm or zero if none (no error
   return) */
```

Every process has one alarm clock set aside for the `alarm` system call. When the alarm goes off, a `SIGALRM` is sent. A child inherits its parent's alarm clock value, but the actual clock isn't shared. The alarm clock remains set across an `exec`.

`alarm` sets the clock to the number of seconds given by `secs`. The previous setting is returned; it will be 0 if no time remained on the clock previously or if there was no previous alarm. The previous setting is used to restore the clock to the way it was before `alarm` was called.

If `secs` is 0, the alarm clock is turned off. It's important to remember to do this. For example, let's say you have a blocking system call like `read` and you only want to block for a limited time. You catch `SIGALRM` (with an empty handler), call `alarm` for, say, 5 seconds, and then issue the `read`, which blocks. When the alarm goes off, the `SIGALRM` interrupts the blocking system call, and `read` returns –1 with `errno` set to `EINTR`. But if `read` returns sooner than 5 seconds, and you forget to turn off the alarm, it will go off later—*much* later, since 5 seconds is a very long time in computer terms—and maybe mysteriously interrupt something else! You'll know something is amiss if you rigorously check error returns, as we do in this book, but, alas, not everyone does.

Here's an example:

```
int main(void)
{
    struct sigaction act;
    char buf[100];
    ssize_t rtn;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
    alarm(5);
    if ((rtn = read(STDIN_FILENO, buf, sizeof(buf) - 1)) == -1) {
        if (errno == EINTR)
            printf("Timed out... type faster next time!\n");
        else
            EC_FAIL
    }
    alarm(0);
    if (rtn == 0)
        printf("Got EOF\n");
    else if (rtn > 0) {
        buf[rtn] = '\0';
        printf("Got %s", buf);
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

```
static void handler(int signum)
{
}
```

In the following sample interaction, I didn't type anything after the first execution of `alarm_test`; I typed "faster" after the second:

```
$ alarm_test
Timed out... type faster next time!
$ alarm_test
faster
Got faster
$
```

Using `alarm` to interrupt a blocking system call is fine in simple cases, like the one I showed, but in more complicated situations you'll probably be blocking in `select` or `poll`, rather than in `read` directly. If you want to time out, use `pselect` if it's available instead of setting an alarm.

A limitation with `alarm` is that there's only one such alarm clock per process. Sections 9.7.4 and 9.7.6 describe timer system calls with more flexibility.

9.7.2 `sleep` System Call

`sleep` is a familiar function that we've used throughout this book. It blocks a thread for a specified number of seconds:

sleep—suspend execution for seconds or until signal

```
#include <unistd.h>

unsigned sleep(
     unsigned secs          /* seconds to sleep */
);
/* Returns unslept amount */
```

The rules for `sleep` in the SUS contain lots of verbiage that allows for a `SIGALRM` to interfere with it, specifically so `sleep` can be implemented as a function in terms of `alarm` and `pause`, or better, `alarm` and `sigsuspend`. Here's a simple way to implement `sleep`:

```
unsigned aup_sleep(unsigned secs)
{
    struct sigaction act;
    unsigned unslept;
```

```
    memset(&act, 0, sizeof(act));
    act.sa_handler = slp_handler;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
    alarm(secs);
    pause();
    unslept = alarm(0);
    return unslept;

EC_CLEANUP_BGN
    EC_FLUSH("aup_sleep")
    return 0;
EC_CLEANUP_END
}

static void slp_handler(int signum)
{
}
```

This version works but has some problems:

- If `SIGALRM`s aren't blocked, the arrival of one before the call to `sigaction` might terminate the process.
- If they are blocked, the function won't even work.
- If the alarm goes off between the calls to `alarm` and `pause`, the pause may last forever.
- The old action for `SIGALRM` isn't restored.
- `alarm` and `sleep` are supposed to be compatible.

The last point means that in a sequence like

```
alarm(10);
...
sleep(20);
```

the alarm going off should interrupt the sleep and execute the handler for `SIGALRM`, and in a sequence like

```
alarm(20);
...
sleep(10);
```

the alarm should be set for 10 more seconds after `sleep` returns.

Handling the interaction of `alarm` and `sleep`, preventing an errant `SIGALRM` from terminating the process, preventing an infinite pause, and restoring the old action and mask require a lot of tricky code that has to handle the three possible cases:

- No `alarm` was set when `sleep` was called.
- The remaining `alarm` time is less than or equal to the requested `sleep` time.
- The remaining `alarm` time is greater than the requested `sleep` time.

This version works:[8]

```
unsigned aup_sleep(unsigned secs)
{
    sigset_t set, oset;
    struct sigaction act, oact;
    unsigned prev_alarm, slept, unslept, effective_secs;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGALRM) )
    ec_neg1( sigprocmask(SIG_BLOCK, &set, &oset) )
    prev_alarm = alarm(0);
    if (prev_alarm != 0 && prev_alarm <= secs)
        effective_secs = prev_alarm;
    else {
        memset(&act, 0, sizeof(act));
        act.sa_handler = slp_handler;
        ec_neg1( sigaction(SIGALRM, &act, &oact) )
        effective_secs = secs;
    }
    alarm(effective_secs);
    set = oset;
    ec_neg1( sigdelset(&set, SIGALRM) )
    if (sigsuspend(&set) == -1 && errno != EINTR)
        EC_FAIL
    unslept = alarm(0);
    slept = effective_secs - unslept;
    ec_neg1( sigaction(SIGALRM, &oact, NULL) )
    if (prev_alarm > slept)
        alarm(prev_alarm - slept);
    ec_neg1( sigprocmask(SIG_SETMASK, &oset, NULL) )
    return unslept;

EC_CLEANUP_BGN
    EC_FLUSH("aup_sleep")
    return 0;
EC_CLEANUP_END
}

static void slp_handler(int signum)
{
}
```

8. Geoff Clare contributed to this version by finding bugs in my original attempt.

Here's what's going on, step-by-step:

1. We block SIGALRM so we can proceed without worrying about one being delivered. We save the old signal mask in oset.
2. We turn off the alarm, in case it's set, and save the remaining time.
3. We calculate how long to sleep (effective_secs), reducing the amount requested if the remaining alarm time was less, and we set an alarm for that time.
4. If no alarm was already set or the sleep time is less than the remaining alarm time, we install the empty handler and save the old action in oact.
5. We call sigsuspend instead of pause so that we can atomically unblock SIGALRM. We leave the other bits in the mask the way they were on entry to aup_sleep.
6. When sigsuspend returns, it is because the alarm went off or because some other signal interrupted it. We don't especially care which it was. We do need the amount remaining on the alarm, though, which we get when we turn it off.
7. We calculate the amount of time actually slept.
8. We reset the old action for SIGALRM.
9. If the time slept is less than the remaining alarm time on entry, we reset the alarm for the new time remaining (some was slept off).
10. We reset the signal mask.
11. We return the unslept time.

If you can understand this function, you've mastered 90% of what's in this chapter and you can give yourself an A. If you find a mistake and mail it to *aup@basepath.com,* score yourself an A+.

9.7.3 Higher-Resolution Sleeping

sleep sleeps for some number of seconds, but that's way too long for many purposes. There are two other calls that sleep for more precise intervals:

usleep—suspend execution for microseconds or until signal

```
#include <unistd.h>

int usleep(
    useconds_t usecs            /* microseconds to sleep */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`usleep` is almost like `sleep`, except that it's an error to use it to sleep for a full second or longer. That is, `usecs` must be less than a million. If you know you have the Timers option (`_POSIX_TIMERS`), `nanosleep` is even more precise and doesn't have the restriction:

nanosleep—suspend execution for nanoseconds or until signal

```
#include <time.h>

int nanosleep(
    const struct timespec *nsecs, /* nanoseconds to sleep */
    struct timespec *remain       /* remaining time or NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`nanosleep` takes a `timespec` structure, which was defined in Section 1.7.2. To recap, it has a `tv_sec` member for seconds and a `tv_nsec` member for nanoseconds.

If it was interrupted, it returns –1 with `errno` set to `EINTR`, as usual, and also sets what `remain` points to, to the remaining time. If it returns zero or if `remain` is `NULL`, it doesn't return anything through the argument.

There's another sleeping function in Section 9.7.5 named `clock_nanosleep`.

9.7.4 Basic Interval-Timer System Calls

The `alarm` system call (Section 9.7.1) uses one process-wide interval timer and is in all versions of UNIX. All SUS-compatible systems, and some others, too, also have three more interval timers that you can use independently:

`ITIMER_REAL` Decrements in real time; generates a `SIGALRM` when it expires.

`ITIMER_VIRTUAL` Decrements in process virtual time; generates a `SIGVTALRM` when it expires.

`ITIMER_PROF` Decrements both in process virtual time and when the system is running on behalf of the process; generates a `SIGPROF` when it expires. It is intended for use by profilers, which help tune programs by indicating where they spend their time.

In the next two system calls, the `which` argument must be one of the three listed macros:

getitimer—get value of interval timer

```
#include <sys/time.h>

int getitimer(
     int which,                    /* timer to get */
     struct itimerval *val         /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setitimer—set value of interval timer

```
#include <sys/time.h>

int setitimer(
     int which,                    /* timer to set */
     const struct itimerval *val,  /* value to set */
     struct itimerval *oval        /* returned old value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct itimerval—structure for getitimer and setitimer

```
struct itimerval {
     struct timeval it_interval;   /* reset value */
     struct timeval it_value;      /* current value */
};
```

In an `itimerval` structure, `it_interval` is the time to reset the timer to when it expires, and `it_value` is the value of the current interval. That is, unlike `alarm`, which goes off only once, these timers can automatically reset themselves when they go off. If you call `setitimer` with an `it_value` member of zero, it stops the timer immediately. If you call `setitimer` with an `it_interval` member of zero, it stops the timer after the current interval expires. A `timeval` structure, defined in Section 1.7.1, has two members: `tv_sec` for seconds, and `tv_usec` for microseconds.

If the `oval` argument to `setitimer` is non-`NULL`, it gets the old value. Some examples:

```
/* 3.5 sec. timer, one time only */
itv.it_interval.tv_sec = 0;
itv.it_interval.tv_usec = 0;
itv.it_value.tv_sec = 3;
itv.it_value.tv_usec = 500000;
ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )
```

```
/* 3.5 sec. timer, then repeating 2.25 sec. timers */
itv.it_interval.tv_sec = 2;
itv.it_interval.tv_usec = 250000;
itv.it_value.tv_sec = 3;
itv.it_value.tv_usec = 500000;
ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )

/* stop timer immediately; interval doesn't matter */
itv.it_value.tv_sec = 0;
itv.it_value.tv_usec = 0;
ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )
```

In the middle example, the timer would fire at 3.5 seconds, again at 5.75, again at 8, and so on, every 2.25 seconds thereafter until stopped.

Here's an example that displays an X every 2 seconds of real ("wall clock") time. Note that once the timer is set, the rest of the program is free to go about its business, which in the example is just to read from the terminal and echo back what was read:

```
void timer_try1(void)
{
    struct sigaction act;
    struct itimerval itv;
    char buf[100];
    ssize_t nread;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
    memset(&itv, 0, sizeof(itv));
    itv.it_interval.tv_sec = 2;
    itv.it_value.tv_sec = 2;
    ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )
    while (true) {
        switch( nread = read(STDIN_FILENO, buf, sizeof(buf) - 1) ) {
        case -1:
            EC_FAIL
        case 0:
            printf("EOF\n");
            break;
        default:
            if (nread > 0)
                buf[nread] = '\0';
            ec_neg1( write(STDOUT_FILENO, buf, strlen(buf)) )
            continue;
        }
        break;
```

```
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("timer_try1")
EC_CLEANUP_END
}

void handler(int signum)
{
    write(STDOUT_FILENO, "\nX\n", 3);
}
```

And this was the output:

```
X
ERROR:  0: tm1 [/aup/c9/tmr.c:26] 0
                *** EINTR (4: "Interrupted system call") ***
```

What happened was that the first X came out OK, after 2 seconds, but the `SIGALRM` signal interrupted the `read`, which was blocked. The fix is to set the `SA_RESTART` flag so system calls won't be interrupted:

```
    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    act.sa_flags = SA_RESTART;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
```

Now the output is this, showing that "hello" was typed and then echoed, along with the Xs every 2 seconds:

```
X

X
hello
X

hello

X
...
```

There's another interval timer system call named `ualarm`, but it's obsolete.

9.7.5 Realtime Clocks

All UNIX systems have a basic clock whose value you can read with the `time` and `gettimeofday` system calls (Section 1.7.1), but the same Timers option

(`_POSIX_TIMERS`) that brought us `nanosleep` (Section 9.7.3) includes one or more additional clocks, depending on how many the software and hardware support. To access one of these, you refer to it by its clock ID.

All systems with the Timers option support one ID, `CLOCK_REALTIME`, which keeps track of the time of day. Whether there are others, what their macros are, and whether they're system-wide or per-process is implementation dependent. For example, Solaris also supports a system-wide `CLOCK_HIGHRES` clock that counts off from some point in the past. Reading it doesn't give you the time of day since you don't know how it was set, but it's fine for comparing readings at two different times.

You call `clock_gettime` to get the time in nanoseconds in a returned `timespec` structure (Sections 1.7.2 and 9.7.3):

clock_gettime—get time from clock

```
#include <time.h>

int clock_gettime(
    clockid_t clock_id,       /* clock ID (CLOCK_REALTIME, etc.) */
    struct timespec *tp       /* time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

You get the resolution of a clock in nanoseconds with `clock_getres`:

clock_getres—get clock resolution

```
#include <time.h>

int clock_getres(
    clockid_t clock_id,       /* clock ID */
    struct timespec *res      /* resolution */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

And, with appropriate privileges (superuser for `CLOCK_REALTIME`), you can set a clock:

clock_settime—set clock

```
#include <time.h>

int clock_settime(
    clockid_t clock_id,       /* clock ID */
    const struct timespec *tp /* time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Here is a function that gets the time and resolution from a clock and then displays it, along with the time in seconds from the `time` system call (Section 1.7.1):

```
void clocks(void)
{
    struct timespec ts;
    time_t tm;

    ec_neg1( time(&tm) )
    printf("time() Time: %ld secs.\n", (long)tm);
    printf("CLOCK_REALTIME:\n");
    ec_neg1( clock_gettime(CLOCK_REALTIME, &ts) )
    printf("Time: %ld.%09ld secs.\n", (long)ts.tv_sec, (long)ts.tv_nsec);
    ec_neg1( clock_getres(CLOCK_REALTIME, &ts) )
    printf("Res.: %ld.%09ld secs.\n", (long)ts.tv_sec, (long)ts.tv_nsec);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("clocks")
EC_CLEANUP_END
}
```

The output on a FreeBSD system was:

```
time() Time: 1051646878 secs.
CLOCK_REALTIME:
Time: 1051646878.568628061 secs.
Res.: 0.000000838 secs.
```

This indicates that the resolution is just under a microsecond. On different hardware, the output on Solaris was:

```
time() Time: 1051646409 secs.
CLOCK_REALTIME:
Time: 1051646409.686869683 secs.
Res.: 0.010000000 secs.
```

There the resolution was only a hundredth of a second. Displaying a time with 9 digits to the right of the decimal point is misleading—it would have been better to limit the display to what the resolution is capable of. However, these clocks aren't for display purposes. They're for timing things.

There's a version of `nanosleep` (Section 9.7.3) that uses a specific clock:

clock_nanosleep—suspend execution for nanoseconds or until signal

```
#include <time.h>

int clock_nanosleep(
    clockid_t clock_id,           /* clock ID */
    int flags,                    /* TIMER_ABSTIME or zero */
    const struct timespec *nsecs, /* nanoseconds to sleep */
    struct timespec *remain       /* remaining time or NULL */
);
/* Returns 0 on success or error number on error */
```

The first argument is the clock ID; the last two are identical to `nanosleep`. If `flags` is zero, you sleep for the specified number of nanoseconds, just as with `nanosleep`. But if it's `TIMER_ABSTIME`, you sleep until the absolute time specified by `nsecs`. The last argument, which returns the time remaining, is used only for relative sleeping; that is, when `flags` is zero.

Finally, you can get a clock ID for a process's CPU-time clock on systems with the Process CPU-Time Clocks option (`_POSIX_CPUTIME`):

clock_getcpuclockid—get process CPU-time clock

```
#include <time.h>

int clock_getcpuclockid(
    pid_t pid,                    /* process ID */
    clockid_t *clock_id           /* returned clock ID */
);
/* Returns 0 on success or error number on error */
```

9.7.6 Advanced Interval-Timer System Calls

Just as the realtime clocks go beyond the basic clocks, so do the advanced interval timers go beyond the interval timers discussed in Section 9.7.4. Instead of just a few system-wide timers, with these calls you can have several interval timers per process, all based on the same clock; recall from the previous section that the only guaranteed clock is `CLOCK_REALTIME`.

You start by creating a timer:

timer_create—create per-process timer

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

int timer_create(
    clockid_t clockid,            /* clock ID */
    struct sigevent *sig,         /* NULL or signal to generate */
    timer_t *timer_id             /* returned timer ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If it succeeds, `timer_create` returns a new timer ID through the `timer_id` argument. If non-`NULL`, `sig` specifies a signal to be generated when the timer expires, a value, and how it's to be delivered, as explained in Section 9.5.6. If `sig` is `NULL`, you get a `SIGALRM` generated and the value is set to the timer ID.

You delete a timer with `timer_delete`:

timer_delete—delete per-process timer

```
#include <time.h>

int timer_delete(
     timer_t timer_id               /* timer ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Next come two calls like `getitimer` and `setitimer` (Section 9.7.4), except the resolution in the `itimerspec` structure is in nanoseconds, instead of microseconds:

timer_gettime—get value of per-process timer

```
#include <time.h>

int timer_gettime(
     timer_t timer_id,              /* timer ID */
     struct itimerspec *val         /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

timer_settime—set value of per-process timer

```
#include <time.h>

int timer_settime(
     timer_t timer_id,              /* timer ID */
     int flags,                     /* flags */
     const struct itimerspec *val,  /* value to set */
     struct itimerspec *oval        /* returned old value or NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct itimerspec—structure for timer_ functions

```
struct itimerspec {
     struct timespec it_interval;   /* reset value */
     struct timespec it_value;      /* current value */
};
```

If the `flags` argument to `timer_settime` is zero, the function behaves just like `setitimer`. But if the `TIMER_ABSTIME` flag is set, the `it_value` time is interpreted like the `nsecs` argument to `clock_nanosleep`: as an absolute time when the timer should go off. (That is, like the alarm clock next to your bed.) The `it_interval` member is still used to reset the timer when it goes off.

Only one signal will be queued when the timer expires, and if the `it_interval` member is small enough, it's possible for it to expire several times between the time the signal is generated and when it is delivered or accepted. In this case, you can get a count of the number of extra expirations—overruns—with this system call:

timer_getoverrun—get per-process timer overrun count

```
#include <time.h>

int timer_getoverrun(
    timer_t timer_id            /* timer ID */
);
/* Returns overrun count or -1 on error (sets errno) */
```

Exercises

9.1. Investigate what signals are implemented on your system other than the standard 28 listed in Section 9.1.3.

9.2. Write a program that generates a `SIGPIPE` signal (hint: use a pipe) when `write` is called. Then change it to ignore `SIGPIPE`s and verify that `write` returns a write error instead. What is the `errno` value and what does it mean?

9.3. Implement `pause` in terms of `sigwait` and any other system calls you need.

9.4. Try to write `signal` (Section 9.4) in terms of `sigaction`. It's tricky to deal with the handler—see if you can figure out a way.

9.5. Write `sigset` (Section 9.4) in terms of `sigaction`.

9.6. Write a signal handler that does a global jump when a signal arrives (Section 9.6) and demonstrates that execution continues normally from the `setjmp` or `sigsetjmp`. Why are `longjmp` and `siglongjmp` not on the list of async-signal-safe functions (Section 9.1.7)?

9.7. `longjmp` and `siglongjmp` only clean up the stack; they don't deal with memory allocated dynamically by `malloc` and `realloc`. Discuss the ramifications of this. Is an automatic solution possible? Is it desirable? Suggest a possible design change to (at least) `setjmp`, `sigsetjmp`, `longjmp`, `siglongjmp`, `malloc`, `realloc`, and `free` that handles the problem.

9.8. Write `sleep` in terms of `usleep`, obeying the restriction that `usleep` must sleep for under a second. That is, you must figure out a way to sleep for, say, 3 seconds.

9.9. Using whatever system calls you want from this chapter (e.g., interval-timer calls) and earlier chapters, write a command named `alarmclock` that takes a time and a message as arguments and then, at that time, displays that message on the standard output and rings a bell. Your command should use kernel facilities to wait for the time; other than to set things up, it should not execute until the time arrives. Design it to run in the background even if the user forgets the trailing ampersand.

9.10. Explain (without writing the code) how you would enhance your `alarmclock` command from the previous Exercise to remember its setting even if the system crashes. You also have to figure out how to get the background process restarted. Is there already a UNIX command that does something like this?

9.11. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.