21

SIGNALS: SIGNAL HANDLERS

This chapter continues the description of signals begun in the previous chapter. It focuses on signal handlers, and extends the discussion started in Section 20.4. Among the topics we consider are the following:

- how to design a signal handler, which necessitates a discussion of reentrancy and async-signal-safe functions;
- alternatives to performing a normal return from a signal handler, in particular, the use of a nonlocal goto for this purpose;
- handling of signals on an alternate stack;
- the use of the *sigaction()* `SA_SIGINFO` flag to allow a signal handler to obtain more detailed information about the signal that caused its invocation; and
- how a blocking system call may be interrupted by a signal handler, and how the call can be restarted if desired.

21.1 Designing Signal Handlers

In general, it is preferable to write simple signal handlers. One important reason for this is to reduce the risk of creating race conditions. Two common designs for signal handlers are the following:

- The signal handler sets a global flag and exits. The main program periodically checks this flag and, if it is set, takes appropriate action. (If the main program cannot perform such periodic checks because it needs to monitor one or more file descriptors to see if I/O is possible, then the signal handler can also write a single byte to a dedicated pipe whose read end is included among the file descriptors monitored by the main program. We show an example of this technique in Section 63.5.2.)
- The signal handler performs some type of cleanup and then either terminates the process or uses a nonlocal goto (Section 21.2.1) to unwind the stack and return control to a predetermined location in the main program.

In the following sections, we explore these ideas, as well as other concepts that are important in the design of signal handlers.

21.1.1 Signals Are Not Queued (Revisited)

In Section 20.10, we noted that delivery of a signal is blocked during the execution of its handler (unless we specify the `SA_NODEFER` flag to *sigaction()*). If the signal is (again) generated while the handler is executing, then it is marked as pending and later delivered when the handler returns. We also already noted that signals are not queued. If the signal is generated more than once while the handler is executing, then it is still marked as pending, and it will later be delivered only once.

That signals can "disappear" in this way has implications for how we design signal handlers. To begin with, we can't reliably count the number of times a signal is generated. Furthermore, we may need to code our signal handlers to deal with the possibility that multiple events of the type corresponding to the signal have occurred. We'll see an example of this when we consider the use of the `SIGCHLD` signal in Section 26.3.1.

21.1.2 Reentrant and Async-Signal-Safe Functions

Not all system calls and library functions can be safely called from a signal handler. To understand why requires an explanation of two concepts: reentrant functions and async-signal-safe functions.

Reentrant and nonreentrant functions

To explain what a reentrant function is, we need to first distinguish between single-threaded and multithreaded programs. Classical UNIX programs have a single *thread of execution*: the CPU processes instructions for a single logical flow of execution through the program. In a multithreaded program, there are multiple, independent, concurrent logical flows of execution within the same process.

In Chapter 29, we'll see how to explicitly create programs that contain multiple threads of execution. However, the concept of multiple threads of execution is also

relevant for programs that employ signal handlers. Because a signal handler may asynchronously interrupt the execution of a program at any point in time, the main program and the signal handler in effect form two independent (although not concurrent) threads of execution within the same process.

A function is said to be *reentrant* if it can safely be simultaneously executed by multiple threads of execution in the same process. In this context, "safe" means that the function achieves its expected result, regardless of the state of execution of any other thread of execution.

> The SUSv3 definition of a reentrant function is one "whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after the other in an undefined order, even if the actual execution is interleaved."

A function may be *nonreentrant* if it updates global or static data structures. (A function that employs only local variables is guaranteed to be reentrant.) If two invocations of (i.e., two threads executing) the function simultaneously attempt to update the same global variable or data structure, then these updates are likely to interfere with each other and produce incorrect results. For example, suppose that one thread of execution is in the middle of updating a linked list data structure to add a new list item when another thread also attempts to update the same linked list. Since adding a new item to the list requires updating multiple pointers, if another thread interrupts these steps and updates the same pointers, chaos will result.

Such possibilities are in fact rife within the standard C library. For example, we already noted in Section 7.1.3 that *malloc()* and *free()* maintain a linked list of freed memory blocks available for reallocation from the heap. If a call to *malloc()* in the main program is interrupted by a signal handler that also calls *malloc()*, then this linked list can be corrupted. For this reason, the *malloc()* family of functions, and other library functions that use them, are nonreentrant.

Other library functions are nonreentrant because they return information using statically allocated memory. Examples of such functions (described elsewhere in this book) include *crypt()*, *getpwnam()*, *gethostbyname()*, and *getservbyname()*. If a signal handler also uses one of these functions, then it will overwrite information returned by any earlier call to the same function from within the main program (or vice versa).

Functions can also be nonreentrant if they use static data structures for their internal bookkeeping. The most obvious examples of such functions are the members of the *stdio* library (*printf()*, *scanf()*, and so on), which update internal data structures for buffered I/O. Thus, when using *printf()* from within a signal handler, we may sometimes see strange output—or even a program crash or data corruption—if the handler interrupts the main program in the middle of executing a call to *printf()* or another *stdio* function.

Even if we are not using nonreentrant library functions, reentrancy issues can still be relevant. If a signal handler updates programmer-defined global data structures that are also updated within the main program, then we can say that the signal handler is nonreentrant with respect to the main program.

If a function is nonreentrant, then its manual page will normally provide an explicit or implicit indication of this fact. In particular, watch out for statements that the function uses or returns information in statically allocated variables.

Example program

Listing 21-1 demonstrates the nonreentrant nature of the *crypt()* function (Section 8.5). As command-line arguments, this program accepts two strings. The program performs the following steps:

1. Call *crypt()* to encrypt the string in the first command-line argument, and copy this string to a separate buffer using *strdup()*.
2. Establish a handler for `SIGINT` (generated by typing *Control-C*). The handler calls *crypt()* to encrypt the string supplied in the second command-line argument.
3. Enter an infinite `for` loop that uses *crypt()* to encrypt the string in the first command-line argument and check that the returned string is the same as that saved in step 1.

In the absence of a signal, the strings will always match in step 3. However, if a `SIGINT` signal arrives and the execution of the signal handler interrupts the main program just after the execution of the *crypt()* call in the `for` loop, but before the check to see if the strings match, then the main program will report a mismatch. When we run the program, this is what we see:

```
$ ./non_reentrant abc def
Repeatedly type Control-C to generate SIGINT
Mismatch on call 109871 (mismatch=1 handled=1)
Mismatch on call 128061 (mismatch=2 handled=2)
Many lines of output removed
Mismatch on call 727935 (mismatch=149 handled=156)
Mismatch on call 729547 (mismatch=150 handled=157)
Type Control-\ to generate SIGQUIT
Quit (core dumped)
```

Comparing the *mismatch* and *handled* values in the above output, we see that in the majority of cases where the signal handler is invoked, it overwrites the statically allocated buffer between the call to *crypt()* and the string comparison in *main()*.

Listing 21-1: Calling a nonreentrant function from both *main()* and a signal handler

signals/nonreentrant.c

```
#define _XOPEN_SOURCE 600
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include "tlpi_hdr.h"

static char *str2;              /* Set from argv[2] */
static int handled = 0;         /* Counts number of calls to handler */

static void
handler(int sig)
{
    crypt(str2, "xx");
    handled++;
}
```

```
int
main(int argc, char *argv[])
{
    char *cr1;
    int callNum, mismatch;
    struct sigaction sa;

    if (argc != 3)
        usageErr("%s str1 str2\n", argv[0]);

    str2 = argv[2];                     /* Make argv[2] available to handler */
    cr1 = strdup(crypt(argv[1], "xx"));  /* Copy statically allocated string
                                            to another buffer */
    if (cr1 == NULL)
        errExit("strdup");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    /* Repeatedly call crypt() using argv[1]. If interrupted by a
       signal handler, then the static storage returned by crypt()
       will be overwritten by the results of encrypting argv[2], and
       strcmp() will detect a mismatch with the value in 'cr1'. */

    for (callNum = 1, mismatch = 0; ; callNum++) {
        if (strcmp(crypt(argv[1], "xx"), cr1) != 0) {
            mismatch++;
            printf("Mismatch on call %d (mismatch=%d handled=%d)\n",
                    callNum, mismatch, handled);
        }
    }
}
```

signals/nonreentrant.c

Standard async-signal-safe functions

An *async-signal-safe* function is one that the implementation guarantees to be safe when called from a signal handler. A function is async-signal-safe either because it is reentrant or because it is not interruptible by a signal handler.

Table 21-1 lists the functions that various standards require to be async-signal-safe. In this table, the functions whose names are not followed by a *v2* or *v3* were specified as async-signal-safe in POSIX.1-1990. SUSv2 added the functions marked *v2* to the list, and those marked *v3* were added by SUSv3. Individual UNIX implementations may make other functions async-signal-safe, but all standards-conformant UNIX implementations must ensure that at least these functions are async-signal-safe (if they are provided by the implementation; not all of these functions are provided on Linux).

SUSv4 makes the following changes to Table 21-1:

- The following functions are removed: *fpathconf()*, *pathconf()*, and *sysconf()*.

- The following functions are added: *execl()*, *execv()*, *faccessat()*, *fchmodat()*, *fchownat()*, *fexecve()*, *fstatat()*, *futimens()*, *linkat()*, *mkdirat()*, *mkfifoat()*, *mknod()*, *mknodat()*, *openat()*, *readlinkat()*, *renameat()*, *symlinkat()*, *unlinkat()*, *utimensat()*, and *utimes()*.

Table 21-1: Functions required to be async-signal-safe by POSIX.1-1990, SUSv2, and SUSv3

_Exit() (v3)	*getpid()*	*sigdelset()*
_exit()	*getppid()*	*sigemptyset()*
abort() (v3)	*getsockname() (v3)*	*sigfillset()*
accept() (v3)	*getsockopt() (v3)*	*sigismember()*
access()	*getuid()*	*signal() (v2)*
aio_error() (v2)	*kill()*	*sigpause() (v2)*
aio_return() (v2)	*link()*	*sigpending()*
aio_suspend() (v2)	*listen() (v3)*	*sigprocmask()*
alarm()	*lseek()*	*sigqueue() (v2)*
bind() (v3)	*lstat() (v3)*	*sigset() (v2)*
cfgetispeed()	*mkdir()*	*sigsuspend()*
cfgetospeed()	*mkfifo()*	*sleep()*
cfsetispeed()	*open()*	*socket() (v3)*
cfsetospeed()	*pathconf()*	*sockatmark() (v3)*
chdir()	*pause()*	*socketpair() (v3)*
chmod()	*pipe()*	*stat()*
chown()	*poll() (v3)*	*symlink() (v3)*
clock_gettime() (v2)	*posix_trace_event() (v3)*	*sysconf()*
close()	*pselect() (v3)*	*tcdrain()*
connect() (v3)	*raise() (v2)*	*tcflow()*
creat()	*read()*	*tcflush()*
dup()	*readlink() (v3)*	*tcgetattr()*
dup2()	*recv() (v3)*	*tcgetpgrp()*
execle()	*recvfrom() (v3)*	*tcsendbreak()*
execve()	*recvmsg() (v3)*	*tcsetattr()*
fchmod() (v3)	*rename()*	*tcsetpgrp()*
fchown() (v3)	*rmdir()*	*time()*
fcntl()	*select() (v3)*	*timer_getoverrun() (v2)*
fdatasync() (v2)	*sem_post() (v2)*	*timer_gettime() (v2)*
fork()	*send() (v3)*	*timer_settime() (v2)*
fpathconf() (v2)	*sendmsg() (v3)*	*times()*
fstat()	*sendto() (v3)*	*umask()*
fsync() (v2)	*setgid()*	*uname()*
ftruncate() (v3)	*setpgid()*	*unlink()*
getegid()	*setsid()*	*utime()*
geteuid()	*setsockopt() (v3)*	*wait()*
getgid()	*setuid()*	*waitpid()*
getgroups()	*shutdown() (v3)*	*write()*
getpeername() (v3)	*sigaction()*	
getpgrp()	*sigaddset()*	

SUSv3 notes that all functions not listed in Table 21-1 are considered to be unsafe with respect to signals, but points out that a function is unsafe only when invocation of a signal handler interrupts the execution of an unsafe function, and the handler itself also calls an unsafe function. In other words, when writing signal handlers, we have two choices:

- Ensure that the code of the signal handler itself is reentrant and that it calls only async-signal-safe functions.
- Block delivery of signals while executing code in the main program that calls unsafe functions or works with global data structures also updated by the signal handler.

The problem with the second approach is that, in a complex program, it can be difficult to ensure that a signal handler will never interrupt the main program while it is calling an unsafe function. For this reason, the above rules are often simplified to the statement that we must not call unsafe functions from within a signal handler.

> If we set up the same handler function to deal with several different signals or use the `SA_NODEFER` flag to *sigaction()*, then a handler may interrupt itself. As a consequence, the handler may be nonreentrant if it updates global (or static) variables, even if they are not used by the main program.

Use of *errno* inside signal handlers

Because they may update *errno*, use of the functions listed in Table 21-1 can nevertheless render a signal handler nonreentrant, since they may overwrite the *errno* value that was set by a function called from the main program. The workaround is to save the value of *errno* on entry to a signal handler that uses any of the functions in Table 21-1 and restore the *errno* value on exit from the handler, as in the following example:

```
void
handler(int sig)
{
    int savedErrno;

    savedErrno = errno;

    /* Now we can execute a function that might modify errno */

    errno = savedErrno;
}
```

Use of unsafe functions in example programs in this book

Although *printf()* is not async-signal-safe, we use it in signal handlers in various example programs in this book. We do so because *printf()* provides an easy and concise way to demonstrate that a signal handler has been called, and to display the contents of relevant variables within the handler. For similar reasons, we occasionally use a few other unsafe functions in signal handlers, including other *stdio* functions and *strsignal()*.

Real-world applications should avoid calling non-async-signal-safe functions from signal handlers. To make this clear, each signal handler in the example programs that uses one of these functions is marked with a comment indicating that the usage is unsafe:

```
printf("Some message\n");          /* UNSAFE */
```

21.1.3 Global Variables and the *sig_atomic_t* Data Type

Notwithstanding reentrancy issues, it can be useful to share global variables between the main program and a signal handler. This can be safe as long as the main program correctly handles the possibility that the signal handler may change the global variable at any time. For example, one common design is to make a signal handler's sole action the setting of a global flag. This flag is periodically checked by the main program, which then takes appropriate action in response to the delivery of the signal (and clears the flag). When global variables are accessed in this way from a signal handler, we should always declare them using the `volatile` attribute (see Section 6.8) in order to prevent the compiler from performing optimizations that result in the variable being stored in a register.

Reading and writing global variables may involve more than one machine-language instruction, and a signal handler may interrupt the main program in the middle of such an instruction sequence. (We say that access to the variable is *nonatomic*.) For this reason, the C language standards and SUSv3 specify an integer data type, *sig_atomic_t*, for which reads and writes are guaranteed to be atomic. Thus, a global flag variable that is shared between the main program and a signal handler should be declared as follows:

```
volatile sig_atomic_t flag;
```

We show an example of the use of the *sig_atomic_t* data type in Listing 22-5, on page 466.

Note that the C increment (++) and decrement (--) operators don't fall within the guarantee provided for *sig_atomic_t*. On some hardware architectures, these operations may not be atomic (refer to Section 30.1 for more details). All that we are guaranteed to be safely allowed to do with a *sig_atomic_t* variable is set it within the signal handler, and check it in the main program (or vice versa).

C99 and SUSv3 specify that an implementation should define two constants (in `<stdint.h>`), `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX`, that define the range of values that may be assigned to variables of type *sig_atomic_t*. The standards require that this range be at least −127 to 127 if *sig_atomic_t* is represented as a signed value, or 0 to 255 if it is represented as an unsigned value. On Linux, these two constants equate to the negative and positive limits for signed 32-bit integers.

21.2 Other Methods of Terminating a Signal Handler

All of the signal handlers that we have looked at so far complete by returning to the main program. However, simply returning from a signal handler sometimes isn't desirable, or in some cases, isn't even useful. (We'll see an example of where

returning from a signal handler isn't useful when we discuss hardware-generated signals in Section 22.4.)

There are various other ways of terminating a signal handler:

- Use *_exit()* to terminate the process. Beforehand, the handler may carry out some cleanup actions. Note that we can't use *exit()* to terminate a signal handler, because it is not one of safe functions listed in Table 21-1. It is unsafe because it flushes *stdio* buffers prior to calling *_exit()*, as described in Section 25.1.
- Use *kill()* or *raise()* to send a signal that kills the process (i.e., a signal whose default action is process termination).
- Perform a nonlocal goto from the signal handler.
- Use the *abort()* function to terminate the process with a core dump.

The last two of these options are described in further detail in the following sections.

21.2.1 Performing a Nonlocal Goto from a Signal Handler

Section 6.8 described the use of *setjmp()* and *longjmp()* to perform a nonlocal goto from a function to one of its callers. We can also use this technique from a signal handler. This provides a way to recover after delivery of a signal caused by a hardware exception (e.g., a memory access error), and also allows us to catch a signal and return control to a particular point in a program. For example, upon receipt of a `SIGINT` signal (normally generated by typing *Control-C*), the shell performs a nonlocal goto to return control to its main input loop (and thus read a new command).

However, there is a problem with using the standard *longjmp()* function to exit from a signal handler. We noted earlier that, upon entry to the signal handler, the kernel automatically adds the invoking signal, as well as any signals specified in the *act.sa_mask* field, to the process signal mask, and then removes these signals from the mask when the handler does a normal return.

What happens to the signal mask if we exit the signal handler using *longjmp()*? The answer depends on the genealogy of the particular UNIX implementation. Under System V, *longjmp()* doesn't restore the signal mask, so that blocked signals are not unblocked upon leaving the handler. Linux follows the System V behavior. (This is usually not what we want, since it leaves the signal that caused invocation of the handler blocked.) Under BSD-derived implementations, *setjmp()* saves the signal mask in its *env* argument, and the saved signal mask is restored by *longjmp()*. (BSD-derived implementations also provide two other functions, *_setjmp()* and *_longjmp()*, which have the System V semantics.) In other words, we can't portably use *longjmp()* to exit a signal handler.

> If we define the `_BSD_SOURCE` feature test macro when compiling a program, then (the *glibc*) *setjmp()* follows the BSD semantics.

Because of this difference in the two main UNIX variants, POSIX.1-1990 chose not to specify the handling of the signal mask by *setjmp()* and *longjmp()*. Instead, it defined a pair of new functions, *sigsetjmp()* and *siglongjmp()*, that provide explicit control of the signal mask when performing a nonlocal goto.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
                        Returns 0 on initial call, nonzero on return via siglongjmp()
void siglongjmp(sigjmp_buf env, int val);
```

The *sigsetjmp()* and *siglongjmp()* functions operate similarly to *setjmp()* and *longjmp()*. The only differences are in the type of the *env* argument (*sigjmp_buf* instead of *jmp_buf*) and the extra *savesigs* argument to *sigsetjmp()*. If *savesigs* is nonzero, then the process signal mask that is current at the time of the *sigsetjmp()* call is saved in *env* and restored by a later *siglongjmp()* call specifying the same *env* argument. If *savesigs* is 0, then the process signal mask is not saved and restored.

The *longjmp()* and *siglongjmp()* functions are not listed among the async-signal-safe functions in Table 21-1. This is because calling any non-async-signal-safe function after performing a nonlocal goto carries the same risks as calling that function from within the signal handler. Furthermore, if a signal handler interrupts the main program while it is part-way through updating a data structure, and the handler exits by performing a nonlocal goto, then the incomplete update may leave that data structure in an inconsistent state. One technique that can help to avoid problems is to use *sigprocmask()* to temporarily block the signal while sensitive updates are being performed.

Example program

Listing 21-2 demonstrates the difference in signal mask handling for the two types of nonlocal gotos. This program establishes a handler for `SIGINT`. The program is designed to allow either *setjmp()* plus *longjmp()* or *sigsetjmp()* plus *siglongjmp()* to be used to exit the signal handler, depending on whether the program is compiled with the `USE_SIGSETJMP` macro defined. The program displays the current settings of the signal mask both on entry to the signal handler and after the nonlocal goto has transferred control from the handler back to the main program.

When we build the program so that *longjmp()* is used to exit the signal handler, this is what we see when we run the program:

```
$ make -s sigmask_longjmp          Default compilation causes setjmp() to be used
$ ./sigmask_longjmp
Signal mask at startup:
                <empty signal set>
Calling setjmp()
Type Control-C to generate SIGINT
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                2 (Interrupt)
(At this point, typing Control-C again has no effect, since SIGINT is blocked)
Type Control-\ to kill the program
Quit
```

From the program output, we can see that, after a *longjmp()* from the signal handler, the signal mask remains set to the value to which it was set on entry to the signal handler.

> In the above shell session, we built the program using the makefile supplied with the source code distribution for this book. The *-s* option tells *make* not to echo the commands that it is executing. We use this option to avoid cluttering the session log. ([Mecklenburg, 2005] provides a description of the GNU *make* program.)

When we compile the same source file to build an executable that uses *siglongjmp()* to exit the handler, we see the following:

```
$ make -s sigmask_siglongjmp       Compiles using cc -DUSE_SIGSETJMP
$ ./sigmask_siglongjmp x
Signal mask at startup:
                <empty signal set>
Calling sigsetjmp()
Type Control-C
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                <empty signal set>
```

At this point, SIGINT is not blocked, because *siglongjmp()* restored the signal mask to its original state. Next, we type *Control-C* again, so that the handler is once more invoked:

```
Type Control-C
Received signal 2 (Interrupt), signal mask is:
                2 (Interrupt)
After jump from handler, signal mask is:
                <empty signal set>
Type Control-\ to kill the program
Quit
```

From the above output, we can see that *siglongjmp()* restores the signal mask to the value it had at the time of the *sigsetjmp()* call (i.e., an empty signal set).

Listing 21-2 also demonstrates a useful technique for use with a signal handler that performs a nonlocal goto. Because a signal can be generated at any time, it may actually occur before the target of the goto has been set up by *sigsetjmp()* (or *setjmp()*). To prevent this possibility (which would cause the handler to perform a nonlocal goto using an uninitialized *env* buffer), we employ a guard variable, *canJump*, to indicate whether the *env* buffer has been initialized. If *canJump* is false, then instead of doing a nonlocal goto, the handler simply returns. An alternative approach is to arrange the program code so that the call to *sigsetjmp()* (or *setjmp()*) occurs before the signal handler is established. However, in complex programs, it may be difficult to ensure that these two steps are performed in that order, and the use of a guard variable may be simpler.

Note that using `#ifdef` was the simplest way of writing the program in Listing 21-2 in a standards-conformant fashion. In particular, we could not have replaced the `#ifdef` with the following run-time check:

```
if (useSiglongjmp)
    s = sigsetjmp(senv, 1);
else
    s = setjmp(env);
if (s == 0)
    ...
```

This is not permitted because SUSv3 doesn't allow *setjmp()* and *sigsetjmp()* to be used within an assignment statement (see Section 6.8).

Listing 21-2: Performing a nonlocal goto from a signal handler

signals/sigmask_longjmp.c

```
#define _GNU_SOURCE     /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <setjmp.h>
#include <signal.h>
#include "signal_functions.h"           /* Declaration of printSigMask() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t canJump = 0;
                        /* Set to 1 once "env" buffer has been
                           initialized by [sig]setjmp() */
#ifdef USE_SIGSETJMP
static sigjmp_buf senv;
#else
static jmp_buf env;
#endif

static void
handler(int sig)
{
    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), strsignal(), printSigMask(); see Section 21.1.2) */

    printf("Received signal %d (%s), signal mask is:\n", sig,
            strsignal(sig));
    printSigMask(stdout, NULL);

    if (!canJump) {
        printf("'env' buffer not yet set, doing a simple return\n");
        return;
    }

#ifdef USE_SIGSETJMP
    siglongjmp(senv, 1);
#else
    longjmp(env, 1);
#endif
}
```

```
int
main(int argc, char *argv[])
{
    struct sigaction sa;

    printSigMask(stdout, "Signal mask at startup:\n");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

#ifdef USE_SIGSETJMP
    printf("Calling sigsetjmp()\n");
    if (sigsetjmp(senv, 1) == 0)
#else
    printf("Calling setjmp()\n");
    if (setjmp(env) == 0)
#endif
        canJump = 1;                    /* Executed after [sig]setjmp() */

    else                                /* Executed after [sig]longjmp() */
        printSigMask(stdout, "After jump from handler, signal mask is:\n" );

    for (;;)                            /* Wait for signals until killed */
        pause();
}
```

signals/sigmask_longjmp.c

21.2.2 Terminating a Process Abnormally: *abort()*

The *abort()* function terminates the calling process and causes it to produce a core dump.

```
#include <stdlib.h>

void abort(void);
```

The *abort()* function terminates the calling process by raising a SIGABRT signal. The default action for SIGABRT is to produce a core dump file and terminate the process. The core dump file can then be used within a debugger to examine the state of the program at the time of the *abort()* call.

SUSv3 requires that *abort()* override the effect of blocking or ignoring SIGABRT. Furthermore, SUSv3 specifies that *abort()* must terminate the process unless the process catches the signal with a handler that doesn't return. This last statement requires a moment's thought. Of the methods of terminating a signal handler described in Section 21.2, the one that is relevant here is the use of a nonlocal goto to exit the handler. If this is done, then the effect of *abort()* will be nullified; otherwise,

abort() always terminates the process. In most implementations, termination is guaranteed as follows: if the process still hasn't terminated after raising SIGABRT once (i.e., a handler catches the signal and returns, so that execution of *abort()* is resumed), *abort()* resets the handling of SIGABRT to SIG_DFL and raises a second SIGABRT, which is guaranteed to kill the process.

If *abort()* does successfully terminate the process, then it also flushes and closes *stdio* streams.

An example of the use of *abort()* is provided in the error-handling functions of Listing 3-3, on page 54.

21.3 Handling a Signal on an Alternate Stack: *sigaltstack()*

Normally, when a signal handler is invoked, the kernel creates a frame for it on the process stack. However, this may not be possible if a process attempts to extend the stack beyond the maximum possible size. For example, this may occur because the stack grows so large that it encounters a region of mapped memory (Section 48.5) or the upwardly growing heap, or it reaches the RLIMIT_STACK resource limit (Section 36.3).

When a process attempts to grow its stack beyond the maximum possible size, the kernel generates a SIGSEGV signal for the process. However, since the stack space is exhausted, the kernel can't create a frame for any SIGSEGV handler that the program may have established. Consequently, the handler is not invoked, and the process is terminated (the default action for SIGSEGV).

If we instead need to ensure that the SIGSEGV signal is handled in these circumstances, we can do the following:

1. Allocate an area of memory, called an *alternate signal stack*, to be used for the stack frame of a signal handler.
2. Use the *sigaltstack()* system call to inform the kernel of the existence of the alternate signal stack.
3. When establishing the signal handler, specify the SA_ONSTACK flag, to tell the kernel that the frame for this handler should be created on the alternate stack.

The *sigaltstack()* system call both establishes an alternate signal stack and returns information about any alternate signal stack that is already established.

```
#include <signal.h>

int sigaltstack(const stack_t *sigstack, stack_t *old_sigstack);
                                                    Returns 0 on success, or –1 on error
```

The *sigstack* argument points to a structure specifying the location and attributes of the new alternate signal stack. The *old_sigstack* argument points to a structure used to return information about the previously established alternate signal stack (if there was one). Either one of these arguments can be specified as NULL. For example, we can find out about the existing alternate signal stack, without changing it, by

specifying NULL for the *sigstack* argument. Otherwise, each of these arguments points to a structure of the following type:

```
typedef struct {
    void  *ss_sp;         /* Starting address of alternate stack */
    int    ss_flags;      /* Flags: SS_ONSTACK, SS_DISABLE */
    size_t ss_size;       /* Size of alternate stack */
} stack_t;
```

The *ss_sp* and *ss_size* fields specify the size and location of the alternate signal stack. When actually using the alternate signal stack, the kernel automatically takes care of aligning the value given in *ss_sp* to an address boundary that is suitable for the hardware architecture.

Typically, the alternate signal stack is either statically allocated or dynamically allocated on the heap. SUSv3 specifies the constant SIGSTKSZ to be used as a typical value when sizing the alternate stack, and MINSIGSTKSZ as the minimum size required to invoke a signal handler. On Linux/x86-32, these constants are defined with the values 8192 and 2048, respectively.

The kernel doesn't resize an alternate signal stack. If the stack overflows the space we have allocated for it, then chaos results (e.g., overwriting of variables beyond the limits of the stack). This is not usually a problem—because we normally use an alternate signal stack to handle the special case of the standard stack overflowing, typically only one or a few frames are allocated on the stack. The job of the SIGSEGV handler is either to perform some cleanup and terminate the process or to unwind the standard stack using a nonlocal goto.

The *ss_flags* field contains one of the following values:

SS_ONSTACK
: If this flag is set when retrieving information about the currently established alternate signal stack (*old_sigstack*), it indicates that the process is currently executing on the alternate signal stack. Attempts to establish a new alternate signal stack while the process is already running on an alternate signal stack result in an error (EPERM) from *sigaltstack()*.

SS_DISABLE
: Returned in *old_sigstack*, this flag indicates that there is no currently established alternate signal stack. When specified in *sigstack*, this disables a currently established alternate signal stack.

Listing 21-3 demonstrates the establishment and use of an alternate signal stack. After establishing an alternate signal stack and a handler for SIGSEGV, this program calls a function that infinitely recurses, so that the stack overflows and the process is sent a SIGSEGV signal. When we run the program, this is what we see:

```
$ ulimit -s unlimited
$ ./t_sigaltstack
Top of standard stack is near 0xbffff6b8
Alternate stack is at          0x804a948-0x804cfff
Call    1 - top of stack near 0xbff0b3ac
Call    2 - top of stack near 0xbfe1714c
Many intervening lines of output removed
Call 2144 - top of stack near 0x4034120c
```

```
    Call 2145 - top of stack near 0x4024cfac
    Caught signal 11 (Segmentation fault)
    Top of handler stack near      0x804c860
```

In this shell session, the *ulimit* command is used to remove any RLIMIT_STACK resource limit that may have been set in the shell. We explain this resource limit in Section 36.3.

Listing 21-3: Using *sigaltstack()*

signals/t_sigaltstack.c

```
#define _GNU_SOURCE          /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigsegvHandler(int sig)
{
    int x;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), strsignal(), fflush(); see Section 21.1.2) */

    printf("Caught signal %d (%s)\n", sig, strsignal(sig));
    printf("Top of handler stack near     %10p\n", (void *) &x);
    fflush(NULL);

    _exit(EXIT_FAILURE);               /* Can't return after SIGSEGV */
}

static void             /* A recursive function that overflows the stack */
overflowStack(int callNum)
{
    char a[100000];                    /* Make this stack frame large */

    printf("Call %4d - top of stack near %10p\n", callNum, &a[0]);
    overflowStack(callNum+1);
}

int
main(int argc, char *argv[])
{
    stack_t sigstack;
    struct sigaction sa;
    int j;

    printf("Top of standard stack is near %10p\n", (void *) &j);

    /* Allocate alternate stack and inform kernel of its existence */

    sigstack.ss_sp = malloc(SIGSTKSZ);
    if (sigstack.ss_sp == NULL)
        errExit("malloc");
```

```
    sigstack.ss_size = SIGSTKSZ;
    sigstack.ss_flags = 0;
    if (sigaltstack(&sigstack, NULL) == -1)
        errExit("sigaltstack");
    printf("Alternate stack is at         %10p-%p\n",
            sigstack.ss_sp, (char *) sbrk(0) - 1);

    sa.sa_handler = sigsegvHandler;     /* Establish handler for SIGSEGV */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_ONSTACK;           /* Handler uses alternate stack */
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        errExit("sigaction");

    overflowStack(1);
}
```

signals/t_sigaltstack.c

21.4 The SA_SIGINFO Flag

Setting the SA_SIGINFO flag when establishing a handler with *sigaction()* allows the handler to obtain additional information about a signal when it is delivered. In order to obtain this information, we must declare the handler as follows:

```
void handler(int sig, siginfo_t *siginfo, void *ucontext);
```

The first argument, *sig*, is the signal number, as for a standard signal handler. The second argument, *siginfo*, is a structure used to provide the additional information about the signal. We describe this structure below. The last argument, *ucontext*, is also described below.

Since the above signal handler has a different prototype from a standard signal handler, C typing rules mean that we can't use the *sa_handler* field of the *sigaction* structure to specify the address of the handler. Instead, we must use an alternative field: *sa_sigaction*. In other words, the definition of the *sigaction* structure is somewhat more complex than was shown in Section 20.13. In full, the structure is defined as follows:

```
struct sigaction {
    union {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    } __sigaction_handler;
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};

/* Following defines make the union fields look like simple fields
   in the parent structure */

#define sa_handler __sigaction_handler.sa_handler
#define sa_sigaction __sigaction_handler.sa_sigaction
```

The *sigaction* structure uses a union to combine the *sa_sigaction* and *sa_handler* fields. (Most other UNIX implementations similarly use a union for this purpose.) Using a union is possible because only one of these fields is required during a particular call to *sigaction()*. (However, this can lead to strange bugs if we naively expect to be able to set the *sa_handler* and *sa_sigaction* fields independently of one another, perhaps because we reuse a single *sigaction* structure in multiple *sigaction()* calls to establish handlers for different signals.)

Here is an example of the use of SA_SIGINFO to establish a signal handler:

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_SIGINFO;

if (sigaction(SIGINT, &act, NULL) == -1)
    errExit("sigaction");
```

For complete examples of the use of the SA_SIGINFO flag, see Listing 22-3 (page 462) and Listing 23-5 (page 500).

The *siginfo_t* structure

The *siginfo_t* structure passed as the second argument to a signal handler that is established with SA_SIGINFO has the following form:

```
typedef struct {
    int     si_signo;         /* Signal number */
    int     si_code;          /* Signal code */
    int     si_trapno;        /* Trap number for hardware-generated signal
                                 (unused on most architectures) */
    union sigval si_value;    /* Accompanying data from sigqueue() */
    pid_t   si_pid;           /* Process ID of sending process */
    uid_t   si_uid;           /* Real user ID of sender */
    int     si_errno;         /* Error number (generally unused) */
    void   *si_addr;          /* Address that generated signal
                                 (hardware-generated signals only) */
    int     si_overrun;       /* Overrun count (Linux 2.6, POSIX timers) */
    int     si_timerid;       /* (Kernel-internal) Timer ID
                                 (Linux 2.6, POSIX timers) */
    long    si_band;          /* Band event (SIGPOLL/SIGIO) */
    int     si_fd;            /* File descriptor (SIGPOLL/SIGIO) */
    int     si_status;        /* Exit status or signal (SIGCHLD) */
    clock_t si_utime;         /* User CPU time (SIGCHLD) */
    clock_t si_stime;         /* System CPU time (SIGCHLD) */
} siginfo_t;
```

The _POSIX_C_SOURCE feature test macro must be defined with a value greater than or equal to 199309 in order to make the declaration of the *siginfo_t* structure visible from <signal.h>.

On Linux, as on most UNIX implementations, many of the fields in the *siginfo_t* structure are combined into a union, since not all of the fields are needed for each signal. (See <bits/siginfo.h> for details.)

Upon entry to a signal handler, the fields of the *siginfo_t* structure are set as follows:

si_signo
: This field is set for all signals. It contains the number of the signal causing invocation of the handler—that is, the same value as the *sig* argument to the handler.

si_code
: This field is set for all signals. It contains a code providing further information about the origin of the signal, as shown in Table 21-1.

si_value
: This field contains the accompanying data for a signal sent via *sigqueue()*. We describe *sigqueue()* in Section 22.8.1.

si_pid
: For signals sent via *kill()* or *sigqueue()*, this field is set to the process ID of the sending process.

si_uid
: For signals sent via *kill()* or *sigqueue()*, this field is set to the real user ID of the sending process. The system provides the real user ID of the sending process because that is more informative than providing the effective user ID. Consider the permission rules for sending signals described in Section 20.5: if the effective user ID grants the sender permission to send the signal, then that user ID must either be 0 (i.e., a privileged process), or be the same as the real user ID or saved set-user-ID of the receiving process. In this case, it could be useful for the receiver to know the sender's real user ID, which may be different from the effective user ID (e.g., if the sender is a set-user-ID program).

si_errno
: If this field is set to a nonzero value, then it contains an error number (like *errno*) that identifies the cause of the signal. This field is generally unused on Linux.

si_addr
: This field is set only for hardware-generated `SIGBUS`, `SIGSEGV`, `SIGILL`, and `SIGFPE` signals. For the `SIGBUS` and `SIGSEGV` signals, this field contains the address that caused the invalid memory reference. For the `SIGILL` and `SIGFPE` signals, this field contains the address of the program instruction that caused the signal.

The following fields, which are nonstandard Linux extensions, are set only on the delivery of a signal generated on expiration of a POSIX timer (see Section 23.6):

si_timerid
: This field contains an ID that the kernel uses internally to identify the timer.

si_overrun
: This field is set to the overrun count for the timer.

The following two fields are set only for the delivery of a SIGIO signal (Section 63.3):

si_band
: This field contains the "band event" value associated with the I/O event. (In versions of *glibc* up until 2.3.2, *si_band* was typed as *int*.)

si_fd
: This field contains the number of the file descriptor associated with the I/O event. This field is not specified in SUSv3, but it is present on many other implementations.

The following fields are set only for the delivery of a SIGCHLD signal (Section 26.3):

si_status
: This field contains either the exit status of the child (if *si_code* is CLD_EXITED) or the number of the signal sent to the child (i.e., the number of the signal that terminated or stopped the child, as described in Section 26.1.3).

si_utime
: This field contains the user CPU time used by the child process. In kernels before 2.6, and since 2.6.27, this is measured in system clock ticks (divide by *sysconf(_SC_CLK_TCK)*). In 2.6 kernels before 2.6.27, a bug meant that this field reported times measured in (user-configurable) jiffies (see Section 10.6). This field is not specified in SUSv3, but it is present on many other implementations.

si_stime
: This field contains the system CPU time used by the child process. See the description of the *si_utime* field. This field is not specified in SUSv3, but it is present on many other implementations.

The *si_code* field provides further information about the origin of the signal, using the values shown in Table 21-2. Not all of the signal-specific values shown in the second column of this table occur on all UNIX implementations and hardware architectures (especially in the case of the four hardware-generated signals SIGBUS, SIGSEGV, SIGILL, and SIGFPE), although all of these constants are defined on Linux and most appear in SUSv3.

Note the following additional points about the values shown in Table 21-2:

- The values SI_KERNEL and SI_SIGIO are Linux-specific. They are not specified in SUSv3 and do not appear on other UNIX implementations.
- SI_SIGIO is employed only in Linux 2.2. From kernel 2.4 onward, Linux instead employs the POLL_* constants shown in the table.

> SUSv4 specifies the *psiginfo()* function, whose purpose is similar to *psignal()* (Section 20.8). The *psiginfo()* function takes two arguments: a pointer to a *siginfo_t* structure and a message string. It prints the message string on standard error, followed by information about the signal described in the *siginfo_t* structure. The *psiginfo()* function is provided by *glibc* since version 2.10. The *glibc* implementation prints the signal description, the origin of the signal (as indicated by the *si_code* field), and, for some signals, other fields from the *siginfo_t* structure. The *psiginfo()* function is new in SUSv4, and it is not available on all systems.

Table 21-2: Values returned in the *si_code* field of the *siginfo_t* structure

Signal	*si_code* value	Origin of signal
Any	SI_ASYNCIO	Completion of an asynchronous I/O (AIO) operation
	SI_KERNEL	Sent by the kernel (e.g., a signal from terminal driver)
	SI_MESGQ	Message arrival on POSIX message queue (since Linux 2.6.6)
	SI_QUEUE	A realtime signal from a user process via *sigqueue()*
	SI_SIGIO	SIGIO signal (Linux 2.2 only)
	SI_TIMER	Expiration of a POSIX (realtime) timer
	SI_TKILL	A user process via *tkill()* or *tgkill()* (since Linux 2.4.19)
	SI_USER	A user process via *kill()* or *raise()*
SIGBUS	BUS_ADRALN	Invalid address alignment
	BUS_ADRERR	Nonexistent physical address
	BUS_MCEERR_AO	Hardware memory error; action optional (since Linux 2.6.32)
	BUS_MCEERR_AR	Hardware memory error; action required (since Linux 2.6.32)
	BUS_OBJERR	Object-specific hardware error
SIGCHLD	CLD_CONTINUED	Child continued by SIGCONT (since Linux 2.6.9)
	CLD_DUMPED	Child terminated abnormally, with core dump
	CLD_EXITED	Child exited
	CLD_KILLED	Child terminated abnormally, without core dump
	CLD_STOPPED	Child stopped
	CLD_TRAPPED	Traced child has stopped
SIGFPE	FPE_FLTDIV	Floating-point divide-by-zero
	FPE_FLTINV	Invalid floating-point operation
	FPE_FLTOVF	Floating-point overflow
	FPE_FLTRES	Floating-point inexact result
	FPE_FLTUND	Floating-point underflow
	FPE_INTDIV	Integer divide-by-zero
	FPE_INTOVF	Integer overflow
	FPE_SUB	Subscript out of range
SIGILL	ILL_BADSTK	Internal stack error
	ILL_COPROC	Coprocessor error
	ILL_ILLADR	Illegal addressing mode
	ILL_ILLOPC	Illegal opcode
	ILL_ILLOPN	Illegal operand
	ILL_ILLTRP	Illegal trap
	ILL_PRVOPC	Privileged opcode
	ILL_PRVREG	Privileged register
SIGPOLL/ SIGIO	POLL_ERR	I/O error
	POLL_HUP	Device disconnected
	POLL_IN	Input data available
	POLL_MSG	Input message available
	POLL_OUT	Output buffers available
	POLL_PRI	High-priority input available
SIGSEGV	SEGV_ACCERR	Invalid permissions for mapped object
	SEGV_MAPERR	Address not mapped to object

(continued)

Table 21-2: Values returned in the *si_code* field of the *siginfo_t* structure (continued)

Signal	*si_code* value	Origin of signal
SIGTRAP	TRAP_BRANCH	Process branch trap
	TRAP_BRKPT	Process breakpoint
	TRAP_HWBKPT	Hardware breakpoint/watchpoint
	TRAP_TRACE	Process trace trap

The *ucontext* argument

The final argument passed to a handler established with the SA_SIGINFO flag, *ucontext*, is a pointer to a structure of type *ucontext_t* (defined in <ucontext.h>). (SUSv3 uses a *void* pointer for this argument because it doesn't specify any of the details of the argument.) This structure provides so-called user-context information describing the process state prior to invocation of the signal handler, including the previous process signal mask and saved register values (e.g., program counter and stack pointer). This information is rarely used in signal handlers, so we don't go into further details.

> Another use of *ucontext_t* structures is with the functions *getcontext()*, *makecontext()*, *setcontext()*, and *swapcontext()*, which allow a process to retrieve, create, change, and swap execution contexts, respectively. (These operations are somewhat like *setjmp()* and *longjmp()*, but more general.) These functions can be used to implement coroutines, where the thread of execution of a process alternates between two (or more) functions. SUSv3 specifies these functions, but marks them obsolete. SUSv4 removes the specifications, and suggests that applications should be rewritten to use POSIX threads instead. The *glibc* manual provides further information about these functions.

21.5 Interruption and Restarting of System Calls

Consider the following scenario:

1. We establish a handler for some signal.
2. We make a blocking system call, for example, a *read()* from a terminal device, which blocks until input is supplied.
3. While the system call is blocked, the signal for which we established a handler is delivered, and its signal handler is invoked.

What happens after the signal handler returns? By default, the system call fails with the error EINTR ("Interrupted function"). This can be a useful feature. In Section 23.3, we'll see how to use a timer (which results in the delivery of a SIGALRM signal) to set a timeout on a blocking system call such as *read()*.

Often, however, we would prefer to continue the execution of an interrupted system call. To do this, we could use code such as the following to manually restart a system call in the event that it is interrupted by a signal handler:

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue;                   /* Do nothing loop body */

if (cnt == -1)                  /* read() failed with other than EINTR */
    errExit("read");
```

If we frequently write code such as the above, it can be useful to define a macro such as the following:

```
#define NO_EINTR(stmt) while ((stmt) == -1 && errno == EINTR);
```

Using this macro, we can rewrite the earlier *read()* call as follows:

```
NO_EINTR(cnt = read(fd, buf, BUF_SIZE));

if (cnt == -1)                /* read() failed with other than EINTR */
    errExit("read");
```

> The GNU C library provides a (nonstandard) macro with the same purpose as our NO_EINTR() macro in <unistd.h>. The macro is called TEMP_FAILURE_RETRY() and is made available if the _GNU_SOURCE feature test macro is defined.

Even if we employ a macro like NO_EINTR(), having signal handlers interrupt system calls can be inconvenient, since we must add code to each blocking system call (assuming that we want to restart the call in each case). Instead, we can specify the SA_RESTART flag when establishing the signal handler with *sigaction()*, so that system calls are automatically restarted by the kernel on the process's behalf. This means that we don't need to handle a possible EINTR error return for these system calls.

The SA_RESTART flag is a per-signal setting. In other words, we can allow handlers for some signals to interrupt blocking system calls, while others permit automatic restarting of system calls.

System calls (and library functions) for which SA_RESTART is effective

Unfortunately, not all blocking system calls automatically restart as a result of specifying SA_RESTART. The reasons for this are partly historical:

- Restarting of system calls was introduced in 4.2BSD, and covered interrupted calls to *wait()* and *waitpid()*, as well as the following I/O system calls: *read()*, *readv()*, *write()*, *writev()*, and blocking *ioctl()* operations. The I/O system calls are interruptible, and hence automatically restarted by SA_RESTART only when operating on a "slow" device. Slow devices include terminals, pipes, FIFOs, and sockets. On these file types, various I/O operations may block. (By contrast, disk files don't fall into the category of slow devices, because disk I/O operations generally can be immediately satisfied via the buffer cache. If a disk I/O is required, the kernel puts the process to sleep until the I/O completes.)
- A number of other blocking system calls are derived from System V, which did not initially provide for restarting of system calls.

On Linux, the following blocking system calls (and library functions layered on top of system calls) are automatically restarted if interrupted by a signal handler established using the SA_RESTART flag:

- The system calls used to wait for a child process (Section 26.1): *wait()*, *waitpid()*, *wait3()*, *wait4()*, and *waitid()*.
- The I/O system calls *read()*, *readv()*, *write()*, *writev()*, and *ioctl()* when applied to "slow" devices. In cases where data has already been partially transferred at the

time of signal delivery, the input and output system calls will be interrupted, but return a success status: an integer indicating how many bytes were successfully transferred.

- The *open()* system call, in cases where it can block (e.g., when opening FIFOs, as described in Section 44.7).
- Various system calls used with sockets: *accept()*, *accept4()*, *connect()*, *send()*, *sendmsg()*, *sendto()*, *recv()*, *recvfrom()*, and *recvmsg()*. (On Linux, these system calls are not automatically restarted if a timeout has been set on the socket using *setsockopt()*. See the *signal(7)* manual page for details.)
- The system calls used for I/O on POSIX message queues: *mq_receive()*, *mq_timedreceive()*, *mq_send()*, and *mq_timedsend()*.
- The system calls and library functions used to place file locks: *flock()*, *fcntl()*, and *lockf()*.
- The `FUTEX_WAIT` operation of the Linux-specific *futex()* system call.
- The *sem_wait()* and *sem_timedwait()* functions used to decrement a POSIX semaphore. (On some UNIX implementations, *sem_wait() is* restarted if the `SA_RESTART` flag is specified.)
- The functions used to synchronize POSIX threads: *pthread_mutex_lock()*, *pthread_mutex_trylock()*, *pthread_mutex_timedlock()*, *pthread_cond_wait()*, and *pthread_cond_timedwait()*.

In kernels before 2.6.22, *futex()*, *sem_wait()*, and *sem_timedwait()* always failed with the error `EINTR` when interrupted, regardless of the setting of the `SA_RESTART` flag.

The following blocking system calls (and library functions layered on top of system calls) are never automatically restarted (even if `SA_RESTART` is specified):

- The *poll()*, *ppoll()*, *select()*, and *pselect()* I/O multiplexing calls. (SUSv3 explicitly states that the behavior of *select()* and *pselect()* when interrupted by a signal handler is unspecified, regardless of the setting of `SA_RESTART`.)
- The Linux-specific *epoll_wait()* and *epoll_pwait()* system calls.
- The Linux-specific *io_getevents()* system call.
- The blocking system calls used with System V message queues and semaphores: *semop()*, *semtimedop()*, *msgrcv()*, and *msgsnd()*. (Although System V did not originally provide automatic restarting of system calls, on some UNIX implementations, these system calls *are* restarted if the `SA_RESTART` flag is specified.)
- A *read()* from an *inotify* file descriptor.
- The system calls and library functions designed to suspend execution of a program for a specified period: *sleep()*, *nanosleep()*, and *clock_nanosleep()*.
- The system calls designed specifically to wait until a signal is delivered: *pause()*, *sigsuspend()*, *sigtimedwait()*, and *sigwaitinfo()*.

Modifying the `SA_RESTART` flag for a signal

The *siginterrupt()* function changes the `SA_RESTART` setting associated with a signal.

```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

Returns 0 on success, or −1 on error

If *flag* is true (1), then a handler for the signal *sig* will interrupt blocking system calls. If *flag* is false (0), then blocking system calls will be restarted after execution of a handler for *sig*.

The *siginterrupt()* function works by using *sigaction()* to fetch a copy of the signal's current disposition, tweaking the SA_RESTART flag in the returned *oldact* structure, and then calling *sigaction()* once more to update the signal's disposition.

SUSv4 marks *siginterrupt()* obsolete, recommending the use of *sigaction()* instead for this purpose.

Unhandled stop signals can generate EINTR for some Linux system calls

On Linux, certain blocking system calls can return EINTR even in the absence of a signal handler. This can occur if the system call is blocked and the process is stopped by a signal (SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU), and then resumed by delivery of a SIGCONT signal.

The following system calls and functions exhibit this behavior: *epoll_pwait()*, *epoll_wait()*, *read()* from an *inotify* file descriptor, *semop()*, *semtimedop()*, *sigtimedwait()*, and *sigwaitinfo()*.

In kernels before 2.6.24, *poll()* also exhibited this behavior, as did *sem_wait()*, *sem_timedwait()*, *futex(FUTEX_WAIT)*, in kernels before 2.6.22, *msgrcv()* and *msgsnd()* in kernels before 2.6.9, and *nanosleep()* in Linux 2.4 and earlier.

In Linux 2.4 and earlier, *sleep()* can also be interrupted in this manner, but, instead of returning an error, it returns the number of remaining unslept seconds.

The upshot of this behavior is that if there is a chance that our program may be stopped and restarted by signals, then we may need to include code to restart these system calls, even in a program that doesn't install handlers for the stop signals.

21.6 Summary

In this chapter, we considered a range of factors that affect the operation and design of signal handlers.

Because signals are not queued, a signal handler must sometimes be coded to deal with the possibility that multiple events of a particular type have occurred, even though only one signal was delivered. The issue of reentrancy affects how we can update global variables and limits the set of functions that we can safely call from a signal handler.

Instead of returning, a signal handler can terminate in a variety of other ways, including calling *_exit()*, terminating the process by sending a signal (*kill()*, *raise()*, or *abort()*), or performing a nonlocal goto. Using *sigsetjmp()* and *siglongjmp()* provides a program with explicit control of the treatment of the process signal mask when a nonlocal goto is performed.

We can use *sigaltstack()* to define an alternate signal stack for a process. This is an area of memory that is used instead of the standard process stack when invoking a signal handler. An alternate signal stack is useful in cases where the standard stack has been exhausted by growing too large (at which point the kernel sends a `SIGSEGV` signal to the process).

The *sigaction()* `SA_SIGINFO` flag allows us to establish a signal handler that receives additional information about a signal. This information is supplied via a *siginfo_t* structure whose address is passed as an argument to the signal handler.

When a signal handler interrupts a blocked system call, the system call fails with the error `EINTR`. We can take advantage of this behavior to, for example, set a timer on a blocking system call. Interrupted system calls can be manually restarted if desired. Alternatively, establishing the signal handler with the *sigaction()* `SA_RESTART` flag causes many (but not all) system calls to be automatically restarted.

Further information

See the sources listed in Section 20.15.

21.7 Exercise

21-1. Implement *abort()*.