CHAPTER 11

Signals

Signals were introduced by the first Unix systems to allow interactions between User Mode processes; the kernel also uses them to notify processes of system events. Signals have been around for 30 years with only minor changes.

The first sections of this chapter examine in detail how signals are handled by the Linux kernel, then we discuss the system calls that allow processes to exchange signals.

The Role of Signals

A *signal* is a very short message that may be sent to a process or a group of processes. The only information given to the process is usually a number identifying the signal; there is no room in standard signals for arguments, a message, or other accompanying information.

A set of macros whose names start with the prefix SIG is used to identify signals; we have already made a few references to them in previous chapters. For instance, the SIGCHLD macro was mentioned in the section "The clone(), fork(), and vfork() System Calls" in Chapter 3. This macro, which expands into the value 17 in Linux, yields the identifier of the signal that is sent to a parent process when a child stops or terminates. The SIGSEGV macro, which expands into the value 11, was mentioned in the section "Page Fault Exception Handler" in Chapter 9; it yields the identifier of the signal that is sent to a process when it makes an invalid memory reference.

Signals serve two main purposes:

- To make a process aware that a specific event has occurred
- To cause a process to execute a *signal handler* function included in its code

Of course, the two purposes are not mutually exclusive, because often a process must react to some event by executing a specific routine.

Table 11-1 lists the first 31 signals handled by Linux 2.6 for the 80×86 architecture (some signal numbers, such those associated with SIGCHLD or SIGSTOP, are

architecture-dependent; furthermore, some signals such as SIGSTKFLT are defined only for specific architectures). The meanings of the default actions are described in the next section.

Table 11-1. The first 31 signals in Linux/i386

#	Signal name	Default action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard	Yes
3	SIGQUIT	Dump	Quit from keyboard	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timerclock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated, or got signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No

Table 11-1. The first 31 signals in Linux/i386 (continued)

#	Signal name	Default action	Comment	POSIX
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

Besides the *regular signals* described in this table, the POSIX standard has introduced a new class of signals denoted as *real-time signals*; their signal numbers range from 32 to 64 on Linux. They mainly differ from regular signals because they are always queued so that multiple signals sent will be received. On the other hand, regular signals of the same kind are not queued: if a regular signal is sent many times in a row, just one of them is delivered to the receiving process. Although the Linux kernel does not use real-time signals, it fully supports the POSIX standard by means of several specific system calls.

A number of system calls allow programmers to send signals and determine how their processes respond to the signals they receive. Table 11-2 summarizes these calls; their behavior is described in detail in the later section "System Calls Related to Signal Handling."

Table 11-2. The most significant system calls related to signals

System call	Description
`kill()`	Send a signal to a thread group
`tkill()`	Send a signal to a process
`tgkill()`	Send a signal to a process in a specific thread group
`sigaction()`	Change the action associated with a signal
`signal()`	Similar to `sigaction()`
`sigpending()`	Check whether there are pending signals
`sigprocmask()`	Modify the set of blocked signals
`sigsuspend()`	Wait for a signal
`rt_sigaction()`	Change the action associated with a real-time signal
`rt_sigpending()`	Check whether there are pending real-time signals
`rt_sigprocmask()`	Modify the set of blocked real-time signals
`rt_sigqueueinfo()`	Send a real-time signal to a thread group
`rt_sigsuspend()`	Wait for a real-time signal
`rt_sigtimedwait()`	Similar to `rt_sigsuspend()`

An important characteristic of signals is that they may be sent at any time to a process whose state is usually unpredictable. Signals sent to a process that is not currently executing must be saved by the kernel until that process resumes execution. Blocking a signal (described later) requires that delivery of the signal be held off until it is later unblocked, which exacerbates the problem of signals being raised before they can be delivered.

Therefore, the kernel distinguishes two different phases related to signal transmission:

Signal generation
: The kernel updates a data structure of the destination process to represent that a new signal has been sent.

Signal delivery
: The kernel forces the destination process to react to the signal by changing its execution state, by starting the execution of a specified signal handler, or both.

Each signal generated can be delivered once, at most. Signals are consumable resources: once they have been delivered, all process descriptor information that refers to their previous existence is canceled.

Signals that have been generated but not yet delivered are called *pending signals*. At any time, only one pending signal of a given type may exist for a process; additional pending signals of the same type to the same process are not queued but simply discarded. Real-time signals are different, though: there can be several pending signals of the same type.

In general, a signal may remain pending for an unpredictable amount of time. The following factors must be taken into consideration:

- Signals are usually delivered only to the currently running process (that is, to the current process).
- Signals of a given type may be selectively *blocked* by a process (see the later section "Modifying the Set of Blocked Signals"). In this case, the process does not receive the signal until it removes the block.
- When a process executes a signal-handler function, it usually *masks* the corresponding signal—i.e., it automatically blocks the signal until the handler terminates. A signal handler therefore cannot be interrupted by another occurrence of the handled signal, and the function doesn't need to be reentrant.

Although the notion of signals is intuitive, the kernel implementation is rather complex. The kernel must:

- Remember which signals are blocked by each process.
- When switching from Kernel Mode to User Mode, check whether a signal for a process has arrived. This happens at almost every timer interrupt (roughly every millisecond).
- Determine whether the signal can be ignored. This happens when all of the following conditions are fulfilled:
  - The destination process is not traced by another process (the PT_PTRACED flag in the process descriptor ptrace field is equal to 0).*

* If a process receives a signal while it is being traced, the kernel stops the process and notifies the tracing process by sending a SIGCHLD signal to it. The tracing process may, in turn, resume execution of the traced process by means of a SIGCONT signal.

— The signal is not blocked by the destination process.

— The signal is being ignored by the destination process (either because the process explicitly ignored it or because the process did not change the default action of the signal and that action is "ignore").

- Handle the signal, which may require switching the process to a handler function at any point during its execution and restoring the original execution context after the function returns.

Moreover, Linux must take into account the different semantics for signals adopted by BSD and System V; furthermore, it must comply with the rather cumbersome POSIX requirements.

Actions Performed upon Delivering a Signal

There are three ways in which a process can respond to a signal:

1. Explicitly ignore the signal.
2. Execute the *default action* associated with the signal (see Table 11-1). This action, which is predefined by the kernel, depends on the signal type and may be any one of the following:

   Terminate
   : The process is terminated (killed).

   Dump
   : The process is terminated (killed) and a core file containing its execution context is created, if possible; this file may be used for debug purposes.

   Ignore
   : The signal is ignored.

   Stop
   : The process is stopped—i.e., put in the TASK_STOPPED state (see the section "Process State" in Chapter 3).

   Continue
   : If the process was stopped (TASK_STOPPED), it is put into the TASK_RUNNING state.

3. Catch the signal by invoking a corresponding signal-handler function.

Notice that blocking a signal is different from ignoring it. A signal is not delivered as long as it is blocked; it is delivered only after it has been unblocked. An ignored signal is always delivered, and there is no further action.

The SIGKILL and SIGSTOP signals cannot be ignored, caught, or blocked, and their default actions must always be executed. Therefore, SIGKILL and SIGSTOP allow a user

with appropriate privileges to terminate and to stop, respectively, every process,* regardless of the defenses taken by the program it is executing.

A signal is *fatal* for a given process if delivering the signal causes the kernel to kill the process. The SIGKILL signal is always fatal; moreover, each signal whose default action is "Terminate" and which is not caught by a process is also fatal for that process. Notice, however, that a signal caught by a process and whose corresponding signal-handler function terminates the process is not fatal, because the process chose to terminate itself rather than being killed by the kernel.

POSIX Signals and Multithreaded Applications

The POSIX 1003.1 standard has some stringent requirements for signal handling of multithreaded applications:

- Signal handlers must be shared among all threads of a multithreaded application; however, each thread must have its own mask of pending and blocked signals.
- The kill() and sigqueue() POSIX library functions (see the later section "System Calls Related to Signal Handling") must send signals to whole multithreaded applications, not to a specific thread. The same holds for all signals (such as SIGCHLD, SIGINT, or SIGQUIT) generated by the kernel.
- Each signal sent to a multithreaded application will be delivered to just one thread, which is arbitrarily chosen by the kernel among the threads that are not blocking that signal.
- If a fatal signal is sent to a multithreaded application, the kernel will kill all threads of the application—not just the thread to which the signal has been delivered.

In order to comply with the POSIX standard, the Linux 2.6 kernel implements a multithreaded application as a set of lightweight processes belonging to the same thread group (see the section "Processes, Lightweight Processes, and Threads" in Chapter 3).

In this chapter the term "thread group" denotes any thread group, even if it is composed by a single (conventional) process. For instance, when we state that kill() can send a signal to a thread group, we imply that this system call can send a signal to a conventional process, too. We will use the term "process" to denote either a conventional process or a lightweight process—that is, a specific member of a thread group.

Furthermore, a pending signal is *private* if it has been sent to a specific process; it is *shared* if it has been sent to a whole thread group.

* There are two exceptions: it is not possible to send a signal to process 0 (*swapper*), and signals sent to process 1 (*init*) are always discarded unless they are caught. Therefore, process 0 never dies, while process 1 dies only when the *init* program terminates.

Data Structures Associated with Signals

For each process in the system, the kernel must keep track of what signals are currently pending or masked; the kernel must also keep track of how every thread group is supposed to handle every signal. To do this, the kernel uses several data structures accessible from the process descriptor. The most significant ones are shown in Figure 11-1.

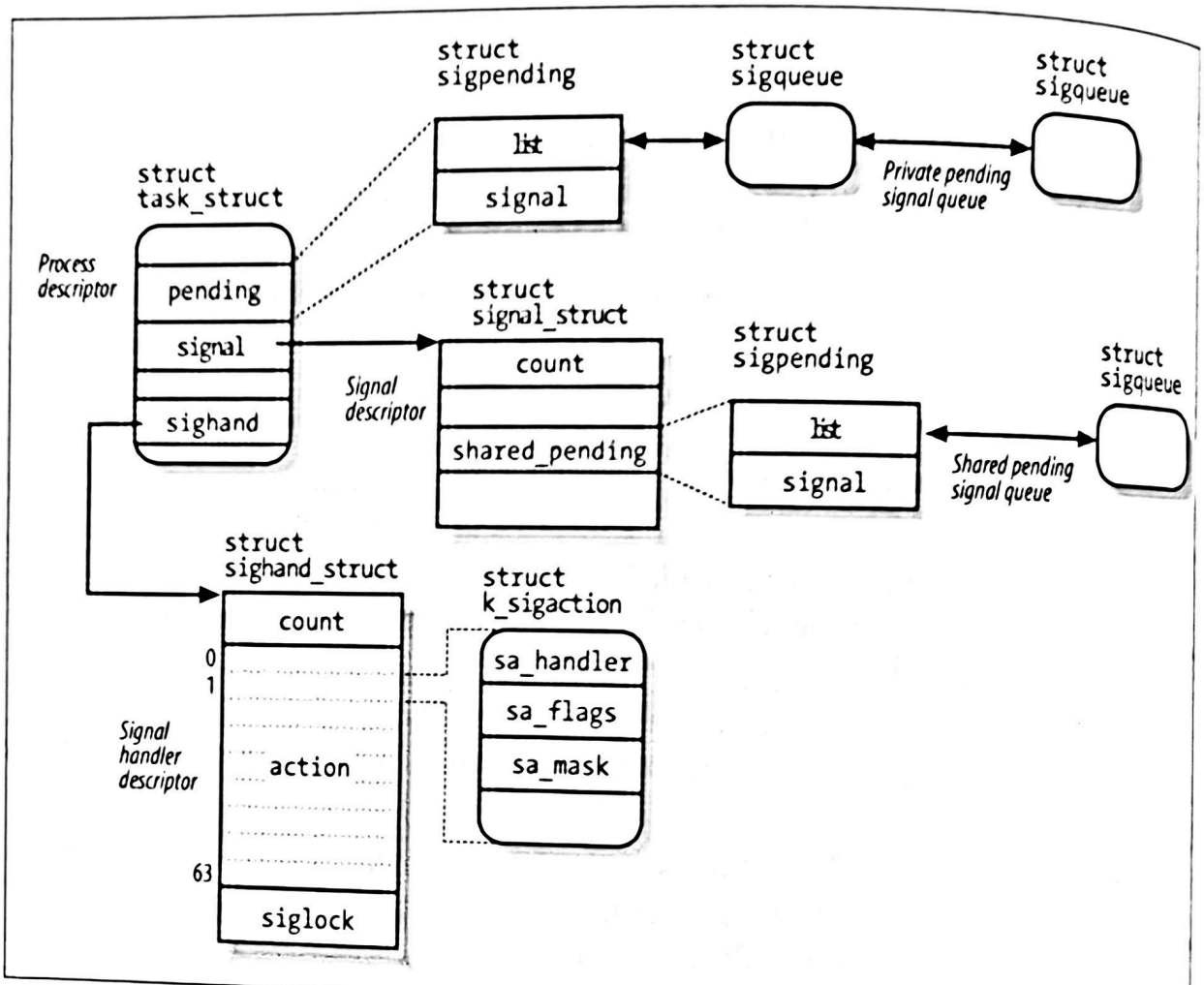

Figure 11-1. The most significant data structures related to signal handling

The fields of the process descriptor related to signal handling are listed in Table 11-3.

Table 11-3. Process descriptor fields related to signal handling

Type	Name	Description
`struct signal_struct *`	`signal`	Pointer to the process's signal descriptor
`struct sighand_struct *`	`sighand`	Pointer to the process's signal handler descriptor
`sigset_t`	`blocked`	Mask of blocked signals
`sigset_t`	`real_blocked`	Temporary mask of blocked signals (used by the `rt_sigtimedwait()` system call)
`struct sigpending`	`pending`	Data structure storing the private pending signals
`unsigned long`	`sas_ss_sp`	Address of alternative signal handler stack

Table 11-3. Process descriptor fields related to signal handling (continued)

Type	Name	Description
size_t	sas_ss_size	Size of alternative signal handler stack
int (*) (void *)	notifier	Pointer to a function used by a device driver to block some signals of the process
void *	notifier_data	Pointer to data that might be used by the notifier function (previous field of table)
sigset_t *	notifier_mask	Bit mask of signals blocked by a device driver through a notifier function

The blocked field stores the signals currently masked out by the process. It is a sigset_t array of bits, one for each signal type:

```
typedef struct {
    unsigned long sig[2];
} sigset_t;
```

Because each unsigned long number consists of 32 bits, the maximum number of signals that may be declared in Linux is 64 (the _NSIG macro specifies this value). No signal can have number 0, so the signal number corresponds to the index of the corresponding bit in a sigset_t variable plus one. Numbers between 1 and 31 correspond to the signals listed in Table 11-1, while numbers between 32 and 64 correspond to real-time signals.

The signal descriptor and the signal handler descriptor

The signal field of the process descriptor points to a *signal descriptor*, a signal_struct structure that keeps track of the shared pending signals. Actually, the signal descriptor also includes fields not strictly related to signal handling, such as the rlim per-process resource limit array (see the section "Process Resource Limits" in Chapter 3), or the pgrp and session fields, which store the PIDs of the group leader and of the session leader of the process, respectively (see the section "Relationships Among Processes" in Chapter 3). In fact, as mentioned in the section "The clone(), fork(), and vfork() System Calls" in Chapter 3, the signal descriptor is shared by all processes belonging to the same thread group—that is, all processes created by invoking the clone() system call with the CLONE_THREAD flag set—thus the signal descriptor includes the fields that must be identical for every process in the same thread group.

The fields of a signal descriptor somewhat related to signal handling are shown in Table 11-4.

Table 11-4. The fields of the signal descriptor related to signal handling

Type	Name	Description
atomic_t	count	Usage counter of the signal descriptor
atomic_t	live	Number of live processes in the thread group

Table 11-4. The fields of the signal descriptor related to signal handling (continued)

Type	Name	Description
wait_queue_head_t	wait_chldexit	Wait queue for the processes sleeping in a wait4() system call
struct task_struct *	curr_target	Descriptor of the last process in the thread group that received a signal
struct sigpending	shared_pending	Data structure storing the shared pending signals
int	group_exit_code	Process termination code for the thread group
struct task_struct *	group_exit_task	Used when killing a whole thread group
int	notify_count	Used when killing a whole thread group
int	group_stop_count	Used when stopping a whole thread group
unsigned int	flags	Flags used when delivering signals that modify the status of the process

Besides the signal descriptor, every process refers also to a *signal handler descriptor*, which is a sighand_struct structure describing how each signal must be handled by the thread group. Its fields are shown in Table 11-5.

Table 11-5. The fields of the signal handler descriptor

Type	Name	Description
atomic_t	count	Usage counter of the signal handler descriptor
struct k_sigaction [64]	action	Array of structures specifying the actions to be performed upon delivering the signals
spinlock_t	siglock	Spin lock protecting both the signal descriptor and the signal handler descriptor

As mentioned in the section "The clone(), fork(), and vfork() System Calls" in Chapter 3, the signal handler descriptor may be shared by several processes by invoking the clone( ) system call with the CLONE_SIGHAND flag set; the count field in this descriptor specifies the number of processes that share the structure. In a POSIX multithreaded application, all lightweight processes in the thread group refer to the same signal descriptor and to the same signal handler descriptor.

The sigaction data structure

Some architectures assign properties to a signal that are visible only to the kernel. Thus, the properties of a signal are stored in a k_sigaction structure, which contains both the properties hidden from the User Mode process and the more familiar sigaction structure that holds all the properties a User Mode process can see. Actually, on the 80×86 platform, all signal properties are visible to User Mode processes.

Thus the k_sigaction structure simply reduces to a single sa structure of type sigaction, which includes the following fields:*

sa_handler
: This field specifies the type of action to be performed; its value can be a pointer to the signal handler, SIG_DFL (that is, the value 0) to specify that the default action is performed, or SIG_IGN (that is, the value 1) to specify that the signal is ignored.

sa_flags
: This set of flags specifies how the signal must be handled; some of them are listed in Table 11-6.†

sa_mask
: This sigset_t variable specifies the signals to be masked when running the signal handler.

Table 11-6. Flags specifying how to handle a signal

Flag Name	Description
SA_NOCLDSTOP	Applies only to SIGCHLD; do not send SIGCHLD to the parent when the process is stopped
SA_NOCLDWAIT	Applies only to SIGCHLD; do not create a zombie when the process terminates
SA_SIGINFO	Provide additional information to the signal handler (see the later section "Changing a Signal Action")
SA_ONSTACK	Use an alternative stack for the signal handler (see the later section "Catching the Signal")
SA_RESTART	Interrupted system calls are automatically restarted (see the later section "Reexecution of System Calls")
SA_NODEFER, SA_NOMASK	Do not mask the signal while executing the signal handler
SA_RESETHAND, SA_ONESHOT	Reset to default action after executing the signal handler

The pending signal queues

As we have seen in Table 11-2 earlier in the chapter, there are several system calls that can generate a signal: some of them—kill() and rt_sigqueueinfo()—send a signal to a whole thread group, while others—tkill() and tgkill()—send a signal to a specific process.

* The sigaction structure used by User Mode applications to pass parameters to the signal() and sigaction() system calls is slightly different from the structure used by the kernel, although it stores essentially the same information.

† For historical reasons, these flags have the same prefix "SA_" as the flags of the irqaction descriptor (see Table 4-7 in Chapter 4); nevertheless there is no relation between the two sets of flags.

Thus, in order to keep track of what signals are currently pending, the kernel associates two pending signal queues to each process:

- The *shared pending signal queue*, rooted at the shared_pending field of the signal descriptor, stores the pending signals of the whole thread group.
- The *private pending signal queue*, rooted at the pending field of the process descriptor, stores the pending signals of the specific (lightweight) process.

A pending signal queue consists of a sigpending data structure, which is defined as follows:

```
struct sigpending {
    struct list_head list;
    sigset_t signal;
}
```

The signal field is a bit mask specifying the pending signals, while the list field is the head of a doubly linked list containing sigqueue data structures; the fields of this structure are shown in Table 11-7.

Table 11-7. The fields of the sigqueue data structure

Type	Name	Description
struct list_head	list	Links for the pending signal queue's list
spinlock_t *	lock	Pointer to the siglock field in the signal handler descriptor corresponding to the pending signal
int	flags	Flags of the sigqueue data structure
siginfo_t	info	Describes the event that raised the signal
struct user_struct *	user	Pointer to the per-user data structure of the process's owner (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3)

The siginfo_t data structure is a 128-byte data structure that stores information about an occurrence of a specific signal; it includes the following fields:

si_signo
: The signal number

si_errno
: The error code of the instruction that caused the signal to be raised, or 0 if there was no error

si_code
: A code identifying who raised the signal (see Table 11-8)

Table 11-8. The most significant signal sender codes

Code Name	Sender
SI_USER	kill( ) and raise( ) (see the later section "System Calls Related to Signal Handling")
SI_KERNEL	Generic kernel function
SI_QUEUE	sigqueue( ) (see the later section "System Calls Related to Signal Handling")

Table 11-8. The most significant signal sender codes (continued)

Code Name	Sender
SI_TIMER	Timer expiration
SI_ASYNCIO	Asynchronous I/O completion
SI_TKILL	tkill( ) and tgkill( ) (see the later section "System Calls Related to Signal Handling")

_sifields
: A union storing information depending on the type of signal. For instance, the siginfo_t data structure relative to an occurrence of the SIGKILL signal records the PID and the UID of the sender process here; conversely, the data structure relative to an occurrence of the SIGSEGV signal stores the memory address whose access caused the signal to be raised.

Operations on Signal Data Structures

Several functions and macros are used by the kernel to handle signals. In the following description, set is a pointer to a sigset_t variable, nsig is the number of a signal, and mask is an unsigned long bit mask.

sigemptyset(set) *and* sigfillset(set)
: Sets the bits in the sigset_t variable to 0 or 1, respectively.

sigaddset(set,nsig) *and* sigdelset(set,nsig)
: Sets the bit of the sigset_t variable corresponding to signal nsig to 1 or 0, respectively. In practice, sigaddset( ) reduces to:

```
set->sig[(nsig - 1) / 32] |= 1UL << ((nsig - 1) % 32);
```

and sigdelset( ) to:

```
set->sig[(nsig - 1) / 32] &= ~(1UL << ((nsig - 1) % 32));
```

sigaddsetmask(set,mask) *and* sigdelsetmask(set,mask)
: Sets all the bits of the sigset_t variable whose corresponding bits of mask are on 1 or 0, respectively. They can be used only with signals that are between 1 and 32. The corresponding functions reduce to:

```
set->sig[0] |= mask;
```

and to:

```
set->sig[0] &= ~mask;
```

sigismember(set,nsig)
: Returns the value of the bit of the sigset_t variable corresponding to the signal nsig. In practice, this function reduces to:

```
return 1 & (set->sig[(nsig-1) / 32] >> ((nsig-1) % 32));
```

sigmask(nsig)
: Yields the bit index of the signal nsig. In other words, if the kernel needs to set, clear, or test a bit in an element of sigset_t that corresponds to a particular signal, it can derive the proper bit through this macro.

`sigandsets(d,s1,s2)`, `sigorsets(d,s1,s2)`, *and* `signandsets(d,s1,s2)`
: Performs a logical AND, a logical OR, and a logical NAND, respectively, between the `sigset_t` variables to which `s1` and `s2` point; the result is stored in the `sigset_t` variable to which `d` points.

`sigtestsetmask(set,mask)`
: Returns the value 1 if any of the bits in the `sigset_t` variable that correspond to the bits set to 1 in `mask` is set; it returns 0 otherwise. It can be used only with signals that have a number between 1 and 32.

`siginitset(set,mask)`
: Initializes the low bits of the `sigset_t` variable corresponding to signals between 1 and 32 with the bits contained in `mask`, and clears the bits corresponding to signals between 33 and 63.

`siginitsetinv(set,mask)`
: Initializes the low bits of the `sigset_t` variable corresponding to signals between 1 and 32 with the complement of the bits contained in `mask`, and sets the bits corresponding to signals between 33 and 63.

`signal_pending(p)`
: Returns the value 1 (true) if the process identified by the `*p` process descriptor has nonblocked pending signals, and returns the value 0 (false) if it doesn't. The function is implemented as a simple check on the `TIF_SIGPENDING` flag of the process.

`recalc_sigpending_tsk(t)` *and* `recalc_sigpending()`
: The first function checks whether there are pending signals either for the process identified by the process descriptor at `*t` (by looking at the `t->pending->signal` field) or for the thread group to which the process belongs (by looking at the `t->signal->shared_pending->signal` field). The function then sets accordingly the `TIF_SIGPENDING` flag in `t->thread_info->flags`. The `recalc_sigpending()` function is equivalent to `recalc_sigpending_tsk(current)`.

`rm_from_queue(mask,q)`
: Removes from the pending signal queue `q` the pending signals corresponding to the bit mask `mask`.

`flush_sigqueue(q)`
: Removes from the pending signal queue `q` all pending signals.

`flush_signals(t)`
: Deletes all signals sent to the process identified by the process descriptor at `*t`. This is done by clearing the `TIF_SIGPENDING` flag in `t->thread_info->flags` and invoking twice `flush_sigqueue()` on the `t->pending` and `t->signal->shared_pending` queues.

Generating a Signal

Many kernel functions generate signals: they accomplish the first phase of signal handling—described earlier in the section "The Role of Signals"—by updating one or more process descriptors as needed. They do not directly perform the second phase of delivering the signal but, depending on the type of signal and the state of the destination processes, may wake up some processes and force them to receive the signal.

When a signal is sent to a process, either from the kernel or from another process, the kernel generates it by invoking one of the functions listed in Table 11-9.

Table 11-9. Kernel functions that generate a signal for a process

Name	Description
`send_sig()`	Sends a signal to a single process
`send_sig_info()`	Like `send_sig()`, with extended information in a `siginfo_t` structure
`force_sig()`	Sends a signal that cannot be explicitly ignored or blocked by the process
`force_sig_info()`	Like `force_sig()`, with extended information in a `siginfo_t` structure
`force_sig_specific()`	Like `force_sig()`, but optimized for `SIGSTOP` and `SIGKILL` signals
`sys_tkill()`	System call handler of `tkill()` (see the later section "System Calls Related to Signal Handling")
`sys_tgkill()`	System call handler of `tgkill()`

All functions in Table 11-9 end up invoking the `specific_send_sig_info()` function described in the next section.

When a signal is sent to a whole thread group, either from the kernel or from another process, the kernel generates it by invoking one of the functions listed in Table 11-10.

Table 11-10. Kernel functions that generate a signal for a thread group

Name	Description
`send_group_sig_info()`	Sends a signal to a single thread group identified by the process descriptor of one of its members
`kill_pg()`	Sends a signal to all thread groups in a process group (see the section "Process Management" in Chapter 1)
`kill_pg_info()`	Like `kill_pg()`, with extended information in a `siginfo_t` structure
`kill_proc()`	Sends a signal to a single thread group identified by the PID of one of its members
`kill_proc_info()`	Like `kill_proc()`, with extended information in a `siginfo_t` structure
`sys_kill()`	System call handler of `kill()` (see the later section "System Calls Related to Signal Handling")
`sys_rt_sigqueueinfo()`	System call handler of `rt_sigqueueinfo()`

do_group_exit(), which executes a clean "group exit" procedure (see the section "Process Termination" in Chapter 3).

Catching the Signal

If a handler has been established for the signal, the do_signal() function must enforce its execution. It does this by invoking handle_signal():

```
handle_signal(signr, &info, &ka, oldset, regs);
if (ka->sa.sa_flags & SA_ONESHOT)
    ka->sa.sa_handler = SIG_DFL;
return 1;
```

If the received signal has the SA_ONESHOT flag set, it must be reset to its default action, so that further occurrences of the same signal will not trigger again the execution of the signal handler. Notice how do_signal() returns after having handled a single signal. Other pending signals won't be considered until the next invocation of do_signal(). This approach ensures that real-time signals will be dealt with in the proper order.

Executing a signal handler is a rather complex task because of the need to juggle stacks carefully while switching between User Mode and Kernel Mode. We explain exactly what is entailed here:

Signal handlers are functions defined by User Mode processes and included in the User Mode code segment. The handle_signal() function runs in Kernel Mode while signal handlers run in User Mode; this means that the current process must first execute the signal handler in User Mode before being allowed to resume its "normal" execution. Moreover, when the kernel attempts to resume the normal execution of the process, the Kernel Mode stack no longer contains the hardware context of the interrupted program, because the Kernel Mode stack is emptied at every transition from User Mode to Kernel Mode.

An additional complication is that signal handlers may invoke system calls. In this case, after the service routine executes, control must be returned to the signal handler instead of to the normal flow of code of the interrupted program.

The solution adopted in Linux consists of copying the hardware context saved in the Kernel Mode stack onto the User Mode stack of the current process. The User Mode stack is also modified in such a way that, when the signal handler terminates, the sigreturn() system call is automatically invoked to copy the hardware context back on the Kernel Mode stack and to restore the original content of the User Mode stack.

Figure 11-2 illustrates the flow of execution of the functions involved in catching a signal. A nonblocked signal is sent to a process. When an interrupt or exception occurs, the process switches into Kernel Mode. Right before returning to User Mode, the kernel executes the do_signal() function, which in turn handles the signal (by invoking handle_signal()) and sets up the User Mode stack (by invoking setup_frame() or setup_rt_frame()). When the process switches again to User Mode, it

starts executing the signal handler, because the handler's starting address was forced into the program counter. When that function terminates, the return code placed on the User Mode stack by the setup_frame( ) or setup_rt_frame( ) function is executed. This code invokes the sigreturn( ) or the rt_sigreturn( ) system call; the corresponding service routines copy the hardware context of the normal program to the Kernel Mode stack and restore the User Mode stack back to its original state (by invoking restore_sigcontext( )). When the system call terminates, the normal program can thus resume its execution.

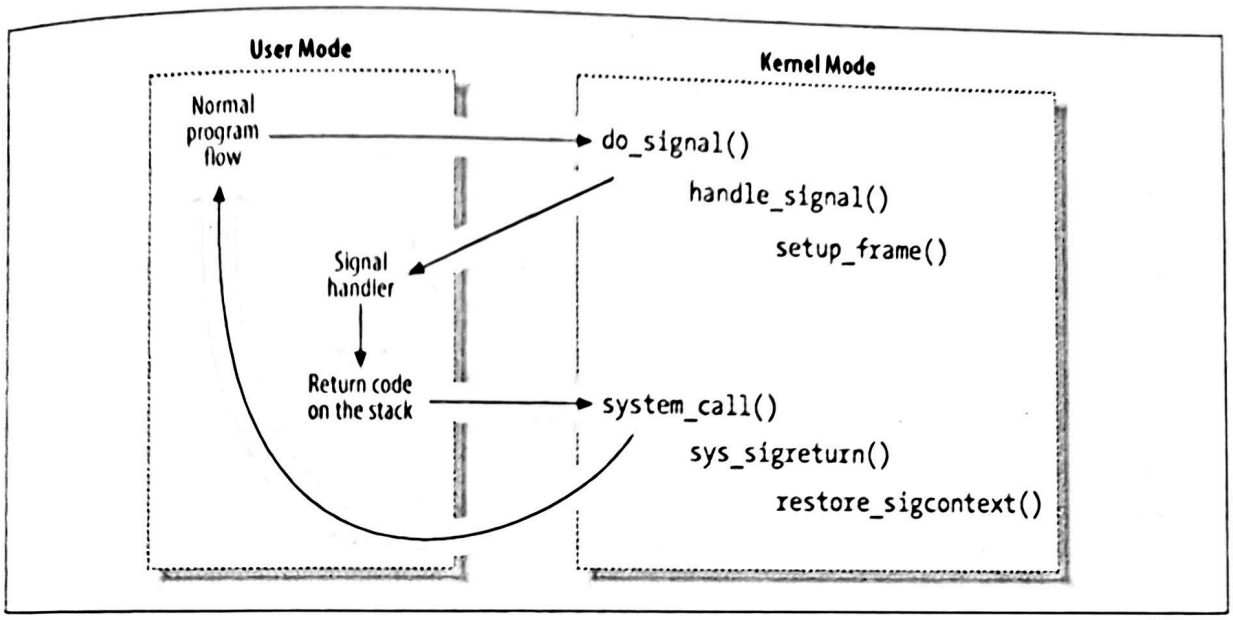

Figure 11-2. Catching a signal

Let's now examine in detail how this scheme is carried out.

Setting up the frame

To properly set the User Mode stack of the process, the handle_signal( ) function invokes either setup_frame( ) (for signals that do not require a siginfo_t table; see the section "System Calls Related to Signal Handling" later in this chapter) or setup_rt_frame( ) (for signals that do require a siginfo_t table). To choose among these two functions, the kernel checks the value of the SA_SIGINFO flag in the sa_flags field of the sigaction table associated with the signal.

The setup_frame( ) function receives four parameters, which have the following meanings:

sig
: Signal number

ka
: Address of the k_sigaction table associated with the signal

oldset
: Address of a bit mask array of blocked signals

Address in the Kernel Mode stack area where the User Mode register contents are saved

The `setup_frame()` function pushes onto the User Mode stack a data structure called a *frame*, which contains the information needed to handle the signal and to ensure the correct return to the `sys_sigreturn()` function. A frame is a `sigframe` table that includes the following fields (see Figure 11-3):

`pretcode`
: Return address of the signal handler function; it points to the code at the `__kernel_sigreturn` label (see below).

`sig`
: The signal number; this is the parameter required by the signal handler.

`sc`
: Structure of type `sigcontext` containing the hardware context of the User Mode process right before switching to Kernel Mode (this information is copied from the Kernel Mode stack of current). It also contains a bit array that specifies the blocked regular signals of the process.

`fpstate`
: Structure of type `_fpstate` that may be used to store the floating point registers of the User Mode process (see the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3).

`extramask`
: Bit array that specifies the blocked real-time signals.

`retcode`
: 8-byte code issuing a `sigreturn()` system call. In earlier versions of Linux, this code was effectively executed to return from the signal handler; in Linux 2.6, however, it is used only as a signature, so that debuggers can recognize the signal stack frame.

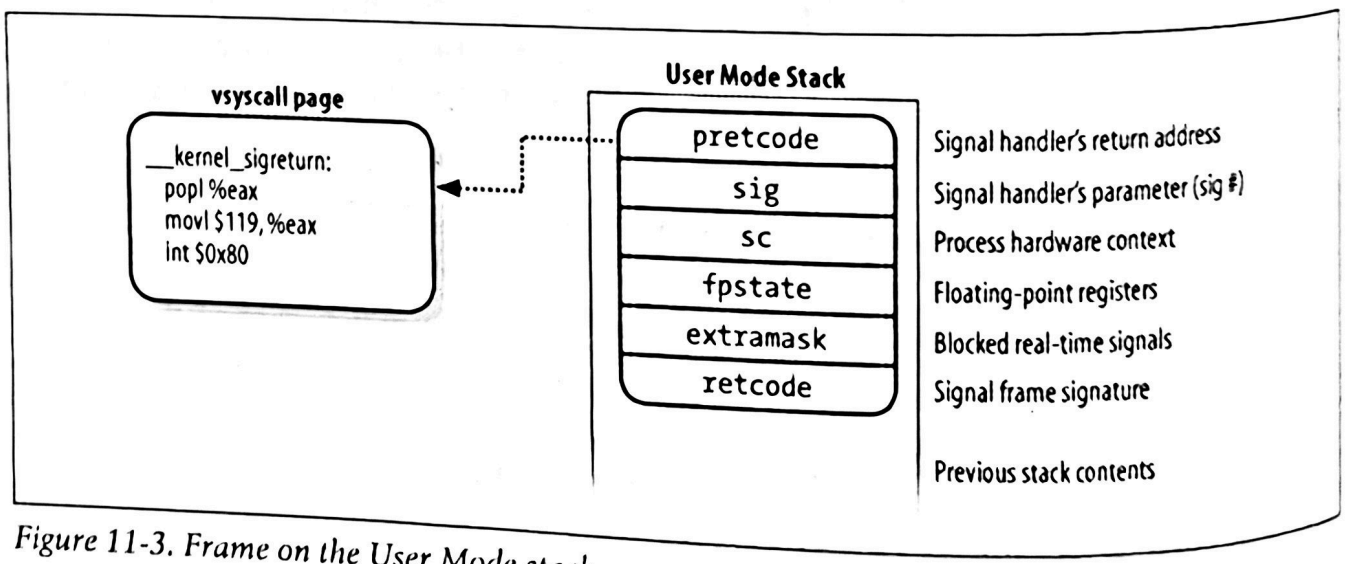

Figure 11-3. Frame on the User Mode stack

The setup_frame( ) function starts by invoking get_sigframe( ) to compute the first memory location of the frame. That memory location is usually* in the User Mode stack, so the function returns the value:

```
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
```

Because stacks grow toward lower addresses, the initial address of the frame is obtained by subtracting its size from the address of the current stack top and aligning the result to a multiple of 8.

The returned address is then verified by means of the access_ok macro; if it is valid, the function repeatedly invokes __put_user( ) to fill all the fields of the frame. The pretcode field in the frame is initialized to &__kernel_sigreturn, the address of some glue code placed in the vsyscall page (see the section "Issuing a System Call via the sysenter Instruction" in Chapter 10).

Once this is done, the function modifies the regs area of the Kernel Mode stack, thus ensuring that control is transferred to the signal handler when current resumes its execution in User Mode:

```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
regs->eax = (unsigned long) sig;
regs->edx = regs->ecx = 0;
regs->xds = regs->xes = regs->xss = __USER_DS;
regs->xcs = __USER_CS;
```

The setup_frame( ) function terminates by resetting the segmentation registers saved on the Kernel Mode stack to their default value. Now the information needed by the signal handler is on the top of the User Mode stack.

The setup_rt_frame( ) function is similar to setup_frame( ), but it puts on the User Mode stack an *extended frame* (stored in the rt_sigframe data structure) that also includes the content of the siginfo_t table associated with the signal. Moreover, this function sets the pretcode field so that it points to the __kernel_rt_sigreturn code in the vsyscall page.

Evaluating the signal flags

After setting up the User Mode stack, the handle_signal( ) function checks the values of the flags associated with the signal. If the signal does not have the SA_NODEFER flag set, the signals in the sa_mask field of the sigaction table must be blocked during the execution of the signal handler:

```
if (!(ka->sa.sa_flags & SA_NODEFER)) {
    spin_lock_irq(&current->sighand->siglock);
```

* Linux allows processes to specify an alternative stack for their signal handlers by invoking the sigaltstack( ) system call; this feature is also required by the X/Open standard. When an alternative stack is present, the get_sigframe( ) function returns an address inside that stack. We don't discuss this feature further, because it is conceptually similar to regular signal handling.

```
        sigorsets(&current->blocked, &current->blocked, &ka->sa.sa_mask);
        sigaddset(&current->blocked, sig);
        recalc_sigpending(current);
        spin_unlock_irq(&current->sighand->siglock);
    }
```

As described earlier, the `recalc_sigpending()` function checks whether the process has nonblocked pending signals and sets its `TIF_SIGPENDING` flag accordingly.

The function returns then to `do_signal()`, which also returns immediately.

Starting the signal handler

When `do_signal()` returns, the current process resumes its execution in User Mode. Because of the preparation by `setup_frame()` described earlier, the `eip` register points to the first instruction of the signal handler, while `esp` points to the first memory location of the frame that has been pushed on top of the User Mode stack. As a result, the signal handler is executed.

Terminating the signal handler

When the signal handler terminates, the return address on top of the stack points to the code in the vsyscall page referenced by the `pretcode` field of the frame:

```
__kernel_sigreturn:
  popl %eax
  movl $__NR_sigreturn, %eax
  int $0x80
```

Therefore, the signal number (that is, the `sig` field of the frame) is discarded from the stack; the `sigreturn()` system call is then invoked.

The `sys_sigreturn()` function computes the address of the `pt_regs` data structure `regs`, which contains the hardware context of the User Mode process (see the section "Parameter Passing" in Chapter 10). From the value stored in the `esp` field, it can thus derive and check the frame address inside the User Mode stack:

```
frame = (struct sigframe *)(regs.esp - 8);
if (verify_area(VERIFY_READ, frame, sizeof(*frame)) {
    force_sig(SIGSEGV, current);
    return 0;
}
```

Then the function copies the bit array of signals that were blocked before invoking the signal handler from the `sc` field of the frame to the `blocked` field of `current`. As a result, all signals that have been masked for the execution of the signal handler are unblocked. The `recalc_sigpending()` function is then invoked.

The `sys_sigreturn()` function must at this point copy the process hardware context from the `sc` field of the frame to the Kernel Mode stack and remove the frame from