Signals

Signals are triggered by events and are posted on a process to notify it that something has happened and requires some action. An event can be generated from a process, a user, or the UNIX kernel. For example, if a process performs a divide-by-zero mathematical operation, or dereferences a NULL pointer, the kernel will send the process a signal to interrupt it. Furthermore, if a user hits the <Delete> or <Ctrl-C> key at the keyboard, the kernel will send the foreground process a signal to interrupt it. Finally, a parent and its child processes can send signals to each other for process synchronization. Thus, signals are the software version of hardware interrupts. Just as there are several levels of hardware interrupts on any given system, there are also different types of signals defined for different events that may occur in a UNIX system.

Signals are defined as integer flags, and the <signal.h> header depicts the list of signals defined for a UNIX system. The table below lists the POSIX-defined signals that are commonly found in most UNIX systems.

Signal name	Use	Core file generated at default
SIGALRM	Alarm timer time-outs. Can be generated by the alarm() API	No
SIGABRT	Abort process execution. Can be generated by the abort() API	Yes
SIGFPE	Illegal mathematical operation	Yes
SIGHUP	Controlling terminal hang-up	No
SIGILL	Execution of an illegal machine instruction	Yes
SIGINT	Process interruption. Commonly generated by the <Delete> or <ctrl-C> keys	No
SIGKILL	Sure kill a process. Can be generated by the *kill -9 <process_id>* command	Yes

Signal name	Use	Core file generated at default
SIGPIPE	Illegal write to a pipe	Yes
SIGQUIT	Process quit. Commonly generated by a control-\ keys	Yes
SIGSEGV	Segmentation fault. Can be generated by de-referencing a NULL pointer	Yes
SIGTERM	Process termination. Can be generated by the "kill <process_id>" command	Yes
SIGUSR1	Reserved to be defined by users	No
SIGUSR2	Reserved to be defined by users	No
SIGCHLD	Sent to a parent process when its child process has terminated	No
SIGCONT	Resume execution of a stopped process	No
SIGSTOP	Stop a process execution	No
SIGTTIN	Stop a background process when it tries to read from its controlling terminal	No
SIGTSTP	Stop a process execution by the control-Z keys	No
SIGTTOU	Stop a background process when it tries to write to its controlling terminal	No

When a signal is sent to a process, it is *pending* on the process to handle it. The process can react to pending signals in one of three ways:

- Accepts the default action of the signal, which for most signals will terminate the process
- Ignore the signal. The signal will be discarded and it has no effect whatsoever on the recipient process
- Invoke a user-defined function. The function is known as a signal handler routine and the signal is said to be *caught* when this function is called. If the function finishes its execution without terminating the process, the process will continue execution from the point it was interrupted by the signal

A process may set up per signal handling mechanisms, such that it ignores some signals, catches some other signals, and accepts the default action from the remaining signals. Furthermore, a process may change the handling of certain signals in its course of execution. For example, a signal may be ignored in the beginning, then set to be caught, and after being caught, set to accept the default action. A signal is said to have been *delivered* if it has been reacted to by the recipient process.

The default action for most signals is to terminate a recipient process (exceptions are the SIGCHLD and SIGPWR signals). Furthermore, some signals will generate a core file for the aborted process so that users can trace back the state of the process when it was aborted. These signals are usually generated when there is an implied program error in the aborted process. For example, the SIGSEGV signal is generated when a process tries to de-reference a NULL pointer. Thus if the process accepts the default action of SIGSEGV, a core file is generated when the process is aborted and the user can use the core file to debug the program.

Most signals can be ignored or caught except the SIGKILL and SIGSTOP signals. The SIGKILL signal can be generated by a user via the *kill -9 <process ID>* command in a UNIX shell. The SIGSTOP signal halts a process execution. For example, when you type <ctrl-Z> at the keyboard, the kernel will send the SIGSTOP signal to the foreground process to stop it. A companion signal to SIGSTOP is SIGCONT, which resumes a process execution after it has been stopped. SIGSTOP and SIGCONT signals are used for job control in UNIX.

A process is allowed to ignore certain signals so that it is not interrupted while doing certain mission-critical work. For example, when a database management process is updating a database file, it should not be interrupted until it is finished, otherwise, the database file will be corrupted. Thus, this process should specify that all common interrupt signals (e.g., SIGINT and SIGTERM) are to be ignored before it starts updating the database file. It should restore signal handling actions for these signals afterward.

Because most signals are generated asynchronously to a process, a process may specify a per signal handler function. These functions are called when their corresponding signals are caught. A common practice of a signal handler function is to clean up a process work environment, such as closing all input and output files, before terminating the process gracefully.

9.1 The UNIX Kernel Supports of Signals

In UNIX System V.3, each entry in the kernel Process Table slot has an array of signal flags, one for each signal defined in the system. When a signal is generated for a process, the kernel will set the corresponding signal flag in the Process Table slot of the recipient process. Furthermore, if the recipient process is asleep (for example, it is waiting for a child process to terminate or is executing the *pause* API), the kernel will awaken the process by scheduling it as well. When the recipient process runs, the kernel will check the process *U*-area that contains an array of signal handling specifications, where each entry of the array corresponds to a signal defined in the system. The kernel will consult the array to find out how the process will react to the pending signal. If the array entry for the signal contains a zero value, the process will accept the default action of the signal. If the array entry contains a 1 value, the process will ignore the signal, and the kernel will discard it. Finally, if the array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine. The kernel will set up the process to execute that function immediately, and the process will return to its current point of execution (or to someplace else if the signal handler does a long jump) if the signal handler function does not terminate the process.

If there are different signals pending on a process, the order in which they are sent to a recipient process is undefined. Furthermore, if multiple instances of a signal are pending on a process, it is implementation-dependent on whether a single instance or multiple instances of the signal will be delivered to the process. In UNIX System V.3, each signal flag in a Process Table slot records only whether a signal is pending, but not how many of them are present.

The way caught signals are handled by UNIX System V.2 and by earlier versions has been criticized as unreliable. Subsequently, BSD UNIX 4.2 (and later versions) and POSIX.1 use different mechanisms to handle caught signals.

Specifically, in UNIX System V.2 and earlier versions, when a signal is caught, the kernel will first reset the signal handler (for that signal) in the recipient process *U*-area, then call the user signal handling function specified for that signal. Thus, if there are multiple instances of a signal being sent to a process at different points, the process will catch only the first instance of the signal. All subsequent instances of the signal will be handled in the default manner.

For a process to continuously catch multiple occurrences of a signal, the process must reinstall the signal handler function every time the signal is caught. However, this is still not a guarantee that the process will catch the signal every time: between the time a signal handler is invoked for a caught signal *X* and the time the signal handler method is reestablished, the process is in a state of accepting the default action for signal *X*. If another instance of signal *X* is delivered to the process during that interval, the process will have to handle the signal in the default manner. This is a race condition, where two events occur simultaneously, and which event will take effect first is unpredictable.

To remedy the unreliability of signal handling in System V.2, BSD UNIX 4.2 (and later versions) and POSIX.1 use a different method: When a signal is caught, the kernel does not reset the signal handler, so there is no need for the process to reestablish the signal handling method. Furthermore, the kernel will block further delivery of the same signal to the process until the signal handler function has completed execution. This ensures that the signal handler function will not be invoked recursively for multiple instances of the same signal. System V.3 introduced the *sigset* API, which behaves in such a reliable manner also.

UNIX System V.4 has adopted the POSIX.1 signal handling method. However, users still have the option to instruct the kernel to use the System V.2 signal handling method on a per-signal basis. This is done via the *signal* APIs, as described next.

9.2 signal

All UNIX systems and ANSI C support the *signal* API, which can be used to define the per-signal handling method. The function prototype of the *signal* API is:

```
#include <signal.h>

void (*signal ( int signal_num, void (*handler)(int) ) ) (int);
```

The formal arguments of the API are: *signal_num* is a signal identifier like SIGINT or SIGTERM, as defined in the <signal.h> header. The *handler* argument is the function pointer of a user-defined signal handler function. This function should take an integer formal argument and does not return any value.

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The *pause* API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include <iostream.h>
#include <signal.h>
/* Signal handler function */
void catch_sig( int sig_num )
{
      signal (sig_num, catch_sig);
      cout << "catch_sig: " << sig_nm << endl;
}
/* Main function */
int main()
{
      signal (SIGTERM, catch_sig);
      signal (SIGINT, SIG_IGN);
      signal (SIGSEGV, SIG_DFL);
      pause();              /* wait for a signal interruption */
}
```

The SIG_IGN and SIG_DFL are manifest constants defined in the <signal.h> header:

```
#define SIG_DFL     void (*)(int)0
#define SIG_IGN     void (*)(int)1
```

The *SIG_IGN* specifies a signal is to be ignored, which means that if the signal is generated to the process, it will be discarded without any interruption of the process.

The *SIG_DFL* specifies to accept the default action of a signal.

The return value of the *signal* API is the previous signal handler for a signal. This can be used to restore the signal handler for a signal after it has been altered:

```
#include <signal.h>
int main()
{
```

```
    void (*old_handler)(int) = signal (SIGINT, SIG_IGN);
    /* do mission critical processing */
    signal (SIGINT, old_handler);   /* restore previous signal handling */
}
```

The *signal* API is not a POSIX.1 standard. However, it is defined by ANSI-C and is available on all UNIX systems. Because the behavior of the *signal* API in System V.2 and earlier versions is different than that in BSD and POSIX.1 systems, it is not recommended to be used by portable applications.The BSD UNIX and the POSIX.1 define a new set of APIs for signal manipulation. The API's behavior is consistent in all UNIX and POSIX.1 systems that support them and they are described in the next two sections.

Note that UNIX System V.3 and V.4 support the *sigset* API, which has the same prototype and similar use as *signal*:

```
#include <signal.h>
void (*sigset ( int signal_num, void (*handler)(int) ) ) (int);
```

The *sigset* arguments and return value are the same as that of *signal*. Both functions set signal handling methods for any named signal. However, whereas the *signal* API is unreliable (as explained in Section 9.1), the *sigset* API is reliable. This means that when a signal is set to be caught by a signal handler via *sigset*, when multiple instances of the signal arrive one of them is handled while the other instances are blocked. Furthermore, the signal handler is not reset to SIG_DFT when the it is invoked.

9.3 Signal Mask

Each process in a UNIX (BSD 4.2 and later, and System V.4) or POSIX.1 system has a signal mask that defines which signals are blocked when generated to a process. A blocked signal depends on the recipient process to unblock it and handle it accordingly. If a signal is specified to be ignored and blocked, it is implementation dependent on whether such a signal will be discarded or left pending when it is sent to the process.

A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the *sigprocmask* API:

```
#include <signal.h>
int sigprocmask ( int cmd, const sigset_t *new_mask, sigset_t* old_mask );
```

The *new_mask* argument defines a set of signals to be set or reset in a calling process signal mask, and the *cmd* argument specifies how the *new_mask* value is to be used by the API. The possible values of *cmd* and the corresponding use of the *new_mask* value are:

***cmd* value**	**Meaning**
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the *new_mask* argument
SIG_BLOCK	Adds the signals specified in the *new_mask* argument to the calling process signal mask
SIG_UNBLOCK	Removes the signals specified in the *new_mask* argument from the calling process signal mask

If the actual argument to *new_mask* argument is a NULL pointer, the *cmd* argument will be ignored, and the current process signal mask will not be altered.

The *old_mask* argument is the address of a *sigset_t* variable that will be assigned the calling process's original signal mask prior to a *sigprocmask* call. If the actual argument to *old_mask* is a NULL pointer, no previous signal mask will be returned.

The return value of a *sigprocmask* call is zero if it succeeds or -1 if it fails. Possible failure may occur because the *new_mask* and/or the *old_mask* actual arguments are invalid addresses.

The *sigset_t* is a data type defined in the <signal.h> header. It contains a collect of bit-flags, with each bit-flag representing one signal defined in a given system.

The BSD UNIX and POSIX.1 define a set of API known as *sigsetops* functions, which set, reset, and query the presence of signals in a *sigset_t*-typed variable:

```
#include <signal.h>

int sigemptyset ( sigset_t* sigmask );
int sigaddset ( sigset_t* sigmask, const int signal_num );
int sigdelset ( sigset_t* sigmask, const int signal_num);
int sigfillset ( sigset_t* sigmask );
int sigismember ( const sigset_t* sigmask, const int signal_num);
```

The *sigemptyset* API clears all signal flags in the *sigmask* argument.

The *sigaddset* API sets the flag corresponding to the *signal_num* signal in the *sigmask* argument.

The *sigdelset* API clears the flag corresponding to the *signal_num* signal in the *sigmask* argument.

The *sigfillset* API sets all the signal flags in the *sigmask* argument.

The return value of the *sigemptyset, sigaddset, sigdelset,* and *sigfillset* calls is zero if the calls succeed or -1 if they fail. Possible causes of failure may be that the *sigmask* and/or the *signal_num* arguments are invalid.

The *sigismember* API returns 1 if the flag corresponding to the *signal_num* signal in the *sigmask* argument is set, zero if it is not set, and -1 if the call fails.

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there. Then it clears the SIGSEGV signal from the process signal mask:

```
#include <stdio.h>
#include <signal.h>
int main ()
{
        sigset_t        sigmask;
        sigemptyset(&sigmask);                  /* initialize set */

        if (sigprocmask(0, 0, &sigmask)==-1) {  /* get current signal mask */
            perror("sigprocmask");
            exit(1);
        }
        else  sigaddset(&sigmask, SIGINT);      /* set SIGINT flag */

        sigdelset(&sigmask,SIGSEGV);            /* clear SIGSEGV flag */
        if (sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
            perror("sigprocmask");              /* set a new signal mask */
}
```

When one or more signals are pending for a process and are unblocked via the *sigprocmask* API, the signal handler methods for those signals that are in effect at the time of the *sigprocmask* call will be applied before the API is returned to the caller. If there are multiple

instances of the same signal pending for the process, it is implementation-dependent whether one or all of those instances will be delivered to the process.

A process can query which signals are pending for it via the *sigpending* API:

```
#include <signal.h>
int sigpending ( sigset_t* sigmask );
```

The *sigmask* argument to the *sigpending* API is the address of a *sigset_t*-typed variable and is assigned the set of signals pending for the calling process by the API. The API returns a zero if it succeeds and a -1 value if it fails.

The *sigpending* API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the *sigprocmask* API to unblock them.

The following example reports to the console whether the SIGTERM signal is pending for the process:

```
#include <iostream.h>
#include <stdio.h>
#include <signal.h>

int main()
{
        sigset_t        sigmask;
        sigemptyset(&sigmask);
        if (sigpending(&sigmask)==-1)
           perror( "sigpending");
        else cout << "SIGTERM signal is: "
           << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set" )
           << endl;
}
```

Note that, in addition to the above APIs, UNIX System V.3 and V.4 also support the following APIs as simplified means for signal mask manipulation:

```
#include <signal.h>

int sighold (int signal_num );
int sigrelse ( int signal_num );
int sigignore ( int signal_num);
int sigpause ( int signal_num );
```

The *sighold* API adds the named signal *signal_num* to the calling process signal mask. It is the same as using the *sigset* API with the SIG_HOLD action:

```
sigset ( <signal_num>, SIG_HOLD );
```

The *sigrelse* API removes the named signal *signal_num* for the calling process signal mask.

The *sigignore* API sets the signal handling method for the named signal *signal_num* to SIG_DFT.

Finally, the *sigpause* API removes the named *signal* signal_num from the calling process signal mask and suspends the process until it is interrupted by a signal.

9.4 sigaction

The *sigaction* API is a replacement for the signal API in the latest UNIX and POSIX systems. Like the *signal* API, the *sigaction* API is called by a process to setup a signal handling method for each signal it wants to deal with. Both APIs pass back the previous signal handling method for a given signal. Furthermore, the *sigaction* API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The *sigaction* API prototype is:

```
#include <signal.h>
int sigaction ( int signal_num, struct sigaction* action,
                                struct sigaction* old_action);
```

The *struct sigaction* data type is defined in the <signal.h> header as:

```
struct sigaction
{
    void        (*sa_handler) (int);
    sigset_t    sa_mask;
    int         sa_flag;
};
```

function pointer of a user defined signal handler function

The *sa_handler* field corresponds to the second argument of the *signal* API. It can be set to SIG_IGN, SIG_DFL, or a user-defined signal handler function. The *sa_mask* field specifies additional signals that a process wishes to block (besides those signals currently specified in the process's signal mask and the *signal_num* signal) when it is handling the *signal_num* signal.

Putting all these together, the *signal_num* argument designates which signal handling action is defined in the *action* argument. The previous signal handling method for *signal_num* will be returned via the *old_action* argument if it is not a NULL pointer. If the *action* argument is a NULL pointer, the calling process's existing signal handling method for *signal_num* will be unchanged.

The following *sigaction.C* program illustrates uses of *sigaction*:

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

void callme( int sig_num )
{
    cout << "catch signal: " << sig_num << endl;
}

int main( int argc, char* argv[] )
{
    sigset_t    sigmask;
    struct sigaction    action, old_action;

    sigemptyset(&sigmask);

    if (sigaddset( &sigmask, SIGTERM)==-1 ||
        sigprocmask(SIG_SETMASK, &sigmask,0)==-1)
      perror("set signal mask");

    sigemptyset(&action.sa_mask);
    sigaddset(&action.sa_mask,SIGSEGV);
```

```
        action.sa_handler = callme;
        action.sa_flags = 0;
        if (sigaction(SIGINT,&action,&old_action)==-1)

           perror( "sigaction");

        pause();                                    /* wait for signal interruption */

        cout << argv[0] << " exists\n";
        return 0;
}
```

In the above example, the process signal mask is set with the SIGTERM signal. The process then defines a signal handler for the SIGINT signal and also specifies that the SIGSEGV signal is to be blocked when the process is handling the SIGINT signal. The process then suspends its execution via the *pause* API.

The sample output of the program is:

```
%   CC sigaction.C -o sigaction
%   sigaction &
[1] 495
%   kill -INT 495
catch signal: 2
sigaction exits
[1]   Done              sigaction
```

If the SIGINT signal is generated to the process, the kernel first sets the process signal mask to block the SIGTERM, SIGINT, and SIGSEGV signals. It then arranges the process to execute the *callme* signal handler function. When the *callme* function returns, the process signal mask is restored to contain only the SIGTERM signal, and the process will continue to catch the SIGILL signal.

The *sa_flag* field of the *struct sigaction* is used to specify special handling for certain signals. POSIX.1 defines only two values for the *sa_flag*: zero or SA_NOCLDSTOP. The SA_NOCLDSTOP flag is an integer literal defined in the <signal.h> header and can be used when the *signal_num* is SIGCHLD. The effect of the SA_NOCHLDSTOP flag is that the kernel will generate the SIGCHLD signal to a process when its child process has terminated, but not when the child process has been stopped. On the other hand, if the *sa_flag* value is zero in a *sigaction* call for SIGCHLD, the kernel will send the SIGCHLD signal to the calling process whenever its child process is either terminated or stopped.

UNIX System V.4 defines additional flags for the *sa_flag* field of *struct sigaction*. These flags can be used to specify the UNIX System V.3 style of signal handling method:

***sa_flag* value**	**Effects on handling of *signal_num***
SA_RESETHAND	If *signal_num* is caught, the *sa_handler* is set to SIG_DFL before the signal handler function is called, and *signal_num* will not be added to the process signal mask when the signal handler function is executed
SA_RESTART	If a signal is caught while a process is executing a system call, the kernel will restart the system call after the signal handler returns. If this flag is not set in the *sa_flag*, after the signal handler returns, the system call will be aborted with a return value of -1 and will set *errno* to EINTR

9.5 The SIGCHLD Signal and the waitpid API

When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process. Depending on how the parent sets up the handling of the SIGCHLD signal, different events may occur:

1. Parent accepts the default action of the SIGCHLD signal: Unlike most signals, the SIGCHLD signal does not terminate the parent process. It affects only the parent process if it arrives at the same time the parent process is suspended by the *waitpid* system call. If that is the case, the parent process will be awakened, the API will return the child's exist status and process ID to the parent, and the kernel will clear up the Process Table slot allocated for the child process. Thus, with this setup, a parent process can call the *waitpid* API repeatedly to wait for each child it created.

2. Parent ignores the SIGCHLD signal: The SIGCHLD signal will be discarded, and the parent will not be disturbed, even if it is executing the *waitpid* system call. The effect of this setup is that if the parent calls the *waitpid* API, the API will suspend the parent until all its child processes have terminated. Furthermore, the child process table slots will be cleared up by the kernel, and the API will return a -1 value to the parent process.

3. Process catches the SIGCHLD signal: The signal handler function will be called in the parent process whenever a child process terminates. Furthermore, if the SIGCHLD signal arrives while the parent process is executing the *waitpid* system call, after the signal handler function returns, the *waitpid* API may be restarted to

collect the child exit status and clear its Process Table slot. On the other hand, the API may be aborted and the child Process Table slot not freed, depending on the parent setup of the signal action for the SIGCHLD signal.

The interaction between SIGCHLD and the *wait* API is the same as that between SIGCHLD and the *waitpid* API. Furthermore, earlier versions of UNIX use the SIGCLD signal instead of SIGCHLD. The SIGCLD signal is now obsolete, but most of the latest UNIX systems have defined SIGCLD to be the same as SIGCHLD for backward compatibility.

9.6 The sigsetjmp and siglongjmp APIs

The *sigsetjmp* and *siglongjmp* APIs have similar functions as their corresponding *setjmp* and *longjmp* APIs. Specifically, both *setjmp* and *sigsetjmp* mark one or more locations in a user program. Later on, the program may call the *longjmp* or *siglongjmp* API to return to any of those marked location. Thus, these APIs provide interfunction *goto* capability.

The *sigsetjmp* and *siglongjmp* APIs are defined in POSIX.1 and on most UNIX systems that support signal masks. The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp ( sigjmpbuf env, int save_sigmask );
int siglongjmp ( sigjmpbuf env, int ret_val );
```

The *sigsetjmp* and *siglongjmp* are created to support signal mask processing. Specifically, it is implementation-dependent on whether a process signal mask is saved and restored when it invokes the *setjmp* and *longjmp* APIs, respectively.

The *sigsetjmp* API behaves similarly to the *setjmp* API, except that it has a second argument, *save_sigmask*, which allows a user to specify whether a calling process signal mask should be saved to the provided *env* argument. Specifically, if the *save_sigmask* argument is nonzero, the caller's signal mask is saved. Otherwise, the signal mask is not saved.

The *siglongjmp* API does all the operations as the *longjmp* API, but it also restores a calling process signal mask if the mask was saved in its *env* argument. The *ret_val* argument specifies the return value of the corresponding *sigsetjmp* API when it is called by *siglongjmp*. Its value should be a nonzero number, and if it is zero the *siglongjmp* API will reset it to 1.

The *siglongjmp* API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and *siglongjmp*

should be called (if a user does not want to resume execution at the code where the signal interruption occurred) to ensure the process signal mask is restored properly when "jumping out" from a signal handling function.

The following *sigsetjmp.C* program illustrates the uses of *sigsetjmp* and *siglongjmp* APIs. The program is modified from the *sigaction.C* program, as depicted in Section 9.4. Specifically, the program sets its signal mask to contain SIGTERM, then sets up a signal trap for the SIGINT signal. The program then calls *sigsetjmp* to store its code location in the *env* global variable. Note that the *sigsetjmp* call returns a zero value when it is called directly in user program and not via *siglongjmp*. The program suspends its execution via the *pause* API. When a user interrupts the process from the keyboard, the *callme* function is called. The *callme* function calls the *siglongjmp* API to transfer program flow back to the *sigsetjmp* function (in the *main* function), which now returns a 2 value.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf      env;

void callme( int sig_num )
{
      cout << "catch signal: " << sig_num << endl;
      siglongjmp( env, 2 );
}

int main()
{
      sigset_t    sigmask;
      struct sigaction     action, old_action;

      sigemptyset(&sigmask);

      if (sigaddset( &sigmask, SIGTERM)==-1 ||
            sigprocmask(SIG_SETMASK, &sigmask,0)==-1)
         perror("set signal mask");
      sigemptyset(&action.sa_mask);
      sigaddset(&action.sa_mask,SIGSEGV);
      action.sa_handler = (void (*)())callme;
      action.sa_flags = 0;

      if (sigaction(SIGINT,&action,&old_action)==-1)
         perror( "sigaction");
```

```
    if (sigsetjmp( env, 1 ) != 0 )        {
                cerr << "Return from signal interruption\n";
                return 0;
    }
    else        cerr << "Return from first time sigsetjmp is called\n";

    pause();                // wait for signal interruption (e.g., from keyboard)
} .
```

The sample output of the above program is:

```
% CC sigsetjmp.C
% a.out &
[1] 377
Return from first time sigsetjmp is called
% kill -INT 377
catch signal: 2
Return from signal interruption
[1]   Done              a.out
%
```

9.7 kill

A process can send a signal to a related process via the *kill* API. This is a simple means of interprocess communication or control. The sender and recipient processes must be related such that either the sender process real or effective user ID matches that of the recipient process, or the sender process has superuser privileges. For example, a parent and a child process can send signals to each other via the *kill* API.

The *kill* API is defined in most UNIX systems and is a POSIX.1 standard. The function prototype of the API is:

```
#include <signal.h>

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

The *signal_num* argument is the integer value of a signal to be sent to one or more processes designated by *pid*. The possible values of *pid* and its use by the *kill* API are:

pid value	Effects on the *kill* API
a positive value	*pid* is a process ID. Sends *signal_num* to that process
0	Sends *signal_num* to all processes whose process group ID is the same as the calling process
-1	Sends *signal_num* to all processes whose real user ID is the same as the effective user ID of the calling process. If the calling process effective user ID is the superuser user ID, *signal_num* will be sent to all processes in the system (except processes-0 and 1). The latter case is used when the system is shutting down -- the kernel calls the *kill* API to terminate all processes except 0 and 1. Note that POSIX.1 does not specify the behavior of the *kill* API when the *pid* value is -1. The above effects are for UNIX systems only
a negative value	Sends *signal_num* to all processes whose process group ID matches the absolute value of *pid*

The return value of *kill* is zero if it succeeds or -1 if it fails.

The following *kill.C* program illustrates the implementation of the UNIX *kill* command using the *kill* API:

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

int main( int argc, char** argv)
{
      int pid, sig = SIGTERM;
      if (argc==3)  {
         if (sscanf(argv[1],"%d",&sig)!=1)  {          /* get signal number */
            cerr << "Invalid number: " << argv[1] << endl;
            return -1;
         }
         argv++, argc--;
      }
      while (--argc > 0)
```

```
        if (sscanf(*++argv,"%d",&pid)==1) {          /* get process ID */
            if (kill (pid, sig)==-1)
                perror("kill");
        }
        else cerr << "Invalid pid: " << argv[0] << endl;
    return 0;
}
```

The UNIX *kill* command invocation syntax is:

kill [-<signal_num>] <Pid> ...

where <*signal_num*> can be an integer number or the symbolic name of a signal, as defined in the <signal.h> header. The <*Pid*> is the integer number of a process ID. There can be one or more <*Pid*> specified, and the *kill* command will send the signal <*signal_num*> to each process that corresponds to a <*Pid*>.

To simplify the above program, any signal specification at the command line must be a signal's integer value. It does not support signal symbolic names. If no signal number is specified, the program will use the default signal SIGTERM, which is the same for the UNIX *kill* command. The program calls the *kill* API to send a signal to each process whose process ID is specified at the command line. If a process ID is invalid or if the *kill* API fails, the program will flag an error message.

9.8 alarm

The *alarm* API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. This is like setting an alarm clock to remind someone to do something after a specified period of time.

The *alarm* API is defined in most UNIX systems and is a POSIX.1 standard. The function prototype of the API is:

```
#include <signal.h>

unsigned int alarm ( unsigned int time_interval );
```

The *time_interval* argument is the number of CPU seconds elapse time, after which the kernel will send the SIGALRM signal to the calling process. If a *time_interval* value is zero, it turns off the alarm clock.

The return value of the *alarm* API is the number of CPU seconds left in the process timer, as set by a previous *alarm* system call. The effect of the previous *alarm* API call is canceled, and the process timer is reset with the new *alarm* call. A process alarm clock is not passed on to its forked child process, but an *exec*'ed process retains the same alarm clock value as was prior to the *exec* API call.

The *alarm* API can be used to implement the *sleep* API:

```
/* sleep.C */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void wakeup()          {};

unsigned int sleep ( unsigned int timer )
{
        struct sigaction action;
        action.sa_handler = wakeup;
        action.sa_flags = 0;
        sigemptyset(&action.sa_mask);

        if (sigaction(SIGALRM, &action,0)==-1)  {
                perror("sigaction");
                return -1;
        }
        (void)alarm( timer );
        (void)pause();
        return 0;
}
```

The *sleep* API suspends a calling process for the specified number of CPU seconds. The process will be awakened by either the elapse time exceeding the *timer* value or when the process is interrupted by a signal.

In the above example, the *sleep* function sets up a signal handler for the SIGALRM, calls the *alarm* API to request the kernel to send the SIGALRM signal (after the *timer* interval), and finally, suspends its execution via the *pause* system call. The *wakeup* signal handler function is called when the SIGALRM signal is sent to the process. When it returns, the *pause* system call will be aborted, and the calling process will return from the *sleep* function.

BSD UNIX defines the *ualarm* function, which is the same function as the *alarm* API, except that the argument and return value of the *ualarm* function are in microsecond units. This is useful for some time-critical applications where the resolution of time must be in microsecond levels.

The *ualarm* function can be used to implement the BSD-specific *usleep* function, which is like the *sleep* function, except its argument is in microsecond units.

9.9 Interval Timers

The *sleep* function that suspends a process for a fixed amount of time is only one use of the *alarm* API. The more general use of the *alarm* API is to set up an interval timer in a process. The interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operations, or to limit the time allowed for the execution of some tasks.

The following program, *timer.C*, illustrates how to set up a real-time clock interval timer using the *alarm* API.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#define  INTERVAL  5

void callme( int sig_no )
{
        alarm( INTERVAL );
        /* do scheduled tasks */
}

int main()
{
    struct sigaction  action;
    sigemptyset(&action.sa_mask);
    action.sa_handler = (void (*)())callme;
    action.sa_flags = SA_RESTART;
    if ( sigaction( SIGALRM,&action,0 )==-1 )  {
       perror( "sigaction");
       return 1;
    }
    if (alarm( INTERVAL ) == -1)
       perror("alarm" );
    else while( 1 )  {
       /* do normal operation */
    }
    return 0;
  }
```

In the above program, the *sigaction* API is called to set up *callme* as the signal handling function for the SIGALRM signal. The program then invokes the *alarm* API to send itself the

SIGALRM after 5 real clock seconds. The program then goes off to perform its normal operation in an infinite loop. When the timer expires, the *callme* function is invoked, which restarts the alarm clock for another 5 seconds and then does the scheduled tasks. When the *callme* function returns, the program continues its "normal" operation until another timer expiration.

The above sample program may be useful in creating a clock synchronization program: Every time the *callme* function is invoked, it polls a remote host for current time, then calls the *stime* API to set the local system clock to be the same as the reference host.

In addition to using the *alarm* API to set up an interval timer in a process, BSD UNIX invented the *setitimer* API, which provides capabilities additional to those of the *alarm* API:

* The *setitimer* resolution time is in microseconds, whereas the resolution time for *alarm* is in seconds

* The *alarm* API can be used to set up one real-time clock timer per process. The *setitimer* API can be used to define up to three different types of timers in a process:

  a. Real time clock timer
  b. Timer based on the user time spent by a process
  c. Timer based on the total user and system times spent by a process

The *setitimer* API is also available in UNIX System V.3 and V.4. However, it is not specified by POSIX. POSIX.1b defines a new set of APIs for interval timer manipulation. These APIs are described in the next section.

The *getitimer* API is also defined in BSD and System V UNIX for users to query the timer values that are set by the *setitimer* API.

The *setitimer* and *getitimer* function prototypes are:

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

int setitimer ( int which, const struct itimerval *val, struct itimerval *old );
int getitimer ( int which, struct itimerval *old );
```

The *which* arguments to the above APIs specify which timer to process. Its possible values and the corresponding timer types are:

which argument value	Timer type
ITIMER_REAL	Timer based on real-time clock. Generates a SIGALRM signal when it expires
ITIMER_VIRTUAL	Timer based on user-time spent by a process. Generates a SIGVTALRM signal when it expires
ITIMER_PROF	Timer based on total user and system times spent by a process. Generates a SIGPROF signal when it expires

The *struct itimerval* data type is defined in the <sys/time.h> header as:

```
struct itimeval
{
        struct timeval      it_interval;        // timer interval
        struct timeval      it_value;           // current value
};
```

For the *setitimer* API, the *val.it_value* is the time to set the named timer, and the *val.it_interval* is the time to reload the timer when it expires. The *val.it_interval* may be set to zero if the timer is to run once only. Furthermore, if the *val.it_value* value is set to zero, it stops the named timer if it is running.

For the *getitimer* API, the *old.it_value* and the *old.t_interval* return the named timer's remaining time (to expiration) and the reload time, respectively.

The *old* argument of the *setitimer* API is like the *old* argument of the *getitimer* API. If this is an address of a *struct itimeval*-typed variable, it returns the previous timer value. If the *old* argument is set to NULL, the old timer value will not be returned.

The ITIMER_VIRTUAL and ITIMER_PROF timers are primary useful in timing the total execution time of selected user functions, as the timer runs only while the user process is running (or the kernel is executing system functions on behalf of the user process for the ITIMER_PROF timer).

The *setitimer* and *getitimer* APIs return a zero value if they succeed or a -1 value if they fail. Moreover, timers set by the *setitimer* API in a parent process are not inherited by its child processes, but these timers are retained when a process *exec*'s a new program.

The following example program, *timer2.C*, is the same as the *timer.C* program, except that it uses the *setitimer* API instead of the *alarm* API. Also, there is no need to call the *setitimer* API inside the signal handling function, as the timer is specified to be reloaded automatically:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <signal.h>
#define  INTERVAL  2

void callme( int sig_no )
{
      /* do some schedule tasks */
}

int main()
{
      struct itimerval val;
      struct sigaction  action;

      sigemptyset(&action.sa_mask);
      action.sa_handler = (void (*)())callme;
      action.sa_flags = SA_RESTART;
      if (sigactionSIGALRM,&action,0)==-1) {
         perror( "sigaction");
         return 1;
      }

      val.it_interval.tv_sec      = INTERVAL;
      val.it_interval.tv_usec     = 0;
      val.it_value.tv_sec         = INTERVAL;
      val.it_value.tv_usec        = 0;

      if (setitimer( ITIMER_REAL, &val, 0 ) == -1)
         perror("alarm" );
      else while( 1 )  {
         /* do normal operation */
      }
      return 0;
}
```

Note that the real time clock timer set by the *setitimer* API is different from that set by the *alarm* API. Thus, a process may set up two real-time clock timers using the two APIs. Furthermore, since the *alarm* and *setitimer* APIs require that users set up signal handling to catch timer expiration, they (when used to set up real-time clock timer) should not be used in conjunction with the *sleep* API. This is because the *sleep* API may modify the signal handling function for the SIGALRM signal.

Daemon Processes

13.1 Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shutdown. We say they run in the background, because they don't have a controlling terminal. Unix systems have numerous daemons that perform day-to-day activities.

In this chapter we look at the process structure of daemons, and how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

13.2 Daemon Characteristics

Let's look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions that we described in Chapter 9. The `ps`(1) command prints the status of various processes in the system. There are a multitude of options—consult your system's manual for all the details. We'll execute

```
ps -axj
```

under 4.3+BSD or SunOS to see the information we need for this discussion. The `-a` option shows the status of processes owned by others, and `-x` shows processes that don't have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID. Under SVR4 a similar command is `ps -efjc`. (On some Unix systems that conform to the Department of Defense security guidelines, we are not able to use `ps` to look at any processes other than our own.) The output from `ps` looks like

```
PPID   PID  PGID   SID TT TPGID  UID  COMMAND
   0     0     0     0 ?    -1    0  swapper
   0     1     0     0 ?    -1    0  /sbin/init -
   0     2     0     0 ?    -1    0  pagedaemon
   1    80    80    80 ?    -1    0  syslogd
   1    88    88    88 ?    -1    0  /usr/lib/sendmail -bd -q1h
   1   105    37    37 ?    -1    0  update
   1   108   108   108 ?    -1    0  cron
   1   114   114   114 ?    -1    0  inetd
   1   117   117   117 ?    -1    0  /usr/lib/lpd
```

We have removed a few columns that don't interest us, such as the accumulated CPU time. The columns headings, in order, are the parent process ID, process ID, process group ID, session ID, terminal name, terminal process group ID (the foreground process group associated with the controlling terminal), user ID, and actual command string.

> The system that these `ps` commands were run on (SunOS) supports the notion of a session ID, which we mentioned with the `setsid` function in Section 9.5. It is just the process ID of the session leader. A 4.3+BSD system, however, will print the address of the `session` structure corresponding to the process group that the process belongs to (Section 9.11).

Processes 0, 1, and 2 are the ones described in Section 8.2. These three are special and exist for the entire lifetime of the system. They have no parent process ID, no process group ID, and no session ID. The `syslogd` daemon is available to any program to log system messages for an operator. The messages may be printed on an actual console device and also written to a file. (We describe the `syslog` facility in Section 13.4.2.) `sendmail` is the standard mailer daemon. `update` is a program that flushes the kernel's buffer cache to disk at regular intervals (usually every 30 seconds). To do this it just calls the `sync`(2) function every 30 seconds. (We described `sync` in Section 4.24.) The `cron` daemon executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by `cron`. We talked about the `inetd` daemon in Section 9.3. It listens on the system's network interfaces for incoming requests for various network servers. The final daemon, `lpd`, handles print requests on the system.

Notice that all the daemons run with superuser privilege (a user ID of 0). None of the daemons has a controlling terminal—the terminal name is set to a question mark and the terminal foreground process group is –1. The lack of a controlling terminal is probably the result of the daemon having called `setsid`. All the daemons other than `update` are process group leaders and session leaders and are the only processes in their process group and session. `update` is the only process in its process group (37) and session (37), but the process group leader (which was probably also the session leader) has already exited. Finally, note that the parent of all these daemons is the `init` process.

13.3 Coding Rules

There are some basic rules to coding a daemon, to prevent unwanted interactions from happening. We state these rules and then show a function, `daemon_init`, that implements them.

1. The first thing to do is call `fork` and have the parent `exit`. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.
2. Call `setsid` to create a new session. The three steps listed in Section 9.5 occur. The process (1) becomes a session leader of a new session, (2) becomes the process group leader of a new process group, and (3) has no controlling terminal.

   > Under SVR4, some people recommend calling `fork` again at this point and having the parent terminate. The second child continues as the daemon. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the SVR4 rules (Section 9.6). Alternately, to avoid acquiring a controlling terminal be sure to specify `O_NOCTTY` whenever opening a terminal device.

3. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted filesystem. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted filesystem, that filesystem cannot be unmounted.

   Alternately, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.
4. Set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.
5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). Exactly which descriptors to close, however, depends on the daemon, so we don't show this step in our example. It can use our `open_max` function (Program 2.3) to determine the highest descriptor and close all descriptors up to that value.

Program 13.1 is a function that can be called from a program that wants to initialize itself as a daemon.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
daemon_init(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0);    /* parent goes bye-bye */

    /* child continues */
    setsid();       /* become session leader */

    chdir("/");     /* change working directory */

    umask(0);       /* clear our file mode creation mask */

    return(0);
}
```

Program 13.1 Initialize a daemon process.

If the `daemon_init` function is called from a `main` program that then goes to sleep, we can check the status of the daemon with the `ps` command:

```
$ a.out
$ ps -axj
 PPID   PID  PGID   SID TT TPGID  UID  COMMAND
    1   735   735   735 ?     -1  224  a.out
```

We can see that our daemon has been initialized correctly. □

13.4 Error Logging

One problem a daemon has is how to handle error messages. It can't just write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error logging facility is required.

The BSD `syslog` facility was developed at Berkeley and used widely in 4.2BSD. Most systems derived from 4.xBSD support `syslog`. We describe this facility in Section 13.4.2.

There has never been a central daemon logging facility in System V. SVR4 supports the BSD-style `syslog` facility, and the `inetd` daemon under SVR4 uses `syslog`. The basis for `syslog` in SVR4 is the `/dev/log` streams device driver, which we describe in the next section.

13.4.1 SVR4 Streams `log` Driver

SVR4 provides a streams device driver, documented in `log`(7) in [AT&T 1990d], with an interface for streams error logging, streams event tracing, and console logging. Figure 13.1 details the overall structure of this facility.

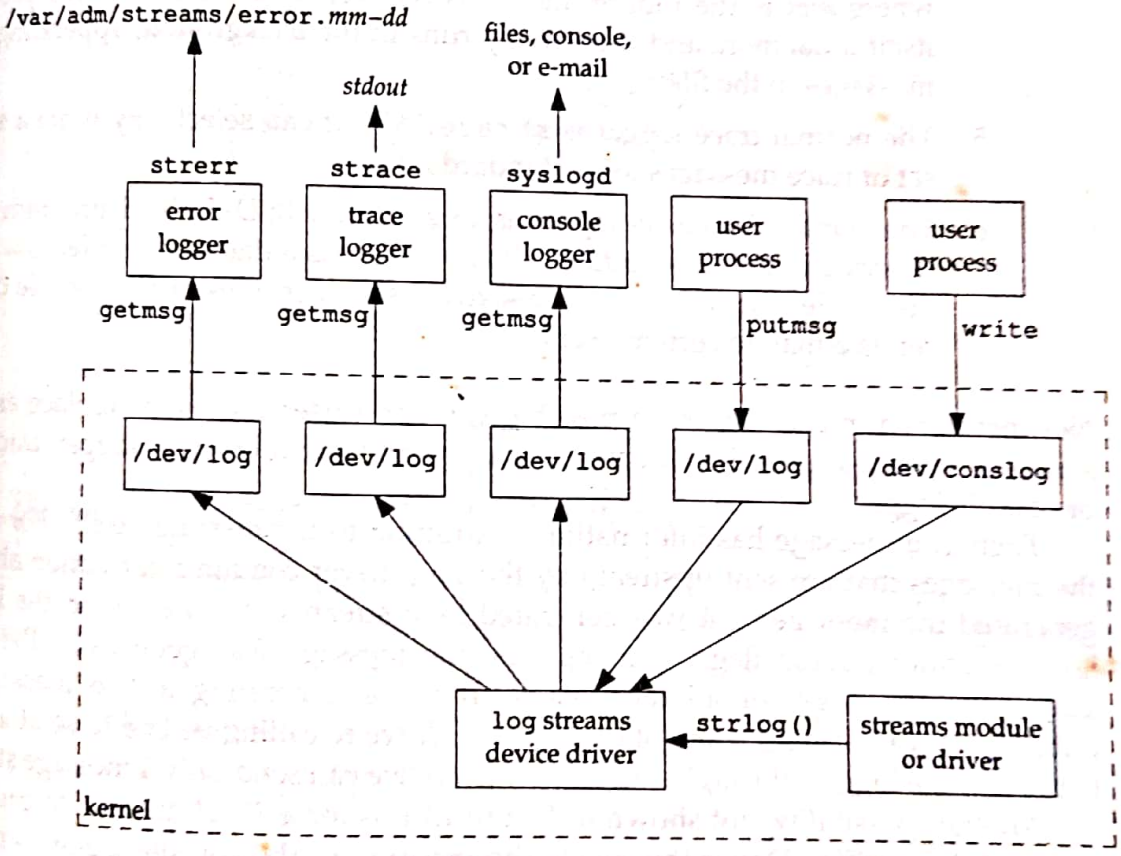

Figure 13.1 The SVR4 `log` facility.

Each `log` message can be destined for one of three loggers: the error logger, the trace logger, or the console logger.

We show three ways to generate `log` messages and three ways to read them.

- Generating `log` messages.
  1. Routines within the kernel can call `strlog` to generate log messages. This is normally used by streams modules and streams device drivers for either error messages or trace messages. (Trace messages are often used in the

debugging of new streams modules or drivers.) We won't consider this type of message generation, since we're not interested in the coding of kernel routines.

2. A user process (such as a daemon) can `putmsg` to `/dev/log`. This message can be sent to any of the three loggers.
3. A user process (such as a daemon) can `write` to `/dev/conslog`. This message is sent only to the console logger.

- Reading `log` messages.
  4. The normal error logger is `strerr`(1M). It appends these messages to a file in the directory `/var/adm/streams`. The file's name is `error.`*mm-dd*, where *mm* is the month and *dd* is the day of the month. This program is itself a daemon, and it normally runs in the background, appending the log messages to the file.
  5. The normal trace logger is `strace`(1M). It can selectively write a specified set of trace messages to its standard output.
  6. The standard console logger is `syslogd`, a BSD-derived program that we describe in the next section. This program is a daemon that reads a configuration file and writes log messages to specified files or the console device or sends e-mail to certain users.

Not mentioned in this list, but a possibility, is for a user process to replace any of the standard system-supplied daemons: we can supply our own error logger, trace logger, or console logger.

Each `log` message has information in addition to the message itself. For example, the messages that are sent upstream by the `log` driver contain information about who generated the message (if it was generated by a streams module within the kernel), a level, a priority, some flags, and the time the message was generated. Refer to the `log`(7) manual page for all the details. If we're generating a `log` message using `putmsg`, we can also set some of these fields. If we're calling `write` to send a message to the console logger (through `/dev/conslog`), we can send only a message string.

Another possibility, not shown in Figure 13.1, is for a SVR4 daemon to call the BSD `syslog`(3) function. Doing this sends the message to the console logger, similar to a `putmsg` to `/dev/log`. With `syslog`, we can set the priority field of the message. We describe this function in the next section.

If the appropriate type of logger isn't running when a log message of that type is generated, the `log` driver just throws away the message.

> Unfortunately, in SVR4 the use of this `log` facility is haphazard. A few daemons use it, but most system-supplied daemons are hardcoded to write directly to the console.
>
> The `syslog`(3) function and `syslogd`(1M) daemon are documented in the BSD Compatibility Library [AT&T 1990c], but they are not in this library—they are in the standard C library, available to all user processes (daemons).

13.4.2 4.3+BSD `syslog` Facility

The BSD `syslog` facility has been widely used since 4.2BSD. Most daemons use this facility. Figure 13.2 details its organization.

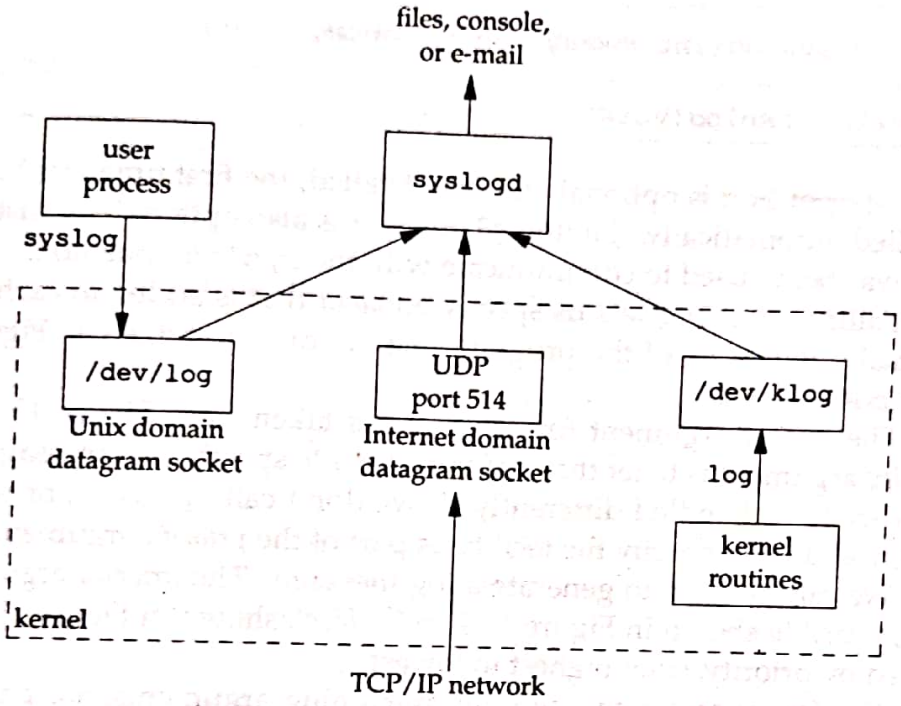

Figure 13.2 The 4.3+BSD `syslog` facility.

There are three ways to generate log messages:

1. Kernel routines can call the `log` function. These messages can be read by any user process that opens and reads the `/dev/klog` device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the `syslog`(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the Unix domain datagram socket `/dev/log`.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the `syslog` function never generates these UDP datagrams—they require explicit network programming by the process generating the log message.

Refer to Stevens [1990] for details on Unix domain sockets and UDP sockets.

Normally the `syslogd` daemon reads all three forms of log messages. This daemon reads a configuration file on start-up, usually `/etc/syslog.conf`, that determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator via e-mail and printed on the console, while warnings may be logged to a file.

Our interface to this facility is through the `syslog` function.

```
#include <syslog.h>

void openlog(char *ident, int option, int facility);

void syslog(int priority, char *format, ...);

void closelog(void);
```

Calling `openlog` is optional. If it's not called, the first time `syslog` is called, `openlog` is called automatically. Calling `closelog` is also optional—it just closes the descriptor that was being used to communicate with the `syslogd` daemon.

Calling `openlog` lets us specify an *ident* that is added to each log message. This is normally the name of the program (e.g., `cron`, `inetd`, etc.). Figure 13.3 describes the four possible *options*.

The *facility* argument for `openlog` is taken from Figure 13.4. The reason for the *facility* argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call `openlog`, or we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to `syslog`.

We call `syslog` to generate a log message. The *priority* argument is a combination of the *facility* shown in Figure 13.4 and a *level*, shown in Figure 13.5. These *levels* are ordered by priority, from highest to lowest.

The *format* argument, and any remaining arguments, are passed to the `vsprintf` function for formatting. Any occurrence of the two characters `%m` in the *format* are first replaced with the error message string (`strerror`) corresponding to the value of `errno`.

The `logger`(1) program is also provided by both SVR4 and 4.3+BSD as a way to send log messages to the `syslog` facility. Optional arguments to this program can specify the *facility*, *level*, and *ident*. It is intended for a shell script running noninteractively that needs to generate log messages.

A form of the `logger` command is being standardized by POSIX.2.

Example

In our PostScript printer daemon in Chapter 17 we will encounter the sequence

```
openlog("lprps", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default *facility* to the line printer system. The actual call to `syslog` specifies an error condition and a message string. If we had not called `openlog`, the second call could have been

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Here we specify the *priority* argument as a combination of a *level* and a *facility*. □

option	Description
LOG_CONS	If the log message can't be sent to `syslogd` via the Unix domain datagram, the message is written to the console instead.
LOG_NDELAY	Open the Unix domain datagram socket to the `syslogd` daemon immediately—don't wait until the first message is logged. Normally the socket is not opened until the first message is logged.
LOG_PERROR	Write the log message to standard error in addition to sending it to `syslogd`. This option is supported only by the 4.3BSD Reno releases and later.
LOG_PID	Log the process ID with each message. This is intended for daemons that `fork` a child process to handle different requests (as compared to daemons such as `syslogd` that never call `fork`).

Figure 13.3 The *option* argument for `openlog`.

facility	Description
LOG_AUTH	authorization programs: `login`, `su`, `getty`, ...
LOG_CRON	`cron` and `at`
LOG_DAEMON	system daemons: `ftpd`, `routed`, ...
LOG_KERN	messages generated by the kernel
LOG_LOCAL0	reserved for local use
LOG_LOCAL1	reserved for local use
LOG_LOCAL2	reserved for local use
LOG_LOCAL3	reserved for local use
LOG_LOCAL4	reserved for local use
LOG_LOCAL5	reserved for local use
LOG_LOCAL6	reserved for local use
LOG_LOCAL7	reserved for local use
LOG_LPR	line printer system: `lpd`, `lpc`, ...
LOG_MAIL	the mail system
LOG_NEWS	the Usenet network news system
LOG_SYSLOG	the `syslogd` daemon itself
LOG_USER	messages from other user processes (default)
LOG_UUCP	the UUCP system

Figure 13.4 The *facility* argument for `openlog`.

level	Description
LOG_EMERG	emergency (system is unusable) (highest priority)
LOG_ALERT	condition that must be fixed immediately
LOG_CRIT	critical condition (e.g., hard device error)
LOG_ERR	error condition
LOG_WARNING	warning condition
LOG_NOTICE	normal, but significant condition
LOG_INFO	informational message
LOG_DEBUG	debug message (lowest priority)

Figure 13.5 The `syslog` *levels* (ordered).

13.5 Client–Server Model

A common use for a daemon process is as a server process. Indeed, in Figure 13.2 we can call the `syslogd` process a server that has messages sent to it by user processes (clients) using a Unix domain datagram socket.

In general a *server* is a process that waits for a *client* to contact it, requesting some type of service. In Figure 13.2 the service being provided by the `syslogd` server is the logging of an error message.

In Figure 13.2 the communication between the client and server is one-way. The client just sends its service request to the server—the server sends nothing back to the client. In the following chapters on interprocess communication we'll see numerous examples where there is a two-way communication between the client and server. The client sends a request to the server, and the server sends a reply back to the client.

13.6 Summary

Daemon processes are running all the time on most Unix systems. To initialize our own process that is to run as a daemon takes some care and an understanding of the process relationships that we described in Chapter 9. In this chapter we developed a function that can be called by a daemon process to initialize itself correctly.

We also discussed the ways a daemon can log error messages, since a daemon normally doesn't have a controlling terminal. Under SVR4 the streams `log` driver is available, and under 4.3+BSD the `syslog` facility is provided. Since the BSD `syslog` facility is also provided by SVR4, in later chapters when we need to log error messages from a daemon, we'll call the `syslog` function. We'll encounter this in Chapter 17 with our PostScript printer daemon.