Signals

Introduction

- ✓ 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.
- ✓ Signals are defined as integer flags, and the <signal.h> header depicts the list of signals defined for a UNIX system.

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGKILL	termination	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (poll)	terminate
SIGPROF	profiling time alarm (setitimer)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate
SIGSTOP	stop	stop process
SIGSYS	invalid system call	terminate+core
SIGTERM	termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTRAP	hardware fault	terminate+core
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process
SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore

SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (setitimer)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (setrlimit)	terminate+core/ignore
SIGXFSZ	file size limit exceeded (setrlimit)	terminate+core/ignore
SIGXRES	resource control exceeded	Ignore

- ✓ 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.
- ✓ 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.
- ✓ Most signals can be caught or ignored except the SIGKILL and SIGSTOP signals.
- ✓ A companion signal to SIGSTOP is SIGCONT, which resumes a process execution after it has been stopped, both 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.
- ✓ Example:- A DBMS process updating a database file should not be interrupted until it is finished, else database file will be corrupted, it should restore signal handling actions for signals when finished mission critical work.
- ✓ Because signals are generated asynchronously to a process, a process may specify a per signal handler function, these function would then be 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.

UNIX Kernel Supports of Signals

- ✓ Process table in the kernel table has a slot containing array of signal flags, one for each signal defined in the system.
- ✓ When a signal is generated kernel will set the corresponding signal flag in the process table slot of the recipient process.
- ✓ If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- ✓ 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.
- ✓ Then kernel consults the array entry of the corresponding signal to find out how the process will react to this signal. If the array entry for the signal contains a value:
- ➢ 0 : The process will accept the default action
- ➢ 1 : The process will ignore the signal
- ➢ Other value: It is used as the function pointer for the user defined signal handler function.
- ✓ *Pending*: When a signal is generated and it is sent to a process, it becomes *pending*. Normally it remains pending for just a short period of time.
- ✓ *Delivered* signal: If a signal has been reacted or action taken for a signal.
- ✓ caught: When signal handler routine is called

Signal

- ✓ All UNIX systems and ANSI-C support the *signal* API is 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 argument of the API are: *sig_num* is a signal identifier like SIGINT or SIGTERM.
- ✓ 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>

```
void catch_sig(int sig_num)        /*signal handler function*/
{
        signal (sig_num,catch_sig);
        cout<<"catch_sig:"<<sig_nm<<endl;
}
 int main()          /* main function*/
{
        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 manifested 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*/
}
```

- ✓ UNIX System V.3 and V.4 support the *signet* 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.

- ✓ The *signal* API is unreliable and *sigset* API is reliable.

Signal Mask

- ✓ 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);
```

- ✓ Returns: 0 if OK, -1 on error
- ✓ 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 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.
- ✓ If the actual argument to *old_mask* is a NULL pointer, no previous signal mask will be returned.
- ✓ The *sigset_t* contains a collection of bit flags, with each bit-flag representing one signal defined in a given system.
- ✓ Possible failure may occur because the *new_mask* and /or the *old_mask* actual arguments are invalid addresses.
- ✓ The BSD UNIX and POSIX.1 define a set of API known as *sigsetops* functions:

```
#include<signal.h>
int sigemptyset (sigset_t* sigmask);
int sigaddset (sigset_t* sigmask, const int sig_num);
int sigdelset (sigset_t* sigmask, const int sig_num);
int sigfillset (sigset_t* sigmask);
int sigismember (const sigset_t* sigmask, const int sig_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.
- ✓ All the above functions return 0 if OK, -1 on error.
- ✓ Possible causes of failure may be that the *sigmask* and/or the *signal_num* arguments are invalid.
- ✓ The *sigismember* API returns 1 if flag is set, 0 if 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.

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

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

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

- ✓ Returns 0 if OK, -1 if 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;
}
```

- ✓ In addition to the above, UNIX also supports following APIs 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.
- ✓ 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.
- ✓ 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.

sigaction

- ✓ 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);

- ✓ Returns: 0 if OK, -1 on error
- ✓ 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;
};
```

- ✓ 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 when it is handling the *signal_num* signal.
- ✓ The *signal_num* argument designates which signal handling action is defined in *action* argument.
- ✓ The previous signal handling method for *signal_num* will be returned via *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 program illustrates the 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();
                cout<<argv[0]<<"exists\n";
                return 0;
}
```

Output:

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

- ✓ 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 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.
- ✓ If the *sa_flag* value is SA_NOCLDSTOP then 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.
- ✓ If the *sa_flag* value is zero then 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_flags* field. These flags can be used to specify the UNIX System V.3 style of signal handling method:

sa_flags value	Effect on handling *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_flags*, after the signal handler returns, the system call will be aborted with a return value of -1 and will set errno to EINTR

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:

- ✓ Parent accepts the **default action** of the SIGCHLD signal:
  - ➢ SIGCHLD does not terminate the parent process.
  - ➢ Parent process will be awakened.
  - ➢ API will return the child's exit status and process ID to the parent.
  - ➢ Kernel will clear up the Process Table slot allocated for the child process.
  - ➢ Parent process can call the waitpid API repeatedly to wait for each child it created.
- ✓ Parent **ignores** the SIGCHLD signal:

- SIGCHLD signal will be discarded.
- Parent will not be disturbed even if it is executing the waitpid system call.
- If the parent calls the waitpid API, the API will suspend the parent until all its child processes have terminated.
- Child process table slots will be cleared up by the kernel.
- API will return a -1 value to the parent process.

✓ Process **catches** the SIGCHLD signal:

- The signal handler function will be called in the parent process whenever a child process terminates.
- If the SIGCHLD arrives while the parent process is executing the waitpid system call, the waitpid API may be restarted to collect the child exit status and clear its process table slots.
- Depending on parent setup, the API may be aborted and child process table slot not freed.

sigsetjmp and siglongjmp Functions

✓ These two functions should always be used when branching from a signal handler.

#include <setjmp.h>

int sigsetjmp(sigjmp_buf *env*, int *savemask*);

Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

void siglongjmp(sigjmp_buf *env*, int *val*);

✓ The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument.

✓ If *savemask* is nonzero, then sigsetjmp also saves the current signal mask of the process in *env*.

✓ When siglongjmp is called, if the *env* argument was saved by a call to sigsetjmp with a nonzero *savemask,* then siglongjmp restores the saved signal mask.

✓ The program demonstrates how the signal mask that is installed by the system when a signal handler is invoked automatically includes the signal being caught. This program also illustrates the use of the sigsetjmp and siglongjmp functions.

```
#include "apue.h"
#include <setjmp.h>
#include <time.h>
static void sig_usr1(int);
static void sig_alrm(int);
```

```
static sigjmp_buf jmpbuf;
static volatile sig_atomic_t canjump;
int main(void)
{
        if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
                err_sys("signal(SIGUSR1) error");
        if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                err_sys("signal(SIGALRM) error");
                pr_mask("starting main: "); /* Figure 10.14 */
                if (sigsetjmp(jmpbuf, 1)) {
                        pr_mask("ending main: ");
                        exit(0);
                }
                canjump = 1; /* now sigsetjmp() is OK */
                for ( ; ; )
                pause();
}
static void sig_usr1(int signo)
{
        time_t starttime;
        if (canjump == 0)
                return; /* unexpected signal, ignore */
                pr_mask("starting sig_usr1: ");
                alarm(3); /* SIGALRM in 3 seconds */
                starttime = time(NULL);
                for ( ; ; ) /* busy wait for 5 seconds */
                if (time(NULL) > starttime + 5)
                        break;
                pr_mask("finishing sig_usr1: ");
                canjump = 0;
                siglongjmp(jmpbuf, 1); /* jump back to main, don't return */
```

```
}
static void sig_alrm(int signo)
{
        pr_mask("in sig_alrm: ");
}
void pr_mask(const char *str)
{
        sigset_t sigset;
        int errno_save;
        errno_save = errno; /* we can be called by signal handlers */
        if (sigprocmask(0, NULL, &sigset) < 0) {
                err_ret("sigprocmask error");
        } else {
                printf("%s", str);
                if (sigismember(&sigset, SIGINT))
                        printf(" SIGINT");
                if (sigismember(&sigset, SIGQUIT))
                        printf(" SIGQUIT");
                if (sigismember(&sigset, SIGUSR1))
                        printf(" SIGUSR1");
                if (sigismember(&sigset, SIGALRM))
                        printf(" SIGALRM");
                /* remaining signals can go here */
                printf("\n");
        }
                errno = errno_save; /* restore errno */
}
```

- ✓ This program demonstrates another technique that should be used whenever siglongjmp is called from a signal handler.
- ✓ Set the variable canjump to a nonzero value only after we've called sigsetjmp.

- ✓ This variable is examined in the signal handler, and siglongjmp is called only if the flag canjump is nonzero.
- ✓ This technique provides protection against the signal handler being called at some earlier or later time, when the jump buffer hasn't been initialized by sigsetjmp.
- ✓ Providing this type of protection usually isn't required with longjmp in normal C code.
- ✓ Since a signal can occur at *any* time, however, we need the added protection in a signal handler.
- ✓ Figure shows a timeline for this program. Divide figure into three parts: the left part (corresponding to main), the center part (sig_usr1), and the right part (sig_alrm).
- ✓ While the process is executing in the left part, its signal mask is 0 (no signals are blocked). While executing in the center part, its signal mask is SIGUSR1. While executing in the right part, its signal mask is SIGUSR1|SIGALRM.

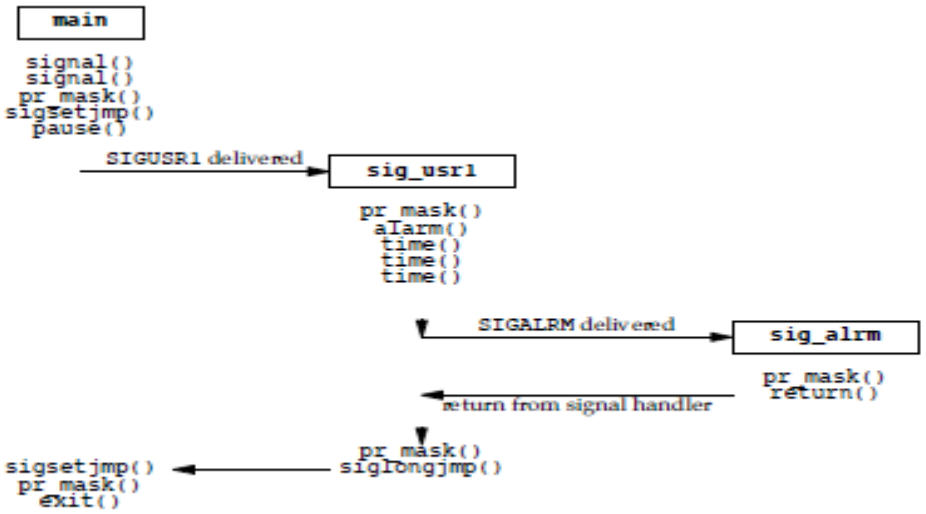

Figure 10.21 Timeline for example program handling two signals

- ✓ Let's examine the output when the program in is executed:

$ **./a.out &** *start process in background*

starting main:

[1] 531 *the job-control shell prints its process ID*

$ **kill -USR1 531** *send the process* SIGUSR1

starting sig_usr1: SIGUSR1

$ in sig_alrm: SIGUSR1 SIGALRM

finishing sig_usr1: SIGUSR1

ending main:

just press RETURN

[1] + Done ./a.out &

- ✓ When a signal handler is invoked, the signal being caught is added to the current signal mask of the process. The original mask is restored when the signal handler returns.
- ✓ Also, siglongjmp restores the signal mask that was saved by sigsetjmp.

kill

- A process can send a signal to a related process via the kill API.
- This is a simple means of inter-process communication or control. The function prototype of the API is:

  #include<signal.h>

  int kill(pid_t pid, int signal_num);

- Returns: 0 on success, -1 on failure.
- 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 su user ID, *signal_num* will be sent to all processes in the system (except processes – 0 and 1). The later case is used when the system is shutting down – kernel calls the kill API to terminate all processes except 0 and 1. **Note: POSIX.1 does not specify the behavior of the kill API when the pid value is -1. This effect is for UNIX systems only.**
A negative value	Sends *signal_num* to all processes whose process group ID matches the absolute value of *pid*.

The following 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) {
                        cerr<<"invalid number:" << argv[1] << endl;
                        return -1;
                }
        argv++,argc--;
        }
while(--argc>0)
if(sscanf(*++argv, "%d", &pid)==1) {
        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. <pid> is the integer number of a process ID.

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.
- ✓ The function prototype of the API is:

  #include<signal.h>

  unsigned int alarm(unsigned int time_interval);

- ✓ Returns: number of CPU seconds left in the process timer, as set by a previous *alarm* system call.
- ✓ 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 *alarm* API can be used to implement the *sleep* API:

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

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 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.

Interval Timers

- ✓ 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 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(SIGALARM,&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 setup *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.
- ✓ In addition to *alarm* API, UNIX also invented the *setitimer* API, which provides additional to those of the *alarm* API:

- The *setitimer* resolution time is in microsecond, 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, which can be used to define up to three different types of timers in a process:

❖ Real time clock timer

❖ Timer based on the user time spent by a process

❖ Timer based on the total user and system times spent by a process

✓ The getitimer API is also defined 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 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 datatype is defined as:

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

✓ 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_value* value is set to zero, it stops the named timer if it is running.

✓ The *val.it_interval* may be set to zero if the timer is to run once only.

- ✓ The *old.it_value* and the *old.it_interval* return the named timer's remaining time to expiration and the reload time respectively.
- ✓ If the *old* argument is set to NULL , the old timer value will not be returned.
- ✓ The *setitimer* and *getitimer* APIs return a zero value if they succeed or a -1 value if they fail.
- ✓ The following program is the same as the above program, except that it uses the *setitimer* API instead of the alarm API.
- ✓ 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 5
void callme(int sig_no)
{
        /*do scheduled tasks*/
 }
 int main()
 {
        struct itimerval val;
        struct sigaction action;
        sigemptyset(&action.sa_mask);
        action.sa_handler=(void(*)( )) callme;
        action.sa_flags=SA_RESTART;
        if(sigaction(SIGALARM,&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;
}
```

POSIX.1b Timers

- ✓ POSIX.1b defines a set of APIs for interval timer manipulations. The POSIX.1b timers are more flexible and powerful than are the UNIX timers in the following ways:
- ➢ Users may define multiple independent timers per system clock.
- ➢ The timer resolution is in nanoseconds.
- ➢ Users may specify the signal to be raised when a timer expires.
- ➢ The time interval may be specified as either an absolute or a relative time.
- ✓ The POSIX.1b APIs for timer manipulations are:

```
#include<signal.h>
#include<time.h>
int timer_create(clockid_t clock, struct sigevent* spec, timer_t* timer_hdrp);
int timer_settime(timer_t timer_hdr, int flag, struct itimerspec* val, struct itimerspec* old);
int timer_gettime(timer_t timer_hdr, struct itimerspec* old);
int timer_getoverrun(timer_t timer_hdr);
int timer_delete(timer_t timer_hdr);
```

- ✓ Returns 0 if succeed and -1 if they fail.
- ✓ The *timer_create* API is used to dynamically create a timer and returns its handler.
- ✓ The clock argument specifies which system clock the new timer should be based on. The clock argument value may be CLOCK_REALTIME for creating a real time clock timer.
- ✓ Other values for the clock argument are system dependent.
- ✓ The spec argument defined what action to take when the timer expires. The struct sigevent data type is defined as:

```
struct sigevent
{
        int sigev_notify;
        int sigev_signo;
        union signal sigev_value;
};
```

- ✓ The *sigev_signo* field specifies a signal number to be raised at the timer expiration.
- ✓ It is valid only when the *sigev_notify* field is set to SIGEV_SIGNAL.
- ✓ If the *sigev_notify* field is set to SIGEV_NONE , no signal is raised by the timer when it expires.
- ✓ The *sigev_value* field is used to contain any user-defined data to identify that a signal is raised by a specific timer.
- ✓ The data structure of the *sigev_value* field is:

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

- ✓ If the *spec* argument is set to NULL and the timer is based on CLOCK_REALTIME, the SIGALRM signal is raised when the timer expires.
- ✓ The *timer_hdrp* argument of the *timer_create* API is an address of a *timer_t*-typed variable to hold the handler of the newly generated timer. This argument should not be set to NULL, as the handler is used to call other POSIX.1b timer APIs.
- ✓ The *timer_settime* starts or stops a timer running.
- ✓ The *timer_gettime* API is used to query the current values of a timer.
- ✓ The *struct itimerspec* data type is defined as:

```
struct itimerspec {
        struct timespec    it_interval;
        struct timespec    it_value;
};
```

and the *struct timespec* data structure is defined as:

```
struct timespec {
```

```
        time_t    tv_sec;
        long    tv_nsec;
    };
```

- ✓ The *itimerspec::it_value* specifies the time remaining in the timer and the *itimerspec::it_interval* specifies the new time to reload the timer after it expires.
- ✓ All times are specified in seconds(timespec::tv_sec field) and in nanoseconds(timespec::tv_nsec field).
- ✓ In the *timer_settime* API, the flag argument value may be 0 or TIMER_RELTIME if the timer start time is relative to the current time.
- ✓ If the *flag* argument value is TIMER_ABSTIME, the timer start time is an absolute time.
- ✓ If the *val.it_value* is zero, it stops the timer from running.
- ✓ If the *val.it_interval* is zero , the timer will not restart after it expires.
- ✓ The *old* argument of the *timer_settime* API is used to obtain the previous timer values.
- ✓ The *old* argument value may be set to NULL , and no timer values are returned.
- ✓ The *old* argument of *timer_gettime* API returns the current values of the named timer.
- ✓ The *timer_getoverrun* API returns the number of signals generated by a timer but was lost.
- ✓ The *timer_destroy* API is used to destroy a timer created by the *timer_create* API.
- ✓ The following program illustrates how to set up an absolute-time timer that will go off at 10:27 A.M , March 20,1996:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<time.h>
#define TIMER_TAG   12
void callme (int signo, siginfo_t*  evp, void* ucontext)
{
        time_t  tim=time(0);
                cerr<< "callme:"<<evp->si_value.sival_int
                <<",signo:"<<signo<<","<<ctime(&tim);
}
```

```
int main()
{
        struct sigaction sigv;
        struct sigevent  sigx;
        struct itimerspec val;
        struct tm do_time;
        timer_t t_id;
        sigemptyset(&sigv.sa_mask);
        sigv.sa_flags=SA_SIGINFO;
        sigv.sa_sigaction=callme;
        if(sifaction(SIGUSR1,&sigv,0)== -1){
                perror("sigaction");
                return 1;
        }
        sigx.sigev_notify=SIGEV_SIGNAL;
        sigx.sigev_signo=SIGUSR1;
        sigx.sigev_value.sival_int=TIMER_TAG;
        if(timer_create(CLOCK_REALTIME,&sigx,&t_id)==-1){
                perror("timer_create");
                return 1;
        }
        do_time.tm_hour =10;
        do_time.tm_min =27;
        do_time.tm_sec =30;
        do_time.tm_mon =3;
        do_time.tm_year =96;
        do_time.tm_mday =20;
        val.it_value.tv_sec = mktime(&do_time);
        val.it_value.tv_nsec=0;
        val.it_interval.tv_sec=15;
        val.it_interval.tv_nsec=0;
```

```
cerr<<"timer will go off at:"<<ctime(&val.it_value.tv_sec);
if(timer_settie(t_id, TIMER_ABSTIME,&val,0)==-1){
        perror("timer_settime");
        return 2;  }
for(int i=0;i<2;i++)
pause();
if(time_delete(t_id) ==-1){
        perror("timer_delete");
        return 3;  }
return 0;  }
```

Output:

```
% CC posix_timer_sbs.C  -o  posix_timer_abs
%  posix_timer_abs
Timer  will go off at: Sat March 20  10:27:30 1996
callme: 12, signo:16 , Sat March 20 10:27:30 1996
callme: 12, signo:16 , Sat March 20 10:27:45 1996
```

- ✓ The above program first sets up the *callme* function as the signal handler for the SIGUSR1 signal.
- ✓ It then creates a timer based on the system real-time clock.
- ✓ The timer should raise the SIGUSR1 signal whenever it expires, and the timer-specific data that should be sent along with the signal is TIMER_TAG.
- ✓ The timer handler returned by the *timer_create* API is stored in the t_id variable.
- ✓ Set the timer to go off on March 20,1996 , at 10:27 Am and 30 seconds and the timer should rerun for every 30 seconds thereafter.
- ✓ The absolute expiration date/time is specified in the *do_time* variable and is being converted to a *time_t* type value via *mktime* function. The *timer_settime* function is called to start the timer running.
- ✓ The program then waits for the timer to expire at the said date/time and expires again 30 seconds later.

- ✓ Before the program terminates, it calls the *timer_delete* to free all system resources allocated for the timer.
- ✓ The program can be modified to use a relative-time timer instead.

```
int main()  {
/*set up sigaction for SIGUSR1*/
…
/*Create a timer using timer_create*/
..
        struct itimerspec val;
        val.it_value.tv_sec=60;
        val.it_value.tv_nsec=0;
        val.it_interval.tv_sec=120;
        val.it_interval.tv_nsec=0;
        if (timer_settime(t_id,0,&val,0)==-1)
                { perror("timer_settime");
                return 2;}
        /*wait for timer expires*/
        …
}
```

Daemon Processes

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 shut down. Because they don't have a controlling terminal, we say that they run in the background.

Daemon Characteristics

- ➢ The ps(1) command prints the status of various processes in the system. There are a multitude of option-consult your system's manual for all the details.
- ➢ If execute ps –axj under BSD-based systems to see the information.

- The –a option shows the status of processes owned by others, -x shows processes that don't have a controlling terminal, -j option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.
- The output from ps looks like

PPID	PID	PGID	SID	TTY	TPGID	UID	COMMAND
0	1	0	0	?	-1	0	Init
1	2	1	1	?	-1	0	[keventd]
1	3	1	1	?	-1	0	[kapmd]
0	5	1	1	?	-1	0	[kswapd]
0	6	1	1	?	-1	0	[bdflush]
0	7	1	1	?	-1	0	[kupdated]
1	1009	1009	1009	?	-1	32	portmap
1	1048	1048	1048	?	-1	0	syslogd -m 0
1	1335	1335	1335	?	-1	0	xinetd -pidfile /var/run/xinetd.pid
1	1403	1	1	?	-1	0	[nfsd]
1	1405	1	1	?	-1	0	[lockd]
1405	1406	1	1	?	-1	0	[rpciod]
1	1853	1853	1853	?	-1	0	Crond
1	2182	2182	2182	?	-1	0	/usr/sbin/cupsd

- Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.
- Process 1 is *init*, is a system daemon responsible for starting system services specific to various run levels.
- The *keventd* daemon provides process context for running scheduled functions in the kernel.
- The *kapmd* daemon provides support for the advanced power management features available with various computer systems.

- The *kswapd* daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.
- The *bdflush* daemon flushes dirty buffers from the buffer cache back to disk when available memory reaches a low-water mark.
- The *kupdated* daemon flushes dirty pages back to disk at regular intervals to decrease data loss in the event of a system failure.
- The *portmapper* daemon provides the service of mapping RPC program numbers to network port numbers.
- The *syslogd* daemon is available to any program to log system messages for an operator.
- The *inetd* listens on the system's network interfaces for incoming requests for various network servers.
- The *nfsd, lockd, rpciod* daemons provide support for the Network File System (NFS).
- The *cron* daemon executes commands at specified dates and times.
- The *cupsd* daemon is a print spooler; it handles print requests on the system.
- 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 parent of most of these daemons is the init process.

Coding Rules

- ✓ **Call umask to 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.
- ✓ **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.
- ✓ **Call setsid to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- ✓ **Change the current working directory to the root directory**. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist

until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

- ✓ **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- ✓ **Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

➢ The following function that can be called from a program that wants to initialize itself as a daemon.

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>
void daemonize(const char *cmd)
{
        int i, fd0, fd1, fd2;
        pid_t pid;
        struct rlimit rl;
        struct sigaction sa;
        /*
        * Clear file creation mask.
        */
                umask(0);
        /*
        * Get maximum number of file descriptors.
        */
```

```
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
                err_quit("%s: can't get file limit", cmd);
/*
* Become a session leader to lose controlling TTY.
*/
        if ((pid = fork()) < 0)
                err_quit("%s: can't fork", cmd);
        else if (pid != 0) /* parent */
                exit(0);
                setsid();
/*
* Ensure future opens won't allocate controlling TTYs.
*/
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
        if (sigaction(SIGHUP, &sa, NULL) < 0)
                err_quit("%s: can't ignore SIGHUP", cmd);
        if ((pid = fork()) < 0)
                err_quit("%s: can't fork", cmd);
        else if (pid != 0) /* parent */
        exit(0);
/*
* Change the current working directory to the root so
* we won't prevent file systems from being unmounted.
*/
if (chdir("/") < 0)
        err_quit("%s: can't change directory to /", cmd);
/*
* Close all open file descriptors.
*/
```

```
        if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
        for (i = 0; i < rl.rlim_max; i++)
        close(i);
    /*
    * Attach file descriptors 0, 1, and 2 to /dev/null.
    */
        fd0 = open("/dev/null", O_RDWR);
        fd1 = dup(0);
        fd2 = dup(0);
    /*
    * Initialize the log file.
    */
        openlog(cmd, LOG_CONS, LOG_DAEMON);
        if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d", fd0, fd1, fd2);
        exit(1);
        }
    }
```

Output:

$ **./a.out**

$ **ps –axj**

PPID	PID	PGID	SID	TTY	TPGID	UID	CMD
1	3346	3345	3345	?	-1	501	./a.out

$ **ps -axj | grep 3345**

1	3346	3345	3345	?	-1	501	./a.out

Error Logging

- ✓ One problem a daemon has is how to handle error messages. It can't simply 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. A central daemon error-logging facility is required.

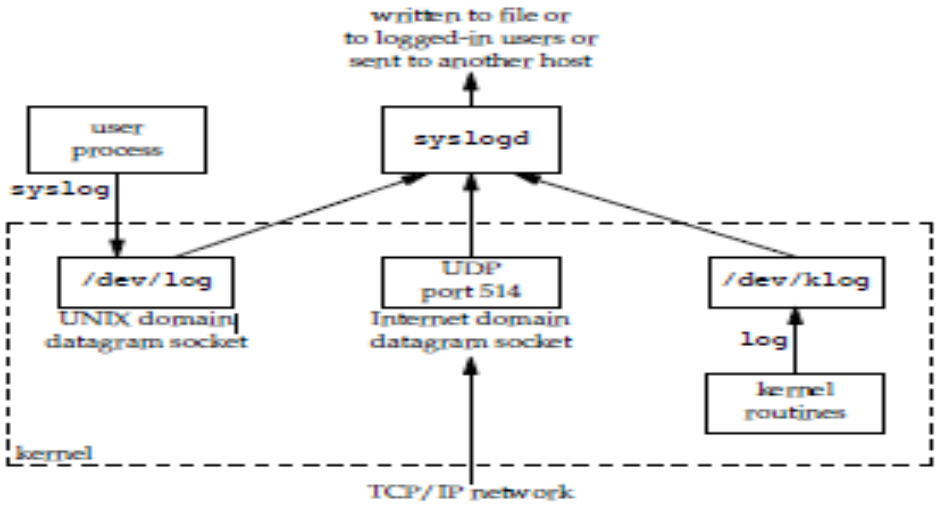

Figure 13.2 The BSD syslog facility

- ✓ There are three ways to generate log messages:
- ➢ Kernel routines can call the log function. These messages can be read by any user process that opens and reads the */dev/klog* device.
- ➢ Most user processes (daemons) call the *syslog(3)* function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket */dev/log*.
- ➢ 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.
- ✓ Normally, the *syslogd* daemon reads all three forms of log messages.
- ✓ On start-up, this daemon reads a configuration file, usually */etc/syslog.conf*, which determines where different classes of messages are to be sent.
- ✓ For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file.
- ✓ Our interface to this facility is through the syslog function.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
```

int setlogmask(int maskpri);

- ✓ Returns previous log priority mask value.
- ✓ 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* specify an *ident* that is added to each log message, which is name of the program.
- ✓ The *option* argument is a bitmask specifying various options.
- ✓ Below figure describes the available options, including a bullet in the XSI column if the option is included in the *openlog* definition in the Single UNIX Specification.

option	XSI	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_NOWAIT`	•	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch `SIGCHLD`, since the application might have retrieved the child's status by the time that `syslog` calls `wait`.
`LOG_ODELAY`	•	Delay the opening of the connection to the `syslogd` daemon until the first message is logged.
`LOG_PERROR`		Write the log message to standard error in addition to sending it to `syslogd`. (Unavailable on Solaris.)
`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`

- ✓ The *facility* argument for openlog is taken from below figure.
- ✓ 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 if 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* and a *level*. These *level*s are ordered by priority, from highest to lowest.

facility	XSI	Description
LOG_AUDIT		the audit facility
LOG_AUTH		authorization programs: login, su, getty, ...
LOG_AUTHPRIV		same as LOG_AUTH, but logged to file with restricted permissions
LOG_CONSOLE		messages written to /dev/console
LOG_CRON		cron and at
LOG_DAEMON		system daemons: inetd, routed, ...
LOG_FTP		the FTP daemon (ftpd)
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_NTP		the network time protocol system
LOG_SECURITY		the security subsystem
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)

- ✓ The *format* argument and any remaining arguments are passed to the vsprintf function for formatting.
- ✓ Any occurrences of the two characters %m in *format* are first replaced with the error message string (strerror) corresponding to the value of errno.
- ✓ The setlogmask function can be used to set the log priority mask for the process.
- ✓ This function returns the previous mask. When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask.
- ✓ Attempts to set the log priority mask to 0 will have no effect.
- ✓ The logger(1) program is also provided by many systems as a way to send log messages to the syslog facility.
- ✓ The logger command is intended for a shell script running noninteractively that needs to generate log messages.

Example

- ✓ In a (hypothetical) line printer spooler daemon, you might encounter the sequence

```
openlog("lpd", 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 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*.
- ✓ In addition to *syslog*, many platforms provide a variant that handles variable argument lists.

```
#include <syslog.h>
#include <stdarg.h>
void vsyslog(int priority, const char *format, va_list arg);
```

Client-Server Model

- ✓ In general, a server is a process that waits for a client to contact it, requesting some type of service.
- ✓ In figure the BSD syslog facility, the service being provided by the syslogd server is the logging of an error message.
- ✓ In figure the BSD syslog facility, the communication between the client and the server is one-way.
- ✓ The client sends its service request to the server; the server sends nothing back to the client.
- ✓ The client sends a request to the server, and the server sends a reply back to the client.