

## MODULE 5

### SIGNALS AND DAEMON PROCESSES

Signals: The UNIX Kernel Support for Signals, signal, Signal Mask, sigaction, The SIGCHLD Signal and the waitpid Function, The sigsetjmp and siglongjmp Functions, Kill, Alarm, Interval Timers, POSIX.1b Timers. Daemon Processes: Introduction, Daemon Characteristics, Coding Rules, Error Logging, Client-Server Model.

Signal is a notification sent to a process to notify it of some event. Signals are generated when an event occurs that requires attention. It can be considered as a software version of a hardware interrupt.

- interrupts whatever the process is doing and force it to handle a signal
- Has an integer number that represents it, and symbolic name.
- Signals are asynchronous- means that the event can occur at any time. May be unrelated to the execution of the process e.g. user types ctrl-C, or the modem hangs
- For example, during floating point error: kernel sends process a signal number 8. Such events are often called interrupts i.e., they interrupt the normal flow of the program to service an interrupt handler.
- Easiest way to send signal to foreground process press Control-C or Control-Z. When terminal driver recognizes a Control-C it sends SIGINT signal to all of the processes in the current foreground job. Control-Z causes SIGTSTP to be sent by default. SIGINT terminates a process SIGTSTP suspends a process.

#### Signal Sources:

Hardware - division by zero

Kernel – notifying an I/O device for which a process has been waiting is available

Other Processes – a child notifies its parent that it has terminated

User – key press (i.e., Ctrl-C)

Signal name	Description	Default action
<b>SIGABRT</b>	abnormal termination (abort)	terminate+core
<b>SIGALRM</b>	timer expired (alarm)	terminate
<b>SIGBUS</b>	hardware fault	terminate+core
<b>SIGCHLD</b>	change in status of child	ignore
<b>SIGCONT</b>	continue stopped process	continue/ignore
<b>SIGFPE</b>	arithmetic exception	terminate+core
<b>SIGHUP</b>	hang up	terminate
<b>SIGILL</b>	illegal instruction	terminate+core
<b>SIGINT</b>	terminal interrupt character	terminate
<b>SIGKILL</b>	termination	terminate

<b>SIGPIPE</b>	write to pipe with no readers	terminate
<b>SIGPROF</b>	profiling time alarm (setitimer)	terminate
<b>SIGPWR</b>	power fail/restart	terminate/ignore
<b>SIGQUIT</b>	terminal quit character	terminate+core
<b>SIGSEGV</b>	invalid memory reference	terminate+core
<b>SIGSTOP</b>	stop	stop process
<b>SIGSYS</b>	invalid system call	terminate+core
<b>SIGTERM</b>	termination	terminate
<b>SIGTRAP</b>	hardware fault	terminate+core

The default action for most of the signals is to terminate a recipient process. Some signals will generate a core file for aborted process so that users can trace back the state of the process when it was aborted.

### **How to handle signals?**

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:

- Accept 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.

## **THE UNIX KERNEL SUPPORT OF SIGNALS**

- 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.
- 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.
- If array entry contains a zero value, the process will accept the default action of the signal.  
If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.  
If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

Note: In addition to the text, data, and stack segment, the OS also maintains for each process a region called the **u-area** (User Area). The **u-area** contains information specific to the process (e.g. open files, current directory, signal action, accounting information) and a

system stack segment for process use.

## **SIGNAL**

The function prototype of the signal API is:

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

The formal arguments of the API are:

sig\_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

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)
{
    cout<<"catch_sig:"<<sig_num<<endl;
}

int main()
{
    signal(SIGTERM,catch_sig); signal(SIGINT,SIG_IGN); signal(SIGSEGV,SIG_DFL);
    pause();          /*wait for a signal interruption*/
}
```

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.

### **Program1**

```
void main ()
{
    signal(SIGINT,SIG_IGN);
    printf ("Looping forever...\n");
    while(1)
        printf ("This line executes till you terminate the process\n");
}

/* output :
```

```
cc sig1.c
./a.out
```

This line executes till you terminate the process

This line executes till you terminate the process

This line executes till you terminate the process...(prints this line till u press ^Z)

^Z

```
[2]+ Stopped      ./a.out */
```

### **Program 2**

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

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

```
void main ()
```

```
{
```

```
signal(SIGINT,SIG_DFL);
```

```
printf ("Looping forever...\n");
```

```
while(1)
```

```
printf ("This line executes till you u press ctrl+C \n");
```

```
}
```

```
/* output
```

This line executes till you u press ctrl+C

This line executes till you u press ctrl+C...

^C

```
cse@ubuntu:~$ */
```

### **program 3**

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

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

```
void call_me(int sig_no)
```

```
{
```

```
    printf("caught signal %d\n", sig_no);
```

```
}
```

```
void main()
```

```
{
```

```
    signal(SIGINT,call_me);
```

```
    pause();
```

```
}
```

```
/* Output:
```

```
./a.out
^C
caught signal 2*/
```

### **SIGNAL MASK**

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);
```

Returns: 0 if OK, 1 on error

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

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
<b>SIG_SETMASK</b>	Overrides the calling process signal mask with the value specified in the <code>new_mask</code> argument.
<b>SIG_BLOCK</b>	Adds the signals specified in the <code>new_mask</code> argument to the calling process signal mask.
<b>SIG_UNBLOCK</b>	Removes the signals specified in the <code>new_mask</code> argument from the calling process signal mask.

Note:

- 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.

### **sigsetops functions:**

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 ]

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");
}
```

## **sigpending API**

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;

}
```

## **SIGACTION**

The **sigaction()** system call is used to change the action taken by a process on receipt of a specific signal. 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 2<sup>nd</sup> argument of the signal API, which 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 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;
}
```

**Extra program:**

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

void sighandler (int signum)
{
    printf("Sighandler: I am in sighandler before sleep\n");
    sleep(3);
    printf("Sighandler: I am in sighandler after sleep\n");
}

void main()
{
    struct sigaction act;
    sigset_t sigmask;
    int rc;
    rc = sigemptyset(&sigmask);
    rc = sigaddset(&sigmask, SIGQUIT); /* Block this signal if the process is in its handler */
}
```



```
act.sa_handler = sighandler;
act.sa_mask = sigmask;

if (sigaction(SIGINT, &act, NULL) < 0)
    perror("Sigaction is failed\n");
while (1)
{
    printf("main: in loop\n");
    sleep(2);
}
}
```

**Output1:**

```
cse@ubuntu:~$ cc sigact1.c
cse@ubuntu:~$ ./a.out
sigemptyset return value: 0
sigaddset return value: 0
main: in loop
main: in loop
^C
Sighandler: I am in sighandler before sleep
^\\
Sighandler: I am in sighandler after sleep
Quit

cse@ubuntu:~$ ./a.out
sigemptyset return value: 0
sigaddset return value: 0
main: in loop
main: in loop
main: in loop
^CSighandler: I am in sighandler before sleep
Sighandler: I am in sighandler after sleep
main: in loop
^\\Quit
cse@ubuntu:~$
```

**Observation:**

If ctrl+C is generated (SIGINT) is generated, during the execution of the process, signal handler is executed.

Now pressing ctrl+\ (SIGQUIT) will be blocked until handler completes execution.

Once ctrl+c handler finishes its task, then ctrl+\ handler is executed by terminating the process.

**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:

- SIGCHLD does not terminate the parent process.
  - Parent process will be awakened.
  - If waitpid() API is used, it 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.
2. 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.
3. 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.

## THE sigsetjmp AND siglongjmp APIs

The function prototypes of the APIs are:

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int val);
```

- sigsetjmp marks one or more locations in a user program and siglongjmp is used to return to any of those marked locations. Thus, these APIs provide interfunction goto capability.
- 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 is similar to setjmp, with 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 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 to ensure the process signal mask is restored properly when "jumping out" from a signal handling function.

The following program illustrates the uses of sigsetjmp and siglongjmp APIs.

```
#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)
    {
        cout<<"return from signal interruption";
        return 0;
    }
    else
        cout<<"return from first time sigsetjmp is called";
    pause();
}
```

## **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 > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which
pid == 1	The signal is sent to all processes on the system for

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<2)
    {
        cout<<"insufficient arguments:" ;
        return -1; }

    pid=atoi(argv[1]);kill(pid,sig);
}
```

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 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: 0 or number of seconds until previously set alarm.

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

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

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

```
#include<unistd.h>
```

```
void wakeup( )
```

```
{ printf("awakened");}
```

```
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;
```

```
}
```

```
alarm (timer);
```

```
pause( );
```

```
return 0;
```

```
}
```

### **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)
```


```
{
```

```
if( sig_no==SIGALRM)
kill(getpid(),SIGTERM);
}
```

```
int main()
{
```

```
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    action.sa_handler=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;
}
```



This code can be replaced with:

```
int main()
{
    signal(SIGALRM,callme);
    alarm(INTERVAL);
    while(1)
    {
        Printf("this is normal operation\n");
        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);
```

## **DAEMON PROCESSES**

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.

### **DAEMON CHARACTERISTICS**

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

### **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.

Example Program:

```
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int daemon_initialise( )
{
    pid_t pid;
    if (( pid = fork() ) < 0)
        return -1;
    else if ( pid != 0)
        exit(0);    /* parent exits */

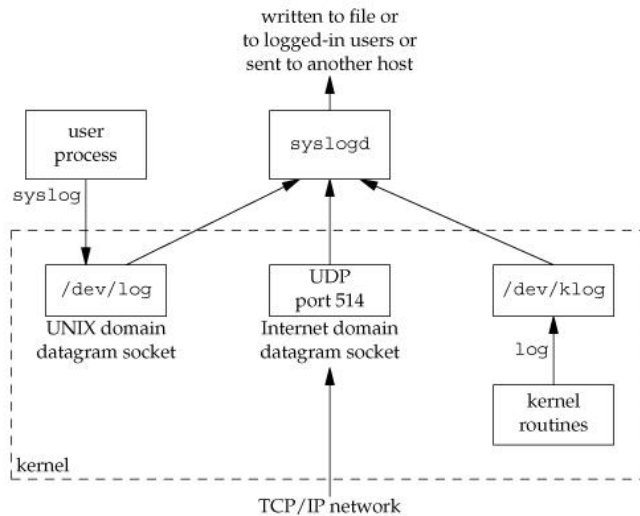
    /* child continues */
    setsid();
    chdir("/");
    umask(0);
    close(0); close(1); close(2);
    return 0;
}
```

### **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.

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);
  
```

### 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 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 the server is one-way. The client sends its service request to the server; the server sends nothing back to the client. In the

upcoming chapters, we'll see numerous examples of two-way communication between a client and a server. The client sends a request to the server, and the server sends a reply back to the client.

```
// program to demonstrate pending signal
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main( void )
{
    sigset_t new,old,pend;
    sigemptyset(&new);
    sigaddset(&new,SIGINT);
    sigprocmask(SIG_BLOCK,&new,&old);
    printf("\n old set %ld\n",old);
    printf("\n new signal set %ld\n",new);

    sleep(3);
    sigpending(&pend);
    if(sigismember(&pend,SIGINT))
    {
        printf("\nsigint pending\n");
        printf("\n signal set %ld\n",pend);
    }
    sigemptyset(&new);
    sigprocmask(SIG_UNBLOCK,&new,0);
    printf("\nsigint unblocked\n");
}

/* output 1
./a.out
old set 0
new signal set 2
^C
sigint pending
signal set 2
sigint unblocked

output2
./a.out
old set 0
new signal set 2
sigint unblocked*/
```

```
// program to demonstrate sigaction API. How to send a process to background, and send signal to it
```

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

```
void callme(int sig_num)
{
    printf("catch signal:%d\n",sig_num);
}
```

```
int main(int ar,char *arr[])
{
    sigset_t sigmask;
    struct sigaction action,old_action;
    sigemptyset(&sigmask);
    sigaddset(&sigmask,SIGTERM);
    sigprocmask(SIG_SETMASK,&sigmask,0);
    sigemptyset(&action.sa_mask);
    sigaddset(&action.sa_mask,SIGSEGV);
    action.sa_handler=callme;
    action.sa_flags=0;
    sigaction(SIGINT,&action,&old_action);
    pause();
    printf("\n%s exits",arr[0]);
    return 0;
}
```

```
/*output
cse@ubuntu:~$ cc sigact.c -o s
cse@ubuntu:~$ ./s &
[3] 5447
cse@ubuntu:~$ kill -INT 5447
catch signal:2 */
```

```
// program to demonstrate SIGCHLD
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
```

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

void signalHandler(int signal)
{
    printf("Cought signal %d!\n",signal);
    if (signal==SIGCHLD) {
        printf("Child ended\n");
        wait(NULL);
    }
}

int main()
{
    signal(SIGALRM,signalHandler);
    //signal(SIGUSR1,signalHandler);
    signal(SIGCHLD,signalHandler);
    if (!fork()) {
        printf("Child running...\n");
        sleep(2);
        printf("Child sending SIGALRM...\n");
        kill(getppid(),SIGALRM); /*send alarm signal to parent*/
        sleep(5);
        printf("Child exiting...\n");
        return 0;
    }
    printf("Parent running, PID=%d. Press ENTER to exit.\n",getpid());
    getchar();
    printf("Parent exiting...\n");
    return 0;
}
```

/\* output

./a.out

Parent running, PID=5734. Press ENTER to exit.

Child running...

Child sending SIGALRM...

Cought signal 14!

Child exiting...

Cought signal 17!

Child ended

Parent exiting... \*/

Output 2

./a.out

Parent running, PID=5816. Press ENTER to exit.( enter button pressed)

Child running...

```
Parent exiting...
cse@ubuntu:~$ Child sending SIGALRM...
Child exiting...
```

This is an example of how signals and signal handlers work. The key function in this example is `signal()`. This will help you implement the "&" operator.

Normally, when the user types in a command, your shell should `fork()` and `execvp()` the appropriate programs, then `waitpid()` for them to end before waiting for the next command. But, if the command has a "&" at the end, it should present the prompt and wait for the next command right away. In that case, there is the problem that the parent process should call `wait()` or `waitpid()` as soon as a child finishes, in order to "clean up" the memory the child held. This is solved with signals.

When calling `signal()`, a process is instructed that it should execute the appropriate signal handler whenever a specific signal is delivered to it. Signals are delivered to note various events, like "interrupt from keyboard" (`SIGINT`) or "child ended" (`SIGCHLD`). For more information, type "man 3 signal" or "man 7 signal".

The idea is that whenever a child finishes, `SIGCHLD` is "automatically" delivered to the parent process. So, the parent can call `wait()` or `waitpid()` as soon as it receives `SIGCHLD`!

In the following example, note the header of `signalHandler()`. The header of any signal handler should be like that (any signal handler should have an `int` as a parameter and return `void`).

The example is pretty simple. A signal handler is set for `SIGALRM`, `SIGUSR1` and `SIGCHLD` (type "man 7 signal" for details). It sends `SIGALRM` to the parent and then finishes (again, sending `SIGCHLD`).