MODULE-5

Chapter 11 Signals and Daemon Processes

Topics

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.

Introduction

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

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

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 affect 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.

11.1 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.

11.2 SIGNAL

The function prototype of the signal API is:

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

The formal argument 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)
{
	signal (sig_num,catch_sig);
	cout<<"catch_sig:"<<sig_num<<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 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.

11.3 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 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.
- 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.

  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");
}
```

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);
```

11.4 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 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;
}
```

11.5 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:

- 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.

11.6 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);
```

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

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

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 process ID.

11.8 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. 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( )
{ ; }
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;
 }
```

11.9 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 addition to alarm API, UNIX also invented 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:

ITIMER_REAL	decrements in real time and generates a SIGALRM signal when it expires
ITIMER_VIRTUAL	decrements in virtual time (time used by the process) and generates a SIGVTALRM signal when it expires.
ITIMER_PROF	decrements in virtual time and system time for the process and 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*/
};
```

Example program:

```
#include<stdio.h>
#include<unistd.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;
}
```

The setitimer and getitimer APIs return a zero value if they succeed or a -1 value if they fail.

11.10 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);
```

Chapter 12 DAEMON PROCESSES

12.1 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 shut down.

12.2 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.

12.3 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 = for() ) < 0)
return -1;
else
if ( pid != 0)
exit(0); /* parent exits */
/* child continues */
setsid( );
chdir("/");
umask(0);
return 0;
}
```

12.4 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.

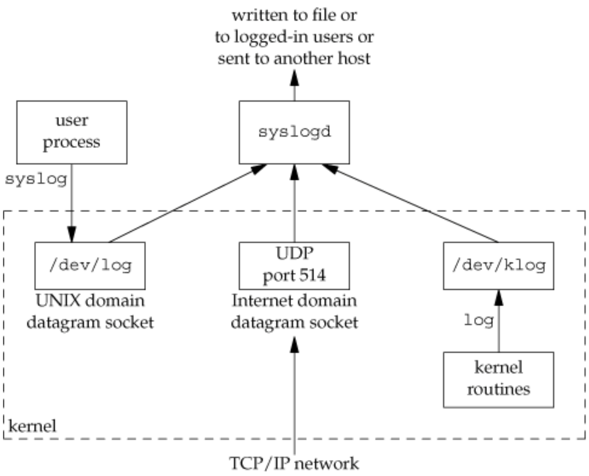

Fig 12.1 The BSD syslog facility

There are three ways to generate log messages:

- Kernel routines can call the logfunction. These messages can be read by any user process that opens and
  reads the /dev/klogdevice.
- 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);
```

11.5 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 12.1, the service being provided by the syslogdserver is the logging of an error message.

In Figure 12.1, 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.