Module – 5

Signals and Daemon Processes

➢ Signals

Signals are software interrupts. They are triggered by events and are posted on process to notify it that something has happened and requires some action.

An event can be generated from a process, a user or the UNIX kernel.

For example if a process performs a divide by zero mathematical operation or dereferences a NULL pointer, the kernel will send the process a signal to interrupt it.

Parent and child processes send the signals to each other for process synchronization.

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: signal_num is a signal identifier like SIGINT or SIGTERM.
The handler argument is the function pointer of a user-defined signal handler function.

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:

1. Accept the **default action** of the signal, which for most signals will terminate the process.
2. Ignore the signal. The signal will be discarded and it has no effect whatsoever on the recipient process.
3. 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.

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;
  }
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.

The commonly found signals in UNIX system.

Name	Description	Core file generated at default
SIGABRT	Abort process execution	YES
SIGALRM	timer expired (alarm timer)	NO
SIGFPE	Illegal mathematical operation	YES
SIGHUP	Controlling terminal hangup	NO
SIGCHLD	Send to parent process when child process terminates	NO
SIGINT	Process interruption	NO
SIGSEGV	invalid memory reference (Segmentation fault)	YES
SIGPIPE	Illegal write to a pipe	YES
SIGTERM	Process termination	YES
SIGQUIT	Process quit	YES
SIGKILL	Kill a process	YES
SIGCONT	Resume execution of a stopped process	NO

SIGSTOP	Stop process execution	NO
SIGTSTP	Stop a process execution by control+Z key	NO

SIGILL	Execution of illegal machine instruction	YES
SIGUSR1	user-defined signal	NO
SIGUSR2	user-defined signal	NO
SIGTTIN	Stop a background process when its try to read from its controlling terminal	NO
SIGTTOU	Stop a background process when its try to write to its controlling terminal	NO

➢ 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.

➢ Signal mask

Each process in UNIX system has a signal mask that defines which signals are blocked when generated to a process.

A process initially inherits the parent’s signal mask when it is created, but any pending signals for the parent process are not passed on.

A process may query or set its signal mask via the sigprocmask API.

#include <signal.h>

int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);

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.

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 is a datatype defined in a <signal.h>, it contains a collection of bit flags

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

#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)
                perror("sigprocmask");
                exit(1);
        }
        else sigaddset(&sigmask,SIGINT); /*set SIGINT flag*/
        sigdelset(&sigmask, SIGSEGV);        /*clear SIGSEGV flag*/
        if(sigprocmask(SIG_SETMASK,&s
                igmask,0)==-1)
                perror("sigprocmask");
}
```

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

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.

#include<signal.h>

int *sigpending*(sigset_t* sigmask);

Returns 0 if OK, -1 if fails.

- sigmask argument is the address of the sigset_t variable which contain set of signals are pending for the calling process by the API.

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);
```

➤ Sigaction

The sigaction API is the replacement for the signal API in latest UNIX and POSIX system. Like the signal API, the sigaction API is called by a process to setup a signal handling method for each signal it wants to deal with.

Furthermore the sigaction API blocks the signal it is catching, allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

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

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 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
   - 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.

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

setjmp or sigsetjmp mark one or more locations in a user program. Later on the program may call the longjmp or siglongjmp API to return to any one of those marked location.

The functions prototype 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's 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 sigsetjmp and setjmp function is that sigsetjmp has an additional argument (i.e. savemask).

If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. Otherwise the signal mask is not saved.

The siglongjmp API does all the operations as the longjmp API. But it also restores a calling process signal mask if the mask was saved in env argument.

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.

➢ 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 sender and the receiver process must be related such that either the sender process real or effective user ID matches that of the receiver process or the sender process has the super user privilege.

For example a parent and the child process can send signal to each other via kill API.

The kill API prototype 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 (+ve value)	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 (-1)	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.

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.

There can be one or more pid specified, and the kill command will send the signal <signal_num> to each process that corresponds to a pid.

➢ 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 alarm API is:

```
#include<signal.h>
unsigned int alarm(unsigned int time_interval);
```

Returns: the return value of the alarm API is the number of CPU seconds left in the process timer, as set by the previous alarm system call.

A process alarm clock is not passed on to its child process.

The time_interval argument is the number of CPU seconds elapse time, the kernel will send SIGALRM signal to the calling process. If a time_interval value is zero then 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;
 }
```

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

In the above example, the sleep function sets up a signal handler for the SIGALRM, calls the alarm API to request the kernel to send the SIGALRM signal (after the timer interval) and finally suspends its execution via 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*/
}
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 set up callme as the signal handling function for the SIGALRM signal.

The program then invokes the alarm API to send itself the SIGALRM after 5 clock seconds. Then the program goes to perform its normal operation.

When the timer expires, the callme unction is invoked which restart the alarm clock after 5 seconds and then does the scheduling task.

When callme function returns, the program continuous its normal operation until another timer expired.

In addition to alarm API, UNIX also invented the **setitimer API**, which provides additional capabilities to those of alarm API.

- The setitimer resolution time is in microseconds, whereas the resolution time for alarm is in seconds.
- The alarm API is used to set up one real time clock timer per process. The setitimer API can be used to define up to three different types of timers in a process:

  i. Real time clock timer
  ii. Timer based on the user time spent by a process
  iii. 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);

both the above APIs return a zero value if they succeed or a -1 value if they fail.

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.
ITIMER_VIRTUAL	Timer based on user time spent by a process.
ITIMER_PROF	Timer based on total user and system times spent by a process.

The struct itimerval datatype is defined in <sys/time.h> header as struct itimerval

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

➢ POSIX.1b timers

The POSIX.1b defines a set of APIs for interval timer manipulations.

The POSIX.1b timers are more flexible and powerful than UNIX timers in the following ways.

- Users may define multiple independent timers per system clock
- The timer resolution is in nanoseconds
- User may specify, on a per-time basis, the signal to be raised when a timer expires.
- The timer interval may be specified as either an absolute or relative time.

There is a limit how many POSIX timers can be created per process. The maximum limit is the TIMER_MAX constant as defined in the <limits.h>

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);

- The **timer_create** API is used to dynamically create a timer and returns its handler.

The **timer_settime** API starts or stops a timer running.

- The **timer_gettime** API returns the current values of the named timer.
- The **timer_getoverrun** API returns the number of signals generated by timer but was lost(overrun).
- The **timer_delete** API is used to destroy a timer created by the timer_create API.

➢ 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.

They run in the background, because they don't have a controlling terminal.

➢ Daemon characteristics

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.
- Processes 0, 1 and 2 are the special processes and exist for the entire life time of the system. They don't have parent process ID, process group ID and no session ID.
- sendmail is a standard mailer demon.
- update is a demon which flushes the kernel's buffer cache to disk at regular interval (usually every 30 seconds).
- cron demon executes the command at specified dates and times.
- syslogd demon log system messages for an operator.
- The parent of all these demons is the init process.

➢ Coding rules

1. 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.

2. 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.

3. 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.

4. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent.

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;
```

```
    }
```

➢ Error logging

One problem a daemon has that 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. And we also don't want each daemon writing its own error messages into a separate files. So a central daemon error-logging facility is required.

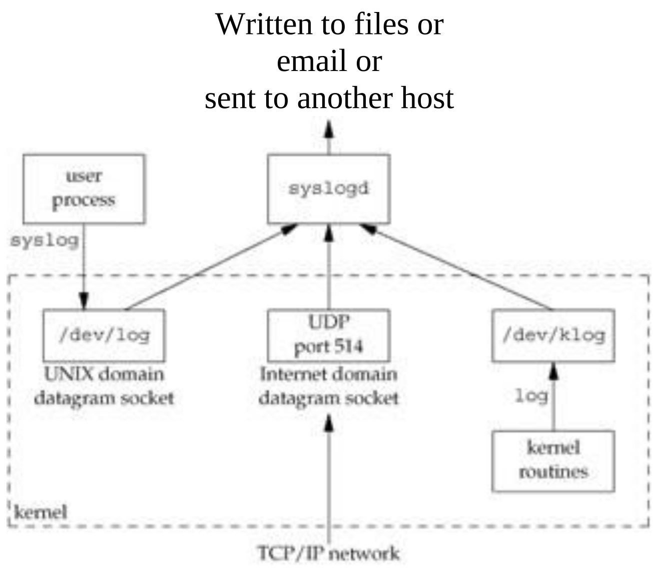

Figure: BSD syslog facility

There are three ways to generate log messages:

1. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device.

2. 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.

3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514.

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.

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);
```

calling openlog is optional. When syslog is called first time, openlog will be called automatically.
Calling closelog is also an optional, it just closes the descriptor that was used to communicate with syslogd daemon.

The **ident** argument in openlog is the name of the program that is added to each log message.

The **facility** argument specifies that messages from different facilities are to be handled differently.

Syslog is called to generate the log message. The **priority** argument is the combination of facility and level. These levels are ordered by priority from highest to lowest.

The **format** argument is used to formatting.