SIGNALS & DAEMON PROCESS

UNIT - IV

- **Signals and Daemon Processes:**
- Introduction
- Signals: The UNIX Kernel Support for Signals,
- signal, Signal Mask, sigaction,
- The SIGCHLD Signal and waitpid API,
- The sigsetjmp and siglongjmp Functions,
- kill, alarm, Interval Timers.

- **Daemon Processes:**
- Introduction,
- Daemon Characteristics,
- Coding Rules,
- Error Logging,
- Client-Server Model.

SIGNALS AND DAEMON PROCESSES

SIGNALS

- Signals are software interrupts.
- If a process performs divide by zero mathematical operation, the kernel will send the signal to interrupt it.
- User hits the <DELETE> or<Ctrl-C> - kernel sends a signal to interrupt it.
- Parent and child processes.

SIGNALS

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGFPE	arithmetic exception	terminate+core
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGKILL	termination	terminate
SIGPIPE	write to pipe with no readers	terminate
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTOP	stop	stop process

SIGNALS

Name	Description	Default action
SIGTTOU	background write to control tty	stop process
SIGUSR1	user-defined signal	Terminate
SIGUSR2	user-defined signal	Terminate
SIGTERM	termination	Terminate

- 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 affect 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.

Unix kernel supports of signals

1. In Unix System V.3, each entry in the kernel process table slot has an array of signal flags, one for each defined in the system.

2. 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.

3. If the recipient process is asleep (waiting a child to terminate or executing *pause* API) the kernel will awaken the process by scheduling it.

4. 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.

5. The kernel will consult the array to find out how the process will react to the pending signal.

6. If the array entry contains a zero value, the process will accept the default action of the signal, and the kernel will discard it.

7. If the array entry contains a one value, the process will ignore the signal.

8. Finally, if the array entry contains any other value, it is used as the function pointer for a used defined signal hander routine.

9. The kernel will setup the process to execute the function immediately, and the process will return to its current point of execution (or to some other place if signal hander does a long jump), if the signal hander does not terminate the process.

10. If there are different signals pending on a process, the order in which they are sent to a recipient process in undefined.

11. If multiple instances of a signal are pending on a process, it is implementation – dependent on whether a single instance or multiple instances of the signal will be delivered to the process.

12. In UNIX System V.3, each signal flag in a process table slot records only whether a signal is pending, but not how many of them are present.

SIGNALS

- The default action for most signals is to terminate a recipient process.
- Some signals are ignored.
- Process ignores so that it is not interrupted while doing certain mission-critical work- process-update database file should not be interrupted else the file will be corrupted.

SIGNALS

The function prototype of the signal API is:

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

The prototype for the signal function states that the function requires two arguments and returns a pointer to a function that returns nothing(void).

The signal function's first argument, sig_no, is a signal identifier- SIGINT/SIGTERM.

The second argument is a function pointer of a user defined signal handler function.

It takes integer value and does not return any value.

The function prototype of the signal API is:

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

If we examine the system's header <signal.h>, we probably find declarations of the form

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

SIG_IGN – signal is to be ignored.

SIG_DFL – accept the default action of a signal

```
/* 7.1.cpp */
#include <iostream>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
using namespace std;
void catch_signal( int signum )
{
     cout << "I got signal number : " << signum << endl;
     signal ( SIGINT, SIG_DFL);
}
int main()
{
     signal( SIGINT, catch_signal );

}
```

Output:

```
$ ./a.out
^C
I got signal number: 2
^C
$
```

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() /*main function*/
{
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.

Unix system supports sigset API- same as signal.

#include<signal.h>

void(*sigset (int signal_num,void (*handler)(int)))(int);

Arguments and return value is same as signal.

Signal API-unreliable

Sigset API-reliable

Multiple instances of signals arrive- one of them is handled other instances are blocked

SIGNAL MASK

- The collection of signals that are currently blocked is called the *signal mask*
- Each process- has signal mask that defines which signals are blocked when generated to a process.
- You can block or unblock signals with total flexibility by modifying the signal mask.
- A blocked signal depends on the recipient process to unblock it & handle it accordingly.
- If a signal is to be ignored & blocked- implementation dependent whether such signal will be discarded or left pending
- 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
- cmd argument specifies determines how the signal mask is changed.
- The last argument, *old_mask*, is used to return information about the old process signal mask. If you just want to change the mask without looking at it, pass a null pointer as the *oldset* argument.

SIGNAL MASK

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	Block the signals in set—add them to the existing mask.
SIG_UNBLOCK	Unblock the signals in *set*—remove them from the existing mask.

SIGNAL SETS

BSD UNIX & POSIX.1 define a set of API :

#include <signal.h>
int sigemptyset(sigset_t *set);
- clears all signal flags

int sigfillset(sigset_t *set);
-sets all the signal flags

int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
-add and delete respectively signal *signo* from *set*.

Returns: 0 if OK, -1 on error.
***set* (Input) A pointer to a signal set.**
***signo* (Input) A signal from the list defined in Control Signals Table.**

SIGNAL SETS

int sigismember(const sigset_t *set, int signo);

- tests whether *signo* is a member of *set*.

Returns: 1 if true- if signo signal flag is set, 0 if false, –1 on error

- The function sigemptyset() initializes the signal set pointed to by set so that all signals are excluded
- The function sigfillset() initializes the signal set so that all signals are included.
- All applications have to call either sigemptyset() or sigfillset() once for each signal set, before using the signal set.
- Once we have initialized a signal set, we can add and delete specific signals in the set.
- The function sigaddset() adds a single signal to an existing set, and sigdelset() removes a single signal from a set.

SIGNAL MASK

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

```
#include <stdio.h>
#include <signal.h>  int main()
{
sigset_t    mask;
sigemptyset(&mask);
/*initialize set*/
if (sigprocmask(0, 0, &mask) == -1)
{ /*get current signal mask*/
perror("sigprocmask");  exit(1);
}
```

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

SIGPENDING FUNCTION

The sigpending function returns the set of signals that are blocked from delivery and currently pending for the calling process.

The set of signals is returned through the set argument

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

Returns: 0 if OK, –1 on error.

```
#include <iostream>
#include <stdio.h>
#include <signal.h>
using namespace std;
int main()
{
  int i;
  sigset_t sigmask;
  sigemptyset( &sigmask );
  if( sigpending( &sigmask ) == -1 )
        perror( "sigpending" );
  else
        int i = sigismember(&sigmask, SIGTERM)? 1 : 0 ;
  if( i == 1 )
        cout << "SIGTERM is set" << endl;
  else
        cout << "SIGTERM is not set" << endl;
  return 0;
}
```

sigaction() Function

- The sigaction API is a replacement for the signal API in the latest UNIX and POSIX systems.
- The sigaction API is called by a process to set up a signal handling method for each signal it wants to deal with.
- sigaction API returns the previous signal handling method for a given signal.

Sigaction() Function

#include <signal.h>

int sigaction(int signo, const struct sigaction *action,struct sigaction *old_action);

Returns: 0 if OK, –1 on error

- The **sigaction**() system call is used to change the action taken by a process on receipt of a specific signal. *signo* specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.
- If *action* is null pointer, the calling process's existing signal handling method for signo will be unchanged.

sigaction() Function

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

```
struct sigaction
{
void (*sa_handler)(int);      /* addr of signal handler, or SIG_IGN, or SIG_DFL
sigset_t   sa_mask;           */
int sa_flags;                 /* additional signals that a process wantsto block
};                            */
                              /* sa_flags specifies a set of flags which modify the
                              behaviour of the signal handling process.*/
                              SA_NOCLDSTOP: If signum is SIGCHLD, do not
                              receive notification when child processes stop or
```

sigaction() Function

- The sa_handler field can be setto SIG_IGN, SIG_DFL, or a user defined signal handler function.
- The sa_mask field specifies additional signals that process wishes to block when it is handling signo signal.
- The signalno argument designates which signal handling action is defined in the *action* argument.
- The previous signal handling method for signalno will be returned via the old action argument if it is not a NULL pointer.
- If actionargument is a NULL pointer, the calling process‘s existing signal handling method for signalno will be unchanged.

```
#include <signal.h>
#include <iostream.h>  void callme ( int
sig_num )
{
cout <<“catch signal:”<<sig_num<< endl;
}
int main(void)
{
sigset_t sigmask;
struct sigaction action, old_action;
sigemptyset(&sigmask);
```

```
if ( sigaddset( &sigmask, SIGTERM) == -1    ||
sigprocmask( SIG_SETMASK, &sigmask, 0) == -1)
perror(“Set signal mask”);
sigemptyset( &action.sa_mask);  sigaddset(
&action.sa_mask, SIGSEGV);  action.sa_handler =
callme;  action.sa_flags = 0;
if (sigaction (SIGINT, &action, &old_action) == -1)
perror(“sigaction”);
pause(); /* wait for signal interruption*/  return
0;
}
```

sigaction FUNCTION

- In the program, the process signal mask is set with 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 terminates its execution via the pause API.

The output of the program would be as:

```
% cc     sigaction.c –o  sigaction
% ./sigaction &  [1]     495
% kill –INT 495  catch signal: 2  sigaction exits
[1]  Done    sigaction
```

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(parent suspended by waitpid system call).
   - 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.

THE SIGCHLD SIGNAL AND THE waitpid API

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, signal handler function returns, 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

- Its like –goto

- Sigsetjmp mark one or more locations in a user program.
- Siglongjmp return to any of those marked location.

- Function prototype

```
#define _POSIX_SOURCE
#include <setjmp.h>
int sigsetjmp(sigjmpbuf env, int savemask);//return 0 if called directly & nonzero returning
                                           from siglongjmp
int siglongjmp(sigjmpbuf env, int ret_val);
```

- Sigsetjmp has a second argument save_sigmask, which allows user to specify whether signal mask should be saved to the env argument. If the save_mask argument is nonzero then signal mask is saved.Otherwise signal mask is not saved.
- In Siglongjmp where env argument saves the location and ret_value argument specifies return value of sigsetjmp API when it is called by siglongjmp.Its value should be nonzero number and if it is zero siglongjmp API will reset it to 1.

```
#include <signal.h>
#include <iostream.h>
void callme ( int sig_num )
{
cout <<“catch signal:”<<sig_num<< endl;
siglongjmp(env,2);
}
int main(void)
{
sigset_t sigmask;
struct sigaction action, old_action;
sigemptyset(&sigmask);
if ( sigaddset( &sigmask, SIGTERM) == -1
    ||sigprocmask( SIG_SETMASK,
&sigmask, 0) == -1)
perror(“Set signal mask”);
sigemptyset( &action.sa_mask);
sigaddset( &action.sa_mask, SIGSEGV);
action.sa_handler =(void(*)()) callme;
action.sa_flags = 0;
if (sigaction (SIGINT, &action, &old_action) == -1)
perror(“sigaction”);
If(sigsetjmp(env,1)!=0) {
cerr<<“Return from signal interruption\n”;
return 0;
}
else  cerr<<“Return from first time sigsetjmp is
called\n”;
pause(); /* wait for signal interruption*/  return 0;
}
```

The output of the program would be as:

```
% cc     sigsetjmp.c
% a.out &
[1] 411
Return from first time sigsetjmp   is called
% kill –INT 411
catch signal: 2
Return from signal interruption
[1] Done    a.out
```

kill Function

A process can send a signal to a related process via the kill API. 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. (Parent kills all its childrens)

```
int main(int argc,char** argv)
{
 int pid, sig = SIGTERM;
if(argc==3)
{
if(sscanf(argv[1],”%d”,&sig)!=1) //get a signal number
{
cerr<<”invalid number:” << argv[1] << endl;
return -1;
}
argv++,argc--;
}
while(--argc>0)
if(sscanf(*++argv, “%d”, &pid)==1)   //get a process id
{
if(kill(pid,sig)==-1)
perror(“kill”); }
else cerr<<”invalid pid:” << argv[0] <<endl;
return 0;
}
```

The UNIX kill command invocation syntax is: Kill [-<signal_num>] <pid>..... Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

alarm Functions

- The alarm function allows us to set a timer that will expire at a specified time in the future.
- When the timer expires, the SIGALRM signal is generated.
- If we ignore or don't catch this signal, its default action is to terminate the process.

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

- Return value- the number of CPU seconds left in the process timer.
- The time_interval argument is the number of CPU seconds elapse time, the moment the time period elapses, the kernel will send the SIGALRM signal to the calling process.
- If a time_interval value is zero, it turns off the alarm clock.

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

- If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function.

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_mask;
flags=0;
sigemptyset(&action.sa_mask);
if(sigaction(SIGALARM,&action,0)==-1)
{
perror("sigaction");
return -1;
}
(void) alarm (timer); //timer value is 10 sec
(void) pause( );
return 0;
}
```

Interval Timers

- Use of alarm API- using sleep suspends a process for fixed amount of time.
- Alarm API – can used to set up interval timer in a process.
- Interval timer can be used to make a process do some tasks at a fixed interval/execution of some operations/do some task.

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=0;
 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 timer based on real- time clock. Generates a SIGALRM signals when it expires.

- ITIMER_VIRTUAL timer based on user-time spent by a process. Generates 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*/
};
```

- val.it_value is the time to set the named timer.-0 value stops the named timer if it is running.
- val.it_interval is the time to reload the timer when it expires.- set to 0 if timer is to run once

- getitimer API, the old.it_value and the old.t_interval return the named timers remaining time(to expiration) and the reload time respectively.

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.
- Because they don't have a controlling terminal, we say that they run in the background.
- UNIX systems have numerous daemons that perform day-to-day activities.

DAEMON PROCESSES

Deamon Characteristic

- $ps -axj

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

Coding Rules

Following are the coding rules to code a daemon:

1. Call umask to set the file mode creation mask to a known value, usually 0
2. Call fork and have the parent exit.
3. Call setsid to create a new session.
4. Change the current working directory to the root directory.
5. Unneeded file descriptors should be closed.
6. Some daemons open file descriptors 0, 1, and 2 to /dev/null

Error Logging

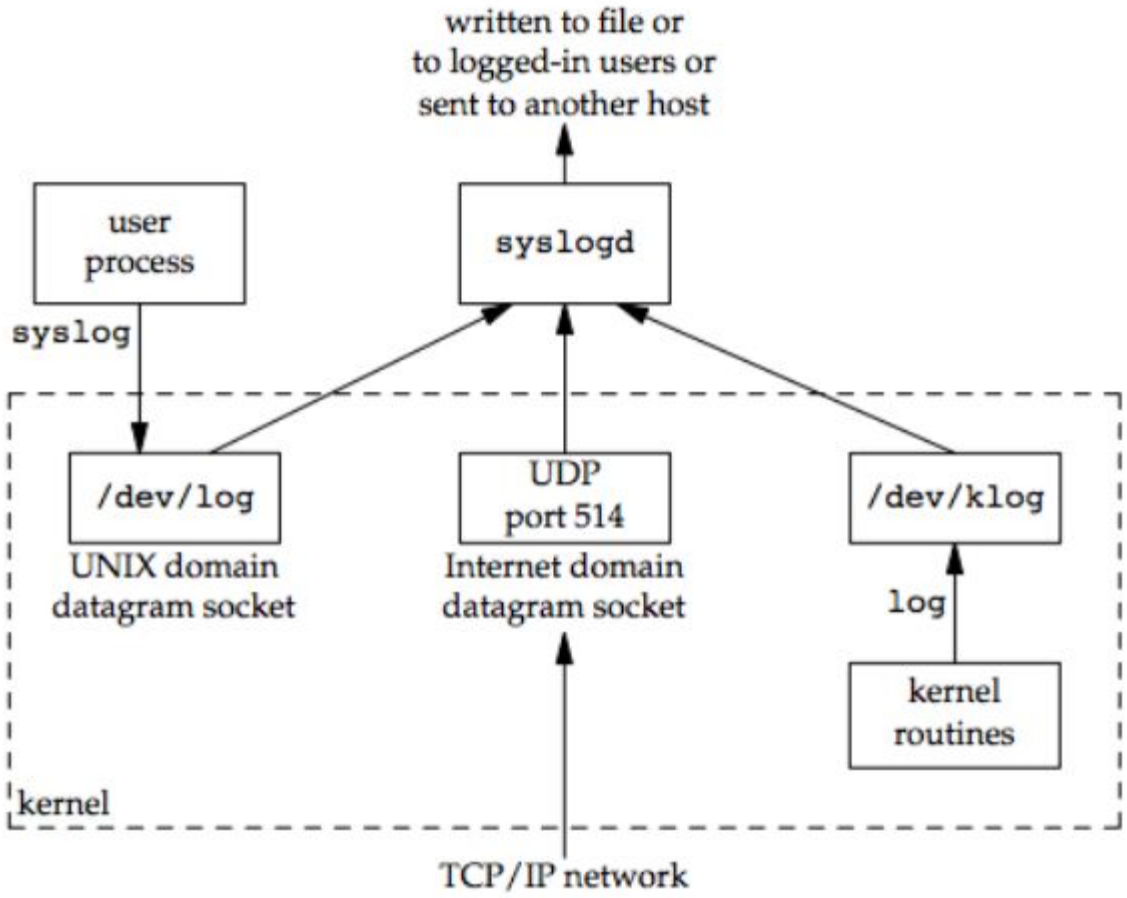
One problem a daemon has is how to handle error messages. It cannot (simply) write to:

- Standard error: it shouldn't have a controlling terminal.
- Console device: on many workstations the console device runs a windowing system.
- Separate files: it's a headache to keep up which daemon writes to which log file and to check these files on a regular basis.

SIGNALS AND DAEMON PROCESSES

A central daemon error-logging facility is required. The BSD syslog facility has been widely used since 4.2BSD. Most daemons use this facility.

The following figure illustrates its structure:



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 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

Client–Server Model:

A common use for a daemon process is as a server process. syslogd process (is a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

A server is a process that waits for a client to contact it, requesting some type of service. The service provided by the syslogd server is the logging of an error message.