

MODULE 5

SIGNALS & DAEMON PROCESSES

Introduction

- ✓ Signals are triggered by events and are posted on a process to notify it that some thing has happened and requires some action.
- ✓ Signals can be generated from a process, a user, or the UNIX Kernel.
- ✓ Example:
 1. A user performs a divide by zero or dereferences a NULL Pointer.
 2. A user hits <Delete> or <Ctrl+C> key at the keyboard.
- ✓ A parent & child process can send signals to each other for process synchronization.
- ✓ Thus signals are the software version of hardware interrupts.
- ✓ Signals are defined as integer flags in the <signal.h> header file.
- ✓ The following table shows the list of POSIX-defined signals found in most UNIX System.

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	thread 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	thread library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (poll)	terminate
SIGPROF	profiling time alarm (set timer)	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	checkpointthaw	ignore
SIGTRAP	hardwarefault	terminate+core
SIGTSTP	terminalstopcharacter	stopprocess
SIGTTIN	backgroundreadfromcontroltty	stopprocess
SIGTTOU	backgroundwritetocontroltty	stopprocess
SIGURG	urgentcondition(sockets)	ignore
SIGUSR1	user-definedsignal	terminate
SIGUSR2	user-definedsignal	terminate
SIGVTALRM	virtualtimealarm(setitimer)	terminate
SIGWAITING	threadlibraryinternaluse	ignore
SIGWINCH	terminalwindowssizechange	ignore
SIGXCPU	CPUlimitexceeded(setrlimit)	terminate+core/ignore
SIGXFSZ	filesize-limitexceeded(setrlimit)	terminate+core/ignore
SIGXRES	resourcecontrolexceeded	ignore

- ✓ When a signal is sent to process, it is pending on the process to handle it.
- ✓ The process can react to signals in one of the three ways.
 1. Accept the default action of the signal – most signals terminate the process.
 2. Ignore the signal.
 3. Invoke a user defined function – The function is called signal handler routine and the signal is said to be caught when the function is called. If the function finishes the execution without terminating the process, the process will continue the execution from the point it was interrupted by the signal.
- ✓ Most signals can be caught or ignored except the SIGKILL & SIGSTOP signals.
- ✓ A companion signal to SIGSTOP is SIGCONT, which resumes the process execution after it has been stopped, both SIGSTOP & SIGCONT signals are used for job control in UNIX.
- ✓ A process is allowed to ignore certain signals so that it is not interrupted while doing certain mission-critical work.
- ✓ Ex: A DBMS Process Updating a database file should not be interrupted until it finishes else database file will be corrupted, it should restore signal handling actions for signals when finished mission-critical work.
- ✓ Because signals are generated asynchronously to a process, a process may specify a per signal handler function, these functions would then be called when their corresponding signals are caught.
- ✓ A common practice of a signal handler function is to clean up a process work environment, such as closing all input-output files, before terminating gracefully.

The UNIX Kernel Support for 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 user-defined signal handler 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 handler does a long jump), if the signal handler 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 is 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.

Signal:

1. All Unix Systems and ANSI – C support the signal API, which can be used to define the per-signal handling method.
2. The function prototype of the signal is:

```
#include<signal.h>

void(*signal(intsignal_num,void(*handler)(int))int);
```

3. `signal_num` is the signal identifier like `SIGINT` or `SIGTERM` defined in the `<signal.h>` handler is the function pointer of a user defined signal handler function. This function should take an integer formal argument and does not return any value.

4. Example below attempts to catch the `SIGTERM`, ignores the `SIGINT` and accepts the default action of the `SIGSEGV` signal.

5. The `pause` API suspends the calling process until it is interrupted by the signal and the corresponding signal handler does return.

Program:

```
#include<iostream.h>#include<signal.h>
>
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 signal interruption
}
```

6. The `SIG_IGN` & `SIG_DFL` are manifest constants defined in `<signal.h>`

```
#define SIG_IGN  void(*) (int) 1 //Ignore the signal
#define SIG_DFL  void(*) (int) 0 //Default action
```

7. The return value of `signal` API is the previous signal handler for the signal.

8. UNIX system V.3 and V.4 support the `sigset` API, which has the same prototype and similar use as `signal`.

```
#include<signal.h>

void(*sigset(intsignal_num,void(*handler)(int))int);
```

9. The `sigset` arguments and return value is the same as that of `signal`.
10. Both the functions set signal handling methods for any named signal, but `signal` API is unreliable and `sigset` is reliable.
11. this means that when a signal is set to be caught by a signal handler via `sigset` when multiple instances of the signal arrive one of them is handled while other instances are blocked. Further the signal handler is not reset to `SIG_DFT` when it is invoked.

SignalMask

1. Each process in UNIX or POSIX system has signal mask that defines which signals are blocked when generated to a process.
2. A blocked signal depends on the recipient process to unblock it and handle it accordingly.
3. If a signal is specified to be ignored and blocked, it is implementation dependent on whether the signal will be discarded or left pending when it is sent to the process.
4. 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.
5. 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);
```

`new_mask` defines a set of to be set or reset in a calling process signal mask. `cmd` specifies how the new mask value is to be used by the APL. The possible values `cmd` are:

Command	Meaning
SIG_SETMASK	Override the calling process signal mask with the value specified in the <code>new_mask</code> argument.
SIG_BLOCK	Adds the signals specified in the <code>new_mask</code> argument to the calling process signal mask.
SIG_UNBLOCK	Remove the signals specified in the <code>new_mask</code> argument to the calling process signal mask.

6. 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.

7. The `old_mask` argument is the address of a `sigset_t` variable that will be assigned the calling process's original signal mask prior to a `sigprocmask` call. If the actual argument to `old_mask` is a `NULL` pointer, no previous signal mask will be returned.

8. The return value of `sigprocmask` call is zero if it succeeds or -1 if it fails.

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

10. 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` typed variable.

```
#include<signal.h>
int sigemptyset(sigset_t *sigmask);
int sigaddset(sigset_t *sigmask, const int signal_num)
int sigdelset(sigset_t *sigmask, const int signal_num):
int sigfillset(sigset_t *sigmask);
int sigismember(const sigset_t *sigmask, const int signal_num);
```

11. The `sigemptyset` API clears all signal flags in the `sigmask` argument.

12. The `sigaddset` API sets the flag corresponding to the `signal_num` signal in `sigmask`.

13. The `sigdelset` API clears the flag corresponding to the `signal_num` signal in `sigmask`.

14. The `sigfillset` API sets all the flags in the `sigmask`

15. The return value of the `sigemptyset`, `sigaddset`, `sigdelset`, and `sigfillset` calls is zero if the call succeeds or -1 if they fail.

16. The `sigismember` API returns 1 if the flag corresponding to the `signal_num` signal in the `sigmask` is set, zero if not set, and -1 if the call fails.

17. 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.

Program:

```
#include<stdio.h>
#include<signal.h>
int main()
{
```

```

sigset_t sigmask; sigemptyset(&sigmask);
/*initialise set*/
if(sigprocmask(0,0,&sigmask)==-1)/*getcurrentsignalmask*/
{
    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");
}

```

18. If there are multiple instances of the same signal pending for the process, it is implementation dependent whether one or all of those instances will be delivered to the process

19. A process can query which signals are pending for it via the `sigpending` API

```

#include<signal.h>

int sigpending(sigset_t*sigmask);

```

`sigmask` is assigned the set of signals pending for the calling process by the `API.sigpending` returns a zero if it succeeds and a -1 value if it fails.

20. UNIX system V 3 and V.4 support the following APIs as simplified means 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);

```

21. The `sighold` API adds the named signal `signal_num` to the calling process signal mask

22. The `sigrelse` API removes the named signal `signal_num` to the calling process signal mask

23. The `sigignore` API sets the signal handling method for the named signal `signal_num` to SIG DFT.

24. The `sigpause` API removes the named signal `signal_num` from the calling process signal mask and suspends the process until it is interrupted by a signal.

Sigaction:

1. The `sigaction` API is a replacement for the `signal` API in the latest UNIX and POSIX systems.

2. The `sigaction` API is called by a process to set up a signal handling method for each signal it wants to deal with.

3. `sigaction` API returns the previous signal handling method for a given signal.

4. The `sigaction` API prototype is:

```
#include<signal.h>

int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

5. 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;
};
```

6. The `sa_handler` field can be set to SIG_IGN, SIG_DFL, or a user-defined signal handler function.

7. The `sa_mask` field specifies additional signals that process wishes to block when it is handling `signal_num` signal.

8. The `signal_num` argument designates which signal handling action is defined in the action argument

9. The previous signal handling method for `signal_num` will be returned via the `old_action` argument if it is not a NULL pointer.

10. If action argument is a NULL pointer, the calling process's existing signal handling method for `signal_num` will be unchanged.

11. The following program illustrates the use of `sigaction`:

Program

```
#include<iostream.h>#in
clude<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;
}
```

12. In the above example, 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.

13. The output of the above program would be as:

```
%ccsigaction.c-osigaction
%./sigaction&[
1]495
%kill-INT495
catch signal: 2
sigaction exits
[1]Done sigaction
```

14. The `sa_flag` field of the struct `sigaction` is used to specify special handling for certain signals.

15. POSIX defines only two values for the `sa_flag`: zero or `SA_NOCHLDSTOP`.

16. The `SA_NOCHLDSTOP` flag is an integer literal defined in the `<signal.h>` header and can be used when `signal_num` is `SIGCHLD`.

17. The effect of the `SA_NOCHLDSTOP` flag is that the kernel will generate the `SIGCHLD` signal to a process when its child process has terminated, but not when the child process has been stopped.

18. If the `sa_flag` value is set to zero in `sigaction` call for `SIGCHLD`, the kernel will send the `SIGCHLD` signal to the calling process whenever its child process is either terminated or stopped.

19. UNIX system V4 defines additional flags for the `sa_flags` field. These flags can be used to specify the UNIX System V3 style of signal handling method:

sa_flags value	Effect on handling signal_num
SA_RESETHAND	If signal_num is caught, the sa_handler is set to SIG_DFL before the signal handler function is called, and signal_num will not be added to the process signal mask when the signal handler function is executed.
SA_RESTART	If a signal is caught while a process is executing a system call, the kernel will restart the system call after the signal handler returns. If this flag is not set in the sa_flags, after the signal handler returns, the system call will be aborted with a return value of -1 and will set <code>errno</code> to <code>EINTR</code> .

The SIGCHLD Signal and The waitpid Function:

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.
- ✓ API will return the child's exit status and process ID to the parent.
- ✓ Kernel will clear up the Process Tables slots 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 & siglongjmp APIs:

1. The sigsetjmp and siglongjmp API has similar functions as their corresponding setjmp and longjmp APIs.

2. The sigsetjmp and siglongjmp APIs are defined in POSIX.1 and on most UNIX systems that support signal mask.

3. The function prototypes of the APIs are:

```
#include <setjmp.h>
```

```
int sigsetjmp(sigjmpbuf env, int save_sigmask); int  
siglongjmp (sigjmpbuf env, int ret_val );
```

4. 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.

5. The `sigsetjmp` API behaves similarly to the `setjmp` API, except that it has a second argument, `save_sigmask`, which allows user to specify whether a calling process signal mask should be saved to the provided `env` argument.

6 If the `save_sigmask` argument is nonzero, the caller's signal mask is saved, else signal mask is not saved.

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

8 The `ret_val` argument specifies the return value of the corresponding `sigsetjmp` API when called by `siglongjmp` API. Its value should be nonzero number, and if it is zero the `siglongjmp` API will reset it to 1.

9. 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 that the process signal mask is restored properly when "jumping out" from a signal handling function.

10. The following C program illustrates the use 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_SETM
ASK,&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<<"returnfromsignalinterruption";
    return0;
}
else
    cerr<<"returnfromfirsttimesigsetjmpiscalled";
pause();
}

```

11. The program sets its signal mask to contain SIGTERM, and then sets up a signal trap for the SIGINT signal.

12. The program then calls sigsetjmp to store its code location in the env global variable. Note the sigsetjmp call returns a zero value when directly called in user program and not via siglongjmp.

13. The program suspends its execution via the pause API.

14. Whenever the user interrupts the process from the keyboard, the callme function is called .

15. The callme function calls siglongjmp API to transfer flow back to the sigsetjmp function in main, which now returns a 2 value.

16. The sample output of the above program is:

```
%ccsigsetjmp.c
%./a.out&
[1]377
```

Return from first time sigsetjmp is called % kill -INT

377catch signal: 2

```
Returnfromsignalinterruption
[1]Donea.out
```

%

Kill:

1. A process can send signal to a related process via the kill API.
2. This is a simple means of IPC or control.
3. The sender and recipient processes must be related such that either sender process real or effective user ID matches that of the recipient process, or the sender has su privileges.
4. For example, a parent and child process can send signals to each other via the kill API.
5. The kill API is defined in most UNIX systems and is a POSIX.1 standard.
6. The function prototype is as

```
#include <signal.h>
int kill(pid_t pid, int signal_num);
```

7. The sig_num argument is the integer value of a signal to be sent to one or more processes designated by pid.
8. The possible values of pid and its use by the kill API are:

Apidvalue	EffectsontheKillAPI
A Positive Value	pid is a process ID. Sends Signal_num to that process.
0	Sends signal num to all processes whose process group ID is the same as the calling process.
-1	Sends signal num to all processes whose real user ID is the same as the effective user ID of the calling process. If the calling process effective user ID is su user ID, signal num will be sent to all processes in the system (except processes 0 and 1). The later case is

	used when the system is shutting down - kernel calls the kill API to terminate all processes except 0 and 1. Note: POSIX.1 does not specify the behavior of the kill API when the pid value is - 1. This effect is for UNIX systems only.
Negative value	Send signal_num to all processes whose process group ID matches the absolute value of pid

9. The return value of kill is zero if it succeeds or -1 if it fails.

10. The following C program illustrates the implementation of the UNIX kill Command.

```
#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;
}
```

Alarm:

1. 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.
2. The alarm API is defined in most UNIX systems and is a POSIX.1 standard.
3. The function prototype of the API is as follows:

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

4. The time_interval argument is the number of CPU seconds elapsing time, after which the kernel will send the SIGALRM signal to the calling process.
5. If the time_interval value is zero, it turns off the alarm clock.
6. The return value of the alarm API is the number of CPU seconds left in the process timer, as set by a previous alarm system call.
7. The effect of the previous alarm API call is canceled, and the process timer is reset with the new alarm call.
8. A process alarm clock is not passed on to its forked child process, but an exec'd process retains the same alarm clock value as was prior to the exec API call.
9. The alarm API can be used to implement the sleep API. Program:

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

10 The sleep function above 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 the pause system call

11. 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

12. BSD UNIX defines the ualarm function, which is the same as the alarm API except that the argument and return value of the ualarm function are in microsecond units.

Interval Timers:

1. The use of the alarm API to set up an interval timer in a process.
2. The interval timer can be used to schedule a process to do some tasks at fixed time intervals, to time the execution of some operations, or limit the time allowed for the execution of some tasks.
3. 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
(SIGALRM,&action,0)==-1)
    {
        perror("sigaction");
        return 1;
    }
    if(alarm(INTERVAL)==-1)
        perror("alarm");
    else while(1)
    {
```

```

        /*donormaloperation*/
    }
    return 0;
}

```

4. The signal API is called to set up call me as the signal handling function for the SIGALRM signal.

5. The program then invokes the alarm API to send itself the SIGALRM signal after 5 real clock seconds.

6. The program then goes off to perform its normal operation in an infinite loop.

7. When the timer expires, the call me function is invoked, which restarts the alarm clock for another 5 seconds and then does the scheduled tasks.

8. When the call me function returns, the program continues its "normal" operation until another timer expiration.

9. BSD UNIX invented the setitimer API, which provides capabilities additional to those of the alarm API.

10. The setitimer resolution time is in microseconds, whereas the resolution time for alarm is in seconds.

11. The alarm API can be used to set up real-time clock timer per process. The setitimer API can be used to define up to three different types of timers in a process:

a. Realtime clock timer.

b. Timer based on the user time spent by a process.

c. Timer based on the total user and system time spent by a process.

12. The getitimer API is also defined in BSD and System V UNIX for users to query the timer values that are reset by the setitimer API.

13. The setitimer & 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);

```

14. The which arguments specify the timer to process, the possible values are:

Which argument value	Timertype
ITIMER_REAL	Timerbasedonreal-timeclock.Generatesa SIGALRMsignalwhenexpires.
ITIMER_VIRTUAL	Timerbased on user-time spent by a process. GeneratesaSIGVTALRMsignalwhenitexpires.
ITIMER_PROF	Timerbasedontotaluser&Systemtimesspentbya process. Generates a SIGPROF signal when it expires.

15. The struct itimerval data type is defined in the <sys/time.h> header as:

```
struct itimerval
{
    structtimervalit_interval; struct
    timerval it_value;
}
```

16. ForsetitimerAPI,theval.it_valueisthetimetosetthenamedtimer,and theval.it_intervalisthetimetoreloadthetimerwhenitexpires.

17. Theval.it_intervalmaybesettozeroifthetimeristorunonlyonceand iftheval.it_valueissettozero,itstopsthenamedtimerifitisrunning.

18. ForgetitimerAPI,old.it_valueandtheold.it_intervalreturnthenamedtimer'sre mainingtimeandthereloadtime,respectively,

19. TheoldargumentofthesetitimerAPIisliketheoldargumentofthe getitimer API.

20. Ifthisisanaddressofastructitimervaltypedvariable,itreturnsthe previous timer value, if set to NULL the old timer value will not be returned.

21. The ITIMER_VIRTUAL and ITIMER_PROF timers are primary useful in timing the total execution time of selected user functions, as the timer runs only while the user process is running or the kernel is executing system functions on behalf of the user process for the ITIMER_PROF timer.

22. BoththeAPIsreturnzeroonsuccessor-1valueiftheyfail.

23. TimerssetbythesetitimerAPIinaparentprocessarenotinheritedbyits child processes, but these timers are retained when a process exec's a new program.

24. ThefollowingprogramillustratestheuseofsetitimerAPL

```

#include <stdio.h>
#include <unistd.h>
#include
<signal.h>#include<sys/
time.h>

#define INTERVAL 5
void callme(int sig_no)
{
    /*doscheduledtasks*/
}
intmain()
{
    struct itimerval val;
    struct sigaction action;
    sigemptyset(&action.sa_mask);action.sa_h
    andler=(void(*)()) callme;
    action.sa_flags=SA_RESTART;if(sigaction(S
    IGALARM,&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");
    elsewhile(1)
    {
        /*donormaloperation*/
    }
    return 0;
}

```

25. Since the setitimer and alarm APIs require that users set up signal handling to catch timer expiration, they should not be used in conjunction with the sleep API, because sleep API may modify the signal handling function for the SIGALRM signal.

POSIX.1bTimers:

1. POSIX.1b defines a set of APIs for interval timer manipulation.
2. The POSIX.1b timers are more flexible and powerful than UNIX timers in following ways:
 - a. Users may define multiple independent timers per system clock.
 - b. The timer resolution is in nanoseconds.
 - c. Users may specify, on a timer basis, the signal to be raised when a timer expires.
 - d. The timer interval may be specified as either an absolute or a relative time.
3. There is a limit on how many POSIX timers can be created per process, this is `TIMER_MAX` constant defined in `<limits.h>` header.

4. POSIX timers created by a process are not inherited by its child process, but are retained across the `exec` system call.

5. A POSIX.1 timer does not use the `SIGALRM` signal when it expires, safely with the `sleep` API in the same program can be used.

6. The POSIX.1b APIs for timer manipulation are:

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

```
int timer_create(clockid_t clock, struct sigevent *spec, timer_t *timer_hdp);
int timer_settime(timer_t *timer_hdp, int flag, struct itimerspec *val, struct
itimerspec *old);
int timer_gettime(timer_t timer_hdp, struct itimerspec *old);
int timer_getoverrun(timer_t timer_hdp);
int timer_delete(timer_t timer_hdp);
```

7. The `timer_create` API is used to dynamically create a timer and returns its handler.

8. The `clock` argument specifies which system clock would be the new timer based on, its value may be `CLOCK_REALTIME` for creating a real time clock timer - this is defined by POSIX.1b - other values are system dependent.

9. The `spec` argument defines what action to take when the timer expires. The `struct sigevent` data type is defined as

```

struct sigevent
{
    int sigev_notify;
    int sigev_signo;
    union sigval sigev_value;
}

```

10. The `sigev_signo` field specifies a signal number to be raised at the timer expiration. It is valid only when the `sigev_notify` field is set to `SIGEV_SIGNAL`.

11. If `sigev_notify` field is set to `SIGEV_NONE`, no signal is raised by the timer when it expires

12. Because multiple timers may generate the same signal, the `sigev_value` field is used to contain any user defined data to identify that a signal is raised by a specific timer. The data structure of the `sigev` field is

```

union sigval
{
    int    sival_int;
    void  *sival_ptr;
}

```

13. For example, a process may assign each timer a unique integer ID assigned to the `spec->sigev_value.sival_int` field

14. To pass this data along with the signal (`sigev_signo`) when it is raised, the `SA_SIGINFO` flag should be set in a signal action call, which sets up the handling for the signal and the handling function prototype should be:

```
void <signal_handler>(int signo, siginfo_t *evp, void *context);
```

when the signal handler is called, the `evp->si_value` contains the data of the `spec->sigev_value`. The `siginfo_t` data type is defined in `<siginfo.h>`

15. If `spec` is set to `NULL` and the timer is based on `CLOCK_REALTIME`, then `SIGALRM` signal is raised when the timer expires.

16. Finally the `timer_hdr` argument is an address of a `timer_t` type variable to hold the handler of the newly generated timer. This should not be `NULL` as it is used to call other `POINTS.1b` timer APIs.

17. All `POSIX.1b` timer APIs return zero on success and -1 if they fail.

18. The `timer_settime` starts and stops a timer running. The `timer_gettime` is used to query the current values of the timer,

19. The struct `itimerspec` data type is defined as:

```
struct itimerspec
```

```

    {
        struct timespec    it_interval;
        struct timespec    it_interval;
    };

```

And the struct timespec is defined as:

```

struct timespec
{
    time_t tv_sec;
    long   tv_nsec;
};

```

20. The `itimerspec.it_value` specifies the timer remaining in the timer, and `itimerspec.it_interval` specifies the new time to reload the timer when it expires. All times are specified in seconds via `timerspec.tv_sec` and in nanoseconds via `timerspec.tv_nsec`.

21. The flag parameter in `timer_settime` may be 0 or `TIMER_REALTIME` if the timer start time (val argument) is relative to the current time

22. If the flag value is `TIMER_ABSTIME`, the timer start time is an absolute time.

23. The old argument is used to obtain previous timer values, if NULL no timer values are returned

24. The old argument `timer_gettime` returns the current values of named timer.

25. The `timer_getoverrun` API returns the number of signals generated by a timer but was lost (overrun). Timer signals are not queued by the kernel if they are raised but not processed by target process, instead the kernel records the number of overrun signals per timer.

26. The `timer_destroy` is used to destroy a timer created by `timer_create` API.

Daemon Processes: Introduction

1. 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.

2. They do not have a controlling terminal; so we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities

3. Here we look at the process structure of daemons and how to write a daemon.

4. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

Daemon Characteristics:

1. We look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions.

2. The `ps` command prints the status of various processes in the system. We will execute: `ps -ax` under BSD UNIX

3. The `-a` option shows the status of processes owned by others, and `-x` shows processes that do not have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.

4. The system processes depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception to this is `init`, since it is a user-level command started by the kernel at boot time.)

5. Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.

6. Process 1 is usually `init`, is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons.

7. On Linux, the `eventd` daemon provides process context for running scheduled functions in the kernel.

8. The `kapmd` daemon provides support for the advanced power management features available with various computer systems.

9. The `swapd` daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time.

10. The Linux kernel flushes cached data to disk using two additional daemons: `bdflush` and `kupdatd`.

11. The `portmapper` daemon, `portmap`, provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers.

12. The syslogd daemon is available to any program to log system messages for an operator. The messages may be printed on a console device and also written to a file.

13. The inetd daemon (xinetd) listens on the system's network interfaces for incoming requests for various network servers.

14. The nfsd, lockd, and rpciod daemons provide support for the Network File System (NFS).

15. The cron daemon (crond) executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by cron.

16. The cupsd daemon is a print spooler, it handles print requests on the system.

17. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called setsid.

18. All the user-level daemons are process group leaders and session leaders and are the only processes in their process group and session. Finally, note that the parent of most of these daemons is the init process.

Coding Rules:

1. Some basic rules to coding a daemon prevent unwanted interactions from happening.

2. We state these rules and then show a function, daemonize, that implements them.

a. The first thing to do is call `umask` to set the file mode creation mask to 0. The file mode creation mask that is inherited could be set to deny certain permissions.

b. 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 are guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.

c. Call `setsid` to create a new session. Three steps occur. The process

- ✓ becomes a session leader of a new session,
- ✓ becomes the process group leader of a new process group
- ✓ has no controlling terminal.

d. 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.

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

f. 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 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;
    elseif(pid!=0)
        exit(0); /*parent exits*/
    /* child continues */
    setsid( );
    chdir("/");
    umask(0);
    return 0;
}
```

ErrorLogging

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.

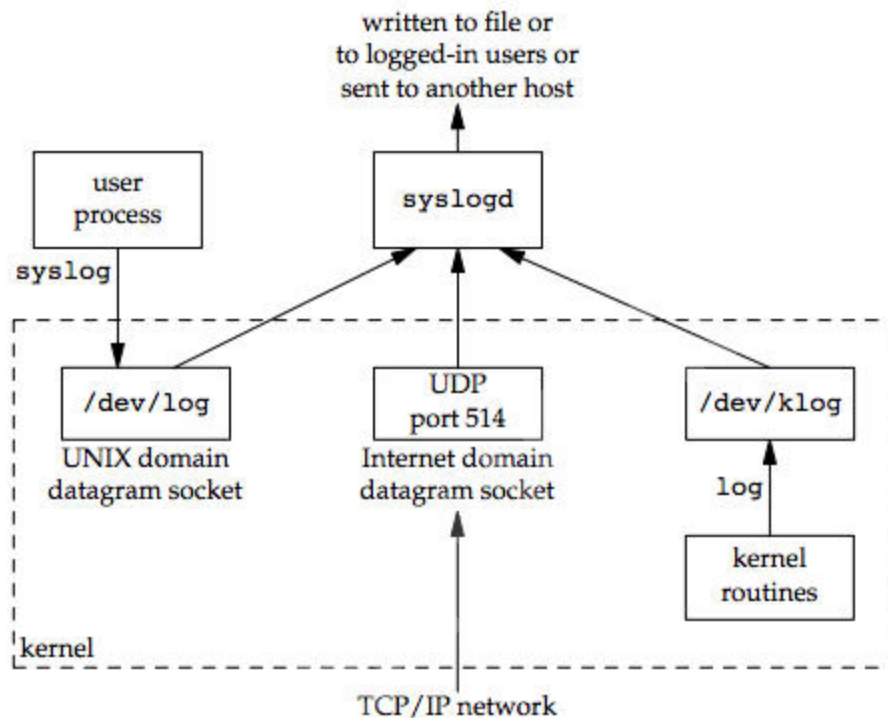


Fig: The BSD syslog facility

There are three ways to generate log messages:

- ✓ Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device.
- ✓ Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
- ✓ A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file. Our interface to this facility is through the syslog function.

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

SingleInstanceDaemons:

1. Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation
2. The daemon might need exclusive access to a device, for example. In the case of the cron daemon, if multiple instances were running, each copy might try to start a single scheduled operation, resulting in duplicate operations and probably an error
3. If the daemon needs to access a device, the device driver will sometimes prevent multiple opens of the corresponding device node in /dev.
4. This restricts us to one copy of the daemon running at a time. If no such device is available, however, we need to do the work ourselves
5. The file and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running
6. If each daemon creates a file and places a write lock on the entire file, only one such write lock will be allowed to be created.
7. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running
8. File and record locking provides a convenient mutual exclusion mechanism
9. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, removing the need for us to clean up from the previous instance of the daemon.

Example

10. The functions shown below will illustrate the use of file and record locking to ensure that only one copy of a daemon is running..
11. Each copy of the daemon will try to create a file and write its process ID
init This will allow administrators to identify the processes easily.
12. If the file is already locked, the lock file function will fail with error set to EACCES or EAGAIN, so we return 1, indicating that the daemon is already

running. Otherwise, we truncate the file, write our process ID to it, and return 0.

13. Truncating the file prevents data from the previous daemon appearing as if it applies to the current daemon.

PROGRAM: Ensure that only one copy of a daemon is running

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>
#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
extern int lockfile(int);
int already_running(void)
{
    int fd;
    char buf[16];
    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0)
    {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE,
            strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0)
    {
        if (errno == EACCES || errno == EAGAIN)
        {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE,
            strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld",
        (long) getpid());
    write(fd, buf,
        strlen(buf)+1);
    return(0);
}
```

Daemon Conventions:

- ✓ If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually `name.pid`, where `name` is the name of the daemon or the service. For example, the name of the cron daemon's lock file is `/var/run/crond.pid`.
- ✓ If the daemon supports configuration options, they are usually stored in `/etc`. The configuration file is named `name.conf`, where `name` is the name of the daemon or the name of the service. For example, the configuration for the syslog daemon is `/etc/syslog.conf`.
- ✓ Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (`/etc/rc*` or `/etc/init.d/*`). If the daemon should be restarted automatically when it exits, we can arrange for `init` to restart it if we include a `respawn` entry for it in `/etc/inittab`.
- ✓ If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch `SIGHUP` and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive `SIGHUP`. Thus, they can safely reuse it.

Client Server Model

1. A common use for a daemon process is as a server process.
2. We can call the syslogd process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.
3. In general, a server is a process that waits for a client to contact it, requesting some type of service.
4. The service being provided by the syslogd server is the logging of an error message.