# UNIX Programming (18CS56)

# Module 5

## (I) Signals

- Signals are triggered by events and posted on a process to notify it that something has happened and requires some action.
- The event can be generated from a process, a user or the UNIX kernel.
- Parent and child processes can send signals to each other for process synchronization.
- Signals are the software version of hardware interrupts.
- Signals are defined as integer flags.
- *<signal.h>* header depicts the list of signals defined for a UNIX system.
- The following table lists some of the important signals used in UNIX systems –

| **Signal Name** | **Use** | **Generates core file at default?** |
|---|---|---|
| **SIGALRM** | Alarm timer time-outs. Generated by *alarm()* API. | No |
| **SIGABRT** | Abort process execution. Generated by *abort()* API. | Yes |
| **SIGFPE** | Illegal mathematical operation | Yes |
| **SIGHUP** | Controlling terminal hang-up | No |
| **SIGILL** | Execution of an illegal machine instruction | Yes |
| **SIGINT** | Process interruption. Generated by the Delete or Ctrl+C keys. | No |
| **SIGKILL** | Kill a process. Generated by *kill –9 <pid>* command. | Yes |
| **SIGPIPE** | Illegal write to a pipe | Yes |
| **SIGQUIT** | Process quit. Generated by Ctrl+\ keys. | Yes |
| **SIGSEGV** | Segmentation fault. Generated by de-referencing a NULL pointer. | Yes |
| **SIGTERM** | Process termination. Generated by *kill <pid>* command. | Yes |
| **SIGUSR1** | Reserved to be defined by users | No |
| **SIGUSR2** | Reserved to be defined by users | No |
| **SIGCHLD** | Sent to a parent process when its child process has terminated | No |
| **SIGCONT** | Resume execution of a stopped process | No |
| **SIGSTOP** | Stop a process execution | No |

| | | |
|---|---|---|
| **SIGTTIN** | Stop a background process when it reads from its controlling terminal | No |
| **SIGTTOU** | Stop a background process when it writes to its controlling terminal | No |
| **SIGTSTP** | Stop a process execution by the Ctrl+Z keys | No |

- When a signal is sent to a process, it is *pending* on the process to handle it.
- Process can react to pending signals in one of three ways:
  – Accepts the default action of the signal. Usually terminates the process.
  – Ignore the signal. Signal will be discarded, and it has no effect on recipient process.
  – Invoke a user-defined function. The function is known as signal handler routine and signal is said to be *caught* when this function is called.
  If the function finishes its execution without terminating the process, the process will continue execution from the point it was interrupted by the signal.
- Process may set up per-signal handling mechanisms – ignores some signals, catches some other signals and accepts default action from the remaining signals.
- Process may change handling of certain signals in its course of execution.
- Signal is said to have been *delivered* if it has been reacted to by the recipient process.
- The default action for most signals is to terminate a recipient process.
- Some signals will generate a core file for the aborted process – users can trace back the state of the process when it was aborted.
  These signals are usually generated when there is an implied program error in the aborted process.
- Most signals can be ignored or caught except SIGKILL and SIGSTOP signals.
- Companion signal to SIGSTOP is SIGCONT, which resumes a process execution after it has been stopped.
- A process is allowed to ignore certain signals so that it is not interrupted while doing certain mission-critical work.
- Signal handler function cleans up the work environment of a process before terminating the process gracefully.

## (II) Kernel Support for Signals

- UNIX System V.3 – each entry in kernel process table slot has array of signal flags, one for each signal defined in the system.
- When a signal is generated for a process, kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If recipient process is asleep, kernel will awaken the process by scheduling it.

- When recipient process runs, kernel will check the process U-area that contains an array of signal handling specifications.
- Each entry of the array corresponds to a signal defined in the system.
- Kernel will consult the array to find out how the process will react to the pending signal.

- UNIX System V.2 – when a signal is caught, kernel will first reset the signal handler in the recipient process U-area, then call the user signal handling function specified for that signal.
- If multiple instances of a signal are being sent to a process at different points, the process will catch only the first instance of the signal.
- All subsequent instances of the signal will be handled in the default manner.
- If a process is continuously catching multiple occurrences of a signal, it must reinstall the signal handler function every time the signal is caught.
- In the time between signal handler invoking and re-establishment of signal handler method, another instance of signal may be delivered to the process.
  This leads to race condition.

- To solve the unreliability of signal handling in UNIX System V.2, BSD UNIX 4.2 and POSIX.1 use an alternate method.
- When a signal is caught, the kernel does not reset the signal handler, so there is no need for the process to re-establish the signal handling method.
- The kernel will block further delivery of the same signal to the process until the signal handler function has completed execution.
- This ensures that signal handler function will not be invoked recursively for multiple instances of the same signal.

## (III) signal API

- *signal* API is used to define the per-signal handling method.
- Prototype of the *signal* API is:

```
#include<signal.h>

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

- *signal_num* – signal identifier such as SIGINT or SIGTERM, as defined in *<signal.h>*
- *handler* – function pointer of a user-defined signal handler function.

- The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal.
  The *pause* API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return.

```
#include<iostream.h>
#include<signal.h>
//signal handler function
void catch_sig(int sig_num){
        signal(sig_num, catch_sig);
        cout << "catch_sig: " << sig_nm;
}
//main
int main(){
        signal(SIGTERM, catch_sig);
        signal(SIGINT, SIG_IGN);
        signal(SIGSEGV, SIG_DFL);
        pause(); //waits for signal interruption
}
```

- SIG_IGN and SIG_DFL are manifest constants defined in <*signal.h*>

```
#define SIG_DFL void (*)(int)0
#define SIG_IGN void (*)(int)1
```

- SIG_IGN – specifies that a signal must be ignored.
  If the signal is generated to the process, it will be discarded without any interruption of process.
- SIG_DFL – specifies that the default action of a signal must be accepted.

- Return value of *signal* API – previous signal handler for a signal.
  It can be used to restore the signal handler for a signal after it has been altered.

```
#include<signal.h>

int main(){
        void (*old_handler)(int) = signal(SIGINT, SIG_IGN);
        //do mission critical processing

        //restore previous signal handling
        signal(SIGINT, old_handler);
}
```

- UNIX System V.3 and V.4 support *sigset* API.

```
#include<signal.h>

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

- Arguments and return value of *sigset* are same as that of *signal*.
- Both functions set signal handling methods for any named signal.
- *signal* API is unreliable – race condition due to handling multiple instances of signal. *sigset* API is reliable – one of the multiple instances is handled, whereas remaining instances are blocked.

## (IV) Signal Mask

- A signal mask defines which signals are blocked when generated to a process.
- The blocked signal depends on recipient process to unblock it and handle accordingly.
- If a signal has to be specified to be ignored or blocked, it is implementation-dependent on whether such a signal will be discarded or left pending when it is sent to the 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 using *sigprocmask* API:

```
#include<signal.h>

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

- *new_mask* – defines a set of signals to be set or reset in a calling process signal mask.
- *cmd* – specifies how the *new_mask* value is to be used by the API.
- Returns: 0 if it succeeds, -1 if it fails.
- Possible failure - *new_mask/old_mask* are invalid addresses.
- The following table lists the various values that can be assigned to *cmd* argument –

| ***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, *cmd* argument will be ignored and current process signal mask will not be altered.
- *old_mask* argument – address of a *sigset_t* variable that will be assigned the original signal mask of calling process prior to *sigprocmask* call.
- If actual argument to *old_mask* is NULL pointer, no previous signal mask will be returned.

- *sigset_t* is a data type defined in <*sigset.h*> - contains a collection of bit flags, with each bit flag representing one signal defined in a given system.

- *sigsetops* functions - set of API defined by BSD UNIX and POSIX.1 to 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);
```

- *sigemptyset* - clears all signal flags in *sigmask* argument.
- *sigaddset* - sets the flag corresponding to *signal_num* signal in the *sigmask* argument.
- *sigdelset* - clears the flag corresponding to *signal_num* signal in the *sigmask* argument.
- *sigfillset* - sets all signal flags in *sigmask* argument.
- *sigismember* - checks if the flag corresponding to *signal_num* signal in the sigmask argument is set or not set.
- Return value of *sigemptyset*, *sigaddset*, *sigdelset* and *sigfillset* calls is 0 if the calls succeed or -1 if they fail.
- Possible causes of failure may be that the *sigmask* and/or *signal_num* arguments are invalid.
- The *sigismember* API returns 1 if the flag corresponding to the *signal_num* signal in the *sigmask* argument is set, 0 if it is 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.

Then it clears the SIGSEGV signal from the process signal mask.

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

int main(){
	sigset_t sigmask;
	sigemptyset(&sigmask); //initialize set
	if(sigprocmask(0,0,&sigmask) == -1){ //get current signal mask
		perror("sigprocmask");
		exit(1);
	}
	else
		sigaddset(&sigmask, SIGINT); //set SIGINT flag
	sigdelset(&sigmask, SIGSEGV); //clear SIGSEGV flag
	if(sigprocmask(SIG_SETMASK,&sigmask,0) == -1)
		perror("sigprocmask"); //set new signal mask
}
```

- When one or more signals are pending for a process and are unblocked via *sigprocmask* API, the signal handler methods for those signals that are in effect at the time of *sigprocmask* call will be applied before the API is returned to caller.
- If multiple instances of the same signal are pending for the process, then it is implementation-dependent whether one or all the instances will be delivered to the process.
- A process can query which signals are pending for it using *sigpending* API:

```
#include<signal.h>

int sigpending(sigset_t *sigmask);
```

- *sigmask* – address of a *sigset_t* typed variable; assigned the set of signals pending for the calling process by the API.
- The API returns 0 if it succeeds and -1 if it fails.
- The *sigpending* API can be useful to find out whether one or more signals are pending for a process and to set up signal handling methods for these signals before the process calls the *sigprocmask* API to unblock them.

- The following example reports to the console whether the SIGTERM signal is pending for the process:

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

int main(){
        sigset_t sigmask;
        sigemptyset(&sigmask);
        if(sigpending(&sigmask) == -1)
                perror("sigpending");
        else
                cout << "SIGTERM signal is: "
                     << (sigismember(&sigmask, SIGTERM)?"set":"no set");
}
```

- The signal mask manipulation functions are –

```
#include<signal.h>

int sighold(int signal_num);
int sigrelse(int signal_num);
int sigignore(int signal_num);
int sigpause(int signal_num);
```

- *sighold* API – adds the named signal *signal_num* to the calling process signal mask.
- It is the same as using the *sigset* API with the SIG_HOLD action:

```
sigset(<signal_num>, SIG_HOLD);
```

- *sigrelse* API – removes the named signal *signal_num* for the calling process signal mask.
- *sigignore* API – sets the handling method for the named signal *signal_num* to SIG_DFL
- *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.

## (V) sigaction

- *sigaction* is the replacement for *signal* API in the latest UNIX and POSIX systems.
- *sigaction* API is called by a process to set up a signal handling method for each signal it wants to deal with.
- It passes back the previous signal handling method for a given signal.
- It blocks the signal it is catching, allowing a process to specify additional signals to be blocked when the API is handling a signal.

- Prototype of *sigaction* API is given by:

```
#include<signal.h>

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

- *struct sigaction* data type is defined in <signal.h> as:

```
struct sigaction{
        void (*sa_handler)(int);
        sigset_t sa_mask;
        int sa_flag;
};
```

- *sa_handler* – corresponds to second argument of *signal* API.
- Can be set to SIG_IGN, SIG_DFL, or user-defined signal handler function.
- *sa_mask* – specifies additional signals that a process wishes to block when it is handling the *signal_num* signal.
- It does not block the signals currently specified in the signal mask of the process and *signal_num* signal.
- *signal_num* - designates which signal handling action is defined in the *action* argument.
- Previous signal handling method for *signal_num* will be returned via *old_action* argument if it is not a NULL pointer.
- If *action* argument is a NULL pointer, existing signal handling method of the calling process for *signal_num* will be unchanged.

- The following program illustrates the use of *sigaction:*

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

void callme(int sig_num){
        cout << "catch signal: " << sig_num;
}

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_flag = 0;
        if(sigaction(SIGINT,&action,&old_action) == -1)
                perror("sigaction");
        pause(); //wait for signal interruption
        cout << argv[0] << " exists";
        return 0;
}
```

- The process signal mask is set with the 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 suspends its execution via the *pause* API.

- If the SIGINT signal is generated to the process, the kernel first sets the process signal mask to block the SIGTERM, SIGINT and SIGSEGV signals.
- It then arranges the process to execute the *callme* signal handler function.
- When the *callme* function returns, the process signal mask is restored to contain only the SIGTERM signal, and the process will continue to catch the SIGILL signal.

- *sa_flag* - used to specify special handling for certain signals.
- POSIX.1 defines two values for *sa_flag*: 0 and SA_NOCLDSTOP.
- If *sa_flag* is 0, kernel will send the SIGCHLD signal to the calling process whenever its child process is either terminated or stopped.
- If *sa_flag* is SA_NOCLDSTOP, kernel will send the SIGCHLD signal to the calling process whenever its child process is terminated, but not when it is stopped.

- Additional flags for the *sa_flag* field in UNIX System V.4 implementation – SA_RESETHAND and SA_RESTART
- SA_RESETHAND – If *signal_num* is caught, *sa_handler* is set to SIG_DFL before the signal handler function is called.
  *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, kernel will restart the system call after the signal handler returns.
  If this flag is not set in the *sa_flag* field, the system call will be aborted with a return of -1 after the signal handler returns and *errno* will be set to EINTR.

## (VI) SIGCHLD signal and the waitpid Function

- When a child process terminates or stops, kernel will generate a SIGCHLD signal to its parent process.
- Three different events may occur depending on how the parent sets up the handling of the SIGCHLD signal.
- *Event 1* – parent accepts the default action of the SIGCHLD signal:
  - SIGCHLD signal does not terminate the parent process.
  - Affects only the parent process if it arrives at the same time when the parent process is suspended by the *waitpid* system call.
  - Parent process will be awakened, *waitpid* will return the child's exit status and PID to the parent, and the kernel will clear up the process table slot for child.
  - With this setup, parent can call *waitpid* repeatedly to wait for each child it created.
- *Event 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 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 and API will return a value of –1 to the parent process.
- *Event 3* – parent catches the SIGCHLD signal:
  - Signal handler function will be called in the parent process whenever a child process terminates.
  - If SIGCHLD signal arrives while the parent process is executing a *waitpid* system call, the *waitpid* API may be restarted to collect the child exit status and clear its process table slot after the signal handler function returns.
  - API may be aborted, and the child process table slot is not freed, depending on the parent setup of the signal action for the SIGCHLD signal.

- Interaction between SIGCHLD and *wait* API is the same as that between SIGCHLD and *waitpid* API.
- Earlier versions of UNIX used the SIGCLD signal instead of SIGCHLD.
- SIGCLD signal is now obsolete, but most of the latest UNIX systems have defined SIGCLD to be the same as SIGCHLD for backward compatibility.

## (VII) sigsetjmp and siglongjmp APIs

- *setjmp* and *sigsetjmp* – marks one or more locations in a user program.
- *longjmp* and *siglongjmp* – called to return to any of the marked locations.
- These APIs give the provision of inter-function *goto* capability.

- These are defined in POSIX.1 and on most UNIX systems that support signal masks.
- The function prototypes are:

```
#include<setjmp.h>

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

- *sigsetjmp* and *siglongjmp* are created to support signal mask processing.
- It is implementation-dependent on whether a signal process mask is saved and restored when it invokes the *setjmp* and *longjmp* APIs.

- *sigsetjmp* – behaves similarly to the *setjmp* API, but it has a second argument *save_sigmask*.
  - Allows a user to specify whether a calling process signal mask should be saved to the provided *env* argument.
  - If the *save_sigmask* argument is non-zero, the caller's signal mask is saved. Otherwise, signal mask is not saved.
- *siglongjmp* – behaves similarly to the *longjmp* API, but it also restores a calling process signal mask if the mask was saved in its *env* argument.
  - *ret_val* argument – specifies the return value of the corresponding *sigsetjmp* API when it is called by *siglongjmp*.
  - *ret_val* value should be a non-zero number. If it is zero, *siglongjmp* API will reset it to 1.
- *siglongjmp* API is usually called from user-defined signal handling functions.
- This is because the 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.

- The following program illustrates the uses of *sigsetjmp* and *siglongjmp* APIs.
- The program sets its signal mask to contain SIGTERM, then sets up a signal trap for the SIGINT signal.
- The program then calls *sigsetjmp* to store its code location in the *env* global variable.
- The *sigsetjmp* call returns a 0 value when it is called directly in the user program and not via *siglongjmp*.
- The program suspends its execution via the *pause* API.
- When a user interrupts the process from the keyboard, the *callme* function is called.
- The *callme* function calls the *siglongjmp* API to transfer program flow back to the *sigsetjmp* function in the *main* function, which returns a value of 2.

```
#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;
        siglongjmp(env, 2);
}

int main(){
        sigset_t sigmask;
        struct sigaction action, old_action;
        sigemptyset(&sigmask);
        if(sigaddset(&sigmask,SIGTERM) == -1 ||
            sigprocmask(SIG_SETMASK,&sigmask,0) == -1)
                perror("set signal mask");
        sigemptyset(&action.sa_mask);
        sigaddset(&action.sa_mask, SIGSEGV);
```

## (VIII) kill

- A process can send a signal to a related process using *kill* API.
- Sender and recipient processes must be related such that either the sender process real/effective UID matches that of recipient process, or sender process has superuser privileges.
- Example: parent and child process sending signals to each other.
- *kill* API function prototype:

```
#include<signal.h>

int kill(pid_t pid, int signal_num);
```

- Returns: 0 if successful, -1 if it fails.

- 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){
                        //get signal number
                        cerr << "Invalid number: " << argv[1];
                        return -1;
                }
                argv++, argc--;
        }
        while(--argc > 0){
                if(sscanf(*++argv,"%d",&pid) == 1){
                        //get process ID
                        if(kill(pid, sig) == -1)
                                perror("kill");
                }
                else
                        cerr << "Invalid pid" << argv[0];
        }
        return 0;
}
```

- *signal_num* – integer value of a signal that must be sent to one or more processes designated by *pid*.
- Possible values of *pid:* positive value, 0, -1, negative value.
- The following table gives the details about these values:

| ***pid* value** | **Effects on *kill* API** |
|---|---|
| **Positive value** | *pid* is a process ID. Sends *signal_num* to that process. |
| **0** | Sends *signal_num* to all processes whose process GID is same as the calling process |
| **-1** | Sends *signal_num* to all process whose real UID is same as the effective UID of calling process.<br>If calling process effective UID is the superuser UID, *signal_num* will be sent to all processes in the system (except process 0 and process 1). |
| **Negative value** | Sends *signal_num* to all processes whose process GID matches absolute value of *pid* |

- UNIX *kill* command invocation syntax is given as follows:
  ***kill [–<signal_num>] <pid>***
- Here, <*signal_num*> - integer number or symbolic name of a signal, as defined in <*signal.h*> header.
- <*pid*> - integer number of a PID.
  There can be one or more PID specified – *kill* will send signal <*signal_num*> to each process that corresponds to a <*pid*>.
- Example: *kill –9 1234*

- Any signal specification at the command line must be a signal's integer value.
- It does not support signal symbolic names.
- If no signal number is specified, the program will use the default signal SIGTERM, which is the same for the UNIX *kill* command.
- The program calls the *kill* API to send a signal to each process whose PID is specified at the command line.
- If a PID is invalid or if the *kill* API fails, the program will flag an error message.

## (IX) alarm

- *alarm* can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds.
- *alarm* API function prototype:

```
#include<signal.h>

unsigned int alarm(unsigned int time_interval);
```

- Returns: number of CPU seconds left in the process timer, as set by previous *alarm* system call.
- *time_interval* – number of CPU seconds elapsed time, after which the kernel will send the SIGALRM signal to the calling process.
  If *time_interval* = 0, it turns off the alarm clock.
- Effect of previous *alarm* is cancelled and process timer is reset with new *alarm* call.
- Process alarm clock is not passed on to its *fork*ed child, but an *exec*ed process retains the same alarm clock value as was prior to the *exec* API call.
- *alarm* API is used to implement *sleep* API – suspends a calling process for the specified number of CPU seconds.
- Process will be awakened by either the elapsed time exceeding the *timer* value or when the process is interrupted by a signal.

- The implementation is given below:

```
#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;
}
```

- The *sleep* function sets up a signal handler for SIGALRM, calls the *alarm* API to request the kernel to send the SIGALRM signal after the *timer* interval and suspends its execution via the *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.

- BSD UNIX defines the *ualarm* function.
- It is same as that of *alarm* API, but argument and return values are in microseconds.
- This is particularly useful for time-critical applications where the resolution of time must be in microsecond levels.
- This can be used to implement BSD-specific *usleep* function, which is similar to *sleep*, but its argument is in microseconds.

## (X) Interval Timers

- Interval timers are implemented using the *alarm* API.
- These 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_num){
        alarm(INTERVAL);
        //do scheduled tasks
}

int main(){
        struct sigaction action;
        sigemptyset(&action.sa_mask);
        action.sa_handler = (void (*)())callme;
        action.sa_flag = SA_RESTART;
        if(sigaction(SIGALRM,&action,0) == -1){
                perror("sigaction");
                return 1;
        }
        if(alarm(INTERVAL) == -1)
                perror("alarm");
        else
                while(1){
                        //do normal operation
                }
        return 0;
}
```

- 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 real clock seconds.
- The program then goes off to perform its normal operation in an infinite loop.
- When the timer expires, the *callme* function is invoked, which restarts the alarm clock for another 5 seconds and then does the scheduled tasks.
- When the *callme* function returns, the program returns continues its normal operation until another timer expiration.

- *setitimer* API – additional capabilities defined in BSD UNIX. Also available in UNIX System V.3 and V.4, but not specified by POSIX.
- POSIX.1b has new set of APIs for interval timer manipulation.

- Additional features of *setitimer* API are:
  - *alarm* resolution time is in seconds.
  *setitimer* resolution time is in microseconds.
  - *alarm* can be used to set up 1 real-time clock timer per process.
  *setitimer* can set up 3 - real-time clock timer, timer based on user time spent by a process, timer based on total user time and system times spent by a process.
- *setitimer* and *getitimer* function prototypes are given as follows:

```
#include<sys/time.h>

int setitimer(int which, const struct itimerval *val,
                     struct itimerval *old);
int getitimer(int which, struct itimerval *old);
```

- Return values for both: 0 if successful, -1 if fail.
- Timers set by *setitimer* in a parent process will not be inherited by child processes, but are retained when a process *exec*s a new program.
- *which* argument – specifies which timer to process.
- The following table specifies the various values for *which* argument –

| ***which* value** | **Timer type** |
|---|---|
| **ITIMER_REAL** | Timer based on real-time clock. Generates a SIGALRM signal 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. |

- ITIMER_VIRTUAL and ITIMER_PROF timers – useful in timing the total execution time of selected user functions, as the timer runs only while the user process is running.

- *struct itimerval* defined in *<sys/time.h>* header as –

```
struct itimerval{
        struct timeval it_interval; //timer interval
        struct timeval it_value; //current value
};
```

- For *setitimer* API – *it_value* is the time to set the named timer, *it_interval* is the time to reload the timer when it expires.
- For *getitimer* API – *it_value* is the named timer's remaining time to expiration, *it_interval* is the time to reload the timer when it expires.

- The following timer illustrates how to set up a real-time clock interval timer using the *setitimer* API.
- The timer is specified to be reloaded automatically.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/time.h>
#include<signal.h>
#define INTERVAL 2

void callme(int sig_num){
        //do scheduled tasks
}

int main(){
        struct itimerval val;
        struct sigaction action;
        sigemptyset(&action.sa_mask);
        action.sa_handler = (void (*)())callme;
        action.sa_flag = SA_RESTART;
        if(sigaction(SIGALRM,&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(IIMER_REAL,&val,0) == -1)
                perror("alarm");
        else
                while(1){
                        //do normal operation
                }
        return 0;
}
```

- The real time clock timer set by the *setitimer* API is different from that set by the *alarm* API.
- Thus, a process may set up two real-time clock timers using the two APIs.
- Furthermore, since the *alarm* and *setitimer* APIs require that users set up signal handling to catch timer expiration, they should not be used in conjunction with the *sleep* API.
- This is because the *sleep* API may modify the signal handling function for the SIGALRM signal.

## (XI) POSIX.1b Timers

- POSIX.1b timers are a set of APIs for interval timer manipulation defined in POSIX.1b implementation.
- These are more flexible and powerful than UNIX timers –
  - Users may define multiple independent timers per system clock.
  - Timer resolution is in nanoseconds.
  - Users may specify the signal to be raised when a timer expires, on a per-timer basis.
  - Timer interval may be specified as either an absolute or a relative time.
- The limit on maximum number of POSIX timers per process is given by TIMER_MAX constant defined in <*limits.h*> header.
- POSIX timers created by a process are not inherited by its child process, but are retained across the *exec* system call.
- Unlike UNIX timers, POSIX timer can be used safely with *sleep* API if it does not use the SIGALRM signal when it expires.
- 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_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);
```

- *timer_create* API – used to dynamically create a timer and return its handler.
- *clock* argument – specifies which system clock the new timer should be based on.
- *clock* value may be CLOCK_REALTIME for creating a real-time clock timer.
- *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;
};
```

- The data structure of the *sigev_value* field is:

```
union sigval{
        int sival_int;
        void *sival_ptr;
};
```

- *timer_settime* API – used to start or stop a running timer.

- *flag* argument in *timer_settime* API may be 0 or TIMER_RELTIME if the timer start time is relative to the current time.
- If the *flag* argument is TIMER_ABSTIME, the timer start time is an absolute time.
- The ANSI C *mktime* function may be used to generate the absolute time for setting a timer.
- If the *val.it_value* is 0, it stops the timer from running.
- If the *val.it_interval* is 0, the timer will not restart after it expires.
- The *old* argument of *timer_settime* API is used to obtain the previous timer values.
- It may be set to NULL, and no timer values are returned.
- *timer_gettime* API – used to query the current values of a timer.
- *old* argument of *timer_gettime* API returns the current values of the named timer.
- The *struct itimerspec* data type is defined as:

```
struct itimerspec{
        struct timespec it_interval;
        struct timespec it_value;
};
```

- The *itimerspec::it_value* specifies the time remaining in the timer, and the *itimerspec::it_interval* specifies the new time to reload the timer after it expires.
- The *struct timespec* data structure is defined as:

```
struct timespec{
        time_t tv_sec;
        long tv_nsec;
};
```

- All times are specified in seconds via the *timespec::tv_sec* field, and in nanoseconds via the *timespec::tv_nsec* field.
- *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 are not being handled by their target processes.
- Instead, the kernel records the number of these overrun signals per timer.
- The *timer_getoverrun* API can be used to determine the amount of time elapsed between the timer started or handled to the present time, based on the overrun count of a named timer.
- The overrun count in a timer is reset whenever a process handles the timer signal.
- *timer_delete* API – used to destroy a timer created by *timer_create* API.

- The following program illustrates how to set up an absolute-time timer that would have gone off at 23:59:59, on 31 December 1999.

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<time.h>
#define TIMER_TAG 12

void callme(int signo,siginfo_t *evp,void *ucontext){
        time_t tim = time(0);
        cerr << "callme: " << evp->si_value.sival_int
             << ", signo: " << signo << ", " << ctime(&tim);
}

int main(){
        struct sigaction sigv;
        struct sigevent sigx;
        struct itimerspec val;
        struct tm do_time;
        timer_t t_id;

        sigemptyset(&sigv.sa_mask);
        sigv.sa_flag = SA_SIGINFO;

        sigv.sa_sigaction = callme;
        if(sigaction(SIGUSR1,&sigv,0) == -1){
                perror("sigaction");
                return 1;
        }
        sigx.sigev_notify = SIGEV_SIGNAL;
        sigx.sigev_signo = SIGUSR1;
        sigx.sigev_value.sival_int = TIMER_TAG;
        if(timer_create(CLOCK_REALTIME,&sigx,&t_id) == -1){
                perror("timer create");
                return 1;
        }
        //set timer to go off at December 31, 1999, 23:59:59
        do_time.tm_hour = 23;
        do_time.tm_min = 59;
        do_time.tm_sec = 59;
        do_time.tm_mon = 12;
        do_time.tm_year = 99;
        do_time.tm_mday = 31;

        val.it_value.tv_sec = mktime(&do_time);
        val.it_value.tv_nsec = 0;
```

```
        val.it_interval.tv_sec = 15;
        val.it_interval.tv_nsec = 0;
        cerr<<"Timer will go off at:"<<ctime(&val.it_value.tv_sec);

        if(timer_settime(t_id,TIMER_ABSTIME,&val,0) == -1){
                perror("timer_settime");
                return 2;
        }

        //do something, then wait for the timer to expire twice
        for(int i=0; i<2; i++)
                pause();
        if(timer_delete(t_id) == -1){
                perror("timer_delete");
                return 3;
        }
        return 0;
}
```

- The above program first sets up the *callme* function as the signal handler for the SIGUSR1 signal.
- It then creates a timer based on the system real-time clock.
- The program specifies that the timer should raise the SIGUSR1 signal whenever it expires, and the timer-specific data that should be sent along with the signal is TIMER_TAG.
- The timer handler returned by the *timer_create* API is stored in the *t_id* variable.
- The next step is to set the timer to go off on 31 December 1999, at 23:59:59 hours and the timer should re-run for every 30 seconds thereafter.
- The absolute expiration date/time is specified in the *do_time* variable (of type *struct tm*) and is being converted to a *time_t*-type value via the *mktime* function.
- After all these are done, the *timer_settime* function is called to start the timer running.
- The program then waits for the timer to expire at the said date/time and expires again 30 seconds later.
- Finally before the program terminates, it calls the *timer_delete* to free all system resources allocated for the timer.

- The above program can be modified to use a relative-time timer instead, as follows. The *main* function is modified to set the timer to go off 60 minutes from now and repeat every 120 seconds after.

```
int main(){
        //set up sigaction for SIGUSR1
        ...
        //create a timer using timer_create
        ...
        struct itimerspec val;
        val.it_value.tv_sec = 60; //expire 60s from now
        val.it_value.tv_nsec = 0;
        val.it_interval.tv_sec = 120; //repeat every 120s
        val.it_interval.tv_nsec = 0;
        if(timer_settime(t_id,0,&val,0) == -1){
                perror("timer_settime");
                return 2;
        }
        //wait for timer to expire
        ...
}
```

- The differences between the *main* functions of this program and the previous program are:
  - the *do_time* variable and *mktime* API are not being used.
  - the *val.it_value* is set directly with the relative time (from the present) when the timer will first expire.
  - the second argument to *timer_settime* call is set to 0 instead of TIMER_ABSTIME.

## (XII) Daemons

- Daemons are processes that live for a long time.
- These are often started when system is bootstrapped and terminate only when the system is shut down.
- There is no controlling terminal for daemons; they always run in the background.
- Perform day-to-day activities in the UNIX system.
- Process names of a daemon usually end with "*d*".
- Examples: *syslogd* – a daemon that implements system logging facility, *sshd* – a daemon that serves incoming SSH connections.
- Daemons respond to network requests, hardware activity, or other programs by performing some task.
- *cron* daemon performs defined tasks at scheduled times.
- Alternative terms for daemons – service (Windows and later versions of Linux), started task (IBM z/OS), ghost job (XDS UTS).
- Network services – daemons that connect to a computer network.

## (XIII) Daemon Characteristics

- *ps* command – prints the status of various processes in the system.
- Options:
  -a: displays the status of process owned by others
  -x: displays the processes that do not have a controlling terminal
  -j: displays the job-related information – session ID, process group ID, controlling terminal and terminal process group ID.
- Typical output will contain – parent process ID, process ID, process group ID, session ID, terminal name, terminal process group ID, user ID and command string.

- Kernel processes – any process whose parent process ID is 0.
- These processes are special and generally exist for the entire lifetime of the system.
- These processes run with superuser privileges and have no controlling terminal and no command line.
- Process 1 – *init* process.
- It is the system daemon responsible for starting system services specific to various run levels.
- These services are usually implemented with the help of their own daemons.

- *keventd* daemon – provides process context for running scheduled functions in the kernel.
- *kapmd* daemon – provides support for the advanced power management features available with various computer systems.
- *kswapd* daemon – supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed. (a.k.a pageout daemon)
- *bdflush* daemon – flushes dirty buffers from the buffer cache back to disk when available memory reaches a low-water mark.
- *kupdated* daemon – flushes dirty pages back to disk at regular intervals to decrease data loss in the event of a system failure.

- *portmap* daemon – portmapper daemon which provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers.
- *syslogd* daemon – available to any program to log system messages for an operator. The messages may be printed on a console device and written to a file.
- *inetd* daemon – listens on the system's network interfaces for incoming requests for various network servers.
- *nfsd, lockd, rpciod* daemons – provide support for the Network File System (NFS).

- *cron* daemon – executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by *cron*.

- *cupsd* daemon – handles print requests on the system (print spooler).
- Most of the daemons run with superuser privilege (a user ID of 0).
- None of the daemons has a controlling terminal: the terminal name is set to a question mark, and the terminal foreground process group is 1.
- All the user-level daemons are process group leaders and session leaders and are the only processes in their process group and session.
- The parent of most of these daemons is the *init* process.

## (XIV) Coding Rules

- Certain rules are written to code a daemon so as to prevent unwanted interactions from happening in the system.
- *Rule 1* – Call *umask* to set the file mode creation mask to 0.
- Inherited file mode creation mask is set to deny certain permissions.
- If the daemon process is going to create files, the user may want to set specific permissions.
- Example: if a process specifically creates files with group-read and group-write permissions enabled, the file mode creation mask should be set accordingly.

- *Rule 2* – Call *fork* and make the parent *exit*.
- If the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done.
- The child inherits the process group ID of the parent but gets a new process ID, which implies that the child is not a process group leader.
- This rule is a prerequisite for the call to *setsid* that is done next.

- *Rule 3* – Call *setsid* to create a new session.
- The process becomes a session leader of a new session.
- The process becomes the process group leader of a new process group.
- The process has no controlling terminal.

- *Rule 4* – Change the current working directory to the root directory.
- Current working directory inherited from parent could be on a mounted file system.
- Daemons normally exist until the system is rebooted.
  If it stays on a mounted file system, that file system cannot be unmounted.

- Alternate – some daemons might change the current working directory to some specific location, where they will do all their work.
- Example: line printer spooling daemons often change to their spool directory.

- *Rule 5* – Unneeded file descriptors should be closed.
- This rule prevents the daemon from holding open any descriptors that it may have inherited from its parent, which could be a shell or some other process.
- *open_max* function or *getrlimit* function determine the highest descriptor and close all descriptors up to that value.

- *Rule 6* – 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.
- As 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, output from the daemon should not show up on the terminal, and input from users should not be read by daemon.

- Based on the coding rules stated above, the following function is used to “daemonize” a process:

```
#include "apue.h"
#include<syslog.h>
#include<fcntl.h>
#include<sys/resource.h>

void daemonize(const char *cmd){
        int i, fd0, fd1, fd2;
        pid_t pid;
        struct rlimit rl;
        struct sigaction sa;

        //clear file creation mask
        umask(0);
        //get maximum number of file descriptors
        if(getrlimit(RLIMIT_NOFILE,&rl) < 0)
                err_quit("%s: cannot get file limit",cmd);
        //become session leader to lose controlling TTY
        if((pid=fork()) < 0)
                err_quit("%s: cannot fork",cmd);
        else if(pid != 0) //parent
                exit(0);
        setsid();
```

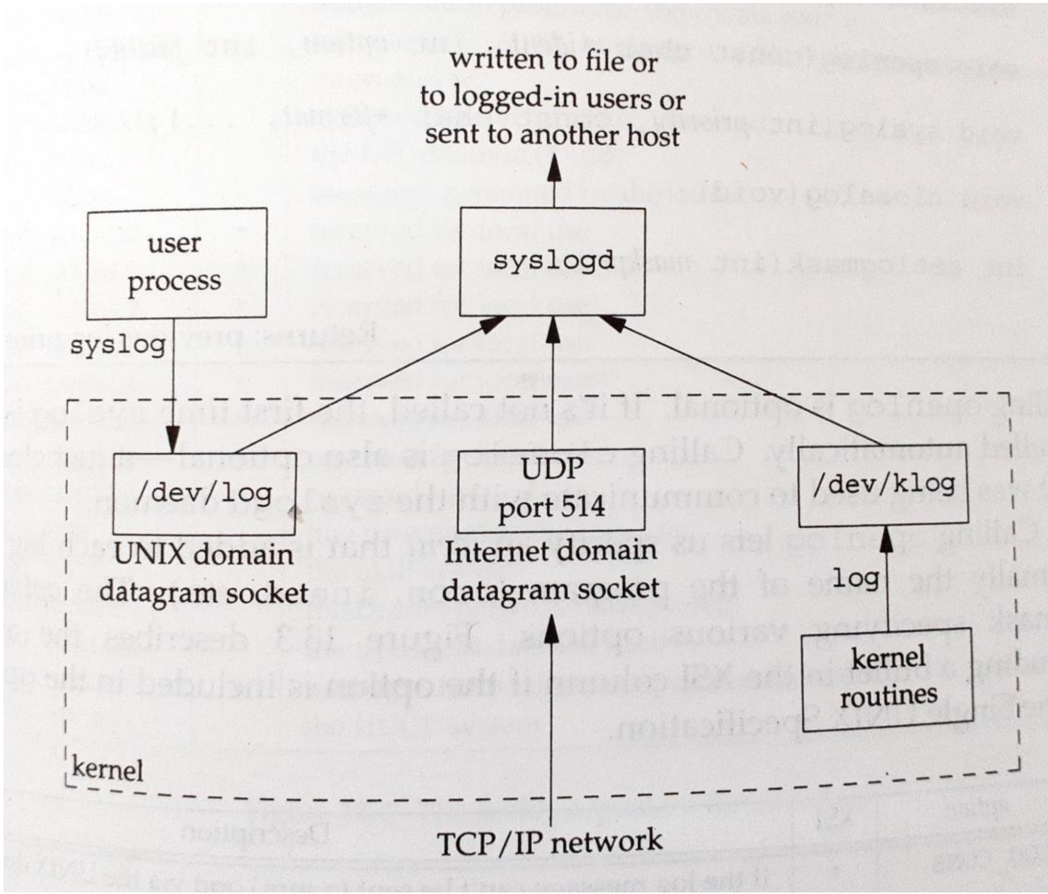
```
        //ensure future opens will not allocate controlling TTYs
        sa.sa_handler = SIG_IGN;
        sigemptyset(&sa.sa_mask);
        sa.sa_flag = 0;
        if(sigaction(SIGHUP, &sa, NULL) < 0)
                err_quit("%s: cannot ignore SIGHUP");
        if((pid=fork()) < 0)
                err_quit("%s: cannot fork",cmd);
        else if(pid != 0) //parent
                exit(0);
        //change current working directory to the root
        if(chdir("/") < 0)
                err_quit("%s: cannot change directory to /");
        //close all open file descriptors
        if(rl.rlim_max == RLIM_INFINITY)
                rl.rlim_max = 1024;
        for(i=0; i<rl.rlim_max; i++)
                close(i);
        //attach file descriptors 0,1 and 2 to /dev/null
        fd0 = open("/dev/null", O_RDWR);
        fd1 = dup(0);
        fd2 = dup(0);
        //initialize log file
        openlog(cmd,LOG_CONS,LOG_DAEMON);
        if(fd0 != 0 || fd1 != 1 || fd2 != 2){
                syslog(LOG_ERR,"unexpected file descriptors %d %d %d",
                        fd0, fd1, fd2);
                exit(1);
        }
}
```

## (XV) Error Logging

- The main problem faced by a daemon is the handling of error messages.
- Daemon cannot write to standard error, as it should not have a controlling terminal.
- Daemons should not write to console device, as many workstations have windowing system running on console device.
- Daemons should not write error messages into a separate file, as it is impossible for admin to keep up with which daemon writes to which log file, and to check these logs on a regular basis.
- Thus, a central daemon error-logging facility is required.

- *syslog* facility developed at Berkeley, for the BSD 4.2 is used for error logging.
- Most systems derived from BSD support *syslog*.
- It is included as an XSI extension in Single UNIX Specification.
- Most daemons use *syslog* to log their error messages.

- The structure of *syslog* facility is given below –




- 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* 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.

  *syslog* never generates these UDP datagrams: they require explicit network programming by the process generating the log message.
- *syslogd* daemon reads all three forms of log messages.
- On start-up, *syslogd* reads a configuration file, usually */etc/syslog.conf*, which determines where different classes of messages are to be sent.
- 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.

- Interface to the logging facility is through these functions:

```
#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);
```

- Calling *openlog* is optional.
  If it is not called, when *syslog* is called for the first time, *openlog* is called automatically.
- Calling *closelog* is also optional.
  It just closes the descriptor that was being used to communicate with the *syslogd* daemon.
- *ident* – added to each log message.
  This is normally the name of the program (*cron*, *inetd*, etc.)
- *option* – bit mask specifying various options.
- The various options are listed in the table below:

| ***option*** | **Description** |
|---|---|
| **LOG_CONS** | If the log message cannot be sent to *syslogd* via the UNIX domain datagram, the message is written to the console instead. |
| **LOG_NDELAY** | Open the UNIX domain datagram socket to the *syslogd* daemon immediately; do not wait until the first message is logged. |
| **LOG_ODELAY** | Delay the open of the connection to the *syslogd* daemon until the first message is logged. |
| **LOG_NOWAIT** | Do not wait for child processes that might have been created in the process of logging the message. |
| **LOG_PERROR** | Write the log message to standard error in addition to sending it to *syslogd*. |
| **LOG_PID** | Log the process ID with each message. This is intended for daemons that fork a child process to handle different requests. |

- *facility* – lets the configuration file specify that messages from different facilities are to be handled differently.
- If we do not call *openlog*, or if we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to *syslog*.
- Single UNIX Specification defines only a subset of the facility codes typically available on a given platform.

- The following table gives details about the available facility codes:

| *facility* | **Description** |
|---|---|
| **LOG_AUTH** | Authorization programs such as *login, su, getty* |
| **LOG_AUTHPRIV** | Same as LOG_AUTH, but logged to file with restricted permissions |
| **LOG_CRON** | *cron* and *at* |
| **LOG_DAEMON** | System daemons such as *inetd, routed* |
| **LOG_FTP** | FTP daemon *ftpd* |
| **LOG_KERN** | Messages generated by the kernel |
| **LOG_LOCAL0** | Reserved for local use |
| **LOG_LOCAL1** | Reserved for local use |
| **LOG_LOCAL2** | Reserved for local use |
| **LOG_LOCAL3** | Reserved for local use |
| **LOG_LOCAL4** | Reserved for local use |
| **LOG_LOCAL5** | Reserved for local use |
| **LOG_LOCAL6** | Reserved for local use |
| **LOG_LOCAL7** | Reserved for local use |
| **LOG_LPR** | Line printer system functions such as *lpd, lpc* |
| **LOG_MAIL** | Mail system |
| **LOG_NEWS** | Usenet network news system |
| **LOG_SYSLOG** | *syslogd* daemon itself |
| **LOG_USER** | Messages from other user processes |
| **LOG_UUCP** | UUCP system |

- *level* – ordered by priority, from highest to lowest.
- The following table lists the *level* values:

| *level* | **Description** |
|---|---|
| **LOG_EMERG** | Emergency – system is unusable (highest priority) |
| **LOG_ALERT** | Condition that must be fixed immediately |
| **LOG_CRIT** | Critical condition (such as hard device error) |
| **LOG_ERR** | Error condition |
| **LOG_WARNING** | Warning condition |
| **LOG_NOTICE** | Normal, but significant condition |
| **LOG_INFO** | Informational message |
| **LOG_DEBUG** | Debug message (lowest priority) |

- *priority* – combination of *facility* and *level*
- The *format* argument and any remaining arguments are passed to the *vsprintf* function for formatting.
- Any occurrence of the two characters *%m* in the *format* are first replaced with the error message string (*strerror*) corresponding to the value of *errno*.
- The *setlogmask* function can be used to set the log priority mask for the process.
- This function returns the previous mask.
- When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask.
- Attempts to set the log priority mask to 0 will have no effect.
- *logger* program – provided by many systems to send log messages to the *syslog* facility.
- Optional arguments – specifies the *facility*, *level*, and *ident*.
- Single UNIX Specification does not define any options.
- *logger* is intended for a shell script running non-interactively that needs to generate log messages.

- Example: in a line printer spooler daemon, the following sequence might be seen:

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

- The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default facility to the line printer system.
- The call to *syslog* specifies an error condition and a message string.
- If *openlog* was not called, the second call could have been

```
syslog(LOG_ERR|LOG_LPR, "open error for %s: %m", filename);
```

- *vsyslog* – a variant of *syslog* that handles variable argument lists.

```
#include<syslog.h>
#include<stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

- However, this function is not included in the Single UNIX Specification.
- Most *syslogd* implementations will queue messages for a short time.
- If a duplicate message arrives during this time – the *syslog* daemon will not write it to the log.
- Instead, the daemon will print out a message similar to "last message repeated N times."

## (XVI) Client-Server Model

- Common use for a daemon process is to be a server process.
- *syslogd* process – a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.
- Server – a process that waits for a client to contact it, requesting some type of service.
- The service being provided by the *syslogd* server is the logging of an error message.
- Communication between the client and the server is one-way.
  The client sends its service request to the server; the server sends nothing back to the client.