UNIT – 4

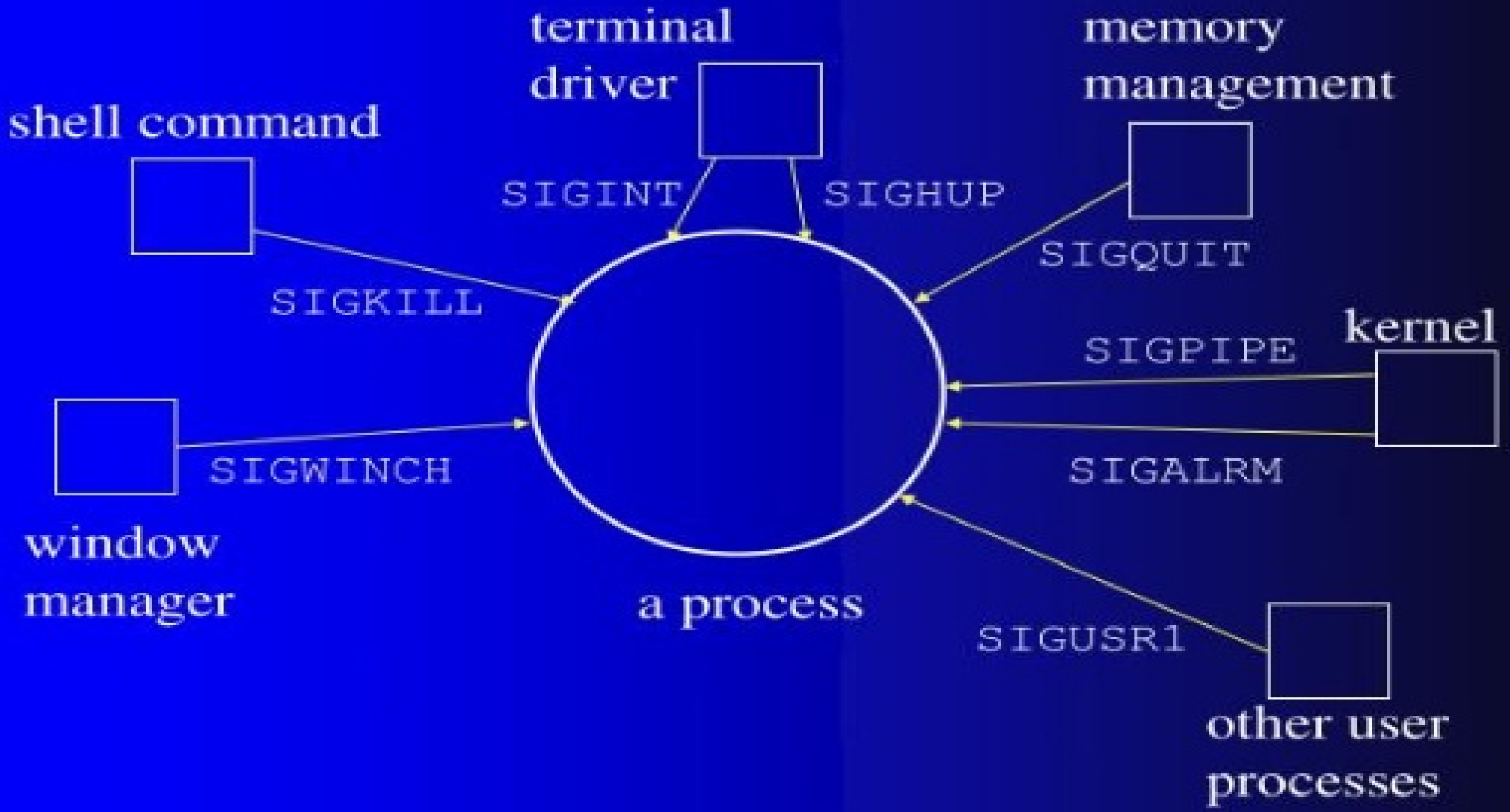
Signals :

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

Daemon Process :

- Introduction,
- Daemon Characteristics,
- Coding Rules,
- Error Logging,
- Client- Server Model

Signal Sources



SIGNALS

- Signals are software interrupts.
- Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.
- When a signal is sent to a process, it is pending on the process to handle it. The process can react to pending signals in one of three ways:
  1. Accept the default action of the signal, which for most signals will terminate the process.
  2. Ignore the signal. The signal will be discarded and it has no affect whatsoever on the recipient process.
  3. Invoke a user-defined function. The function is known as a signal handler routine and the signal is said to be caught when this function is called.

SIGNALS

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

SIGNALS

Name	Description	Default action
SIGTTOU	background write to control tty	stop process
SIGUSR1	user-defined signal	Terminate
SIGUSR2	user-defined signal	Terminate
SIGTERM	termination	Terminate
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process

SIGNALS

THE UNIX KERNEL SUPPORT OF SIGNALS

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.
- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

SIGNALS

The function prototype of the signal API is:

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

Returns: previous disposition of signal (see following) if OK, SIG_ERR on error

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

SIGNALS

The function prototype of the signal API is:

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

The sig_no argument is just the name of the signal.
The value of handler is
(a) the constant SIG_IGN,
(b) the constant SIG_DFL, or
(c) the address of a function to be called when the signal occurs.

SIGNALS

The function prototype of the signal API is:

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

If we specify SIG_IGN, we are telling the system to ignore the signal.
(Remember that we cannot ignore the two signals SIGKILL and SIGSTOP)

When we specify SIG_DFL, we are setting the action associated with the signal to its default value.

When we specify the address of a function to be called when the signal occurs, we are arranging to "catch" the signal. We call the function either the signal handler or the signal-catching function.

SIGNALS

The function prototype of the signal API is:

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

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

The signal function's first argument, sig_no, is an integer.

The second argument is a pointer to a function that takes a single integer argument and returns nothing.

The function whose address is returned as the value of signal takes a single integer argument (the final (int)).

SIGNALS

The function prototype of the signal API is:

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

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

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

These constants can be used in place of the "pointer to a function that takes an integer argument and returns nothing," the second argument to signal, and the return value from signal.

The three values used for these constants need not be -1, 0, and 1.

They must be three values that can never be the address of any declarable function.

SIGNALS

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_n
 um<<endl;
}
```

```
 int main() /*main function*/
{
signal(SIGTERM,catch_sig)
;  signal(SIGINT,SIG_IGN);
signal(SIGSEGV,SIG_DFL);
pause( );/*wait for a signal
interruption*/
}
```

The SIG_IGN specifies a signal is to be ignored, which means that if the signal is generated to the process, it will be discarded without any interruption of the process.

SIGNALS

```
#include<stdio.h>
#include<signal.h>
/*signal handler function*/
static void sig_usr(int signo)
/* arg is signal number */
{
if (signo == SIGUSR1)
      printf("received SIGUSR1\n");
 else if (signo == SIGUSR2)
      printf("received SIGUSR2\n");
else
printf("received signal %d\n",
     signo);
}
```

```
int    main(void)
{
if (signal(SIGUSR1, sig_usr) == SIG_ERR)
       perror("can't catch SIGUSR1");
if (signal(SIGUSR2, sig_usr) == SIG_ERR)
      perror("can't catch SIGUSR2");
for ( ; ; )
 pause();
}
```

SIGNALS

```
$ ./a.out &              start process in background
[1] 7216                 job-control shell prints job number and process ID
$ kill -USR1 7216        send it SIGUSR1
received SIGUSR1
$ kill -USR2 7216        send it SIGUSR2
received SIGUSR2
$ kill 7216               now send it SIGTERM
[1]+ Terminated   ./a.out
```

When we send the SIGTERM signal, the process is terminated, since it doesn't catch the signal, and the default action for the signal is termination.

SIGNALS

kill and raise Functions

The kill function sends a signal to a process or a group of processes.
The raise function allows a process to send a signal to itself.

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

Both return: 0 if OK, –1 on error

The call

```
raise(signo);
```

is equivalent to the call

```
kill(getpid(),
signo);
```

SIGNALS

kill and raise Functions

#include <signal.h>

int kill(pid_t pid, int signo);

There are four different conditions for the pid argument to kill.

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

SIGNALS AND DAEMON PROCESSES

```
#include<iostream.h>
#include<signal.h>
Void handler(int);
Int main()
{
 int pid;
If((pid=fork())==0)
{
 singal (SIGINT,SIG_DFL);
Sleep(2);
}
 else
{
 kill(pid, SIGINT);
 sleep(5);
Cout<“parent exiting”;
}
Void handler()
{
 cout<<“signal received by child”;
}
```

SIGNALS

alarm and pause Functions

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

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

Returns: 0 or number of seconds until previously set alarm.

The seconds value is the number of clock seconds in the future when the signal should be generated.

SIGNALS

alarm and pause Functions

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

- If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function.
- That previously registered alarm clock is replaced by the new value.
- If a previously registered alarm clock for the process has not yet expired and if the seconds value is 0, the previous alarm clock is canceled.
- The number of seconds left for that previous alarm clock is still returned as the value of the function.
- Although the default action for SIGALRM is to terminate the process, most processes that use an alarm clock catch this signal.

```
#include<iostream.h>
#include<signal.h>
Using namespace std;
Void sleep(int i);
{
 alarm(i);
 pause();
}

Int main()
{
 cout<<“amit is waiting”;
 sleep(5);
Cout<“amit finished waiting”;
Return 0;
}
```

SIGNALS

alarm and pause Functions

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>
 int pause(void);
```

Returns: –1 with
errno set to EINTR

The only time
pause returns is if a
signal handler is
executed and that
handler returns.
In that case, pause

SIGNALS

alarm and pause Functions

Using alarm and pause, we can put a process to sleep for a specified amount of time. The sleep() can be implemented using alarm() and pause().

```
#include <signal.h>
 #include
<unistd.h>
static void
sig_alrm(int signo)
{
/* nothing to do, just return to
 wake up the pause */
}
```

```
 unsigned int sleep(unsigned int nsecs)
{
 if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
alarm(nsecs); /* start the timer */
pause();    /* next caught signal wakes us up
 */  return(alarm(0));
/* turn off timer, return unslept time */
}
```

SIGNALS

SIGNAL SETS

We need a data type to represent multiple signals—a signal set

POSIX.1 defines the data type sigset_t to contain a signal set and the following five functions to manipulate signal sets.

```
#include <signal.h>
int sigemptyset(sigset_t
*set);  int sigfillset(sigset_t
*set);
int sigaddset(sigset_t  *set, int
signo);  int sigdelset(sigset_t     *set,
int                          signo);
```

Returns: 0 if OK, -1 on error.

SIGNALS

SIGNAL MASK

Signal mask of a process is the set of signals currently blocked from delivery to that process.

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

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

```
#include <signal.h>
int sigprocmask(int cmd,  const sigset_t *new_mask,  sigset_t *old_mask);
```

Returns: 0 if OK, -1 on error

.

SIGNALS

SIGNAL MASK

The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API.

The possible values of cmd and the corresponding use of the new_mask value are:

.

Cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument.
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask.
SIG_UNBLOCK	Removes the signals specified in the new_mask argument from the calling process signal mask.

SIGNALS

SIGNAL MASK

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

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

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

SIGNALS

SIGNAL MASK

The program prints the names of the signals in the signal mask of the calling process

```
#include <stdio.h>
#include <signal.h>
 int main()
{
sigset_t    sigset;
sigemptyset(&sigset);
/*initialize set*/
if (sigprocmask(0,
NULL, &sigset) < 0)
      perror("sigprocma
      sk error");
if (sigismember(&sigset, SIGINT))
           printf("SIGINT ");
if (sigismember(&sigset, SIGQUIT))
         printf("SIGQUIT ");
if (sigismember(&sigset, SIGUSR1))
           printf("SIGUSR1 ");
if (sigismember(&sigset, SIGALRM))
         printf("SIGALRM ");
}
```

SIGNALS

SIGPENDING FUNCTION

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

The set of signals is returned through the set argument

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

Returns: 0 if OK, –1 on error.

SIGNALS

SIGPENDING FUNCTION

- The process blocks SIGQUIT, saving its current signal mask (to reset later), and then goes to sleep for 5 seconds.
- Any occurrence of the quit signal during this period is blocked and won't be delivered until the signal is unblocked.
- At the end of the 5-second sleep, we check whether the signal is pending and unblock the signal.

SIGNALS - SIGPENDING FUNCTION

```
#include <signal.h>
#include <unistd.h>
static void    sig_quit(int signo)
{
printf("caught SIGQUIT\n");
if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
  perror("can't reset SIGQUIT");
}
int    main(void)
{
sigset_t newmask, oldmask, pendmask;
 if (signal(SIGQUIT, sig_quit) ==
SIG_ERR)
   perror("can't catch SIGQUIT");
/* Block SIGQUIT and save current signal
mask*/
sigemptyset(&newmask);
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
     perror("SIG_BLOCK error");
 sleep(5);
     /* SIGQUIT here will remain
pending */
if (sigpending(&pendmask) < 0)
    perror("sigpending error");
if (sigismember(&pendmask, SIGQUIT))
      printf("\nSIGQUIT pending\n");
/* Reset signal mask which unblocks SIGQUIT*/
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    perror("SIG_SETMASK error");
printf("SIGQUIT unblocked\n");
 sleep(5);
/* SIGQUIT here will terminate with
 core file */
exit(0);
```

SIGNALS

Sigaction() Function

- The sigaction() function allows us to examine or modify (or both) the action associated with a particular signal.
- This function supersedes the signal() function from earlier releases of the UNIX System.

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

Returns: 0 if OK, –1 on error

SIGNALS

sigaction() Function

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

SIGNALS

sigaction() Function

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

```
struct sigaction
{
 void (*sa_handler)(int);      /* addr of signal handler, or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask;            /* additional signals to block */
 int sa_flags;                /* signal options */
 }
```

SIGNALS

sigaction() Function

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

SIGNALS - sigaction FUNCTION

```
#include <signal.h>
#include <iostream.h>
void callme ( int sig_num )
{
cout <<“catch
signal:”<<sig_num<<
endl;
}
int   main(void)
{
sigset_t sigmask;
struct sigaction action, old_action;
 sigemptyset(&sigmask);
 if ( sigaddset( &sigmask, SIGTERM) == -1   ||
 sigprocmask( SIG_SETMASK, &sigmask, 0) == -1)
     perror(“Set signal mask”);
sigemptyset( &action.sa_mask);
 sigaddset( &action.sa_mask, SIGSEGV);
  action.sa_handler = callme;
 action.sa_flags = 0;
 if (sigaction (SIGINT, &action, &old_action) == -1)
      perror(“sigaction”);
pause(); /* wait for signal interruption*/
 return 0;
}
```

SIGNALS - sigaction FUNCTION

- In the program, the process signal mask is set with SIGTERM signal.
- The process then defines a signal handler for the SIGINT signal and also specifies that the SIGSEGV signal is to be blocked when the process is handling the SIGINT signal.
- The process then terminates its execution via the pause API.

SIGNALS

THE SIGCHLD SIGNAL AND THE waitpid API

When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process. Depending on how the parent sets up the handling of the SIGCHLD signal, different events may occur:

1. Parent accepts the **default action** of the SIGCHLD signal:
   - SIGCHLD does not terminate the parent process.
   - Parent process will be awakened.
   - API will return the child's exit status and process ID to the parent.
   - Kernel will clear up the Process Table slot allocated for the child process.
   - Parent process can call the waitpid API repeatedly to wait for each child it created.

SIGNALS

THE SIGCHLD SIGNAL AND THE waitpid API

2. Parent **ignores** the SIGCHLD signal:
   - SIGCHLD signal will be discarded.
   - Parent will not be disturbed even if it is executing the waitpid system call.
   - If the parent calls the waitpid API, the API will suspend the parent until all its child processes have terminated.
   - Child process table slots will be cleared up by the kernel.
   - API will return a -1 value to the parent process.
3. Process **catches** the SIGCHLD signal:
   - The signal handler function will be called in the parent process whenever a child process terminates.
   - If the SIGCHLD arrives while the parent process is executing the waitpid system call, the waitpid API may be restarted to collect the child exit status and clear its process table slots.
   - Depending on parent setup, the API may be aborted and child process table slot not freed.

The sigsetjmp and siglongjmp APIs:

The *sigsetjmp* and *siglongjmp* APIs have similar functions as their corresponding *setjmp* and *longjmp* APIs.

These are used to transfer control from one function to another hence they are called non-local goto statements. The function prototype of these functions are:

```
#include <setjmp.h>
    int    sigsetjmp(sigjmp_buf  jmpb, int save_sigmask );
    void  longjmp(sigjmp_buf  jmpb, int retval);
```

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

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

The *retval* argument specifies the return value of the corresponding *sigsetjmp* API when it is called by *siglongjmp.* It value should be a non-zero number, if it is zero the *siglongjmp* API will reset it to 1.

The *siglongjmp* API is usually called from user defined signal handling functions

SIGNALS

abort() Function

abort function causes abnormal program termination

```
#include <stdlib.h>
 void abort(void);
```

This function never
returns.

This function sends the SIGABRT signal to the caller.
Processes should not ignore this signal.
ISO C states that calling abort will deliver an
unsuccessful termination notification to the
host environment by calling raise(SIGABRT).

SIGNALS

system() Function

#include <stdlib.h>
int system(const char *command);

This function returns is -1 on error.
If the value of *command* is NULL, **system**() returns nonzero if the shell is available, and zero if not.

system() executes a command specified in *command* by calling **/bin/sh -c** *command*, and returns after the command has been completed. During execution of the command, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored

Eg: system("ls –l");

SIGNALS

sleep() Function

sleep - sleep for the specified number of seconds

#include <unistd.h>
unsigned int sleep(unsigned int seconds);

Returns: 0 or number of unslept seconds.

This function causes the calling process to be suspended until either
1. The amount of wall clock time specified by seconds has elapsed.
2. A signal is caught by the process and the signal handler returns.

Eg:

```
sleep(60); // suspend the process for one minute.
```

Interval timers

- We know that, sleep( ) API is constructed using the alarm( ) and pause( ) APIs.
- But sleep( ) API just suspends a process for a fixed amount of time.
- Other than this, the alarm( ) API can also be used to set up an interval timer.
- The interval timer is used to schedule a process to do some tasks at a fixed time interval.

```
#include <sys/time.h>
 int getitimer(int which, struct itimerval *value);

int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

```
struct itimerval
{
struct timeval it_interval; /* next value */
 struct timeval it_value; /* current value */
};

 struct timeval
 {
 long tv_sec; /* seconds */
long tv_usec; /* microseconds */
};
```

Tag	Description
ITIMER_REAL	decrements in real time, and delivers **SIGALRM** upon expiration.
ITIMER_VIRTUAL	decrements only when the process is executing, and delivers **SIGVTALRM** upon expiration.
ITIMER_PROF	decrements both when the process executes and when the system is executing on behalf of the process. Coupled with **ITIMER_VIRTUAL**, this timer is usually used to profile the time spent by the application in user and kernel space. **SIGPROF** is delivered upon expiration

Sample questions

1. What is signal API? Explain the various signal names and their purpose.
2. What is signalmask? Explain with the syntax and a sample program.
3. Explain the sigaction( ) function.
4. Explain the various ways in which a parent can handle different signals when a child process terminates or stops.
5. With the syntax, explain the concept of sigsetjmp ( ) & siglongjmp ( ) functions.
6. Explain the kill and alarm API with syntax.
7. Write a note on interval times.
8. Explain the various signal sets with the sytax.
9. Explain the sleep and abort API with syntax.

DAEMON PROCESSES

- Introduction
- Daemon characteristics
- Coding rules for creating daemons
- Error Logging
- Client – Server Model

DAEMON PROCESSES

- Daemons are processes that live for a long time.
- They are often started when the system is bootstrapped and terminate only when the system is shut down.
- Because they don't have a controlling terminal, we say that they run in the background.
- UNIX systems have numerous daemons that perform day-to-day activities.

DAEMON PROCESSES

Deamon Characteristics

- $ps -axj

PPID	PID	PGID	SID	TTY	TPGID	UID	COMMAND
0	1	0	0	?	-1	0	init
1	2	1	1	?	-1	0	[keventd]
1	3	1	1	?	-1	0	[kapmd]
0	5	1	1	?	-1	0	[kswapd]
0	6	1	1	?	-1	0	[bdflush]
0	7	1	1	?	-1	0	[kupdated]
1	1009	1009	1009	?	-1	32	portmap
1	1048	1048	1048	?	-1	0	syslogd -m 0
1	1335	1335	1335	?	-1	0	xinetd -pidfile /var/run/xinetd.pid
1	1403	1	1	?	-1	0	[nfsd]
1	1405	1	1	?	-1	0	[lockd]
1405	1406	1	1	?	-1	0	[rpciod]
1	1853	1853	1853	?	-1	0	crond
1	2182	2182	2182	?	-1	0	/usr/sbin/cupsd

DAEMON PROCESSES

Deamon Characteristics

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don’t have controlling terminal.
- Daemons are session and group leaders.

DAEMON PROCESSES

Deamon Characteristics

- Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure.
- 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.
- Process ID of 1 is usually init.
- It is a system daemon responsible for, among other things, starting system services specific to various run levels.

DAEMON PROCESSES

Deamon Characteristics

- keventd daemon provides process context for running scheduled functions in the kernel.
- The kswapd daemon is also known as the pageout daemon.
- It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.
- The inetd daemon (xinetd) listens on the system's network interfaces for incoming requests for various network servers.
- The nfsd, lockd, and rpciod daemons provide support for the
- Network File System (NFS).
- The cron daemon (crond) executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by cron.
- The cupsd daemon is a print spooler; it handles print requests on the system.

DAEMON PROCESSES

CODING RULES

1. **Call fork and have the parent exit**. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader.
2. **Call umask to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.

CODING RULES

3. **Call setsid to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
4. **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.
5. **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.

DAEMON PROCESSES CODING RULES

6. **Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**.

   Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user.

   Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon.

   If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

```
#include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <sys/types.h>
 #include <sys/stat.h>
 int main(int argc, char* argv[])
{
    FILE *fp= NULL;
    pid_t process_id = 0;
    pid_t sid = 0; // Create child process
    process_id = fork(); // Indication of fork() failure
   if (process_id < 0)
  {
   printf("fork failed!\n"); // Return failure in exit status
  exit(1);
}
```

```
// PARENT PROCESS. Need to kill it.
 if (process_id > 0)
 {
   printf("process_id of child process %d \n", process_id);
    // return success in exit status
   exit(0);
 }
 //unmask the file mode umask(0); //set new session
sid =setsid();
 if(sid < 0)
 {
 // Return failure
exit(1);
 }
signal(SIGHUP,SIG_IGN);
```

```
// Change the current working directory to root.
chdir("/");
// Close stdin. stdout and stderr
 close(STDIN_FILENO);
 close(STDOUT_FILENO);
close(STDERR_FILENO);
 // Open a log file in write mode.
fp = fopen ("Log.txt", "w+");
 while (1)
 {
```

```
//Dont block context switches, let the process sleep for
some time
sleep(1);
fprintf(fp, "Logging info...\n");
 fflush(fp); // Implement and call some function that does
core work for this daemon.
}
fclose(fp);
 return (0);
}
```

DAEMON PROCESSES

CODING RULES

Example Program:

```
#include <unistd,h>
#include <sys/types.h>
 #include <fcntl.h>
 int daemon_initialise(
 )
{
pid_t pid;
if (( pid = fork() ) < 0)
      return –1;
else if ( pid != 0)
    exit(0); /* parent exits */
 /* child continues */
 setsid( );
 chdir(“/”)
;
umask(0);
 return 0;
 }
```

DAEMON PROCESSES

Error Logging

- One problem a daemon has is how to handle error messages.
- It can't simply write to standard error, since it shouldn't have a controlling terminal.
- We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system.
- We also don't want each daemon writing its own error messages into a separate file.
- A central daemon errorlogging facility is required.

1.SVR4 Streams log Driver::

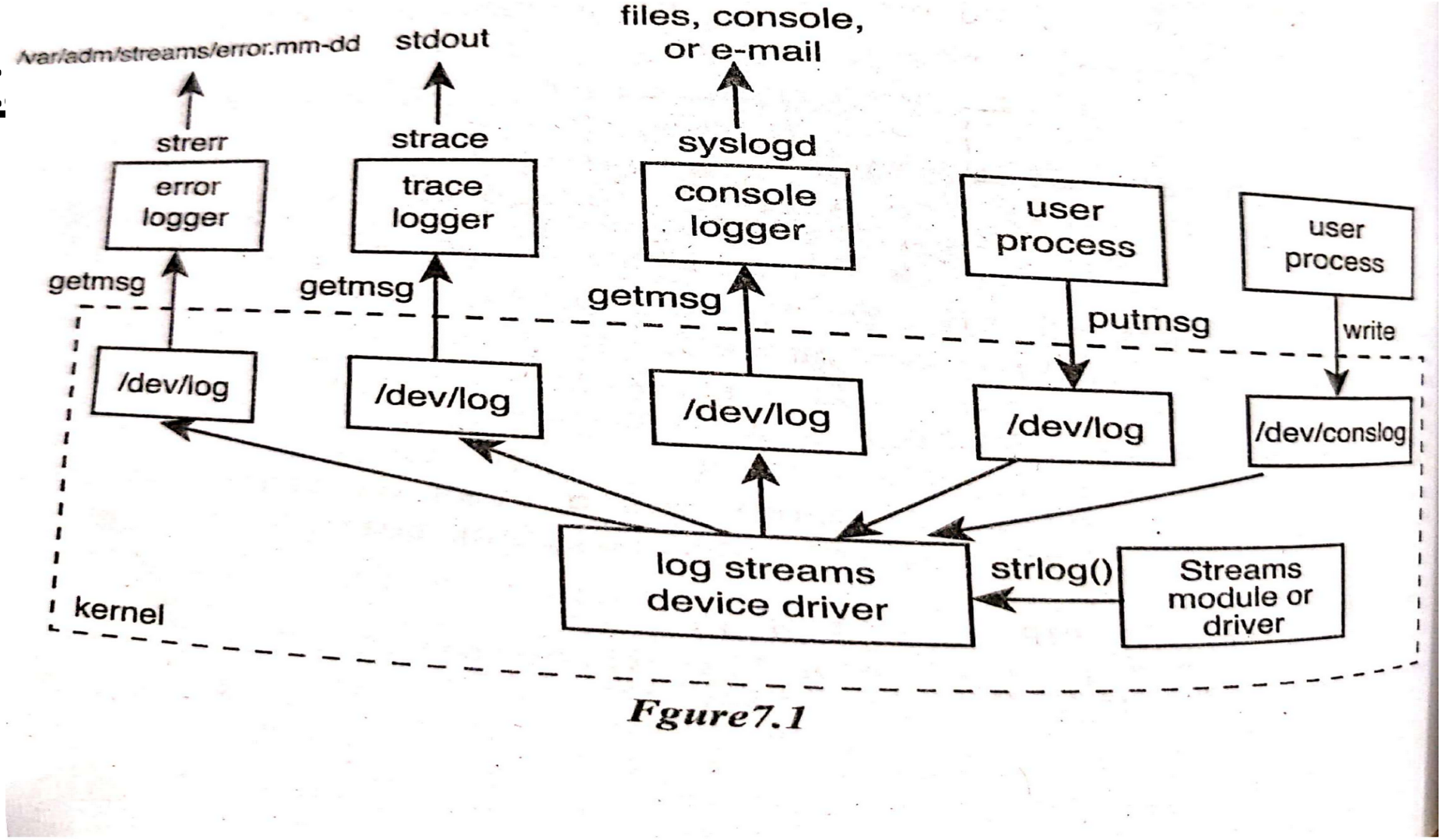
SVR4 provides a streams device driver, with an interface for streams *error logging*, *streams event tracing*, and *console logging*.

Each *log* message can be routed to one of three loggers: the *error logger*, the *trace* logger, *or* the *console logger*.

There are three ways to generate *log* messages and three ways to read them.

DAEMON PROCESSE

Error Logging SVR4 stream 'log' driver



Fgure7.1

Generating log messages:

- Routines within the kernel can call *strlog* to generate log messages. This is normally used by streams modules and streams device drivers for either *error messages or trace* messages.

- User processes (such as a daemon) can *putmsg* to /dev/log. This message can be sent to any of the three loggers.

- A user process (such as a daemon) can write to /dev/conslog. This message is sent only to the console logger.

Reading *log* messages:

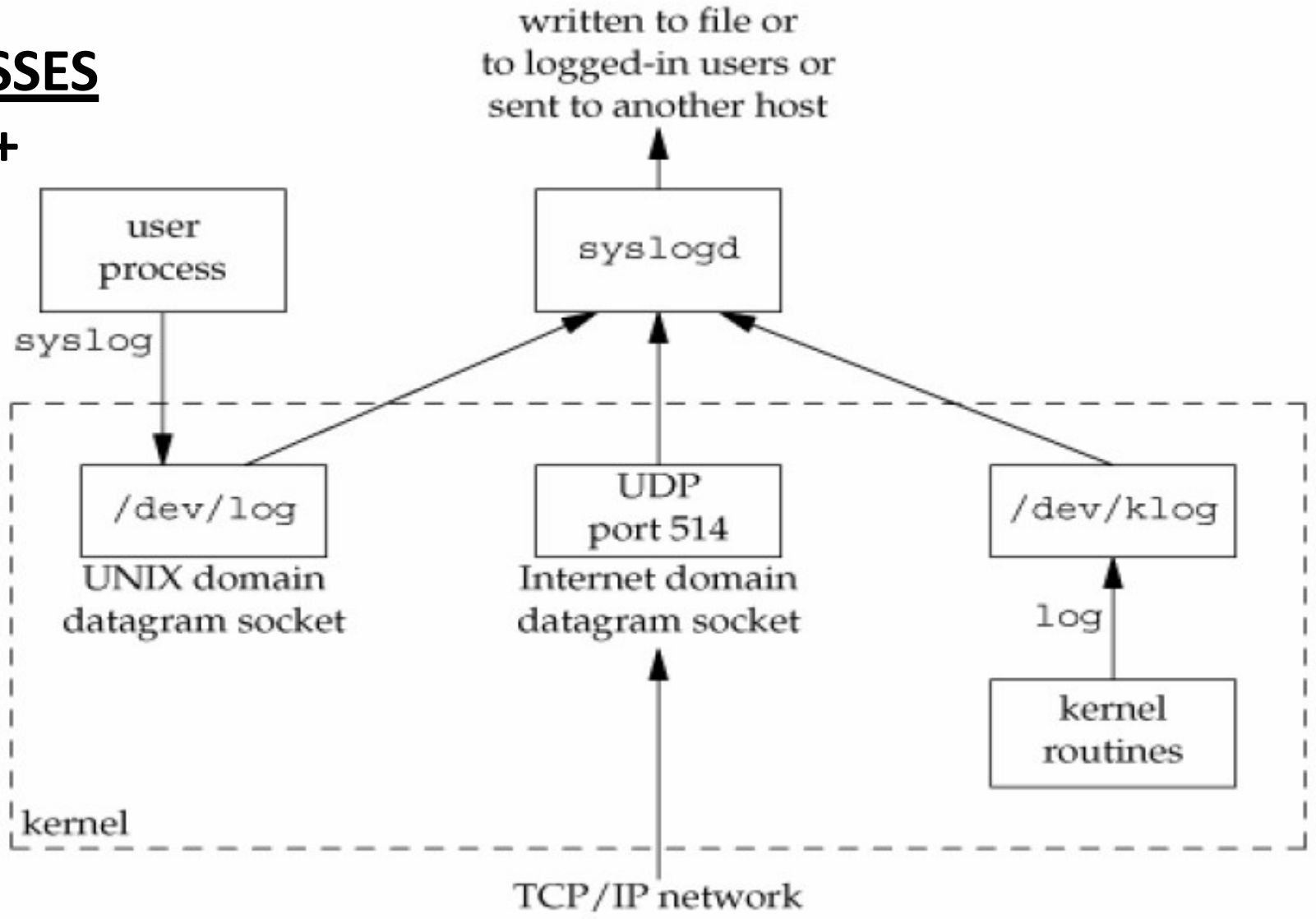
The normal error logger is *strerr*. It appends these messages to a file in the directory */var/adm/strearms*. The file's name is *error.mm-dd*, where *mm* is the month and *dd* is the day of the month. This program, itself a daemon and it runs in the background, appending the *log* messages to the file.

The normal trace logger is *strace*. It can selectively write a specified set of trace messages to its standard output.

The standard console logger is *syslogd*. This program is a daemon that reads a configuration file and writes log messages to specified files or the console device or sends e-mail to certain users.

DAEMON PROCESSES

Error Logging 4.3+ BSD ‘ syslog( ) facility’



The BSD syslog facility

DAEMON PROCESSES

Error Logging

There are three ways to generate log messages:

1. Kernel routines can call the log function.
   These messages can be read by any user process that opens and reads the /dev/klog device.
2. Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514.

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.

Client – Server Model

- The most common use of a daemon process is as a server process.
- In general, a server is a process that waits for a client to contact it, requesting some type of service.
  - e.g. inetd: is a server daemon, that listens on the system's network interfaces for incoming requests for various network servers.
  - Syslogd: this service provides logging of an error messages.

- Hence daemon processes are mainly used for client- server communication

Overview

- What are daemons processes
- Characteristics of daemon process
- Coding rules for daemon
- Error logging facility
    - SVR 4
    - 4.3 BSD
- Client – Server Model