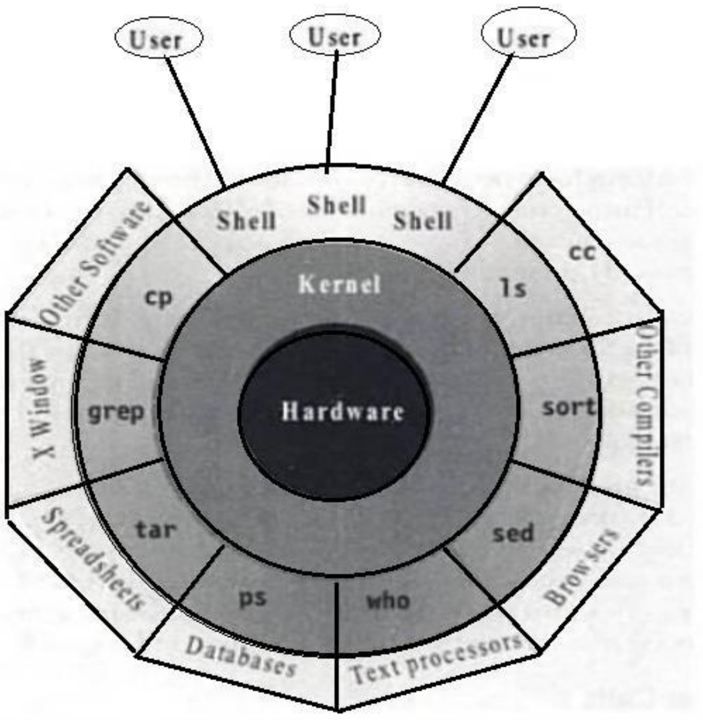
1.(a) Illustrate unix architecture with neat diagram. 08

The UNIX Architecture



- The entire UNIX system is supported by a handful of essentially simple and abstract concepts.
- The UNIX architecture has three important agencies:
  1. Division of labor: Kernel and shell
  2. The file and process
  3. The system calls

1. Division of labor: Kernel and shell

- The fertile ideas in the development of UNIX has two agencies – kernel and shell.
- The kernel interacts with the machine's hardware.
- The shell interacts with the user.

a) **The Kernel**

- The core of the operating system - a collection of routines mostly written in C.
- It is loaded into memory when the system is booted and communicates directly with the hardware.
- User programs (the applications) that need to access the hardware use the services of the kernel, which performs the job on the user's behalf.

- These programs access the kernel through a set of functions called system calls.
  - **Functions of kernel:**
    - ✦ It manages the system's memory, schedules processes, decides their priorities and so on.
    - ✦ Process control
    - ✦ Resources management.
    - ✦ The kernel has to do a lot of this work even if no user program is running.
    - ✦ The kernel is also called as the operating system - a programs gateway to the
      computer's resources.

b) The Shell

- Computers don't have any capability of translating commands into action.
- That requires a command interpreter, also called as the shell.
- Shell is actually interface between the user and the kernel.
- Most of the time, there's only one kernel running on the system, there could be several shells running – one for each user logged in.
- The shell accepts commands from user, if require rebuilds a user command, and finally communicates with the kernel to see that the command is executed.
- **Example**:

```
$ echo VTU        Belagavi
  #Shell rebuilds echo command by removing multiple
spaces      VTU Belagavi
```

- **Types of shell** - There are 3 types of Shell:
  1.) Bourne shell ($) ----> Bash ($)
  2.) C (%) -----> tesh (%)
  3.) korn ($)

2. The File and Process

- Two simple entities support the UNIX – the file and the process.
- “Files have places and processes have life”.
- **The File:**
  - ✦ A file is just an array of bytes and can contain virtually anything.
  - ✦ Every file in UNIX is part of the one file structure provided by UNIX.
  - ✦ UNIX considers directories and devices as members of the file system.

- **The Process:**
  - ✦ The process is the name given to the file when it is executed as a program (Process is program under execution).
  - ✦ We can say process is an “time image” of an executable file.

- ✦ We also treat process as a living organism which have parents, children and are born and die.

3. The System Calls

- The UNIX system-comprising the kernel, shell and applications-is written in C.
- Though there are several commands that use functions called system calls to communicate with the kernel.
- All UNIX flavors have one thing in common – they use the same system calls.
- **System Calls and Utilities:**
    - ✦ **A system call is a function that is called from within a program to request a service from the operating system kernel.**
    - ✦ A system call is just what its name implies - a request for the operating system to do something on behalf of the user's program.
    - ✦ The system calls are functions used in the kernel itself. To the programmer, the system call appears as a normal C function call.
    - ✦ **UNIX system calls are used to manage the file system, control processes, and to provide inter-process communication. The UNIX system interface consists of about 80 system calls.**

(b) Discuss the silent features of UNIX operating system. 08

Refer mqp1

(c) What are internal and external commands in UNIX? Explain them with suitable example. 04

External Commands And Internal Commands

1. **External Commands**:
    - If the command (file) has an independence existence in the /bin directory, it is called external command.
    - Existence of the command can be seen as file
    - **Example:** **ls, ps, cat, who**

      $ type ls # ls is an external

    command ls is /bin/ls • External commands are not built into the shell.

    - These are executable present in a separate file.
    - When an external command has to be executed, a new process has to be spawned and the command gets executed.
    - For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.

2. **Internal commands:**
    - When the **shell executes command(file) from its own set of built-in commands that are not stored as separate files in /bin directory**, it is called internal command.
    - Internal commands are something which is built into the shell.

- For the shell built in commands, the execution speed is really high.
- It is because no process needs to be spawned for executing it.
- For example, when using the "cd" command, no process is created.
- The current directory simply gets changed on executing it.
- If the command exists both as an internal and external one**, shell executes internal command only.**
- **Internal commands will have top priority** compared to external command of same name.
- **Example:** **echo, exit, kill, cd**
  $ type echo # echo is an internal command echo is shell built-in

2.(a) Illustrate command structure usage and behavior with respect to absolute and relative pathnames of following commands with suitable examples. i). mkdir ii). rmdir 10

2. mkdir: Making Directories:

✦ Directories are created with mkdir (make directory) command. The command is followed by names of the directories to be created.

✦ **Example:**

```
bash-4.1$ mkdir
vemana bash-4.1$ ls
            1.c    demo.txt     Documents pcd        roopa.txt  vemana
array.c   Desktop              Downloads   Pictures  Templates ccittu.c
directorycommand  Music         Public  Unix bash-4.1$ cd vemana
```

✦ You can create a number of subdirectories with one mkdir command: bash-4.1**$ mkdir cs is ec** bash-4.1**$ ls**

```
cs  ec   is
```

✦ For instance, the following command creates a directory tree: bash-4.1**$ mkdir cs/3cs cs/4cs cs/2cs** bash-4.1**$ cd cs** bash-4.1**$ ls**

```
2cs  3cs  4cs
```

✦ The order of specifying arguments is important. You cannot create subdirectories before creation of parent directory. For instance **following command doesn't work:**

```
bash-4.1$ cd pcd
bash-4.1$   mkdir   prog   prog/prog1
prog/prog2  bash-4.1$  ls       binomial
prog
bash-4.1$ mkdir file/text file/ppt file     mkdir: cannot create directory
`file/text': No such file or directory     mkdir: cannot create directory
`file/ppt': No such file or directory bash-4.1$
```

3. rmdir: Removing A Directory:

✦ The rmdir (remove directory) command removes the directories.

✦ **Example:** bash-4.1**$ rmdir 3cs** bash-4.1**$ ls**

```
2cs  4cs
```

- If 3cs is empty directory then it will be removed form system: bash-4.1**$ rmdir cs rmdir: failed to remove `cs': Directory not empty** ✦ rmdir expect the arguments reverse of mkdir:

```
bash-4.1$ rmdir cs/2cs cs/4cs cs
bash-4.1$ ls
          ec  is
```

- First subdirectories need to be removed from the system then parent.
- Following command works with rmdir.

- System refuses to create a directory due to following reasons:
   i. **User doesn‘t have permission to create directory.** (i.e. directory write protected).
   ii. The **directory already exists. iii.** There may be **ordinary file by that name in the current directory.**

To remove your current directory 1st we should move to parent directory of current directory. **For Example:**

```
bash-4.1$ cd ec bash-4.1$ ls bash-4.1$ rmdir ec      rmdir:
failed to remove `ec': No such file or directory
bash-4.1$ cd .. bash-4.1$ rmdir ec is bash-4.1$ ls bash-4.1$ rmdir
vemana      rmdir: failed to remove `vemana': No such file or
directory
bash-4.1$ cd ..

bash-4.1$ rmdir
vemana bash-4.1$ ls
         1.c   demo.txt  Documents  pcd    roopa.txt  array.c~    Desktop
Downloads Pictures  Templates ccittu.c~  directorycommand  Music Public
unix
```

- The order of specifying arguments is important. You cannot delete parent directories before deletion of subdirectories
- System refuses to delete a directory due to following reasons:
   i. User doesn‘t have permission to delete directory. (i.e. write protected directory).
   ii. The directory doesn‘t exist in system.
   iii. The directory is your present working directory.
- **The dot (.) and double dots (..) notations in relative pathnames:**
   - User can move from working directory /home/kumar/progs/cprogs to home directory /home/kumar using cd command like:

```
$pwd
/home/kumar/progs/cprogs
   $cd /home/kumar
$pwd
/home/kumar
```

   - Navigation becomes easy by using common ancestor.

- **(.) (a single dot)** - This represents the current directory.
- **(..) (two dots)** - This represents the parent directory.
- Assume user is currently placed in /home/kumar/progs/cprogs:

```
$pwd
/home/kumar/progs/cprogs
 $cd ..
$pwd
/home/kumar/progs
```

- This method is compact and easy when ascending the directory hierarchy. The command **cd ..** Translates to this —change your current directory to parent of current directory.
- The relative paths can also be used as:

```
$pwd
    /home/kumar/progs
$cd ../..
$pwd
/home
```

- The following command copies the file prog1.java present in javaprogs, which is present is parent of current directory to current directory.

```
$pwd
/home/kumar/progs/cprogs
$cp   ../javaprogs/prog1.java .
```

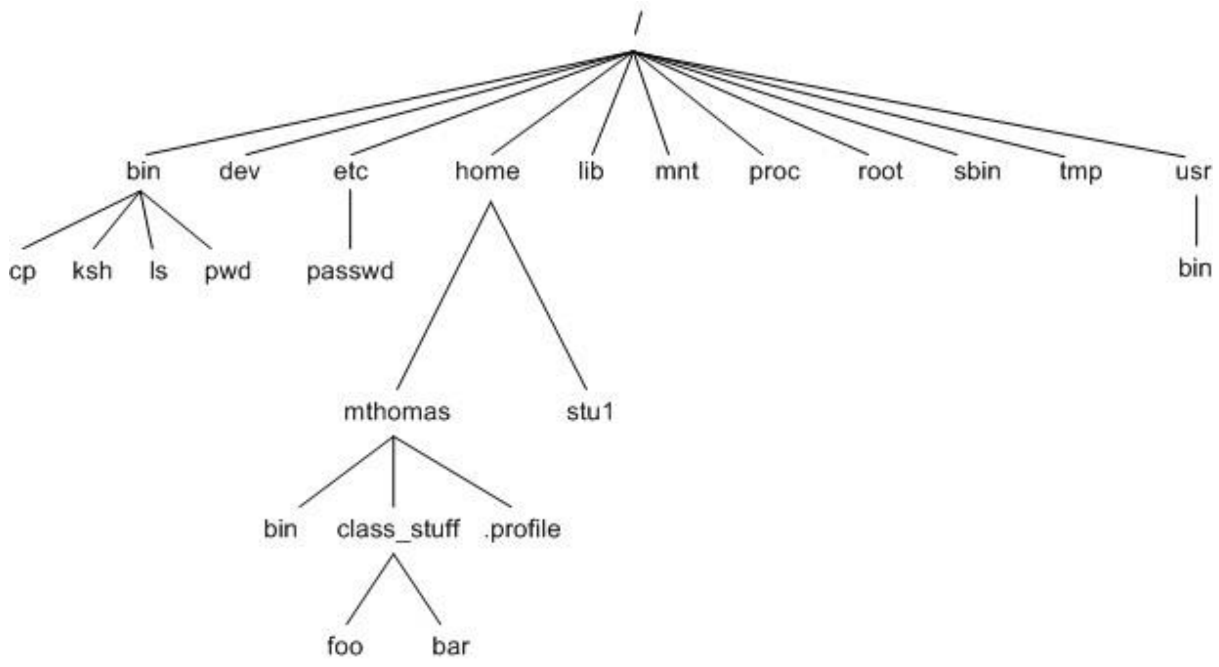
- Now prog1.java is copied to cprogs under progs directory.

(b) Discuss different file types available in UNIX operating system with neat diagram. 8
Refer mqp1

(c) Explain parent-child relationship in UNIX file system. 2

The PARENT-CHILD Relationship

- The files in UNIX are related to one another.
- The file system in UNIX is a collection of all of these files (ordinary, directory and device files) organized in a hierarchical (an inverted tree) structure as shown in below figure.



The feature of UNIX file system is that there is a top, which serves as the reference point for all files.

- This top is called root and is represented by a / (Front slash).
- The root is actually a directory.
- The root directory (/) has a number of subdirectories under it.
- The subdirectories in turn have more subdirectories and other files under them.
- Every file apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root.

3.(a) Which command is used for listing file attributes? Briefly describe the significance of each field of the output 08
Mqp1

(b) Current file permission of a regular file “unix” are rw--w---x. Illustrate both relative and absolute methods required to change permission to the following: i). –wxrwxr-x ii). r-------x iii). –w-r-x-w- iv). --xrw-r—08

(c) Explain wild cards with examples and its various types. 04
Mqp1 (4c **)**

4.(a) Define is shell programming? Write a shell program to create a simple calculator which can perform basic arithmetic operations? 10

Shell programming

- A shell script contains a list of commands which have to be executed regularly.
- Shell script is also known as shell program.
- The user can execute the shell script itself to execute commands in it.
- A shell script runs in interpretive mode. i.e. the entire script is compiled internally in memory and then executed.

- Hence, shell scripts run slower than the high-level language programs.
- ".sh" is used as an extension for shell scripts.
  - Program : To perform simple calculator

Check notes

(b) Explain grep command with all options. 06

Mqp1 (3c)

(c) Write the output for following command. i) grep ^[^3] abcd

ii) grep -v “please delete” filename.txt | wc

iii)ls | wc-l >fcount iv)cat *.c | wc –c 04

5 (a) Describe general unix file API’s with syntax and explain the each field in detail 10

(b) Explain file and record locking in detail. 06

File and Record Locking

Multiple processes performs read and write operation on the same file concurrently

- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.
- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as “**Exclusive lock**”.
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as “**shared lock** “.

- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.

(c) List the number of ways a process can terminate? 04

PROCESS TERMINATION:

There are eight ways for a process to terminate.

1. **Normal termination occurs in five ways:**

   Return from `main`

   Calling `exit`

   Calling `_exit` or `_Exit`

   Return of the last thread from its start routine

   Calling `pthread_exit` from the last thread

2. **Abnormal termination occurs in three ways:**

   Calling `abort`

   Receipt of a signal

   Response of the last thread to a cancellation request

6.(a) Describe the mechanism of process creation with a neat diagram 08

(b) Explain the following commands i)fork ii)vfork iii)exit 06

An existing process can create a new one by calling the `fork` function.

Prototype: *#include<unistdih>*

pid_t fork(void);

***Returns:** 0 in child, process ID of child in parent, 1 on error*

- The new process created by `fork` is called the child process. This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

Vfork

- The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ.

- The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- The vfork function creates the new process, just like fork, **without copying the address space of the parent into the child,** as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.
- Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent.
- Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.
- The prototype of vfork is

#include < unistd. h> #include
<sys/types. h>

pid_t vfork(void);

***Returns:** 0 in child, process ID of child in parent, -1 on error*

<u>exit:</u>

- exit function terminates the process normally, performs certain cleanup processing and then returns to the kernel.
- the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. This causes all buffered output data to be flushed (written to the file).

(c) Define race condition and polling? How to overcome these conditions 06

A **race condition** occurs when multiple processes are competing for the same system resource(s). The outcome depends on the order in which the processes run.

➢ Problems due to race conditions are hard to debug.

➢ We cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

<u>POLLING CAN BE USED TO AVOID RACE CONDITION:</u>

- A process that wants to wait for a child to terminate must call one of the `wait` functions.

- If a process wants to wait for its parent to terminate, a loop of the following form could be used:

```
while (getppid() != 1)
        sleep(1);
```

  This type of loop is called polling

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used, and various forms of interprocess communication (IPC) can also be used.

- It is required that each process tells the other when it has finished its initial set of operations, and that each waits for the other to complete, before heading off on its own.
- To achieve this, five routines `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` are used. these can be either macros or functions

7.(a) Illustrate IPC with all its methods. 08

(b) Explain pipes with all its advantages and limitations? 06

PIPES

- Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.
- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Returns: 0 if OK, 1 on
error.

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

- Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.
- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

(c) Briefly explain the rules who can change group ID’s 06

8.(a) Demonstrate the Client and Server interaction with neat diagram. 10

Mqp1 (7c)

(b) What are Interpreter Files? Give the difference between Interpreter Files and Interpreter. 06

(c) What are semaphores? Mention its two types. 04

A semaphore is a counter used to provide access to a shared data object for multiple processes.

//types

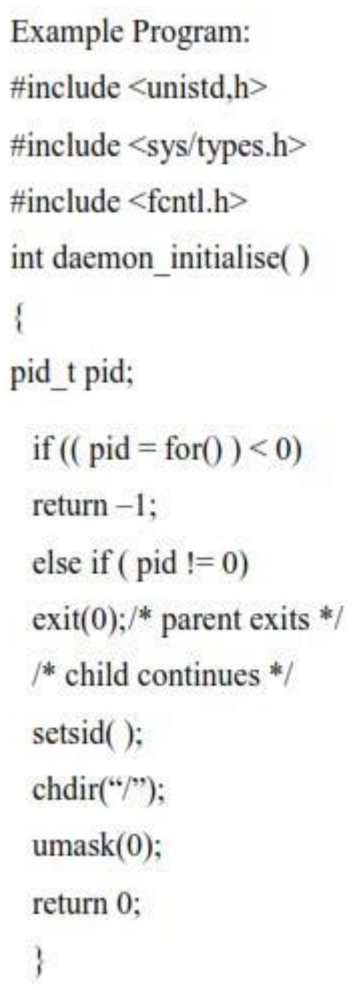
9.(a)What are daemon processes? Enlist their characteristics. Also write a program to transform a normal user process into a daemon process. Explain every step in the program. 10

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.

DAEMON CHARACTERISTICS

The characteristics of daemons are:

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



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

//steps

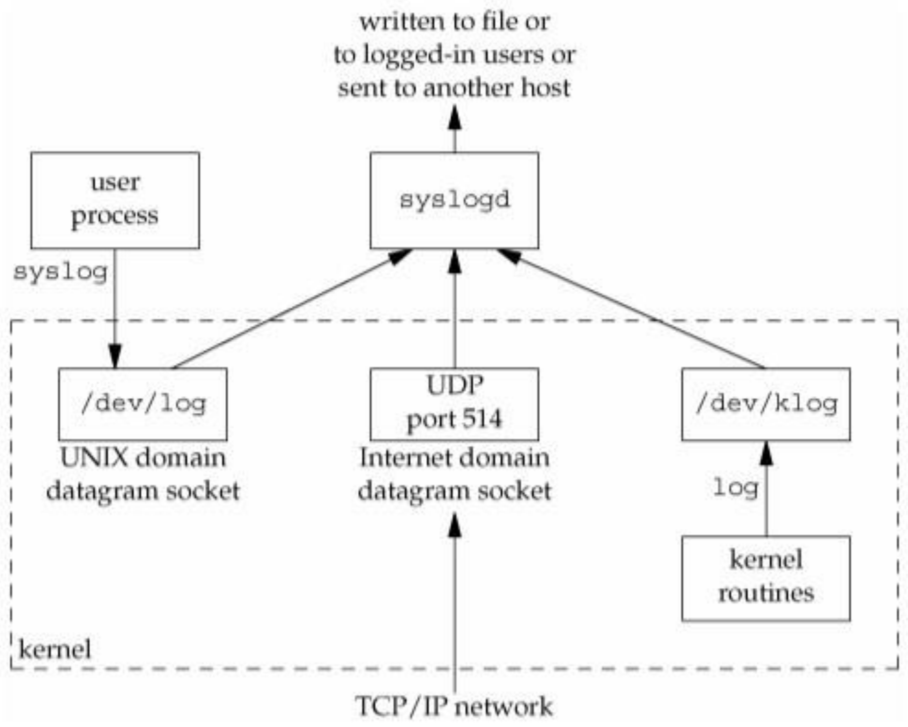
(b) Explain the kill() API and alarm() API? 10

10.(a) What is error logging? With a neat block schematic discuss the error login facility in BSD. 10

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. A central daemon error-logging facility is required.

Figure 13.2. The BSD `syslog` facility



There are three ways to generate log messages:

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

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

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

char *format, ...); void closelog(void); int
setlogmask(int maskpri);

(b) Explain the terms i)signal ii)signal mask 10

SIGNAL:

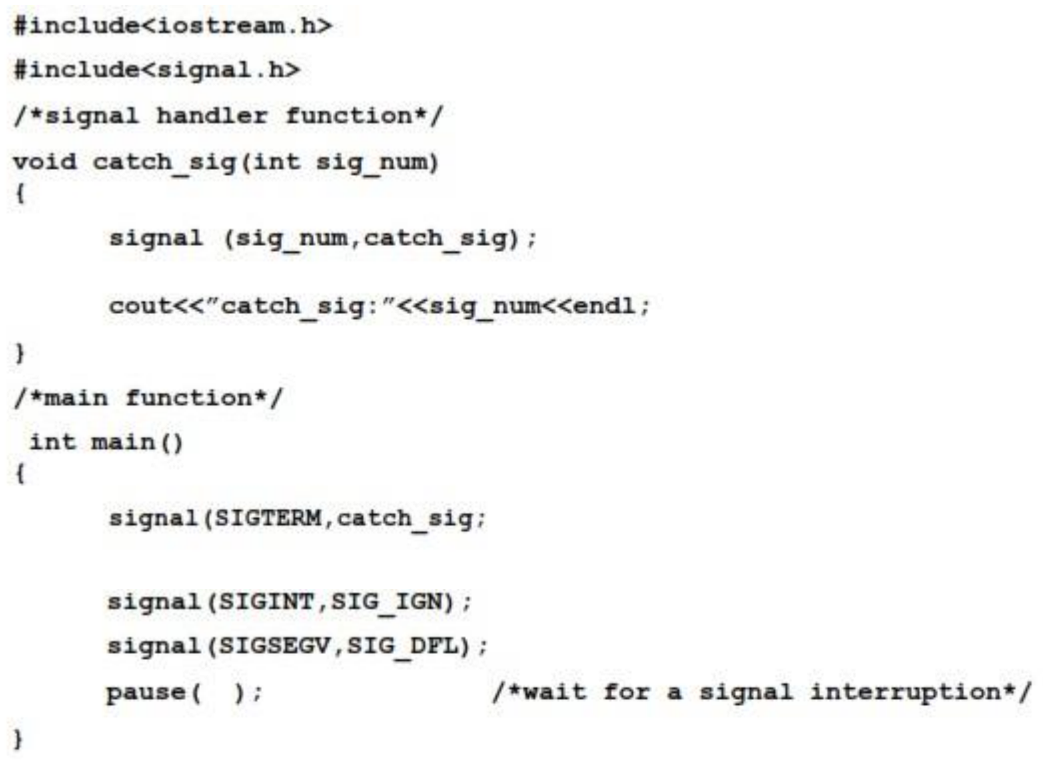
The function prototype of the signal API is:

#include <signal.h>

void (*signal(int sig_no, void (*handler)(int)))(int);

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.

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



```
#include<iostream.h>
#include<signal.h>
/*signal handler function*/
void catch_sig(int sig_num)
{
      signal (sig_num,catch_sig);

      cout<<"catch_sig:"<<sig_num<<endl;
}
/*main function*/
 int main()
{
      signal(SIGTERM,catch_sig;

      signal(SIGINT,SIG_IGN);
      signal(SIGSEGV,SIG_DFL);
      pause(  );               /*wait for a signal interruption*/
}
```

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

The SIG_DFL specifies to accept the default action of a signal.

SIGNAL MASK:

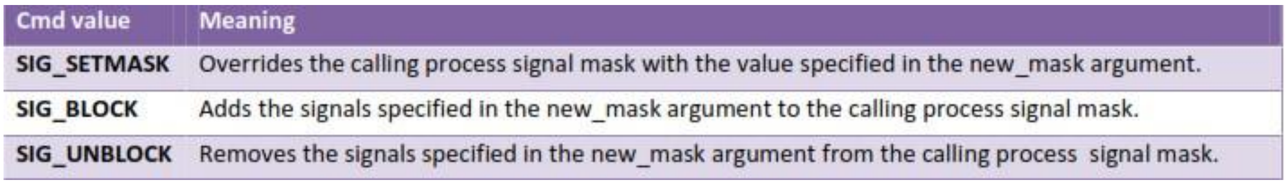
- A process initially inherits the parent’s signal mask when it is created, but any pending signals for the parent process are not passed on.
- A process may query or set its signal mask via the sigprocmask API:

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

    *old_mask);
```

Returns: 0 if OK, 1 on error

- The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, 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.

- the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered.
- If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.

CHECK THIS FOR SETJUMP AND LONGJUMP.

With an example explain the use of setjmp and longjmp functions.

ANS:

In C, we can't `goto` a label that's in another function. Instead, we must use the `setjmp` and `longjmp` functions to perform this type of branching.

these two functions are useful for handling error conditions that occur in a deeply nested function call.

Prototypes:

#include <setjmp.h>

```
int setjmp(jmp_buf env);
```

Returns: 0 if called directly, nonzero if returning from a call to

```
longjmp  void longjmp(jmp_buf env, int val);
```

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly.

In the call to setjmp, the argument env is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp.

Normally, the env variable is a global variable, since we'll need to reference it from another function.

When we encounter an error say, in the add function, we call longjmp with two arguments. The first is the same env that we used in a call to setjmp, and the second, val, is a nonzero value that becomes the return value from setjmp. The reason for the second argument is to allow us to have more than one longjmp for each setjmp.

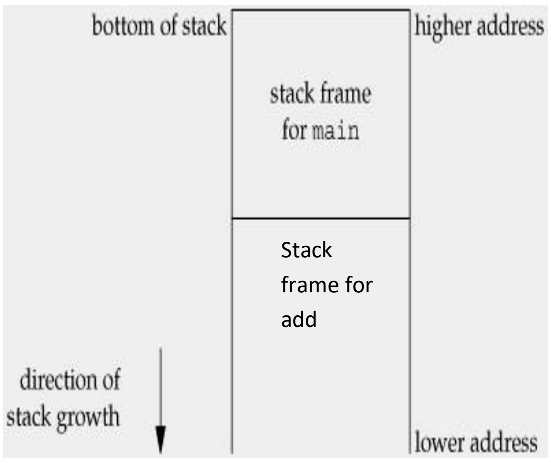
Example of setjmp and longjmp

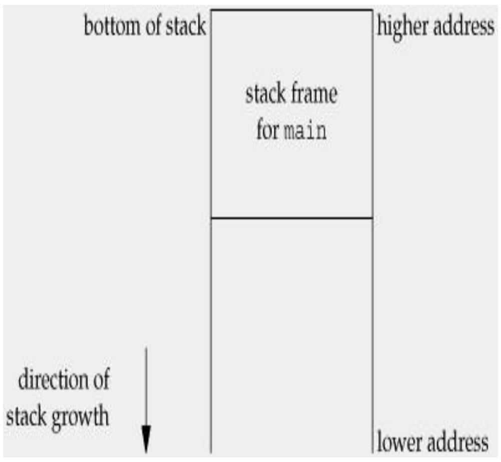
```
#include
"apue.h"
#include
<setjmp.h>

jmp_buf env;
int
main(void)

{
     int i;
     if
((i=setjmp(en
v)) != 0)
printf("error
");
add(a,b);
exit(0);
}
```

```
      ... Void
     add(int a,int
     b)
     {
         int   sum;
     sum=a+b;
     if(sum     <0)
     longjump<(env
     ,1)
}
```





tack frame after add function has been called   stack frame after longjmp has been called

When `main` is executed, we call `setjmp`, which records whatever information it needs to in the variable `env` and returns 0. We then call `add` and assume that an error of some form is detected. `longjmp` causes the stack to be "unwound" back to the `main` function, throwing away the stack frames `add` .Calling `longjmp` causes the `setjmp` in `main` to return, but this time it returns with a value of 1 (the second argument for `longjmp`).