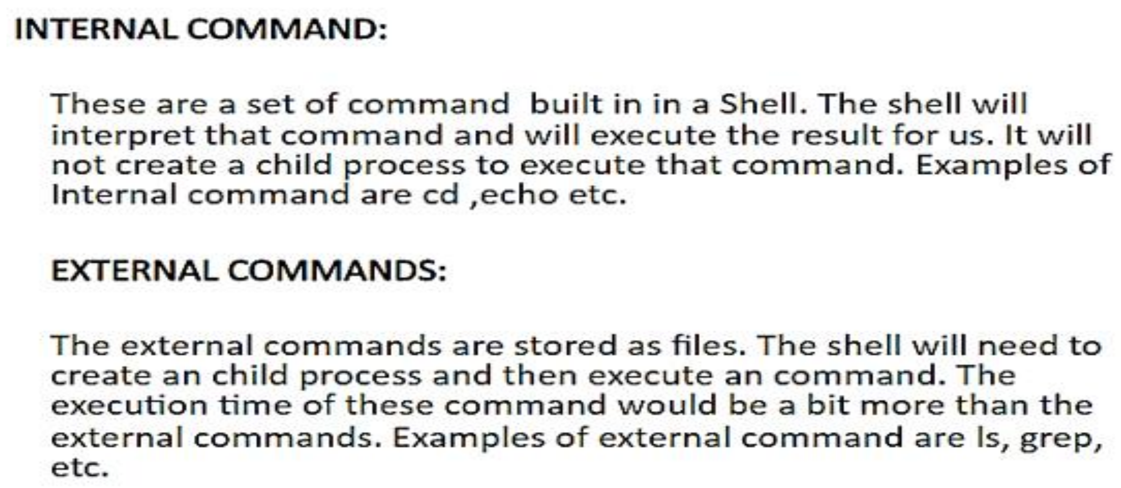
USN											

VTU Examination –March 2023
Solution

Sub:	**UNIX Programming**	Sub Code:	18CS56	Branch:	ISE
Exam Date:	09/03/2023 Duration: 3 Hrs	Max Marks: 100	Sem	**V**	

	Answer any FIVE FULL Questions	MARKS	CO
1 (a)	**Compare internal commands and external commends, files and processes.**	[06]	CO1


INTERNAL COMMAND:

These are a set of command built in in a Shell. The shell will interpret that command and will execute the result for us. It will not create a child process to execute that command. Examples of Internal command are cd ,echo etc.

EXTERNAL COMMANDS:

The external commands are stored as files. The shell will need to create an child process and then execute an command. The execution time of these command would be a bit more than the external commands. Examples of external command are ls, grep, etc.

- File have spaces on disk (inactive state) and Processes have life in system.
- Program under execution is a process.
- A File is a container for storing information.
- All file attributes are kept in a separate area of the hard disk,accessible only by the kernel.

(b)	**Explain all the features of UNIX operating system.**	[09]	CO1

1. Multiuser system

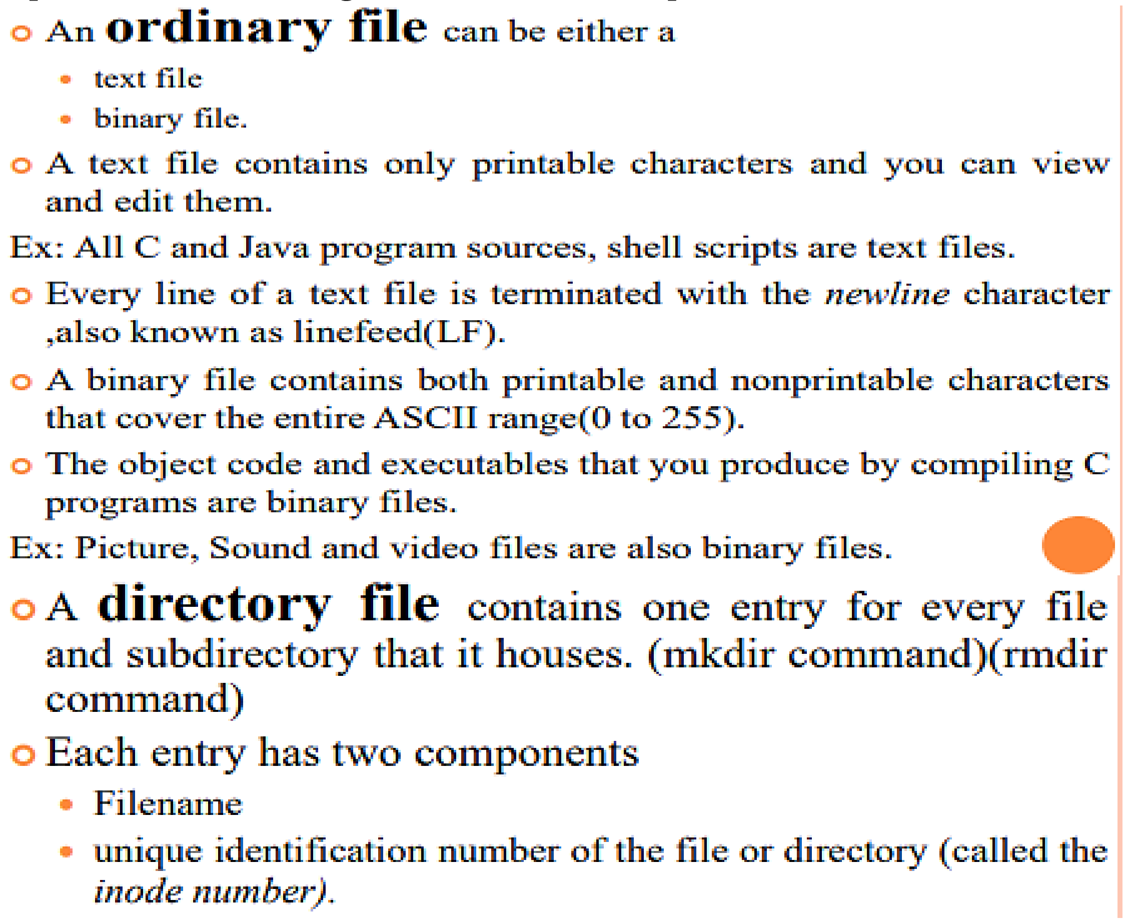
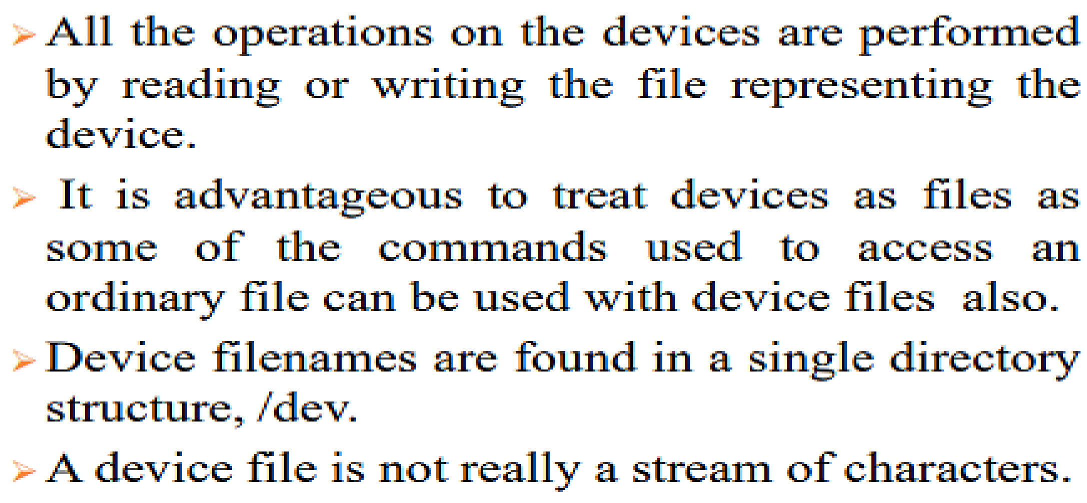
From the fundamental point of view UNIX is a multiprogramming system ; this can happen in 2 ways
- Multiple users can run a system
- A single user can also run Multiple jobs

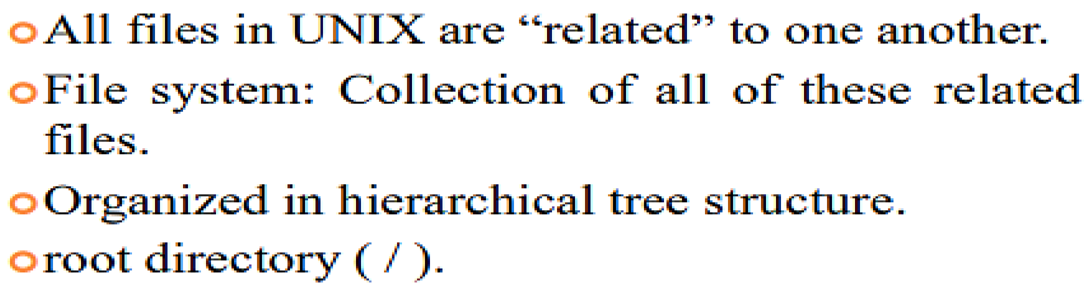
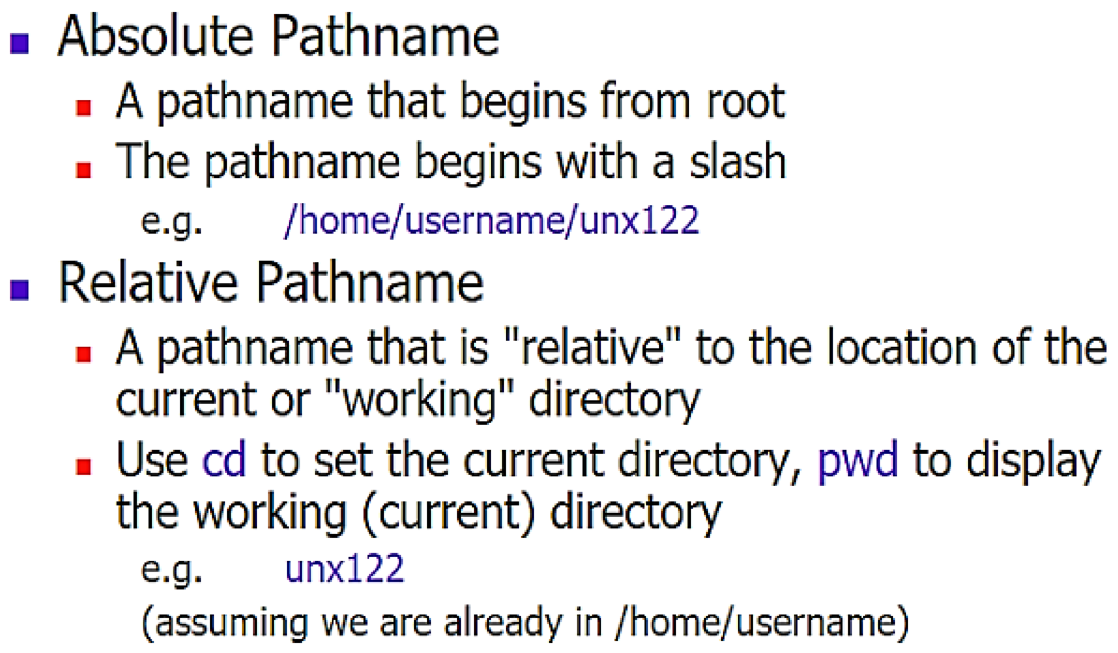
2 .Multitasking System

In a Multitasking environment, a user sees one job running in the foreground & the rest are running in the background.

3. The Building Block Approach

The designers never attempted to pack too many features into a few tools. Instead they developed a few hundred commands each of which performed one simple job only.
E.g: ls | wc
4. UNIX Tool Kit
5. Pattern Matching e.g : ls chap* , ls chap+ , ls chap?
6. Programming facility
7. Documentation: - man command, which remains the most important reference for commands and their configuration files.
Internet, FAQ in net, articles published in magazines & Journals and lecturer notes available by universities on their website.

(c)	**Write the output for the following commands:** 1) cal 10 2021 Prints the Calendar of October 2021. 2) date +”%D%T" Prints present date in the format of mm/dd/yy with time in 24hrs format. 3) type echo Prints the output as echo is a shell builtin. 4) passwd changes the old password to new password. 5) who prints information about currently logged in user on to system.	[05]	CO1
	OR		
2 (a)	**Explain the different categories of files with examples.**  An **ordinary file** can be either a • text file • binary file. A text file contains only printable characters and you can view and edit them. Ex: All C and Java program sources, shell scripts are text files. Every line of a text file is terminated with the *newline* character ,also known as linefeed(LF). A binary file contains both printable and nonprintable characters that cover the entire ASCII range(0 to 255). The object code and executables that you produce by compiling C programs are binary files. Ex: Picture, Sound and video files are also binary files. A **directory file** contains one entry for every file and subdirectory that it houses. (mkdir command)(rmdir command) Each entry has two components • Filename • unique identification number of the file or directory (called the *inode number).*  ➢ All the operations on the devices are performed by reading or writing the file representing the device. ➢ It is advantageous to treat devices as files as some of the commands used to access an ordinary file can be used with device files also. ➢ Device filenames are found in a single directory structure, /dev. ➢ A device file is not really a stream of characters.	[06]	CO1

(b)	**Describe the parent child relationship in UNIX file system and differentiate absolute pathnames with relative path names.**  ○ All files in UNIX are "related" to one another. ○ File system: Collection of all of these related files. ○ Organized in hierarchical tree structure. ○ root directory (/).  ■ Absolute Pathname ■ A pathname that begins from root ■ The pathname begins with a slash e.g. /home/username/unx122 ■ Relative Pathname ■ A pathname that is "relative" to the location of the current or "working" directory ■ Use cd to set the current directory, pwd to display the working (current) directory e.g. unx122 (assuming we are already in /home/username)	[06]	CO1
(c)	**Write the description for the following commands:** i) mkdir college college/ISE college/CSE creates a directory college and two sub-directories as ISE and CSE in College directory. ii)mV f1.C f2.C f3.C cprogs rename the files f1.c f2.c f3.c to cprogs iii)if my pwd is /home/ravi/progs then Cd ../.. prompt will be in home directory iv)ls –l | wc-l lists all the files with 7 attributes and prints the line count of each file. v) cp f1 f2 f3 files copies files f1 f2 f3 to files vi) rm –i chap1 remove the file chap1 in interactive manner. vii) cat >> test.txt creating the output redirection contents into test.txt file. viii) rmdir college/ISE removes the sub directory ISE from college directory.	[08]	CO1

3 (a)	**Explain all the options of ls commands with examples.**
	o list of all filenames in the current directory.
	o It displays the files by using ASCII collating sequence
	o Syntax: ls [options] [arguments]
	o Ex:
	`$ ls -l`
	`total 2`
	`-rw-r--r-- 1 Administrator None  16 Sep 23 11:00 geek.txt`
	`-rw-r--r-- 1 Administrator None 110 Sep 25 14:50 input.txt`
	1.File Type and Permissions : FIRST Column
	➢It indicates the type and permissions associated with the each file .
	➢The first character represents the " file type "
	' – ' represents the ordinary file
	' d 'represents the directory file
	' a / b/ c 'represents the device file
	➢ Remaining character in first column represents the read , write and execute permission to the owner(USER) , group and others .
	2.LINKS : SECOND COLUMN
	➢This indicates the number of links associated with a file.
	➢This is actually the number of filenames maintained by the system for the single copy of a file on disk .
	3. OWNERSHIP : THIRD column
	➢ when we create the file , automatically we are the owner of this file.
	4.Group ownership:4th column represents the group owner of the file .
	5.File Size: 5th column is the amount of data it contains (i.e the total number of characters it has stored in it).
	6.Last Modification Time : 6th 7th 8th columns shows the last modification time of the file.
	A file is said to be modified only if its content get changed , if we change the ownership or permission the modification time will remain unchanged .
	7.File name: last column indicates the filenames arranged in ASCII collating sequence .
		[06]	CO2

(b) Consider a file test.txt with default permissions as -rw-r--r--, grant execute permission to owner, write and execute permission to group members and execute permission to others using both relative and absolute approaches. [04] CO2

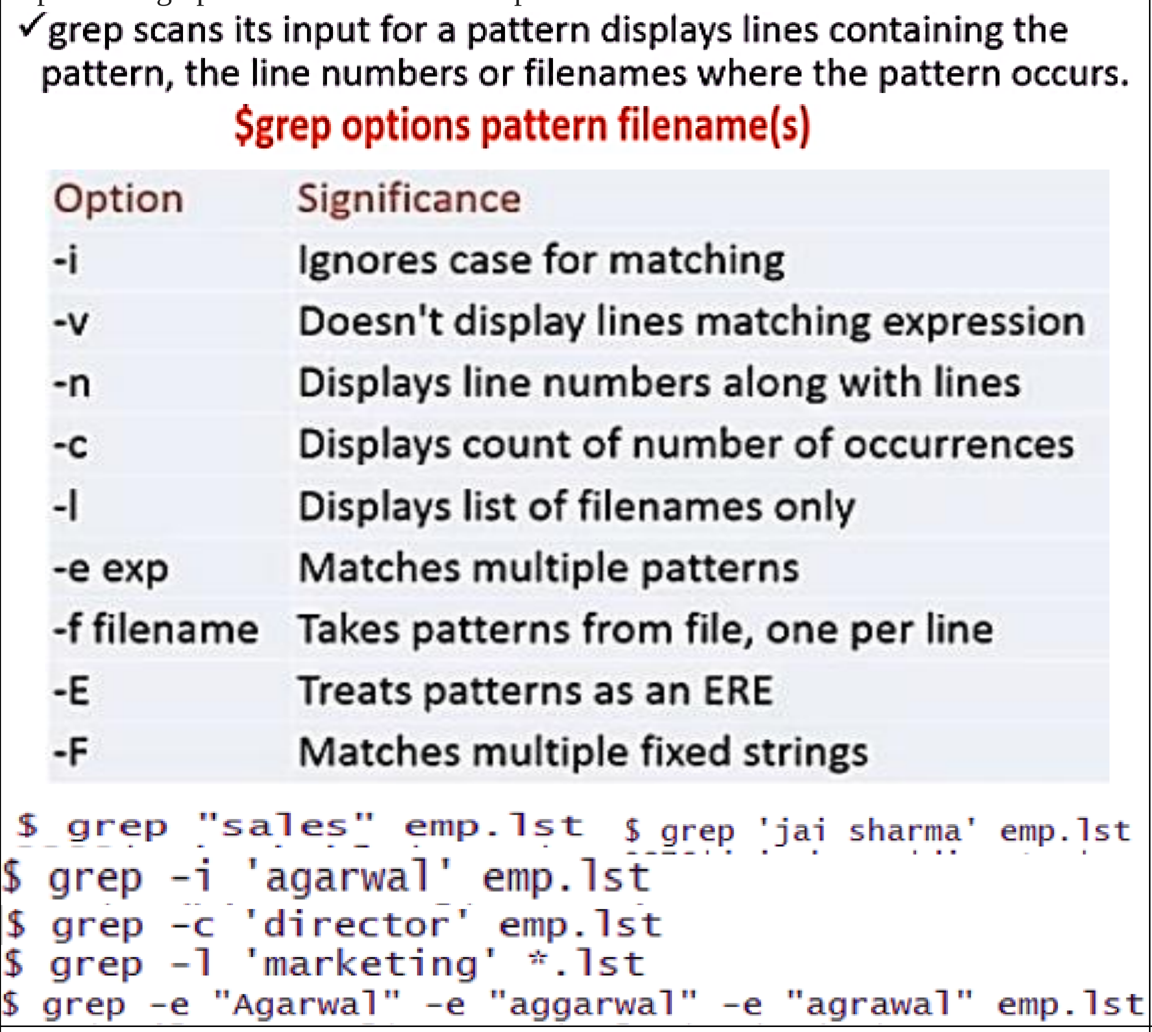
Relative :$chmod u+x g+wx o+x test.txt

Absolute :$chmod 751 test.txt

(c) **Write the output for the following commands.** [05] CO2

1) cp ???? progs
   copies to progs directory all files with 4 character names.
2) rm ‘chap*’
   removes all files of chap
3) mV *. [!C][!P][!P] progs
   moves all files to progs except with the extension .cpp
4) cat *.txt | wc -C
   prints the content of all the files with the extension .txt along with the character count of files.
5) cp chap\ [0-1\]
   copies the files chap contents except with digit 0 to 1.

(d) Explain the grep command with all its options. [05] CO2



✓grep scans its input for a pattern displays lines containing the pattern, the line numbers or filenames where the pattern occurs.

$grep options pattern filename(s)

Option	Significance
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e exp	Matches multiple patterns
-f filename	Takes patterns from file, one per line
-E	Treats patterns as an ERE
-F	Matches multiple fixed strings

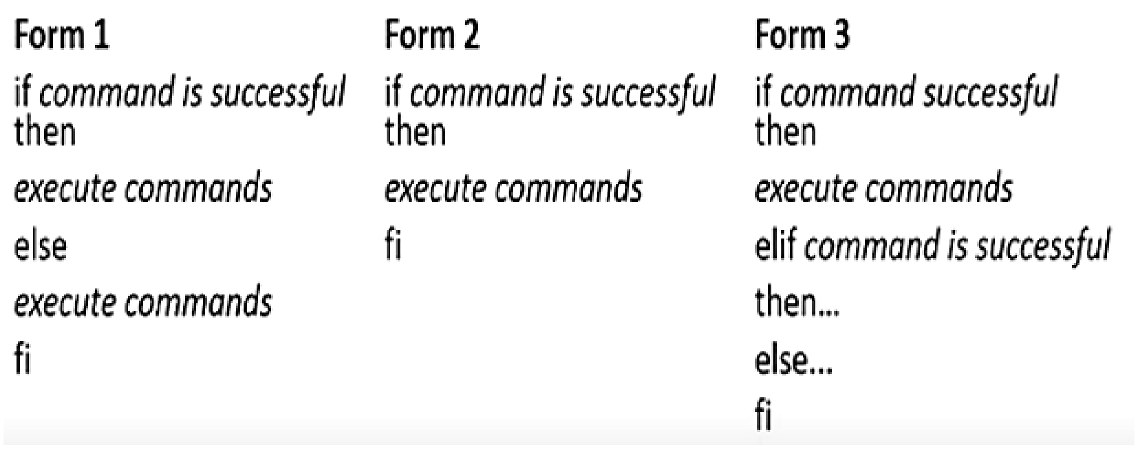
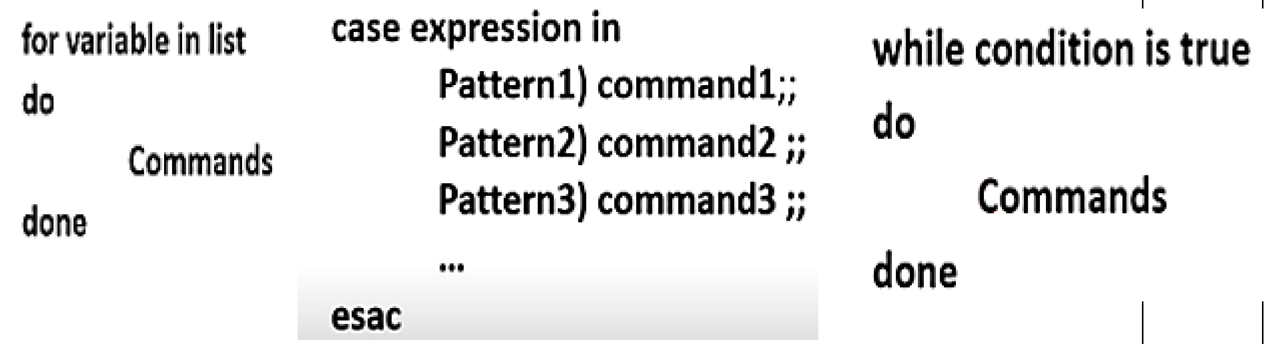
```
$ grep "sales" emp.lst   $ grep 'jai sharma' emp.lst
$ grep -i 'agarwal' emp.lst
$ grep -c 'director' emp.lst
$ grep -l 'marketing' *.lst
$ grep -e "Agarwal" -e "aggarwal" -e "agrawal" emp.lst
```

OR

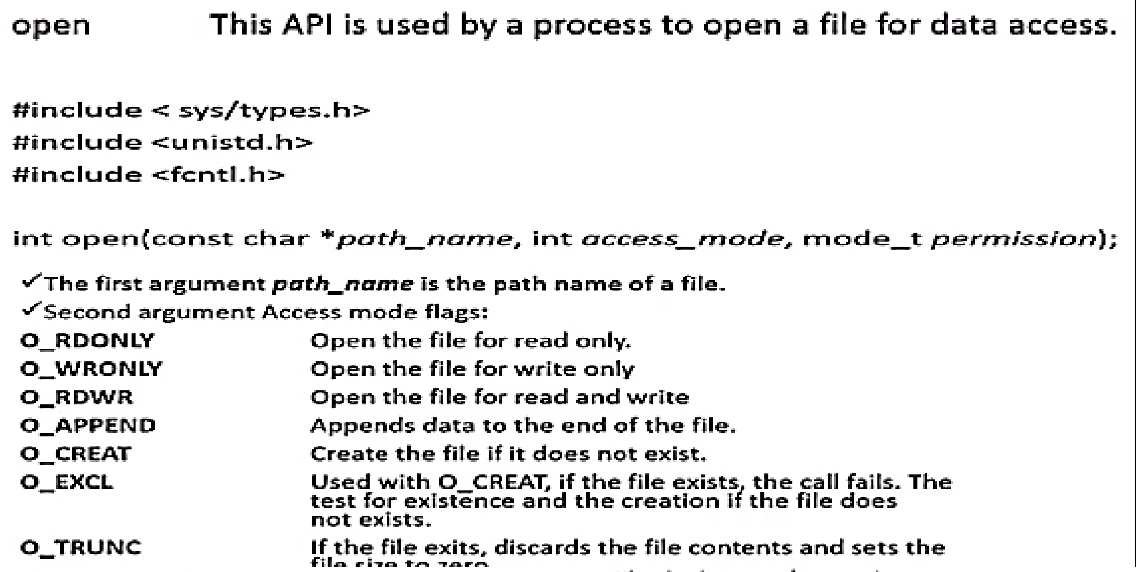
4 (a) Write a program to read pattern and filename from the user and search the pattern in the given file. [05] CO2

```
#!/bin/bash
read -p "Enter file name : " filename
```

	while read line
	do
	echo $line
	done < $filename		

(b)	Write the output for the following commands. i) grep "Anil" std.lst || echo "pattern not found” searches for pattern Anil from std.lst otherwise prints as pattern not found ii) test $x -gt $y Compares two strings x is greater than y and tests for condition. iii) [-Z $stg] test to check whether a string is empty. iv) [-r $file] checks if the file is readable. v) [!-n $stg] Checks if the given string stg operand size is not non-zero; if it is nonzero length, then it returns true otherwise false	[05]	CO2
(c)	Explain all the looping statements with syntax.  Form 1 if *command is successful* then *execute commands* else *execute commands* fi Form 2 if *command is successful* then *execute commands* fi Form 3 if *command successful* then *execute commands* elif *command is successful* then... else... fi  for variable in list do Commands done case expression in Pattern1) command1;; Pattern2) command2 ;; Pattern3) command3 ;; ... esac while condition is true do Commands done	[06]	CO2
(d)	Write a shell script to read multiple patterns from the command line and search these patterns in the given file which is also read from command line by using shift command. [Ex. Command line arguments as below #>script.sh pat1 pat2 pat3, pat4 pat5]. #!/bin/sh echo "Script Name: $0" echo "First Parameter of the script is $1" echo "The second Parameter is $2" echo "The complete list of arguments is $@" echo "Total Number of Parameters: $#" echo "The process ID is $$" echo "Exit code for the script: $?"	[04]	CO2

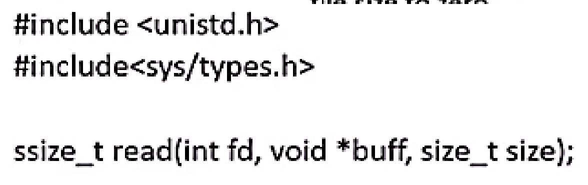
5 (a)	**Explain the General File API's open(), read(), write(), lseek() with their prototype.**	[10]	CO3

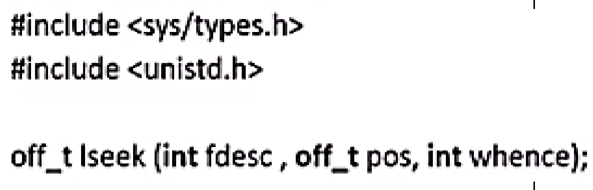

open This API is used by a process to open a file for data access.

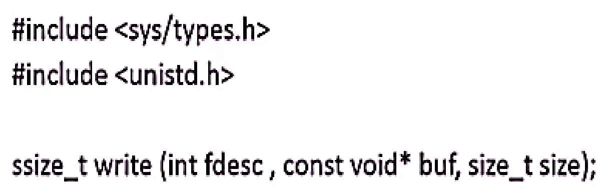
```
#include < sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int open(const char *path_name, int access_mode, mode_t permission);
```

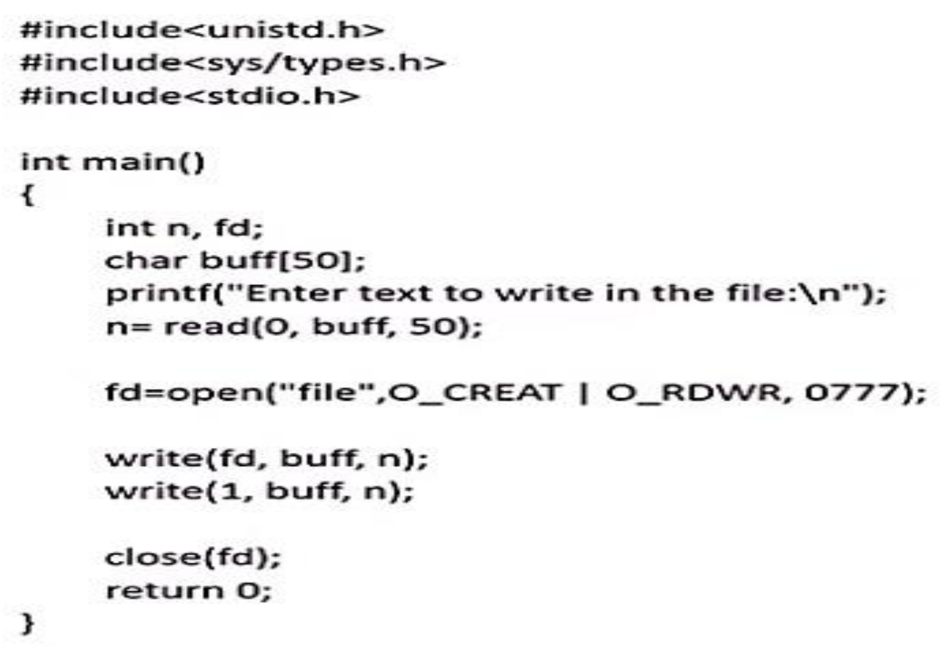
✓The first argument *path_name* is the path name of a file.
✓Second argument Access mode flags:

O_RDONLY	Open the file for read only.
O_WRONLY	Open the file for write only
O_RDWR	Open the file for read and write
O_APPEND	Appends data to the end of the file.
O_CREAT	Create the file if it does not exist.
O_EXCL	Used with O_CREAT, if the file exists, the call fails. The test for existence and the creation if the file does not exists.
O_TRUNC	If the file exits, discards the file contents and sets the


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

ssize_t read(int fd, void *buff, size_t size);
```


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

off_t lseek (int fdesc , off_t pos, int whence);
```


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

ssize_t write (int fdesc , const void* buf, size_t size);
```


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

int main()
{
        int n, fd;
        char buff[50];
        printf("Enter text to write in the file:\n");
        n= read(0, buff, 50);

        fd=open("file",O_CREAT | O_RDWR, 0777);

        write(fd, buff, n);
        write(1, buff, n);

        close(fd);
        return 0;
}
```

(b)	**Describe the memory layout of a C program with a diagram and explain memory allocation API's with their prototypes.**		

A C program has been composed of the following pieces:

[10] CO3

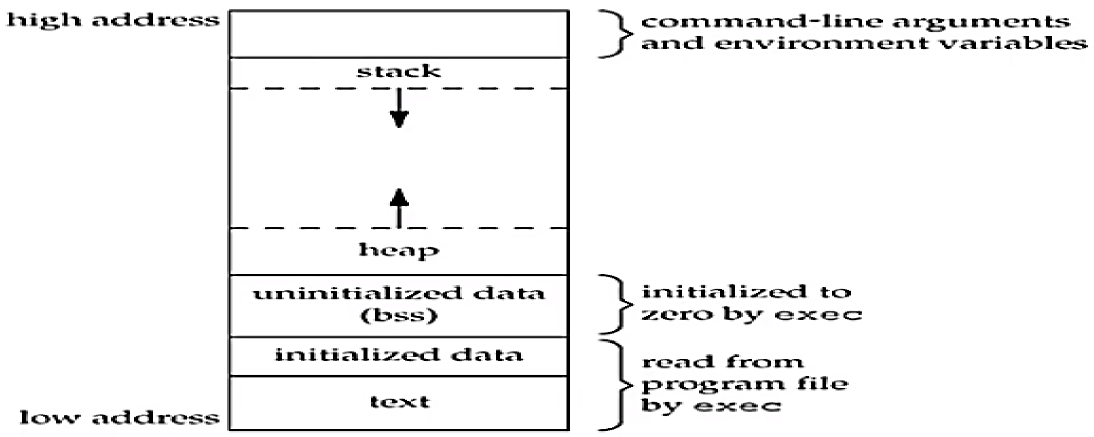
- **Text segment**: The machine instructions that the CPU executes.
- **Initialized data segment**: usually called simply the data segment, containing variables that are specifically initialized in the program.

  For example, the C declaration

  ```
  int maxcount = 99;
  ```
- **Uninitialized data segment**: Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing.

  For example, the C declaration

  ```
  long sum[1000];
  ```
- **Stack**: where automatic variables are stored, along with information that is saved each time a function is called
- **Heap**: where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.



high address
stack
heap
uninitialized data (bss)
initialized data
text
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

ISO C specifies three functions for memory allocation:

1. **malloc**: Which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. **calloc**: Which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
3. **realloc**: Which increases or decreases the size of a previously allocated area.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

All three return: non-null pointer if OK, NULL on error

```
void free(void *ptr);
```

	OR		
6 (a)	**Explain setimp and longimp, getrlimit and setrlimit function with examples.**	[10]	CO3

```
#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);
```

The env variable(the first argument) records the necessary information needed to continue execution.

The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

```
#include<unistd.h>
#include <setjmp.h>
#define TOK_ADD 5
jmp_buf jmpbuffer;
 int main(void)
{
char line[MAXLINE];
if (setjmp(jmpbuffer) != 0)
        printf("error");
while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line); exit(0);
 }
```

```
void cmd_add(void)
{
int token;
token = get_token();
if (token < 0)
        longjmp(jmpbuffer, 1);
}
```

✓Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
```

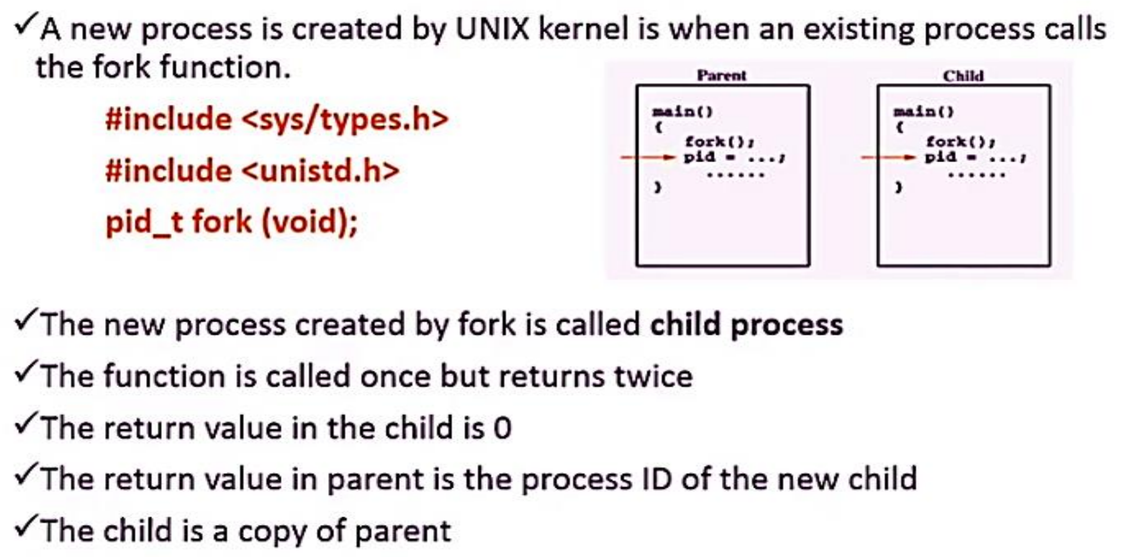
Both return: 0 if OK, nonzero on error

```
struct rlimit
{
        rlim_t rlim_cur; /* soft limit: current limit */
        rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

Resource argument takes one of the following values:

1. RLIMIT_CORE: The maximum size in bytes of a core file.
2. RLIMIT_CPU: The maximum amount of CPU time in seconds.
3. RLIMIT_DATA: The maximum size in bytes of the data segment.
4. RLIMIT_NOFILE: The maximum number of files per process.

(b) Describe how the process is created by using fork() and vfork(). List out the inherited from the parent when the child process is created? [10] CO3

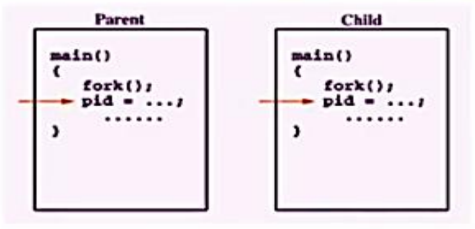


✓A new process is created by UNIX kernel is when an existing process calls the fork function.

#include <sys/types.h>

#include <unistd.h>

pid_t fork (void);



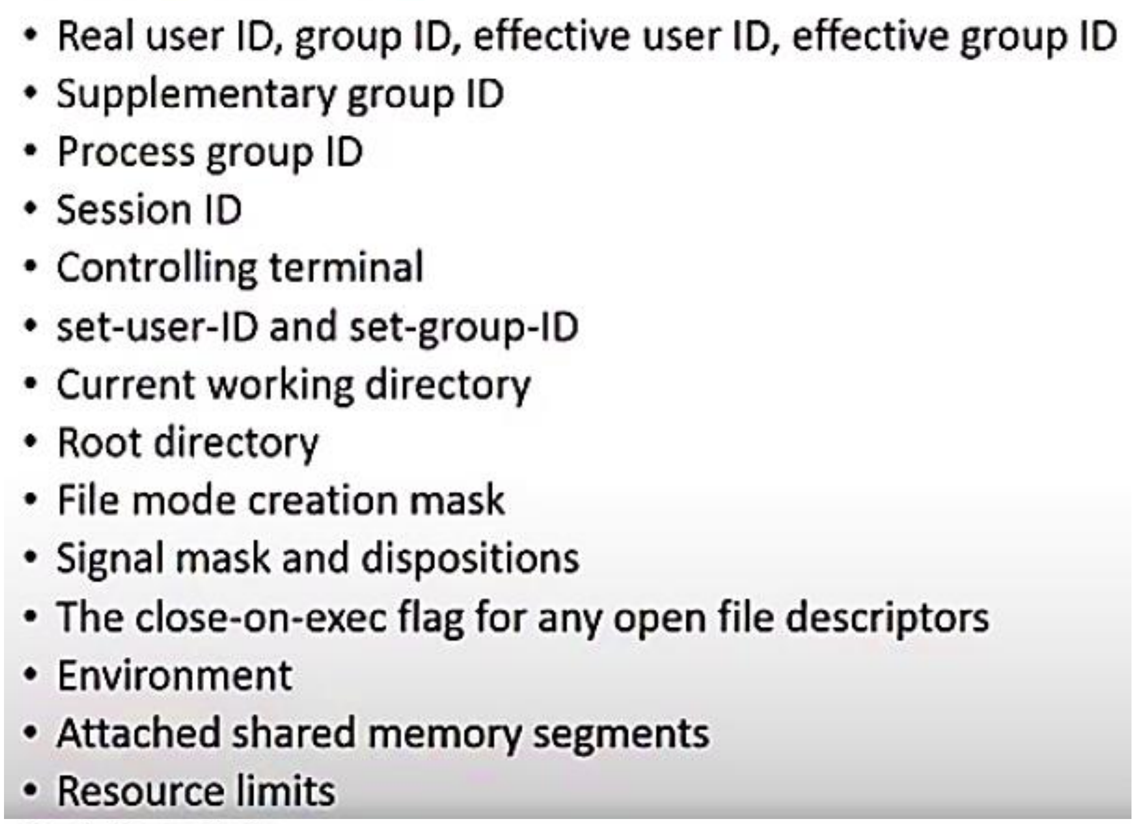
✓The new process created by fork is called **child process**

✓The function is called once but returns twice

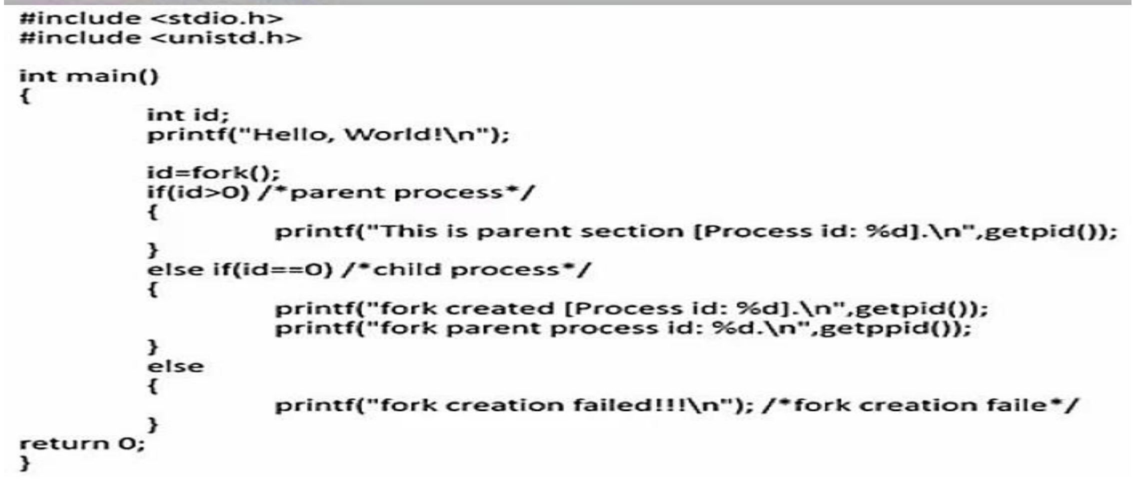
✓The return value in the child is 0

✓The return value in parent is the process ID of the new child

✓The child is a copy of parent



- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Resource limits



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

int main()
{
        int id;
        printf("Hello, World!\n");

        id=fork();
        if(id>0) /*parent process*/
        {
                printf("This is parent section [Process id: %d].\n",getpid());
        }
        else if(id==0) /*child process*/
        {
                printf("fork created [Process id: %d].\n",getpid());
                printf("fork parent process id: %d.\n",getppid());
        }
        else
        {
                printf("fork creation failed!!!\n"); /*fork creation faile*/
        }
return 0;
}
```

Creates new process and block the parent.

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork (void);
```

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

int main()
{
            printf("Before vfork\n");
            vfork();
            printf("After vfork\n");

            return 0;
}
```

7 (a) Explain the implementation of system function using fork(), exec(), wait() API's. [10] CO4

```
#include <stdlib.h>

int system(const char *cmdstring);
```

- ➢ If cmdstring is a null pointer, system returns nonzero only if a command processor is available.
- ➢ System is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the fork fails or waitpid returns an error other than EINTR, system returns −1 with errno set to indicate the error.
2. If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit.
3. Otherwise, all three functions—fork, exec, and waitpid succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid.

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

(b)	Define pipes, write a program to send data from parent to child, using pipe API and also list its limitations.	[10]	CO4

- 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]);

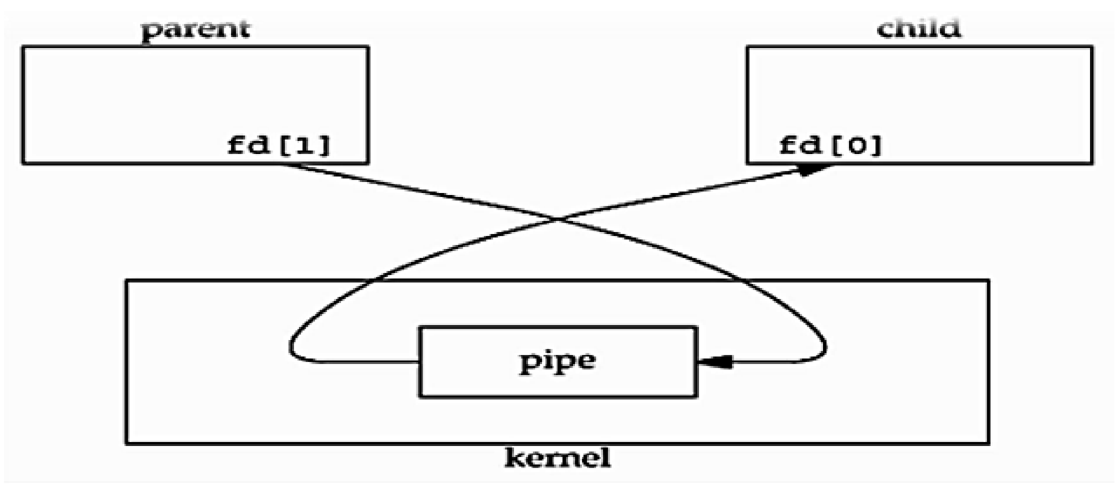
  Returns: 0 if OK, 1 on error.
- Two file descriptors are returned through the filedes argument:

• filedes[0] is open for reading and

• filedes[1] is open for writing.

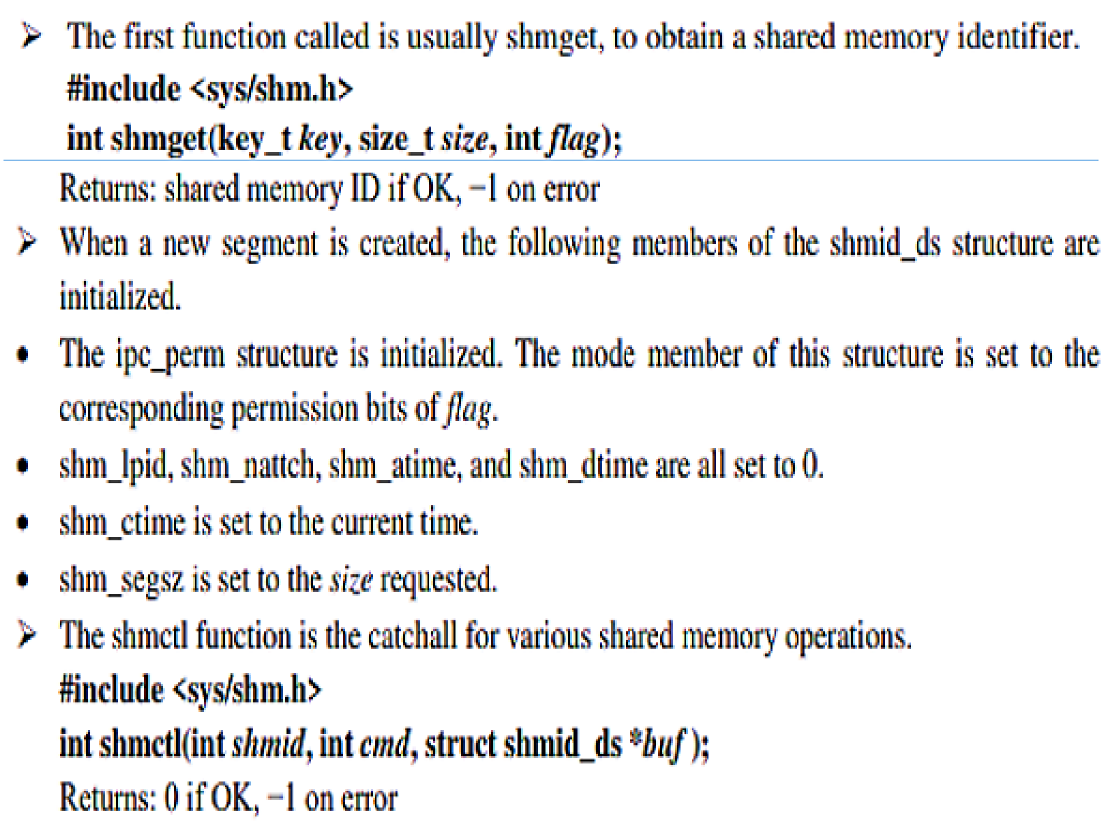
The output of filedes[1] is the input for filedes[0].

```
#include  "apue.h"

int
main(void)
{
        int       n;
        int       fd[2];
        pid_t     pid;
        char      line[MAXLINE];

        if  (pipe(fd)  <  0)
                err_sys("pipe  error");
        if  ((pid  =  fork())  <  0)  {
                err_sys("fork  error");
        }  else  if  (pid  >  0)  {          /*  parent  */
                close(fd[0]);
                write(fd[1],  "hello  world\n",  12);
        }  else  {                          /*  child  */
                close(fd[1]);
                n  =  read(fd[0],  line,  MAXLINE);
                write(STDOUT_FILENO,  line,  n);
        }
        exit(0);
}
```



OR

8 (a)	Define semaphores and explain how the IPC is implemented using various semaphore API's. ➢ A semaphore is a counter used to provide access to a shared data object for multiple processes. ➢ To obtain a shared resource, a process needs to do the following: 1. Test the semaphore that controls the resource. 2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource. 3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1. **#include <sys/sem.h>** **int semget(key_t key, int nsems, int flag);** Returns: semaphore ID if OK, 1 on error. **#include <sys/sem.h>** **int semctl(int semid, int semnum, int cmd,... /* union semun arg */);** **#include <sys/sem.h>** **int semop(int semid, struct sembuf semoparray[], size_t nops);** Returns: 0 if OK, 1 on error.	[10]	CO4
(b)	**Explain the implementation of shared memory IPC mechanism with all its API's and their prototypes.** ☐ Shared memory allows two or more processes to share a given region of memory. ☐ This is the fastest form of IPC, because the data does not need to be copied between the client and the server. ☐ The only trick in using shared memory is synchronizing access to a given region among multiple processes. ☐ If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. ➢ Once a shared memory segment has been created, a process attaches it to its address space by calling shmat. **#include <sys/shm.h>** **void *shmat(int *shmid*, const void **addr*, int *flag*);** Returns: pointer to shared memory segment if OK, −1 on error ➢ The SHM_RND command stands for "round." SHMLBA stands for "low boundary address multiple" and is always a power of 2. **#include <sys/shm.h>** **int shmdt(const void **addr*);** Returns: 0 if OK, −1 on error	[10]	CO4



- ➢ The first function called is usually shmget, to obtain a shared memory identifier.

  #include <sys/shm.h>

  int shmget(key_t *key*, size_t *size*, int *flag*);

  Returns: shared memory ID if OK, −1 on error

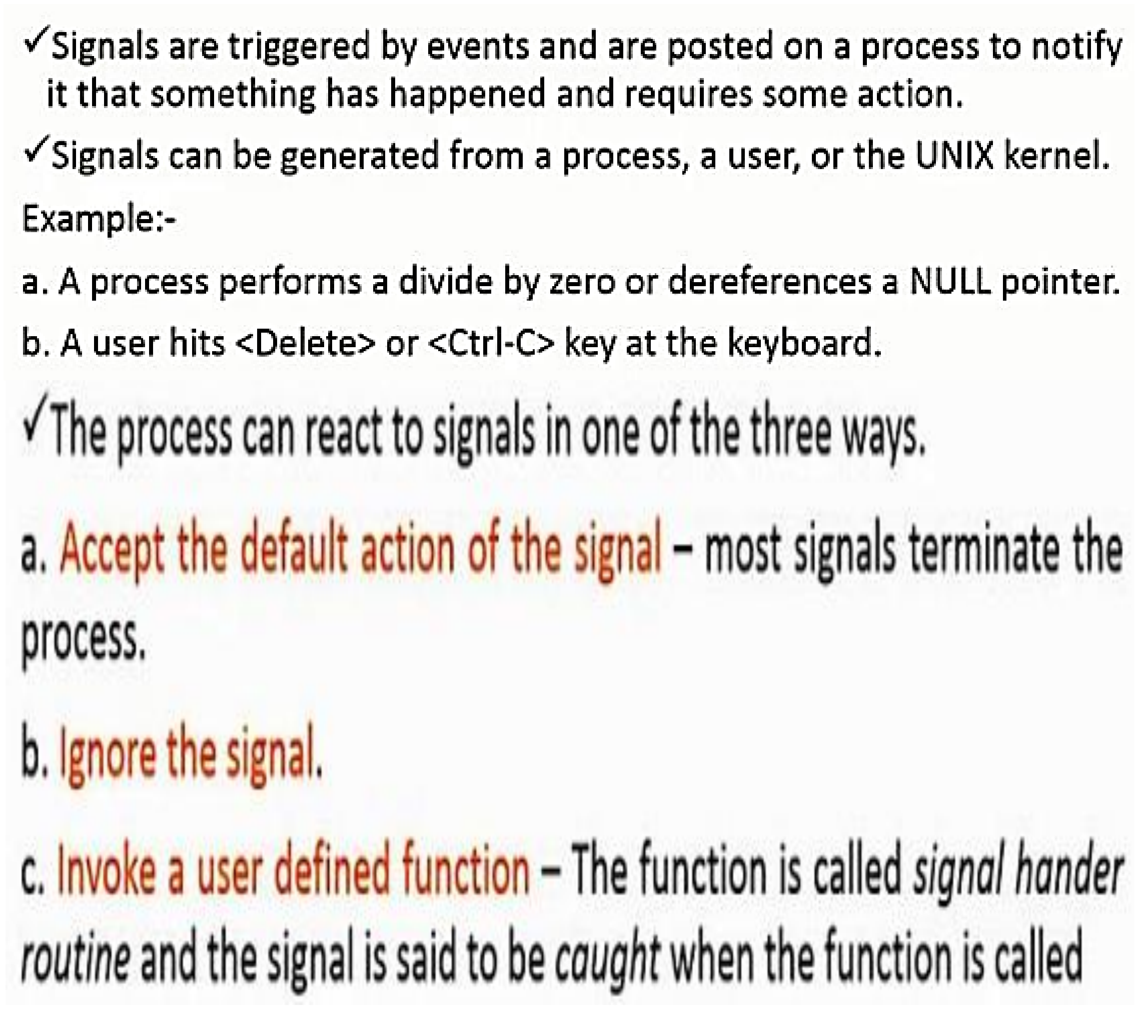
- ➢ When a new segment is created, the following members of the shmid_ds structure are initialized.
  - The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of *flag*.
  - shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
  - shm_ctime is set to the current time.
  - shm_segsz is set to the *size* requested.
- ➢ The shmctl function is the catchall for various shared memory operations.

  #include <sys/shm.h>

  int shmctl(int *shmid*, int *cmd*, struct shmid_ds **buf*);

  Returns: 0 if OK, −1 on error

9 (a) **Define signal and list the actions taken by a process when the signal is raised. Explain the signal API signal (), sigset (), sigaction ().** [10] CO5



✓Signals are triggered by events and are posted on a process to notify it that something has happened and requires some action.

✓Signals can be generated from a process, a user, or the UNIX kernel.

Example:-

a. A process performs a divide by zero or dereferences a NULL pointer.

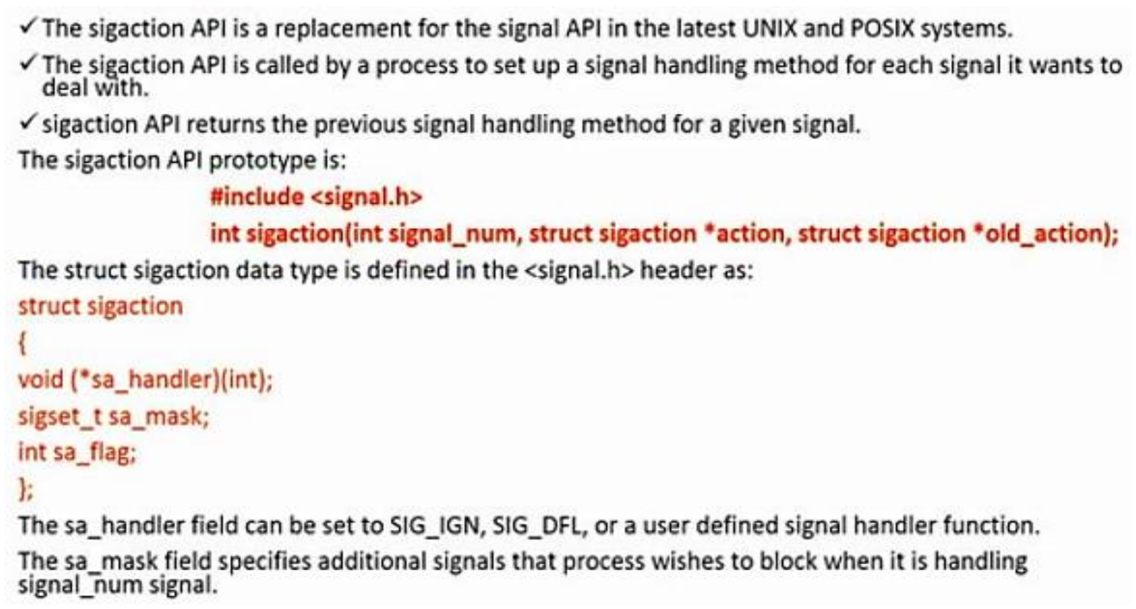
b. A user hits <Delete> or <Ctrl-C> key at the keyboard.

✓The process can react to signals in one of the three ways.

a. Accept the default action of the signal – most signals terminate the process.

b. Ignore the signal.

c. Invoke a user defined function – The function is called *signal hander routine* and the signal is said to be *caught* when the function is called



✓ 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.

The sigaction API prototype is:

```
        #include <signal.h>
        int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

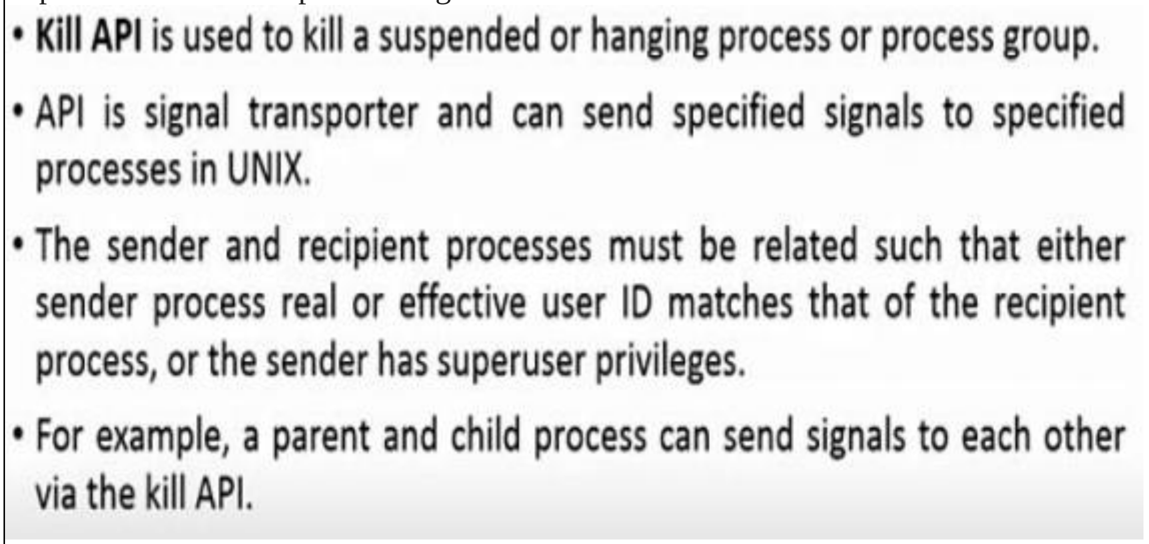
The struct sigaction data type is defined in the <signal.h> header as:

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

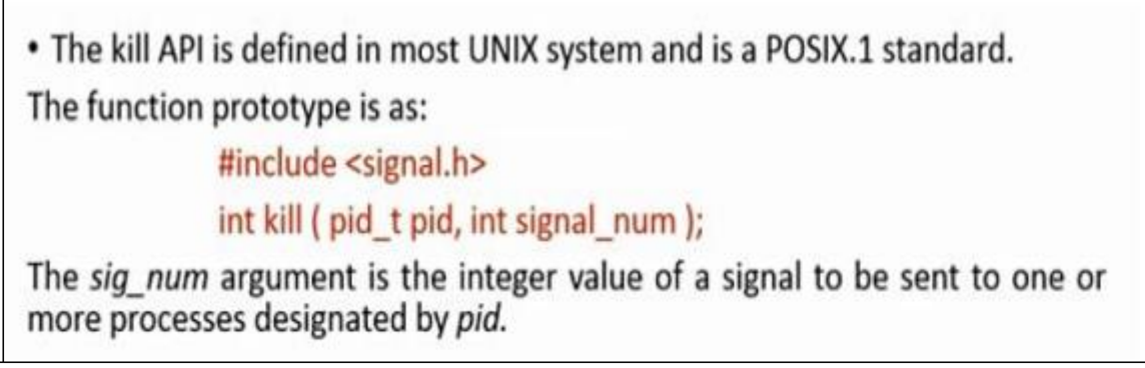
The sa_handler field can be set to SIG_IGN, SIG_DFL, or a user defined signal handler function.

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

(b) Explain how kill API is used for sending a signal to a process and explain the implementation of sleep API using alarm API. [10] CO5



- **Kill API** is used to kill a suspended or hanging process or process group.
- API is signal transporter and can send specified signals to specified processes in UNIX.
- The sender and recipient processes must be related such that either sender process real or effective user ID matches that of the recipient process, or the sender has superuser privileges.
- For example, a parent and child process can send signals to each other via the kill API.

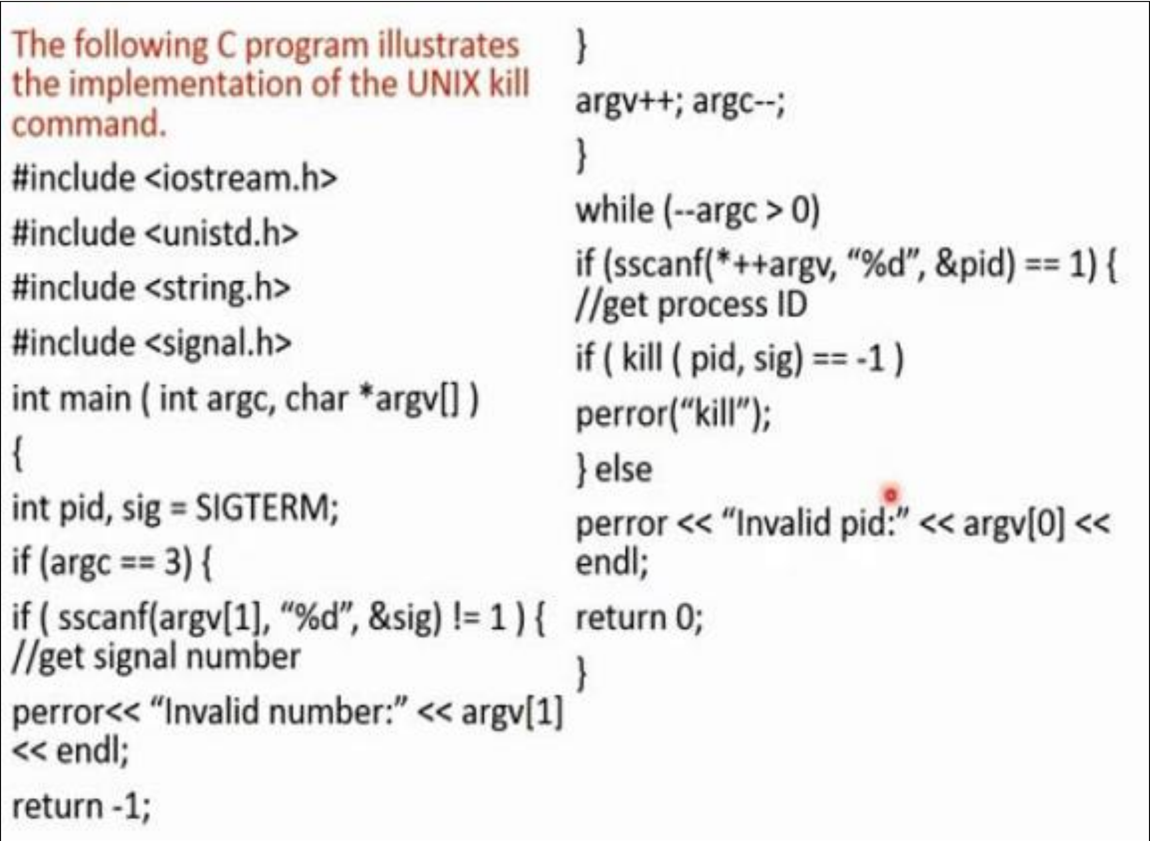


- The kill API is defined in most UNIX system and is a POSIX.1 standard.

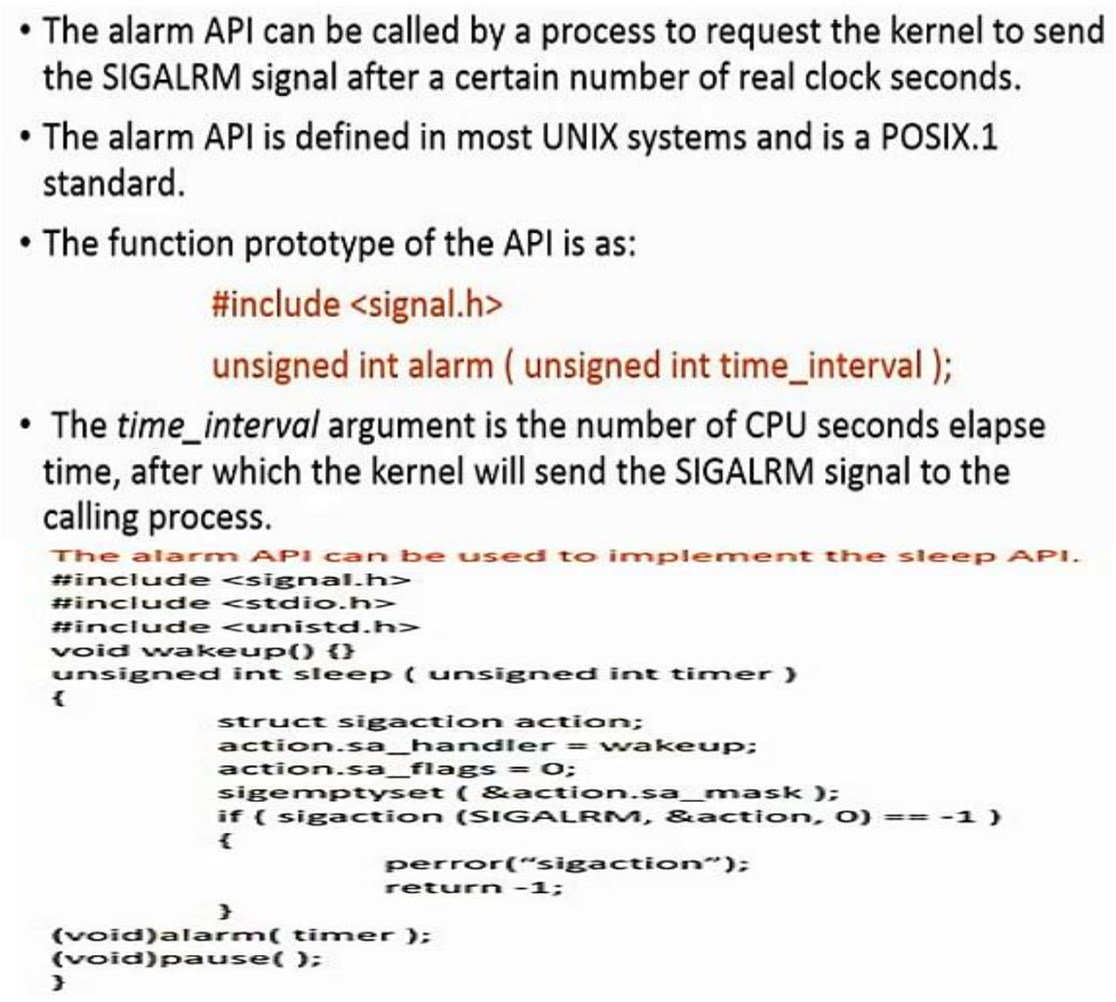
The function prototype is as:

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

The *sig_num* argument is the integer value of a signal to be sent to one or more processes designated by *pid*.


The following C program illustrates the implementation of the UNIX kill command.

```
#include <iostream.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
perror<< “Invalid number:” << argv[1] << endl;
return -1;
}
argv++; argc--;
}
while (--argc > 0)
if (sscanf(*++argv, “%d”, &pid) == 1) { //get process ID
if ( kill ( pid, sig) == -1 )
perror(“kill”);
} else
perror << “Invalid pid:” << argv[0] << endl;
return 0;
}
```


- The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds.
- The alarm API is defined in most UNIX systems and is a POSIX.1 standard.
- The function prototype of the API is as:

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

- The *time_interval* argument is the number of CPU seconds elapse time, after which the kernel will send the SIGALRM signal to the calling process.

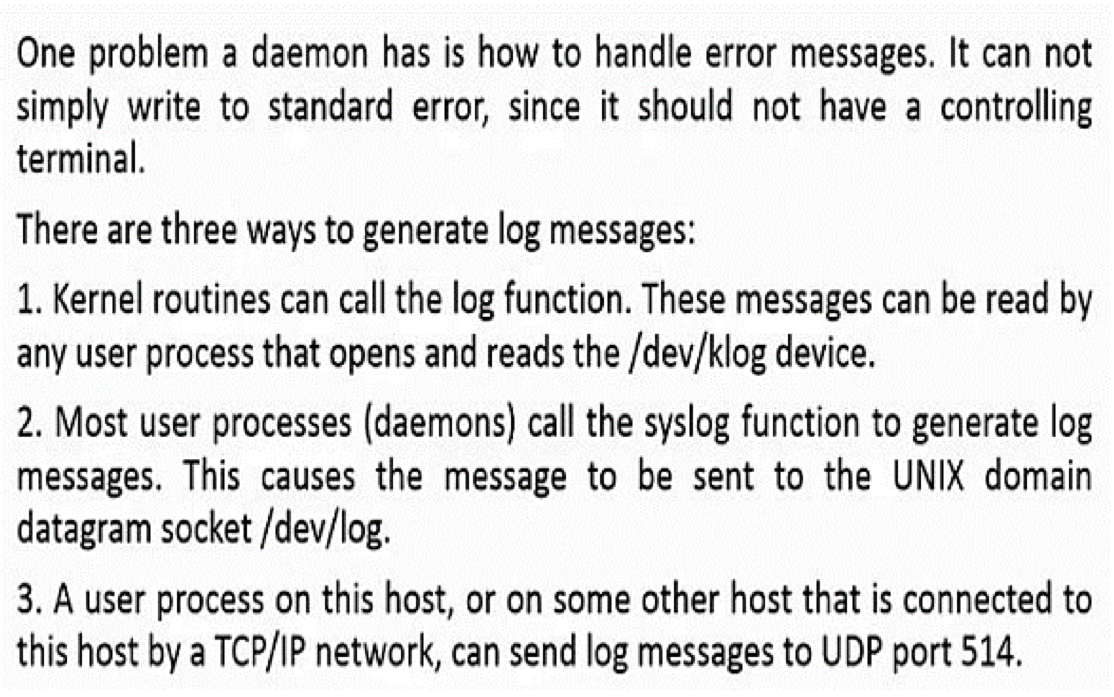
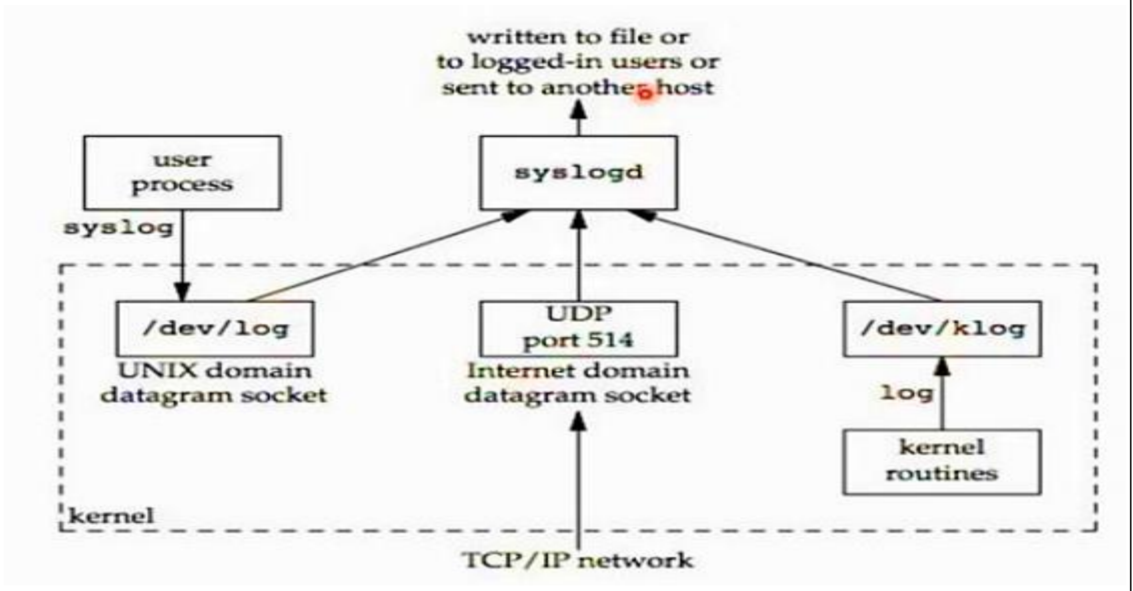
The alarm API can be used to implement the sleep API.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void wakeup() {}
unsigned int sleep ( unsigned int timer )
{
            struct sigaction action;
            action.sa_handler = wakeup;
            action.sa_flags = 0;
            sigemptyset ( &action.sa_mask );
            if ( sigaction (SIGALRM, &action, 0) == -1 )
            {
                        perror(“sigaction”);
                        return -1;
            }
(void)alarm( timer );
(void)pause( );
}
```

OR

10 (a)	Define the Daemon process. Explain all the coding rules to be followed while coding at daemon process. • A **daemon** (also known as background **processes**) is a **Linux** or UNIX program that runs in the background. Almost all **daemons** have names that end with the letter "d". • Daemons are processes that live for a long time. • They are often started when the computer system is started and terminate only when the system is shut down. • They do not have a controlling terminal; so we say that they run in the background. 1. The first thing to do is call umask to set the file mode creation mask to 0. 2. Call fork and have the parent exit. 3. Call setsid to create a new session. 4. Change the current working directory to the root directory. 5. Unneeded file descriptors should be closed. 6. Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect.	[10]	CO5

(b)	**Write a note on interval timer.**
	
	• The interval timer 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_no)
	{
	alarm(INTERVAL); /*do scheduled tasks*/ }
	
	main()
	{
	struct sigaction action; sigemptyset(&action.sa_mask);
	action.sa_handler=(void(*)( )) callme;
	action.sa_flags=SA_RESTART;
	if(sigaction(SIGALARM,&action,0)==-1) { perror("sigaction");
	return 1; } if(alarm(INTERVAL)==-1)
	perror("alarm");
	else while(1)
	{
	/*do normal operation*/
	}
	return 0;
	}
		[05]	CO5

(c)	**Explain the BSD syslog facility for handling Daemons error messages.**  One problem a daemon has is how to handle error messages. It can not simply write to standard error, since it should not have a controlling terminal. 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 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.  written to file or to logged-in users or sent to another host user process syslogd syslog /dev/log UNIX domain datagram socket UDP port 514 Internet domain datagram socket /dev/klog log kernel routines kernel TCP/IP network	[05]	CO5

Faculty Signature CCI Signature HOD Signature