III/IV B.Tech (Regular) DEGREE EXAMINATION
UNIX Programming (14 IT 506/C)
Scheme of Evaluation

Maximum: 60 Marks

Answer ONE question from each unit. (4*12 = 48 Marks)

1. Write briefly about the following 1*12= 12 Marks

a. Difference between kernel and shell.
Ans: Kernel interacts with hardware and most of the tasks like memory management, process scheduling and file management. Shell is interface between kernel and user. When you type in a command at your terminal, the shell interprets the command and calls the program that you want.

b. Difference between single user and multiuser system.
Ans: The main difference between single user and multiuser operating system is that in a single user operating system, only one user can access the computer system at a time while in a multiuser operating system, multiple users can access the computer system at a time.

c. Write the uses of sed command.
Ans: Sed command is used to edit the file with the line and context addressing.

d. Shell variable.
Ans: Shell variable can hold the data and also convert to environmental variable.

e. What does *expr* do in shell script?
Ans: *expr* is a command to do arithmetic operations in UNIX.

f. What is a background process in UNIX?
Ans: A background process is a program that is running without user input and using '&' can make a process "back ground Process".

g. Process group.
Ans: a process group denotes a collection of one or more processes.

h. Difference between system call and command.
Ans: System call is a function that is executed by your operating system, but Command is a program (or a shell built-in) you execute from your command shell.

i. Difference between program and process.
Ans: Process means program is in execution, where as job is a task which is not in execution stage.
(and/or) Process is active, where as job is passive.

j. Socket.
Ans: Socket is a IPC mechanism to communicate between two processes within same or different systems.

k. Write the syntax to create a pipe.
Ans: int pipe (int fd [2])

l. Write the syntax to create a shared memory segment.
Ans: int shmget (key_t,size_t size, int shmflg)

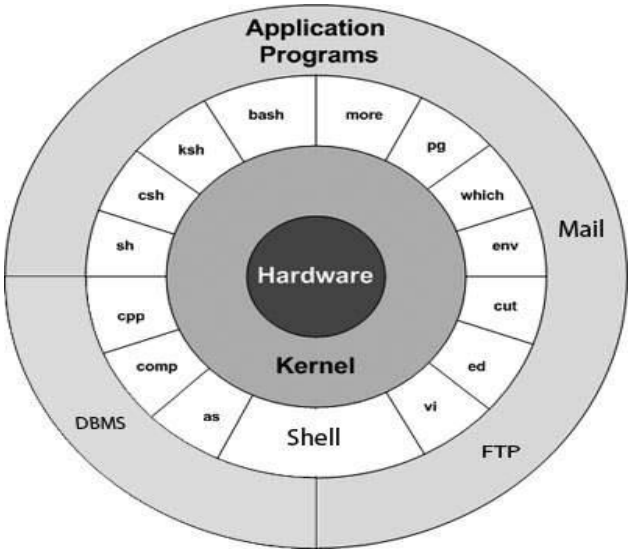
UNIT I

2. a) With a neat diagram, explain the architecture of UNIX operating system.

Ans: [Diagram – 2M, Explanation – 4M]

UNIX architecture contains hardware, kernel, shell and applications.

Hardware: Hardware is the main infrastructure such as I/O peripheral, Hard disk, etc…to work with UNIX.



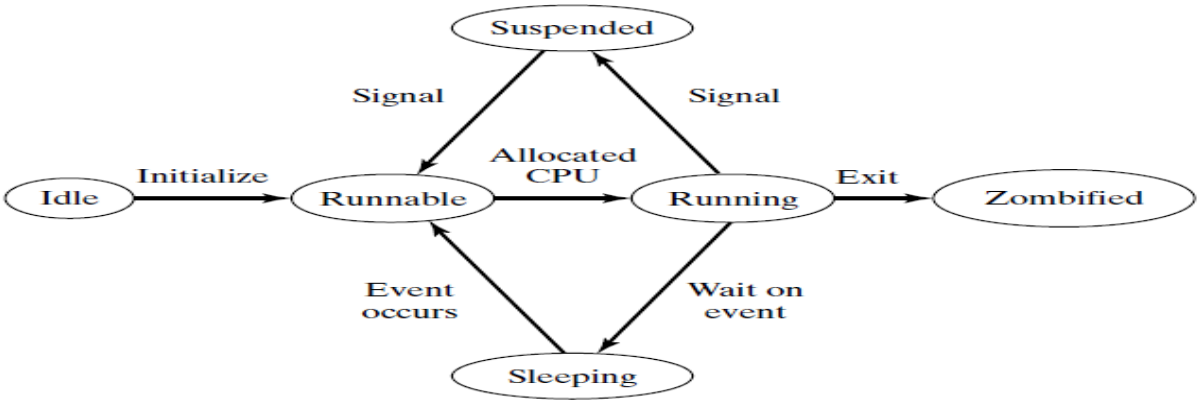
Applications: These applications are run with in shell also. And applications such as Data bases, Text processors, Browsers and etc…

Kernel: The kernel is the core of the operating system. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs 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. It manages the system's memory, schedules processes and decides their priorities. The kernel is called the gateway to the computer's resources.

Shell: The outer parts of the operating system jobs are done using the Shell. It is actually the interface between user and kernel. Even though there's only one kernel running on the system, there could be several shells are in action, one for each user who is logged in. When enter a command through the keyboard, the shell thoroughly examines the keyboard input for special characters.

2. b) With a neat diagram, explain the process life cycle.

Ans: [Diagram – 2M, Explanation – 4M]



Every process in the system can be in one of six states:

•**Running**, which means that the process is currently using the CPU.

•**Runnable**, which means that the process can make use of the CPU as soon as it becomes available.

•**Sleeping**, which means that the process is waiting for an event to occur. For example, if a process executes a read () system call, it sleeps until the I/O request is completed.
•**Suspended**, which means that the process has been "frozen" by a signal such as SIGSTOP. It will resume only when sent a SIGCONT signal. For example, a Control-Z from the keyboard suspends all of the processes in the foreground job.
•**Idle**, which means that the process is being created by a fork () system call and is not yet runnable.
•**Zombified,** which means that the process has terminated, but has not yet returned its exit code to its parent. A process remains a zombie until its parent accepts its return code via the wait () system call

3. a) Write the use of sed editor. Illustrate matching characters in regular expression with examples.
Ans: sed is a multipurpose tool which combines the work of several filters. sed performs non interactive operations on a data stream. The syntax is
sed options `address action` files(s) **---2 Marks**

Context Addressing: The pattern must be bounded by a / on either side.
$ sed –n '/director/p' emp.lst
<output>

$ sed –n '/dasgupta/,/saksena/p' emp.lst
<output>

Using Regular Expressions: Context addresses also uses regular expressions. As like
$ sed –n '/[aA]gg*[ar][ar]wal/p' emp.lst
<output>
Sample example with explanation **---4 Marks**

3. b) Write the use of awk. Illustrate different functions in awk with examples.
Ans: The use of awk is filter the text and produce output as per the user perspective. It will provide program facility also. It can spilt the lines into fields and output formatting.

Functions:
i) Length: It determines the length of its argument, and if no argument is present, the entire line is assumed to be argument.
ex: awk –F"|" 'length > 1024' emp.lst

ii) Index: index(s1,s2) determines the position of a string s2 within a larger string s1. This action especially useful in validating single character fields.
ex: x= index ("abcde", "b")
This returns the value 2
iii) Substr: The substr(stg, m,n) function extracts a substring from a string stg, m represents the starting point of extraction and n indicates the number of characters to be extracted. Because string values can also be used for computation.
ex: awk –F"|" 'substr($5,7,2) >45 && substr($5,7,2) < 52' emp.lst
iv) Split: split(stg,arr,ch) breaks up a string stg on the determine ch and stores the fields in an array arr[ ].
Awk –F\| '{split($5,ar."/")

v) System: Want to print system date at the beginning of the report.
BEGIN {
system("date") } **----6 Marks**

UNIT - II

4. a) Illustrate decision making and loop control statements in Shell.

Ans:

i) if: the syntax is as

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

Example:

```
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

ii) case: The syntax is

```
case word in
   pattern1)
      Statement(s) to be executed if pattern1 matches
      ;;
   pattern2)
      Statement(s) to be executed if pattern2 matches
      ;;
   pattern3)
      Statement(s) to be executed if pattern3 matches
      ;;
esac
```

Example:

```
option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

----3 Marks

iii) while: The syntax is

```
while command
do
```

Statement(s) to be executed if command is true
done

Example:

```
a=0
while [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

iv) for: The for allows a list of commands to e executed several times, using different values of the loop variable during each iteration. Its syntax is

for *name* [in {*word*}*]
do
list
done

The for command loops the value of the variable *name* through each *word* in the word list, evaluating the commands in *list* after each iteration. If no word list is supplied, $@ ($1..) is used instead. A break command causes the loop to end immediately, and a continue command causes the loop to jump immediately to the next iteration.

(example program)

```
$ cat for.sh
for color in red yellow green blue
do echo one color is $color
done
```

Output:

```
one color is red
one color is yellow
one color is green
one color is blue
```

---3 Marks

4. b) List out the common environment variables that control the user environment in Shell.

Ans:

Name	Meaning
$HOME	the full pathname of your home directory
$PATH	a list of directories to search for commands
$MAIL	the full pathname of your mailbox
$USER	your username
$SHELL	the full pathname of your login shell
$TERM	the type of your terminal

```
echo HOME = $HOME, PATH = $PATH ...list two variables.
HOME = /home/glass, PATH = /bin:/usr/bin:/usr/sbin
$ echo MAIL = $MAIL ...list another.
MAIL = /var/mail/glass
```

$ echo USER = $USER, SHELL = $SHELL, TERM=$TERM
USER = glass, SHELL = /bin/sh, TERM=vt100 ------**6 Marks**

5. a) Illustrate with examples the mechanisms for string handling and command line arguments in Shell environment.

Ans: String handling mechanism is based on some conditional expressions. They can be identified with following

-l string: The length of string.
-n string: True if string contains at least one character.
-z string: True if string contains no characters.
str1 = str2: True if str1 is equal to str2.
str1 != str2: True if str1 is not equal to str2.
String: True if string is not null.
Sample example:

```
if [ $1 == $2 ]
then
echo "Two Files are Same"
else
    d=`diff $1 $2`
if [ -z $d ]
then
rm $2
echo "Second file Removed"
else
echo "The Contents of the files are not same"
fi
fi
```

---3 Marks

With using positional parameters and predefined local variables can get data from command line arguments into the shell script.

Name	Value
$@	an individually quoted list of all the positional parameters
$#	the number of positional parameters
$?	the exit value of the last command
$!	the process ID of the last background command
$-	the current shell options assigned from the command line or by the built-in set command (see later)
$$	the process ID of the shell in use

Sample example:

```
echo there are $# command line arguments: $@
cc $1 # compile the first argument.
echo the last exit value was $?
```

----3 Marks

5. b) Write positional parameters in UNIX along with examples.

Ans: A positional parameter is a variable within a shell program; its value is set from an argument specified on the command line that invokes the program. Positional parameters are numbered and are referred to with a preceding ``$": $1, $2, $3, and so on.

Sample example:

```
factorial ()
{
local number=$1
if [ $number -eq 0 ]
then
   Factorial=1
else
   let "next = number - 1"
   factorial $next
   Factorial=`echo $number \* $Factorial | bc`
#   let "Factorial = $number * $Factorial"
fi
return $Factorial 2>/dev/null
}
if [ $# -ne 1 ]
then
   echo "Invalid Arguments"
   echo "Usage: ./fact2.sh  Number "
   exit 1
fi
factorial $1
echo "Factorial of $1 is $Factorial"
```

----**6 Marks**

UNIT – III

6. Explain the syntax and each argument of the following system calls:
(i) create () (ii) open () iii) lseek () iv. stat () v. dup() vi. fcntl()

Ans:

i) Create(): A call to create() creates a new open file description, an entry in the system-wide table of open files. The parameter flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. This request opening the file read-only, write-only, or read/write, respectively. ---**2 Marks**

ii) open(): The syntax is

```
int open(const char *pathname, int flags, mode_t mode);
```

A call to open() creates a new open file description, an entry in the system-wide table of open files. The parameter flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively. ----**2 Marks**

iii) lseek(): lseek() allows you to change a descriptors current file position.fd is the file descriptor,offset is a long integer,mode describes how offset should beinterpreted.

```
Off_t lseek(int fd,off_t offset,int mode)
```

Sample program:

```
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
main(intargc,char *argv[])
```

```
{
        intfd,skval;
        char c;
        if(argc!=2)
        {
                printf("Need 1 argument\n");
                exit(1);
        }
        fd=open(argv[1],O_RDONLY);
        if(fd==-1)
        {
                printf("Can't open file %s\n",argv[1]);
                exit(2);
        }
        while((skval=read(fd,&c,1))==1)
        {
                printf("Char:%c\n",c);
                skval=lseek(fd,99,1);
                printf("New seek value is:%d\t",skval);
        }
        printf("\n");
        exit(0);
}
```

----2 Marks

iv) stat (): Syntax as follows

int stat(const char **path*, struct stat ***buf*);
int fstat(int *filedes*, struct stat ***buf*);
int lstat(const char **path*, struct stat ***buf*);

stat() stats the file pointed to by *path* and fills in *buf*.
lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.
fstat() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *filedes*.

These functions return information about a file, in the buffer pointed to by *buf*. No permissions are required on the file itself, but—in the case of **stat**(), **fstatat**(), and **lstat**()—execute permission is required on all of the directories in *pathname* that lead to the file **----2 Marks**

v) dup(): dup() allow to duplicate file descriptors. The syntax as follows

int dup (int old Fd)

Here, dup() finds the smallest free file descriptor entry and points it to the same file as oldFd. This function returns the index of the new file descriptor if successful and -1 otherwise.

Sample program:

```
#include <stadio.h>
#include <fcntl.h>
main ()
{
 Int fd1, fd2, fd3;
fd1=open("test.txt", O_RDWR | O_TRUNC);
```

```
printf("fd1=%d\n",fd1);
write(fd1, "what's ",6);
fd2=dup(fd1);
printf("fd2=%d\n",fd2);
write(fd2,"up",3);
close(0);
fd3=dup(fd1);
printf("fd3=%d\n",fd3);
write(2, "?\n",2);
}
```
----2 Marks

vi) fcntl(): The syntax is

int fcntl (int fd, int cmd, int arg)

fcntl () performs the operation encoded by cmd on the file associated with the file descriptor fd. arg is an optional argument for cmd. Here are the most common values of cmd:

VALUE	OPERATION
F_SETFD	Set the close-on-exec flag to the lowest bit of arg (0 or 1).
F_GETFD	Return a number whose lowest bit is 1 if the close-on-exec flag is set and 0 otherwise.
F_GETFL	Return a number corresponding to the current file status flags and access modes.
F_SETFL	Set the current file status flags to arg. **----2 Marks**

7. a) Illustrate the difference between fork() and exec() with examples programs.

Ans:

Fork: System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child.

Sample program: (optional)

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    intpid;
    printf("Original Process\tPID=%d   PPID=%d\n",getpid(),getppid());
    pid=fork();
    printf("%d",pid);
    if(pid==0)
    {
        printf("Child\tPID=%d   PPID=%d\n",getpid(),getppid());
    }
    else
```

```
    {
        printf("Parent\tPID=%d   PPID=%d\n",getpid(),getppid());
    }
    printf("PID %d terminates\n",getpid());
    exit(0);
}
```

Exec(): The created child process does not have to run the same program as the parent process does. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script.

```
int execl ( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execv ( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
```

The **execvp()** system call requires two arguments:

1. The first argument is a character string that contains the name of a file to be executed.
2. The second argument is a pointer to an array of character strings. More precisely, its type is **char ****, which is exactly identical to the **argv** array used in the main program:
3. int main(int argc, char **argv)

When **execvp()** is executed, the program file given by the first argument will be loaded into the caller's address space and **over-write** the program there. Then, the second argument will be provided to the program and starts the execution. As a result, once the specified program file starts its execution, the original program in the caller's address space is gone and is replaced by the new program.

execvp() returns a negative value if the execution fails.

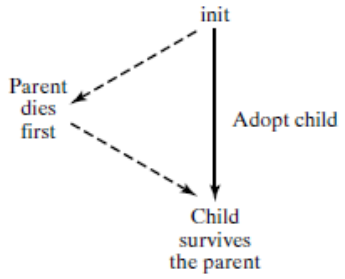
Sample program: (optional)

```
int main( void ) {
    char *argv[3] = {"Command-line", ".", NULL};

    int pid = fork();

    if ( pid == 0 ) {
        execvp( "find", argv );
    }       /* Put the parent to sleep for 2 seconds--let the child finished executing */
    wait( 2 );
    printf( "Finished executing the parent process\n"
        " - the child won't get here--you will only see this once\n");
    return 0;
}
```

-----6 Marks

7. b) What happens to the child process when the parent process kills/dies first? Explain with example.

Ans: If a parent dies before its child, the child is automatically adopted by the original “init” process, PID 1. The “init” process always accepts its children’s termination codes.



----2 Marks

```
#include <stdio.h>
main ()
{
int pid;
printf ("I'm the original process with PID %d and PPID %d.\n",
getpid (), getppid ());
pid = fork (); /* Duplicate. Child and parent continue from here */
if (pid != 0) /* Branch based on return value from fork () */
{
/* pid is non-zero, so I must be the parent */
printf ("I'm the parent process with PID %d and PPID %d.\n",
getpid (), getppid ());
printf ("My child's PID is %d\n", pid);
}
else
{
/* pid is zero, so I must be the child */
sleep (5); /* Make sure that the parent terminates first */
printf ("I'm the child process with PID %d and PPID %d.\n",
getpid (), getppid ());
}
printf ("PID %d terminates.\n", getpid () ); /* Both processes execute
this */
}
```
----4 Marks

8. a) Describe in detail about the signals.

Ans: Signal function is an action of any one of these signal handling function, signal default and signal ignore. **---1 Mark**

Signal function is mainly handling the signal nothing but it's called signal handler. The default signal handler usually performs one of the following actions:

- terminates the process and generates a core file (dump)
- terminates the process without generating a core image file (quit)
- ignores and discards the signal (quit)
- suspends the process (suspend)
- resumes the process **----1 Mark**

List some signals explanation **----4 Marks**

8. b) Illustrate the role of kill and raise functions in signal generation.

Ans: Kill function is used send the signal to other processes such as SIGSTOP and SIGCONT. To follow these signals the process can run like that. The syntax is

int kill(pid_t pid, int sig); **----1 Mark**

The raise function allows a process to send a signal to itself. If the process want send the any signal by itself this raise function is useful. The syntax is

int raise(int sig); **-----1 Mark**

Sample example:

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

```
{
        int pid1,pid2;

        pid1=fork();
        printf("PID1=%d\n",pid1);
        if(pid1==0)             //First child
        {
                while(1)
                {
                        printf("Pid1 is alive\n");
                        sleep(1);
                }
        }
        pid2=fork();
        printf("PID2=%d\n",pid2);
        if(pid2==0)             //Second child
        {
                while(1)
                {
                        printf("Pid2 is alive\n");
                        sleep(1);
                }
        }
        sleep(3);
        kill(pid1,SIGSTOP);  //Suspend first child
        sleep(3);
        kill(pid1,SIGCONT);  //Resume first child
        sleep(3);
        kill(pid1,SIGINT);     //Kill first child
        kill(pid2,SIGINT);     //Kill second child

}
```

-----4 Marks

9. a) Write the advantage of pipes in UNIX. Write a simple program to demonstrate IPC mechanism between child and parent process.

Ans: Pipes are an interprocess communication mechanism that allows two or more processes to send information to each other. They are commonly used from within shells to connect the standard output of one utility to the standard input of another.

All versions of UNIX support unnamed pipes, which are the kind of pipes that shells use. System V also supports a more powerful kind of pipe called a named pipe.

Named pipes [often referred to as first-in, first-out queues (FIFOs)] are less restricted than unnamed pipes, and offer the following advantages:

- They have a name that exists in the file system.
- They may be used by unrelated processes.
- They exist until they are explicitly deleted. **------3 Marks**

```
#include <stdio.h>
#define READ 0 /* The index of the read end of the pipe */
#define WRITE 1 /* The index of the write end of the pipe */
```

```
char* phrase = "Stuff this in your pipe and smoke it";
main ()
{
int fd [2], bytesRead;
char message [100]; /* Parent process' message buffer */
pipe (fd); /*Create an unnamed pipe */
if (fork () == 0) /* Child, writer */
{
close(fd[READ]); /* Close unused end */
write (fd[WRITE],phrase, strlen (phrase) + 1); /* include NULL*/
close (fd[WRITE]); /* Close used end*/
}
else /* Parent, reader*/
{
close (fd[WRITE]); /* Close unusedend */
bytesRead = read (fd[READ], message, 100);
printf ("Read %d bytes: %s\n", bytesRead, message); /* Send */
close (fd[READ]); /* Close usedend */
}
}
```
-----3 Marks

9. b) Explain in detail IPC using semaphores with an example.

Ans:

Semaphores are another IPC mechanism available when developing on UNIX. They allow different processes to synchronize their access to certain resources. The most common and simplest kind of semaphore is called a binary semaphore because they have two states locked or unlocked. ----**1 Mark**

semget(): Gets a semaphore set. The value returned is its id, for use with other calls.
int semget(key_t key, int n, int flags);
key is the key associated with the semaphore set you want.
n is the number of semaphores the set should contain.
flags specifies how how the set should be allocated. SHM R | SHM W is the best thing to pass.

The following members of the semid_ds structure are initialized.
•The ipc_perm structure is initialized this structure is set to the corresponding permission bits of flag.
•sem_otime is set to 0.
•sem_ctime is set to the current time.
•sem_nsems is set to nsems. --------**2 Marks**

semctl(): A bit like shmctl(), but for semaphores. Again, ridiculously overcomplicated.
int semctl(int id, int iSem, int cmd, union semun arg);
id is the semaphore set id.
iSem is the semaphore you want to twiddle. Only valid with some of the commands.
cmd is the command you want to perform.
arg is used for fiddling with semaphore values. With everything but cmd set to SETALL, just pass NULL.

There are two values for cmd worth looking at: SETALL and IPC RMID. For details on the others, type man semctl.

The cmd argument specifies one of the following ten commands to be performed on the set specified by semid. The five commands that refer to one particular semaphore value use semnum to specify one member of the set. The value of semnum is between 0 and nsems-1.

IPC_STAT Fetch the semid_ds structure for this set, storing it in the structure pointed to by arg.buf.

IPC_SET Set the sem_perm.uid, sem_perm.gid, and sem_perm.mode fields from the structure pointed to by arg.buf in the semid_ds structure associated with this set. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.

IPC_RMID Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore.

GETVAL Return the value of semval for the member semnum.

SETVAL Set the value of semval for the member semnum. The value is specified by arg.val.

GETPID Return the value of sempid for the member semnum.

GETNCNT Return the value of semncnt for the member semnum.

GETZCNT Return the value of semzcnt for the member semnum.

GETALL Fetch all the semaphore values in the set. These values are stored in the array pointed to by arg.array.

SETALL Set all the semaphore values in the set to the values pointed to by arg.array. **-----3 Marks**

Scheme prepared by Signature of the HOD, IT Dept.

Paper Evaluators:

S.No	Name Of the College	Name of the Faculty	Signature