INFORMATION TECHNOLOGY DEPARTMENT

OS LAB MANUAL

S.NO.	CONTENTS	PAGE NO.
1.	Introduction to Operating Systems laboratory	IV
	PROGRAMS	
2.	Program 1: Familiarity with Lab environment,User and System level Operating System Commands	1
3.	Program 2: Program to get and set environment variables using system calls	3
4.	Program 3: Program for File and Directory Management	5
5.	Program 4: Program to get the attribute of a file or directory on linux using system calls	9
6.	Program 5: Program to Display process information using process related system calls	11
7.	Program 6: Program to demonstrate usage of linux system calls on Process Management- Fork(),(Orphan and Zombie process) Exec(),Wait(),Sleep() etc..	12
8.	Program 7: Program for Creating and Manipulating threads	16
9.	Program 8: Program to demonstrate usage of Pipes, Shared memory, Message Queues.	18
10.	Program 9: Program for echo server using pipes	21
11.	Program 10: Program for echo server using messages queues	22

19.	Program 11: Program for echo server using shared Memory	25
20.	Program 12: Program for Basic Operations and Arithmetic on Semaphores	27
21.	Program 13: Program for producer consumer using semaphores	29
22.	Program 14: Program for producer consumer problem using message passing	31
23.	Program 15: Program for reader and writer using semaphores	32
24.	Program 16: Program for dining philosopher problem using semaphores	34
25.	Program 17: Program for Linux shell scripts: a) Program to display statements b) Program to find whether a number is odd or even c) Program to find factorial of a number d) Program to reverse a number	36
	Annexure – I : O.U prescribed programs for Operating Systems Laboratory	41

Introduction to Operating System Laboratory
Introduction LINUX operating system

LINUX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops. LINUX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of LINUX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session. There are many different versions of LINUX, although they share common similarities. The most popular varieties of LINUX are Sun Solaris, GNU/Linux, and MacOS X.

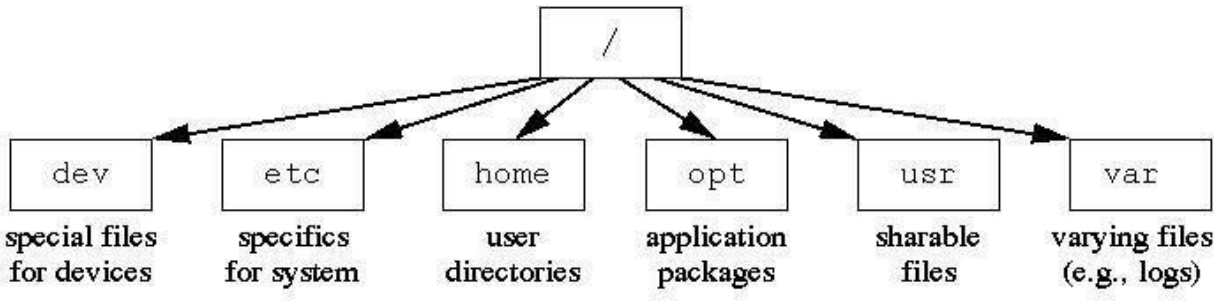
Laboratory Objective

Upon successful completion of this Lab the student will be able to:

1. Become familiar with the User and System level operating system commands
2. Write programs using system calls related to file and process management
3. Implement different IPC mechanisms
4. Find solution for different classical synchronization problems
5. Become familiar with basic concepts of Shell programming and write small shell scripts

Structure of LINUX file system -

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called root (written as a slash /)



etc – Administrative programs and configuration files
dev – Devices drivers (pointers) such as disk drives, keyboard, mouse, etc.
var – Temporary administrative space for logging and other system information
home – Home directories for users
usr – Standard programs and code libraries

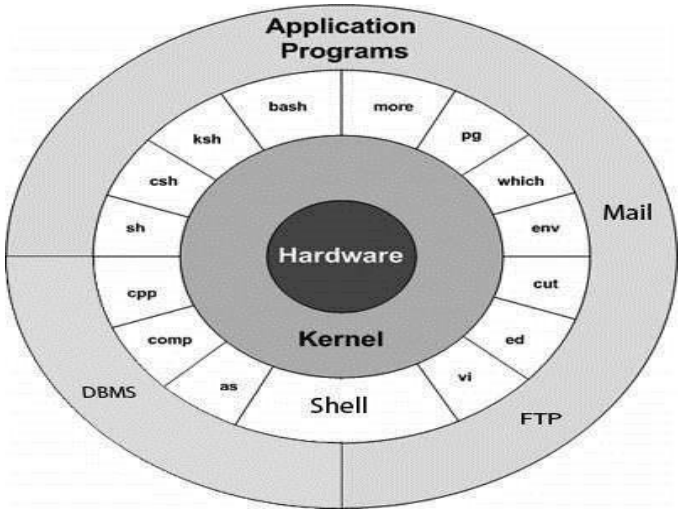
/usr/sbin –Administrative programs
/usr/bin – Standard executable programs
/usr/lib – Code libraries
/usr/local/bin – Additional programs

Features of LINUX operating system -

1) Multitasking: Multitasking is the capability of the operating system to perform various tasks.ie. A single user can perform various tasks. Multiuser capabilities. This allows several users to use the same computer to perform their tasks.
2) **Security: Every user have a login name and password So, accessing another user's data** is impossible without permission
3) Portability: LINUX is portable because it is written in a high level language. So LINUX can be run on different computers.
4) Communication: LINUX supports the following communications.
   i) Between the different terminals connected to the LINUX server.
   ii) Between the users of one computer to the users of another.
5) Programming facility: LINUX is highly programmable, the LINUX shell programming language has all the necessary ingredients like conditional and control structures (Loops) and variables.

LINUX Architecture -

Here is a basic block diagram of a LIN**UX system** −



The main concept that unites all versions of LINUX is the following four basics −

- **Kernel:** The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, task scheduling and file management.

- **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the LINUX variants.

- **Commands and Utilities:** There are various command and utilities which you would use in your day to day activities. **cp, mv, cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.

- **Files and Directories:** All data in LINUX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

System Requirement

Hardware Configuration: LENOVO Think Centre Intel Core I3-3220, RAM : 4GB,
HDD: 500 GB

Software Configuration: Red Hat Linux 6 Client, Microsoft Windows 7

Program 1
Familiarity with Lab environment, User and System level Operating System Commands

Problem Definition
To become Familiar with Lab environment, User and System level Operating System Commands

Problem Description
1) who:-It displays the information about all the users who have logged into the system currently
2) whoami:- It displays Current username, Terminal number, date and time at which user logged into the system
3) pwd:- It displays current working directory
4) date:- It displays system date and time
5) ls - It lists the files and directories stored in the current directory. To list the files in a directory use the following syntax: $ls dirname
6) mkdir – It is used to create directories by using the command: $mkdir dirname
7) clear- It clears the screen
8) cd - It is used to change the current working directory to any other directory specified
9) cd.. -This command is used to come out from the current working directory.
10) rmdir - Directories can be deleted using the **rmdir** command - $rmdir dirname
11) cat – It displays the contents of a file - $cat filename
12) cp - It is used to copy a file - $ cp source_file destination_file
13) mv- It is used to change the name of a file - $ mv old_file new_file
14) rm – It is used to delete an existing file - $ rm filename
15) stat- It is used to display file or file system status - $ stat filename
16) stty – Change and print terminal line settings. Its option – **"stty –a"** prints all current settings in human readable form
17) tty – It prints the filename of the terminal connected to standard input.
18) uname –It prints system information
19) umask – It specifies user file creation mask, implying which of the 3 permissions are to be denied to the owner,group and others.
20) find – It searches for files in a directory hierarchy
    Its form is – **find path-list selection-criteria action**
21) sort – It sorts the lines of text files
22) ps - It displays information about the current processes.
23) set - It displays name and value of each shell environment variable.

File Permission

commands chmod

Changes the file/directory permission mode.

For ex : $ chmod 777 file1, gives full permission to owner, group and others

$ chmod o-w file1

Removes write permission for others.

Pseudocode

Synopsis:

Command [-switches] [arguments]

where:

- Command: The UNIX command
- [-Switches]:Optional parameter,usually single letters,providing aditional functionality to the command
- [arguments]: what the command is to be run against

Note:UNIX commands are case sensitive.

Problem Validation

Output

– $ pwd

$ /home/it13/it1340

Program 2
Program to get and set environment variables using system calls

Problem Definition

Program to GET and SET the Environment variable and to know use of getenv and setenv system calls

Problem Description

UNIX ENVIRONMENT VARIABLES

Variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for particular variables and if they are found will use the values stored. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.Standard UNIX variables are split into two categories, environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE and shell variables have lower case names.

For examples of environment variables are

- USER (your login name)
- HOME (the path name of your home directory)
- HOST (the name of the computer you are using)
- ARCH (the architecture of the computers processor)
- DISPLAY (the name of the computer screen to display X windows)
- PRINTER (the default printer to send print jobs)
- PATH (the directories the shell should search to find a command)

Syntax: Char *getenv(const char *name);

The getenv() function searches the environment list to find the Environment variable *name*,

and returns a pointer to the corresponding *value* string.

Syntax: int setenv(const char * envname,const char * enval,int overwrite)

The setenv() function adds the variable *name* to the environment with the value *value*, if *name* does not already exist. If *name* does exist in the environment, then its value is changed to *value* if *overwrite* is nonzero; if *overwrite* is zero, then the value of *name* is not changed (and setenv() returns a success status).

Pseudocode

1: Retrieve the value of an environment variable

2: Update the value of the retrieved environment variable

Problem Validation

Output -

HOME=/HOME/IT13/IT13067

HOME/IT13067 HOME IS RESET

Program 3
Program for File and Directory Management

Problem Definition

To demonstrate usage of system calls-open(),read(),write(),opendir(),readdir(),closedir() for file and directory management

Problem Description

The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel. To a process then, all input and output operations are synchronous and unbuffered.

System Calls :
"System calls are functions that a programmer can call to perform the services of the operating system. **"**

open() : system call to open a file :open returns a file descriptor, an integer specifying the position of this open file in the table of open files for the current process .
close() : system call to close a file
read() : read data from a file opened for reading
write() : write data to a file opened for writing

The open() system call :

#include<fcntl.h>

int open(const char *path,int oflag);

The return value is the descriptor of the file. Returns -1 if the file could not be opened. The first parameter is path name of the file to be opened and the second parameter is the opening mode specified by bitwise oring one or more of the following values

Value	***Meaning***
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only

O_RDWR	Open for reading and writing
O_APPEND	Open at end of file for writing
O_CREAT	Create the file if it doesn't already exist
O_EXCL	If set and O_CREAT set will cause open() to fail if the file already exists
O_TRUNC	Truncate file size to zero if it already exists

close() system call :

The close() system call is used to close files.

#include <unistd.h>

int close(int fildes);

It is always a good practice to close files when not needed as open files do consume resources and all normal systems impose a limit on the number of files that a process can hold open.

The read() system call :

The read() system call is used to read data from a file or other object identified by a file descriptor. The prototype is

#include<sys/types.h>

size_t read(int fildes,void *buf,size_t nbyte);

fildes is the descriptor, *buf* is the base address of the memory area into which the data is read and *nbyte* is the maximum amount of data to read.The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read.An attempt to read beyond the end of a file results in a return value of zero.

The write() system call :

The write() system call is used to write data to a file or other object identified by a file descriptor. The prototype is

#include<sys/types.h>

size_t write(int fildes, const void *buf, size_t nbyte);

UNIX offers a number of system calls to handle a directory. The following are most commonly used system calls.

1. opendir()

Syntax : DIR * opendir (const char * dirname);

opendir () takes dirname as the path name and returns a pointer to a DIR structure. On error returns NULL.

2. readdir()

Syntax: struct dirent * readdir (DIR *dp) ;

A directory maintains the inode number and filename for every file in its fold. This function returns a pointer to a dirent structure consisting of inode number and filename.'dirent' structure is defined in <dirent.h> to provide at least two members – inode number and directory name.

```
struct  dirent
{
        ino_t d_ino ; // directory inode number
        char d_name[]; // directory name
}
```

3. closedir()

Syntax: int closedir (DIR * dp);

Closes a directory pointed by dp. It returns 0 on success and -1 on error.

Pseudo code (file) -

1 : open file or create and open file in write mode.
2 : Read data from standard input into a buffer.
3 : Write data to file from buffer .
4 : Close the file.
5: Open file for Read only
6 : Read data from file into buffer.
7 :Display buffer data to standard output
8: Close the file

Pseudocode : (Directory)

1 :open directory .
2 :Read the contents of the directory(filenames).
3 :Display the contents of the directory.
4 : Close the directory.

Problem Validation

Result (file):

Input**:**

Enter information to be written to the file

HELLO

Output:
Information read from the file

HELLO

Result(directory):

Output:

Contents of the directory are-

Hello.c

Sample.c

Program 4
Program to get the attribute of a file or directory on linux using system calls

Problem Definition
To write the program to implement the system call stat( ).

Problem Description
stat() system call is used to get file/directory attributes.

Syntax: int stat (char *name, struct stat *buf)

stat ( ) fills the buffer *buf* with information about the file *name*. The stat structure is defined in **"**/usr/include/sys/stat.h**"**. The stat structure contains the following members:

NAME MEANING

st_dev the device number

st_ino the inode number st_mode
the permissions flag st_nlink the
hard link count st_uid the user id

st_gid the group id st_size
the file size

st_atime the last access time st_mtime the
last modification time st_ctime the last
status change time

There are some predefined macros defined in **"**/usr/include/sys/stat.h**"** that takes st_mode as their argument and return true (1) for the following file types:

MACRO RETURNS TRUE FOR FILE TYPE

S_ISDIR directory
S_ISREG regular file
stat returns **0** if **successful** and **–1** otherwise.

Pseudocode –

1: Retrieve the attributes of the file or directory

2: The attributes of the file or directory are Userid, groupid, devicename, inode no, No of Hard links, Size in bytes,Block size, No. of blocks allocated,Time of last access, Last modification and last status change, Type of file,user and group privileges(Read Write and Execute Permissions)

Problem Validation

Output -
Different statistics or attributes of a file are displayed for a given file or directory

Last access time: Tues Apr 11 11:24:48 2015
Last modification time:wed Apr 12 11:03:00 2015
Last status change is:Thus Apr 14 12:09:00 2015
Device 64768
Inode number:32899125
Device type:0
Size in bytes:4096
Block size:4096
No of blocks:16
No of links:2

Program 5
Program to Display process information using process related system calls

Problem Definition
To obtain process information using process related system calls- getpid() and getppid().

Problem Description
To demonstrate usage of process related system calls and display process related information using them.

SYSTEM CALLS USED:

1. getpid( ) Each process is identified by its id value. This function is used to get the id value of a particular process.

2. getppid( ) Used t**o get particular process parent'**s id value.

Pseudocode –

1 : Create a child process
2 : Print process id of the child and its parents process id. 3
: Print process id of the parent and its parents process id

Problem Validation

Output-

Child Processing
Iam child and my process id is 22518
Iam child and my parents process id is 22519
Parent processing
Iam parent and my process id is 22519
Iam parent and my parents process id is 22517

Program 6

Program to demonstrate usage of linux system calls on Process Management- Fork (),(Orphan and Zombie process) Exec(),Wait(),Sleep() etc..

Problem Definition

To Demonstrate process management system calls Fork (),Exec(),Wait, Sleep() etc..

Problem Description

fork() is used to create a new process. The execl () command does not start a new process , it just continues the original process by overlaying memory with a new set of instructions. As this occurs a new program is replaced so there is no way to return to the old program. Some times it is necessary to start a new process , leaving the old process unrelated. This is done with the fork().

Syntax:

pid_t fork(void);

Fork creates an exact replica of parent (calling)process. After fork returns, the parent process and child process now have different PIDs.. At this point , there are two processes with practically identical constitute, and they both continue execution at the statement following fork(). To be able to distinguish between the parent and the child process , fork returns with two values: Zero in the child process. The PID of the child in the parent process.

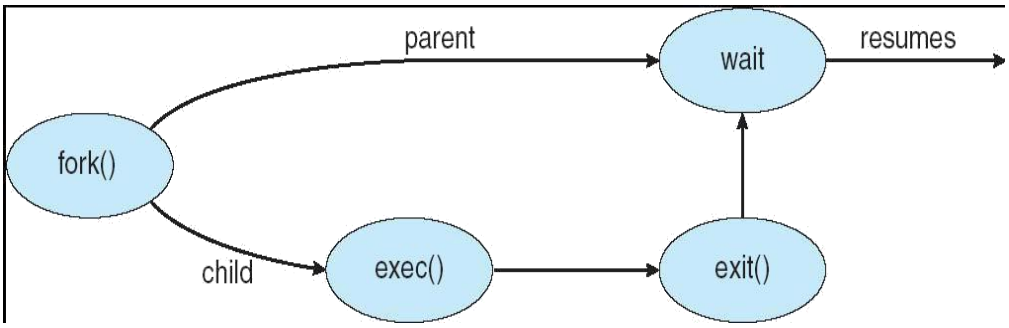
Example :

```
# include
<stdio.h> main()
{
int pid;
pid = fork();
if ( pid == 0)
{
// this is the child process;
}
else
{
// This is the Parent process.

}
```

SYSTEM CALLS USED:

1. **fork ( )** Used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.



2. **wait( )** The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

Syntax : wait(NULL)

3. **exit( )** A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

Syntax:#include<unistd.h>

exit(int status)

4. **sleep()**-sleep for the specified number of seconds

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

Orphan Process:

In orphan process is a computer process whose parent process has finished or terminated, though it remains running itself.In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called re-parenting and occurs automatically. Even though technically the process has the init process as its parent, it is still called an orphan process since the process that originally created it no longer exists.

Zombie Process:

On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still **needed to allow the parent process to read its child's exit status. The term zombie process derives** from the common definition of zombie — an undead person. **In the term's metaphor**, the child process has **"**died**"** but has not yet been **"**reaped**"**. Also, unlike normal processes, the kill command has no effect on a zombie process

EXEC FAMILY:

There are six members in exec family, out of which five are library functions and one is a system call. Ultimately, all the library functions use the system call for their needs. The six library functions are

execl(char * pathname , const char *arg1…….const char *argn , (char *)0)
execv(char *pathname , char *const arg[])
execlp(ch**ar *filename , const char *arg1…….const char *argn ,** (char *)0)
execvp(char *filename , char *const arg[])
execle(char *pathname , const char *arg1…….const char *argn , (char *)0 , char * const env[]) execve(char *pathname , char *const arg[] , char * const env[])

1.execl(): The first member, which is execl, takes the full pathname of the script or command as its first argument, the arguments to be passed for that script or command as its subsequent arguments, and at last, it also accepts a null pointer which marks the end of arguments. The following program explains, how execl can be used.

2.execlp( ) Used after the fork() system call by one of the two p**rocesses to replace the process‟** memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution.The child process overlays its address space with the UNIX command /bin/ls using the execlp system call.

3. execv(): There is only a minor difference between execl and execv. After seeing the following example, you will get to know on your own.

4.execvp(): The exec family of functions replaces the current process image with a new process image. The functions execvp will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash (/) character. The search path is the path specified in the environment by the PATH variable. If this variable isn't specified, the default path ``/bin:/usr/bin:'' is used .

Pseudocode (Orphan process)

1. Create a child process
2. Child process is suspended for some time so that the parent process terminates to demonstrate an orphan process
3. The init process becomes the parent of the orphan process

Pseudocode (Zombie process)

1. Create a child process
2.Parent process is suspended for some time so that the Child process terminates to demonstrate the Zombie process

Problem Validation

Output 1(orphan process)

Child processed:34567 childs parent process id:344567

Parent process ppid: 344567

Orphan process parent is init process whose id is 1

Output 2(zombie process)

Flag	S	uid	pid	ppid	c	PRI	tty	time	cmd
I	z	501	201	202	0	76	pts/3	2:00:00	a.out

Program 7
Program for creating and manipulating threads

Problem Definition
Program for Demonstrating creation and manipulation of threads.

Problem Description
When multiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities

pthread_self(): This function returns the ID of the calling thread. #include <pthread.h>
pthread_t pthread_self(void);

pthread_exit(): This function terminates the calling thread andmakesthe value *value_ptr* available toany successful join with the terminatingthread.

#include <pthread.h>
void pthread_exit(void **value_ptr*);

pthread_create() : This function creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function start_routine passing it arg as first argument. The new thread terminates either explicitly, by calling pthread_exit(), or implicitly, by returning from the start_routine function. The latter case is equivalent to calling pthread_exit() with the result returned by start_routine as exit code.The attr argument specifies thread attributes to be applied to the new thread.

#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *(*start_routine)(void *), void * arg);

pthread_join()
pthread_join suspends the execution of the calling thread until the thread identified by **th** terminates, either by calling pthread_exit() or by being cancelled. If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return. The return value of th is either the argument it gave to pthread_exit(), or PTHREAD_CANCELED if th was cancelled.The joined thread th must be in the joinable state.
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);

Pseudocode -
1: Create a thread and prints its id
2:Assign a task to a thread (sum of first 10 numbers)
3: Wait for thread to terminate and then print the sum

Problem Validation

Output

Compile the thread program by linking –l to the file cc –lpthread program7.c

Thread task id is 1209059648

pthread-create succ execution.

Thread task is 308904800

Sum is 45

Program 8
Program to demonstrate usage of Pipes, Shared memory, Message Queues.

Problem Definition
To write a program for demonstrating usage of Pipes, Shared memory, Message queues.

Problem Description
One of the mechanisms that allow related-processes to communicate is the pipe. A pipe is a one-way mechanism that allows two related processes (i.e. one is an ancestor of the other) to send a byte stream from one of them to the other one.The system assures us of one thing: The order in which data is written to the pipe, is the same order as that in which data is read from the pipe. The system also assures that data won't get lost in the middle, unless one of the processes (the sender or the receiver) exits prematurely.

pipe() -
This system call is used to create a read-write pipe that may later be used to communicate with a process we'll fork off. The call takes as an argument an array of 2 integers that will be used to save the two file descriptors used to access the pipe. The first to read from the pipe, and the second to write to the pipe. Here is how to use this function:

```
int fd[2];
if (pipe(fd) < 0)
perror("Error");
```

If the call to pipe() succeeded, a pipe will be created, fd[0] will contain the number of its read filedescriptor, and fd[1] will contain the number of its write file descriptor.The program first call fork() to create a child process. One (the parent process) reads write to the pipe and child process reads the data from the pipe ans then prints the data to the screen.

Message queues:

The msgget() function initializes a new message queue:

int msgget(key_t key, int msgflg)

It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflg argument must be an octal integer with settings for the queue's permissions and control flags.POSIX message queues allow processes to exchange data in the form of messages.

Shared Memory **-** is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.A process creates a shared memory segment using shmget(),The original owner of a shared memory segment can assign ownership to another user with shmctl(). It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using shmctl(). Once created, a shared segment can be attached to a process address space using shmat(). It can be detached using shmdt() The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment,

as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototype can be found in <sys/shm.h>

One shmid data structure for each shared memory segment in the system. */

```
struct shmid_ds {
        struct ipc_perm shm_perm;       /* operation perms */
        int     shm_segsz;              /* size of segment (bytes) */
        time_t  shm_atime;              /* last attach time */
        time_t  shm_dtime;              /* last detach time */
        time_t  shm_ctime;              /* last change time */
        unsigned short  shm_cpid;       /* pid of creator */
        unsigned short  shm_lpid;       /* pid of last operator */
        short   shm_nattch;             /* no. of current attaches */
`       }
```

shmget () - allocates a System V shared memory segment Syntax:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Pseudocode :(pipe)

1. Create a pipe
2. Write to the pipe
3. Read from the pipe
4. close both the ends of pipe

Pseudocode :(Message queues)

1: Create a message queue
2: Write a message to the message queue
3. Read the message from the message queue and display it
4. Delete the message queue.

Pseudocode:(Shared memory)

1: Create the shared memory
2: Attach the shared memory segment to the address space of the calling process
3:Read information from the standard input and write to the shared memory
4: Read the content of the shared memory and write on to the standard output
5: Delete the shared memory

Problem Validation

Output (pipe)

Data written to the pipe is – Hello

Data read from the pipe is – Hello

Output (Message queue)

Return val of send is 0.
Return val of receive is 56
Message read is hello

Output (Shared Memory)

Enter a message
HELLO
Data from the shared memory HELLO

Program 9
Echo server using pipes

Problem Definition

To implement two-process communication through Echo server application using pipes.

Problem Description

An echo server is usually an application which is used to test if the connection between a client and a server is successful. It consists of a server which sends back whatever text the client sends. The program shows communication between 2 processes using the IPC mechanism – pipes.
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].

Pseudocode -

1. Create two pipes pipe1 and pipe2
2. Create a child process
3. Child closes read end of pipe1
4. Child closes write end of pipe2
5. Child writes to pipe1 and waits for response from parent
6. Parent closes write end of pipe1
7. Parent closes read end of pipe2
8. Parent reads data from pipe1 and writes to pipe2 for child to read
9. Child reads data written by parent to pipe2 through read end of pipe2 and displays on console.

Problem Validation

Output-
Child process
Parent process
Child reads from pipe – Hello

Program 10
Echo server using messages queues

Problem Definition

To implement two-process communication through Echo server application using Message queues.

Problem Description

An echo server is usually an application which is used to test if the connection between a client and a server is successful. It consists of a server which sends back whatever text the client sends. The program shows communication between 2 processes using the IPC mechanism – Message queues.A message queue or queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). The messages can be fetched based on their type field.

```
#include <sys/types.h>
#include <sys/ipc.h> #include
<sys/msg.h>
int msgget(key_t key, int flags);
```

The argument key must have the value IPC_PRIVATE or a valid IPC key value. The flags argument must contain the permission bits for the new queue and IPC_CREAT if the queue is being created.

```
#include <sys/types.h>
#include <sys/ipc.h> #include
<sys/msg.h>
int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
```

The first argument msqid is the IPC ID of the message queue to send the message on.The argument msgp points to a message structure to be sent. The size of the message msgsz is the message size, not including the message type value. The msgflg argument is specified as 0 unless the flag IPC_NOWAIT is used.

The format of the message structure is shown in the next synopsis:

```
struct msgbuf { /* Message Structure */ long
mtype; /* message type */
char mtext[1]; /* body of message */ };
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

The argument msqid is the IPC ID of the message queue to receive the message from. The pointer argument msgp must point to a receiving buffer large enough to hold the received message. The argument msgsz indicates the maximum size of the received message, not including the size of the mtype member. The msgtyp and msgflg members hold the message type (priority) and option flags for this call, respectively.

To perform control operations on a message queue, including its destruction, use the msgctl() function:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

The first argument, msqid, is the message queue IPC ID. The argument cmd is a command constant, and the last argument, buf, is a pointer to a structure.The function msgctl() returns 0 when it is successful. When -1 is returned, errnoholds the error code.The operation commands accepted by this function include -

IPC_RMID Destroy the message queue.
IPC_STAT Query the message queue for information.
IPC_SET Change certain message queue attributes.

Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds
{
 struct ipc_perm msg_perm;
 msgqnum_t msg_qnum;       /* # of messages on queue */
 msglen_t msg_qbytes;      /* max # of bytes on queue */
 pid_t msg_lspid;          /* pid of last msgsnd() */
 pid_t msg_lrpid;          /* pid of last msgrcv() */
 time_t msg_stime;         /* last-msgsnd() time */
 time_t msg_rtime;         /* last-msgrcv() time */
 time_t msg_ctime;         /* last-change time */
   .
   .
   .
 };
```

The kernel maintains a structure of information for each IPC channel. Similar to the information it maintains for files.

```
struct ipc_perm
{
ushort uid;     /* owner's user id*/
ushort gid;     /* owner's group id*/
ushort cuid;     /* creator's user id*/
ushort cgid;     /* creator's group id*/
ushort mode;     /* r/w permission*/
ushort seq;      /* sequence no. */
key_t key;       /* key*/
};
```

Pseudocode -

1. Create a Message queue
2. Create a child process
3. Child process writes a message to the message queue.
4. Parent reads the message from the message queue.
5. Parent then writes the read message to the message queue.
6. Child then reads the message written by parent to the message queue and displays onto the console.

Problem Validation -

Output-
Child process
Parent process
Child reads from Message queue – Hello

Program 11
Echo server using Shared Memory

Problem Definition

To implement two-process communication through Echo server application using Shared Memory

Problem Description

An echo server is usually an application which is used to test if the connection between a client and a server is successful. It consists of a server which sends back whatever text the client sends. The program shows communication between 2 processes using the IPC mechanism – Shared Memory.

Shared Memory is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.Shared memory is created and accessed if it already exists using the shmget()function.

#include <sys/types.h>
#include <sys/ipc.h> #include
<sys/shm.h>
int shmget(key_t key, int size, int flag);

The argument key is the value of the IPC key to use, or the value IPC_PRIVATE. The size argument specifies the minimum size of the shared memory region required. The flag option must contain the permission bits if shared memory is being created. Additional flags that may be used include IPC_CREAT and IPC_EXCL, when shared memory is being created.The return value is the IPC ID of the shared memory region when the call is successful (this includes the value zero). The value -1 is returned if the call fails, with errno set.

Shared memory must be attached to process memory space before you can use it as memory.
This is performed by calling upon shmat()

#include <sys/types.h>
#include <sys/ipc.h> #include
<sys/shm.h>
void * shmat(int shmid, void *addr, int flag);

The argument shmid specifies the IPC ID of the shared memory that you want to attach to your process. The argument addr indicates the address that you want to use for this. A null pointer for addr specifies that the UNIX kernel should pick the address instead. The flag argument permits the option flag SHM_RND to be specified. Specify 0 for flag if no options apply. When shmat() succeeds, a (void *) address is returned that represents the starting address of the shared memory region. If the function fails, the value (void *)(-1) is returned instead.

Pseudocode

1. Create a Shared Memory.
2. Attach shared memory segment to address space of calling process.
3. Create a child process
4. Child process writes to the shared memory and waits for response from parent process.
5. Parent reads contents of shared memory and displays onto the console.
6. Parent then writes to the shared memory.
7. Child then reads the contents of shared memory and displays onto the console.

Problem Validation

Output-
Child process
Parent process
Child process -Welcome
Parent process-Welcome

Program 12
To implement a Program for Basic Operations and Arithmetic on Semaphores

Problem Definition
To understand the usage of semaphores

Problem Description
Semaphores are a synchronization primitive. They are intended to let multiple processes synchronize their operations .To obtain a resource that is controlled by a semaphore, a process needs to test its current value , and if the current value is greater than zero, decrement the value by one. If the current value is zero, the process must wait until the value is greater than zero. To release a resource, that is controlled by a semaphore, a process increments the semaphore value. If some other process has been waiting for the semaphore value to become greater than zero, that other process can now obtain the semaphore.A semaphore set is created or accessed by using the semget() system call. Its function synopsis is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flags);
```

The semget()function requires an IPC key value in the argument key, and permissions and flags in the argument flag. The argument nsems indicates how many semaphores you want to create in this set. For every set of semaphores in the system, the kernel maintains the following structure of information –

```
struct semid_ds {
struct ipc_perm sem_perm; /* operation permission struct */
struct sem *sem_base; /* pointer to first semaphore in set */
u_short sem_nsems; /* number of sems in set */
time_t sem_otime; /* last operation time */
time_t sem_ctime; /* last change time */ };
```

The sem structure is internal data structure used by kernel to maintain the set of values for a given semaphore.

```
struct sem
{
ushort semval;      /*semaphore value, nonnegative */
short sempid;       /* pid of last operation */
ushort semncnt;     /* # awaiting semval > cval */
ushort semzcnt;     /* # awaiting semval = 0 */
}
```

Once a semaphore is opened with semget, operations are performed on one or more of the semaphore values in the set using the semop system call.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
semop(int semid, struct sembuf *opsptr,size_t nops) ;
```

opsptr is a ptr to an array of following structure.

```
struct sembuf
{
short sem_num; /*sem_num. , 0,1…. nsem-1 */
short sem_op;        /* semaphore operation */
short sem_flg;       /* operation flags */
};
```

Pseudocode :

1. Create a semaphore.
2. Assign value to a semaphore.
3. Create a child process.
4. Child process performs a down operation on the semaphore and tries to access a variable .Until the child performs an up operation, the parent process cant execute.
5. Parent process performs a down operation on the semaphore and then performs up operation.
6. Remove the semaphore set

Problem Validation

Output-Child
process Enter
no.
2
CP values of x=2 child leaving
access Parent process
PP accessing

.

Program 13
Program for producer consumer using semaphores

Problem Definition
To implement a program for Producer consumer problem using semaphores

Problem Description
The program implements solution for the classical synchronization problem- Producer consumer using Semaphores and Shared memory. Producer-Consumer problem is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Pseudocode :

1. Create a semaphore.
2. Create a shared memory.
3. Initialize value of the semaphore to 1.
4. Create a child process.
5. Child process (Producer) performs a down operation on the semaphore and writes to the shared memory.
6. Producer performs an up operation on the semaphore for the consumer to consume.
7. Parent process (Consumer) performs a down operation on the semaphore and reads or consumes the data from the shared memory.
8. Consumer then performs an up operation.

Problem Validation

Output-

Producer CP
Enter a string
Hello
Producer CP
Enter a string
MJCET
Consumer PP Hello
Consumer PP MJCET

Program 14
Program for producer consumer problem using message passing

Problem Definition
To implement a program for producer consumer problem using message passing

Problem Description
The program implements solution for the classical synchronization problem- Producer consumer using Message queues. Producer-Consumer problem is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses Message queues.

Pseudocode :

1. Create a semaphore.
2. Create a shared buffer- Message queue.
3. Initialize value of the semaphore to 1.
4. Create a child process.
5. Child process (Producer) performs a down operation on the semaphore and writes to the Message queue.
6. Producer performs an up operation on the semaphore for the consumer to consume.
7. Parent process (Consumer) performs a down operation on the semaphore and reads or consumes the data from the Message queue.
8. Consumer then performs an up operation.

Problem Validation
Output-

```
Producer CP
Enter a string
Hello
Producer CP
Enter a string
MJCET
Consumer PP Hello
Consumer PP MJCET
```

Program 15
Program for reader and writer using semaphores

Problem Definition

To implement a program for reader and writer using semaphores

Problem Description

The program implements solution for the classical synchronization problem- Reader-Writer using Semaphore and Shared memory.

Semaphores can be used to restrict access to the database under certain conditions. In this example, semaphores are used to prevent any writing processes from changing information in the database while other processes are reading from the database.

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the readers-writers problem

In the solution to the first readers-writers problem, the reader processes share the following data structures:

semaphore mutex,
wrt; int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

Pseudocode :

```
semaphore mutex = 1; // Controls access to the reader count
semaphore wrt = 1; // Controls access to the database
int reader_count;     // The number of reading processes accessing the data

Reader()
{
while (TRUE) {                    // loop forever

down(mutex);                      // gain access to reader_count
reader_count = reader_count + 1;      // increment the reader_count

if (reader_count == 1)
```

```
down(wrt); // if this is the first process to read the database,
// a down on wrt is executed to prevent access to the
// database by a writing process

up(mutex);       // allow other processes to access reader_count

read_wrt();      // read the database

down(mutex);                    // gain access to reader_count
reader_count = reader_count - 1;      // decrement reader_count

if (reader_count == 0)
up(wrt); // if there are no more processes reading from the // database, allow writing process
to access the data

up(mutex);     // release exclusive access to reader_count

}


Writer()
{
while (TRUE) {        // loop forever
create_data();    // create data to enter into database (non-critical)
down(wrt);        // gain access to the database
write_db();       // write information to the database
up(wrt);          // release exclusive access to the database
}
```

Problem Validation

Output –

```
    Writer process
    Enter data
    Welcome to OS lab
    Writer finished
    Reader 0 is accessing data
    Welcome to OS lab
    Reader 1 is accessing data
    Welcome to OS lab
    Reader 2 is accessing data
Welcome to OS lab
```

Program 16
Program for dining philosopher problem using semaphores

Problem Definition
To implement a Program for Dining philosopher problem using Semaphores

Problem Description
The program implements Dining philosopher problem which is a circular permutation based synchronization problem using Semaphores and shared memory.The dining-philosophers problem is considered a classic synchronization because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks .When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick[5]; where all the elements of chopstick are initialized to 1. Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick

Pseudocode :

The structure of philosopher i---

```
do
{
wait (chopstick [i] );
wait(chopstick [(i + 1) % 5] ) ;
……

// eat
……..
signal(chopstick [i]);
signal(chopstick [(i + 1) % 5]);

// think

} while (TRUE);
```

Problem Validation

Output –

Philosopher 0 is thinking Philosopher 1 is thinking Philosopher 2 is thinking Philosopher 3 is thinking Philosopher 4 is thinking

Philosopher 1 has acquired chopsticks Philosopher 1 is eating

Philosopher 1 has returned chopsticks Philosopher 3 has acquired chopsticks
Philosopher 1 is thinking
Philosopher 3 is eating

……

Program 17

Program for Linux shell scripts:

a) Program to display statements

b) Program to find whether a number is odd or even c) Program to find factorial of a number

d) Program to reverse a number

SHELL PROGRAMMING

1. Shell or the Command interpreter is the mediator which interprets the commands and then conveys them to the kernel which ultimately executes them.
2. **Kernel is usually stored in a file called 'UNIX' where as the shell program in a file called 'sh'.**
3. Types of shells :-
   i. Bourne shell (sh) or Bourne again shell (bash)
   ii. C shell (csh)
   iii. Korn shell (ksh)

4. A shell program is nothing but a series of unix commands .
5. Instead of specifying one job at a time, the shell is given a to-do-list – a program - that carries out an entire procedure.
6. Such programs are known as shell scripts.
7. Shell programming language incorporates most of the features that most modern day programming languages offer.

Shell variables –

Rules for building shell variables are as follows :

1) A variable name is any combi**nation of alphabets, digits and an underscore ('_').**

2) No commas or blanks are allowed within a variable name.

3) The first character of a variable name must either be an alphabet or an underscore.

4) Variable names should be of any reasonable length.

5) Variable names are case sensitive.

Keywords for accepting input – read

displaying output - echo

Assigning value to variables –

Values can be assigned to variables through read statement or also by using a simple assignment operator. For ex: age=30

Note : While assigning values to variables using assignment operator, no spaces to be given on **either side of it. If the variable doesn't exist it will be created and value assigned**

Variables in Unix are of 2 types :

1. Unix-defined variables or System variables or Environment variables
2. User- defined variables

Note : To print or access value of a variable use '$' .
For ex: To print value of variable 'flag' write - echo $flag

Arithmetic in Shell script -

1. All shell variables are string variables, hence to carry out arithmetic operations use expr command which evaluates arithmetic expressions.
2. More than one assignment can be done in a single statement.
3. Before and at the end of expr keyword use ` (back quote) sign not the (single quote i.e. ') sign which is generally above TAB key.
4. Terms of the expression provided to expr must be separated by blanks. Thus expression expr 10+20 is invalid.
5. **The '*"** symbol must be preceded by a \ ,otherwise the shell treats it as a wildcard character for all files in the current directory

OPERATORS USED IN SHELL SCRIPT –

OPERATOR	MEANING
–gt	Greater than
–lt	Less than
–ge	Greater than or equal to
–le	Less than or equal to
–ne	Not equal to
–eq	Equal to
–a	Logical AND
–o	Logical OR
!	Logical NOT

CONTROL INSTRUCTIONS IN SHELLS -

There are four types of control instructions in shell :

- Sequence Control Instruction.
- Selection or Decision control Instruction
- Repetition or Loop control Instruction
- Case Control Instruction

Decision statements –

<u>If-then-else-fi statements:</u>

if condition then Commands else Commands fi

1) For statements :
for control variable in value1 value 2 value3 do
Command list done

2) While statements : while control command do
Command list Done

3) Until statements : until control command do
Command list done

Case statements :-case value in choice1) commands; choice2)
commands; esac.

Problem Definition
a)To write a shell script to display statements

Pseudocode

echo "What is your name? " read name
echo "Hello $name Welcome to shell programming“

Problem Validation

Output-

$ vi SS1.sh $ sh SS1.sh
$ What is your name John
Hello John Welcome to shell programming

Problem Definition

b)To write a shell script to find whether a number is odd or even

Pseudocode

```
echo "Enter a number " read n
 n1=$(expr $n % 2) if [ $n1 -eq 0 ]
       then
  echo "no. is even" else
  echo "no. is odd" fi
```

Problem Validation

Output -

$ sh SS2.sh
$ Enter a number 16
no. is even

$ sh SS2.sh
$ Enter a number 13
no. is odd

Problem Definition
c) To write a shell script to find to find factorial of a number

Pseudocode

```
f=1  i=1
  echo "enter a number " read n
  while [ $i -le $n ] do
  f=$(expr $f \* $i) i=$(expr $i + 1) done
 echo "factorial is $f"
```

Problem Validation
Output -

```
$ sh SS3.sh
$ Enter a number 5
factorial is 120
```

Problem Definition
d) To write a shell script to find to find reverse of a number

Pseudocode

```
rev=0
  echo "enter a number " read n
  while [ $n -gt 0 ] do
     m=$(expr $n % 10) rev=$(expr $rev \* 10 + $m)
     n=$(expr $n / 10 )
 done
 echo "reverse of given no. is $rev"
```

Problem Validation
Output -

```
$ sh SS4.sh
$ Enter a number 137
reverse of given no. is 731
```

Annexure – I

List of programs according to O.U. curriculum

BIT 331	**OPERATING SYSTEMS LAB**
Instruction	*3 Periods per week*
Duration	3 Hours
University Examination	50 Marks
Sessional	25 Marks

1. Familarity and usage of system calls of LINUX/WINDOWS NT on process management fork(), exec() etc IPC & Synchoronization-pipes, shared memory, messages, semaphores etc. , File management-read, write etc.
2. Creating Threads and Manipulating under Windows-NT platform.
3. Implementing a program to get the attributes of a file/Directory on Linux using related system calls.
4. Implementing a program to get and set the environment variables using system calls.
5. Implementation of Echo server using pipes.
6. Implementation of Echo server using shared memory.
7. Implementation of Echo server using Messages.
8. Implementing Producer Consumer Problem using semaphores.
9. Implementing Producer Consumer Problem using Message passing.
10. Implementing Reader-writers problem using Semaphores.
11. Implementing Dining philosophers problem using semaphores.
12. Implementing Dinning philosophers problem using Windows-NT threads.
13. Implementation of Limited shell on Linux platform.

Suggested Reading:

1. W. Richard Stevens, Unix Network Programming, Prentice Hall/Pearson Education,2009.