OBJECTIVE OF OPERATING SYSTEM LAB

The main objective of this lab is to introduce the concept of processes in UNIX and UNIX like operating systems and also to make the students aware of the features and capabilities of Unix/Linux so that they can utilize its improved functionalities to develop new Unix/Linux based softwares and can also contribute to the development of the operating system itself.

In this lab you are going to learn about

- The working of **fork system calls** (what is fork system call, why is it used). How to use a **single fork** system call, **multiple fork** system call, how to use fork to calculate two different task (i.e. child calculates sum of even numbers and parent calculates sum of odd numbers). The working and implementation of **exit ()** and **wait ()** system call. To introduce the concept of **orphan** and **zombie** processes.

- To introduce the concept of **IPC**'s (inter process communication) or process to process communication. You will learn different methods for implementing process to process communication. Following system calls are used for communication

  i) **pipe**

  ii)**FIFO**

  iii)**message queue**

  iv)**shared memory**

  v)**semaphore**

- Further you will be taught about how to execute a process using **execl()** system call and its implementation.

- A real-time example of pipe system call for communication between processes used in unix (implementation **of ls | wc** using pipes.)

- You will learn about scheduling algorithms of CPU and the need of these algorithms. Different types of scheduling algorithms implemented are **FCFS, SJFS and priority scheduling**.

- You will learn about Page replacement algorithms and the need of these algorithms. Different types of Page replacement algorithms implemented are **FIFO, LRU**.

PRE-REQUISITES FOR THIS LAB:

The student must have a good knowledge about C programming language and how to use a UNIX operating system. He must be familiar with the concepts of UNIX operating system and should be able to perform basic operations like creating, reading, writing, editing files in

UNIX operating system**.**

TOOLS REQUIRED:

Any variant of UNIX or UNIX like operating system.

INDEX

WEEK 1

Demonstration of FORK() System Call

FORK FUNCTION:

An existing process can create a new one by calling the **fork** function .The new process created by **fork** is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call **getppid** to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to **fork.** The child is a copy of the parent.

SYSTEM CALL USED:

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 .This can be done by testing the returned value of fork() .

- If fork() returns a negative value, the creation of process was unsuccessful.
- If fork() returns zero(0), then child has created.
- If fork() returns positive value, which is the process ID of the child process, to the parent.

PROGRAM:

SINGLE FORK

```
#include<stdio.h>                                   //HEADER FILES
#include<unistd.h>
int main()                                          //MAIN FUNCTION
{
        fork();         //CALLING FORK TO CREATE A CHILD PROCESS
        printf("LINUX\n");
        return 0;
}                                                   //END OF MAIN
```

<u>OUTPUT:</u>

```
LINUX
LINUX
```

<u>PROGRAM:</u>

MULTI TIME FORK

```
#include<stdio.h>                                   //HEADER FILES
#include<unistd.h>
int main()                                          //MAIN FUNCTION
{
        fork();         //CALLING FORK TO CREATE A CHILD PROCESS
        printf("LINUX\n");
        fork();         //CALLING FORK TO CREATE A CHILD PROCESS
        printf("UNIX\n");
        fork();         //CALLING FORK TO CREATE A CHILD PROCESS
        printf("RED HAT\n");
        return 0;
}                                                   //END OF MAIN
```

<u>OUTPUT:</u>

```
LINUX
LINUX
UNIX
UNIX
RED HAT
RED HAT
RED HAT
UNIX
RED HAT
RED HAT
```

UNIX
RED HAT
RED HAT
RED HAT

WEEK 2

Parent Process Computes the SUM OF EVEN and Child Process Computes the sum of ODD NUMBERS using fork

Here,we are using the concept of fork. When a child process is created, then sum of even and odd numbers in an array is calculated by child and parent process separately.

First of all, we declare an array which takes integer values. Then we enter the elements in this array. The child process is created by calling fork() system call .The returned value of fork in child is 0 whereas in parent ,it returns the PID of child process. Then we check the condition **(if (pid==0))** ,if it is true, it means child process is created. Now child process calculate sum of even numbers and parent calculate sum of odd numbers.

SYSTEM CALL USED:

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 .This can be done by testing the returned value of fork().

The fork operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process, though if **copy-on-write** semantics are implemented actual physical memory may not be assigned (i.e., both processes may share the same physical memory segments for a while). Both the parent and child processes possess the same code segments, but execute independently of each other.

exit()

ISOC defines exit to provide a way for a process to terminate without running exit handler or signal handlers. Whether or not standard I/O streams are flushed depends on the implementation.

PROGRAM:

```
#include<stdio.h>                           //HEADER FILES
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
#define max 20                              //SYMBOLIC CONSTANTS
int main()                                  //MAIN FUNCTION
{
pid_t pid;
int a[max],n,sum=0,i,status;
printf("\nEnter the no of terms in the array :");
scanf("%d",&n);
printf("\nEnter values in the array : ");
for(i=0;i<n;i++)                        //LOOP
{
        scanf("%d",&a[i]);
        pid=fork();
        wait(&status);                      //WAITING FOR STATUS
        if(pid==0)
        {
                for(i=0;i<n;i++)            //LOOP
                {
                        if(a[i]%2==0)
                        {
                                sum=sum+a[i];       //CALCULATING SUM OF EVEN
                        }
                        printf("Sum of even nos = %d\n",sum);

                }
                exit(0);

        }

        else
        {
                for(i=0;i<n;i++)            //LOOP
                {
                        if(a[i]%2!=0)
                        {
                                sum=sum+a[i];       //CALCULATING SUM OF ODD
                        }
                        printf("Sum of odd nos = %d\n",sum);
                }
        }
```

```
    }
return 0;
}                                                  //END OF MAIN
```

OUTPUT:

```
Enter the no of terms in the array :6

Enter values in the array : 1 2 3 4 5 6
Sum of even nos = 12
Sum of odd nos = 9
```

WEEK 3

Demonstration of WAIT() System Call

WAIT:

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the **SIGCHLD** signal to the parent. Because the termination of a child is an asynchronous event it can happen at any time while the parent is running this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. A process that calls wait() or waitpid() can block, if all of its children are still running

- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched .
- Return immediately with an error, if it doesn't have any child processes.
- The wait function can block the caller (parent) until a child process terminates. If a child has already terminated , the child is called is a zombie, otherwise wait returns immediately with that child's status.

SYSTEM CALLS USED:

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() s/m call.This can be done by testing the returned value of fork()

- If fork() returns a negative value, the creation of process was unsuccessful.
- If fork() returns zero(0), then child has created.

- fork() returns positive value, the process ID of the child process, to the parent.

Wait()

The wait function can block the caller until a child process terminates**.** If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates.

PROGRAM:

```
#include<stdio.h>                                  //HEADER FILES
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()                                         //MAIN FUNCTION
{
        pid_t pid;
        pid=fork();                //CALLING FORK TO CREATE A CHILD PROCESS
        if(pid==0)                                 //IN CHILD PROCESS
        {
                printf("I m Child\n");
                exit(0);
        }

         else                                      //IF PARENT PROCESS
        {
                wait(&status);                     // WAITING FOR CHILD STATUS
                printf("I m Parent\n");
                printf("The Child PID =%d\n", pid);
        }
        return 0;
}                                                  //END OF MAIN
```

OUTPUT:

```
I m Child
I m Parent
The Child PID =2503
```

WEEK 4

Implementation of ORPHAN PROCESS & ZOMBIE PROCESS

ORPHAN PROCESS:

An orphan process in unix and unix like operating system , is a computer process whose parent process has finished or terminated, through itself remains running . A process can become orphaned during remote invocation when the client process crashes after making a request to server.

A process can also be orphaned running on the same machine as its parent process. In a UNIX-like operating system any orphaned process will be immediately adopted by the special “**init**” process as its parent, it is still called an orphan process since the process which originally created no longer exists.

SYSTEM CALLS USED:

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() s/m call.This can be done by testing the returned value of fork()

- If fork() returns a negative value, the creation of process was unsuccessful.
- If fork() returns zero(0), then child has created.
- fork() returns positive value, the process ID of the child process, to the parent.

getpid()

getpid() returns the process ID of the calling process. The ID is guaranteed to be unique and is useful for constructing temporary file name.

getppid()

getppid() returns the process ID of the calling process.

sleep()

A typical sleep system call takes a time value as a parameter, specifying the minimum amount of time that process is to sleep before execution. The parameter typically specifies seconds, although some OS provides finer resolution, such as milliseconds or microseconds.

PROGRAM:

```
#include<stdio.h>                                   //HEADER FILES
#include<unistd.h>
int main()                                          //MAIN FUNCTION
{
        pid_t pid;
        pid=fork();                 //CALLING FORK TO CREATE A CHILD PROCESS




        if(pid==0)
        {
                sleep(6);
                printf("\n I m Child. My PID = %d And PPID = %d",
getpid(),getppid());
        }
        else
        {
                printf("I m Parent. My Child PID = %d And my PID =
%d",pid,getpid());
        }
        printf("\nTerminating PID = %d\n",getpid());

   return 0;
}                                                   //END OF MAIN
```

OUTPUT:

```
I m Parent. My Child PID = 2753 And my PID = 2752
Terminating PID = 2752
I m Child. My PID = 2753 And PPID = 1
Terminating PID = 2753
```

ZOMBIE PROCESS:

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 process that started the (new zombie) process to read its exit status. The term zombie process derives from the common definition of zombie an undead process.

A zombie process is not the same as an orphan process. An orphan process is a process that is still executing, but whose parent has died. They do not become zombie processes; instead, they are adopted by init (process ID 1), which waits on its children .

When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, at which stage the zombie is removed.

SYSTEM CALLS USED:

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() s/m call.This can be done by testing the returned value of fork()

- If fork() returns a negative value, the creation of process was unsuccessful.
- If fork() returns zero(0), then child has created.
- fork() returns positive value, the process ID of the child process, to the parent.

sleep()

A typical sleep s/m call takes a time value as a parameter, specifying the minimum amount of time that process is to sleep before execution. The parameter typically specifies seconds, although some OS provides finer resolution, such as milliseconds or microseconds.

exit()

ISOC defines exit to provide a way for a process to terminate without running exit handler or signal handlers. Whether or not standard I/O streams are flushed depends on the implementation.

PROGRAM:

```
#include<stdio.h>                              //HEADER FILES
#include<unistd.h>
#include<stdlib.h>
int main()                                     //MAIN FUNCTION
{
        pid_t pid;
        pid=fork();                 //CALLING FORK TO CREATE A CHILD PROCESS
        if(pid!=0)
        {
                while(1)
                sleep(50);
        }
        else
        {
                exit(0);
        }
}                                              //END OF MAIN
```

OUTPUT:

```
unix user@ylmfos:~$ ./pb1 & [1]
2761
unix user@ylmfos:~$ ps
  PID TTY          TIME CMD
 2611 pts/0  00:00:00 bash
 2761 pts/0  00:00:00 pb1
 2762 pts/0  00:00:00 pb1 <defunct>
 2763 pts/0  00:00:00 ps
unix user@ylmfos:~$ kill 2761
unix user@ylmfos:~$ ps
```

```
  PID TTY          TIME CMD
 2611 pts/0    00:00:00 bash
 2764 pts/0    00:00:00 ps
[1]+ Terminated              ./pb1
```

WEEK 5

Implementation of PIPE

INTER-PROCESS COMMUNICATION

Inter-process communication is a " process to process " communication in a single system.

5 types of IPC's are:

- PIPE
- FIFO
- MESSAGE QUEUE
- SHARED MEMORY
- SEMAPHORE

PIPE:

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. PIPE is created by calling pipe function. Two file descriptors are returned through the fd argument: fd[0] is open for reading, and fd[1] is open for writing. The output of fd[1] is the input for fd[0].

SYSTEM CALLS USED:

pipe():

Pipe is created by calling pipe function. Two file descriptors are returned through file descriptor argument: fd[0] is open for reading and fd[1] is open for writing. The o/p of fd[1] is i/p for fd[0].

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() s/m call.This can be done by testing the returned value of fork()

- If fork() returns a negative value, the creation of process was unsuccessful.
- If fork() returns zero(0), then child has created.
- fork() returns positive value, the process ID of the child process, to the parent.

exit()

ISOC defines exit to provide a way for a process to terminate without running exit handler or signal handlers. Whether or not standard I/O streams are flushed depends on the implementation.

PROGRAM:

```
#include<stdio.h>                              //HEADER FILES
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main()                                     //MAIN FUNCTION
{
        pid_t pid;
        char arr[100],str[100];
        int fd[2],nbr,nbw;
        pipe(fd);                    //CREATING A PIPE
        pid=fork();                  //CALLING FORK TO CREATE A CHILD PROCESS
        if(pid==0)
        {
                printf("\nEnter a string: ");
                gets(str);
                nbw=write(fd[1],str,strlen(str));
                printf("Child wrote %d bytes\n",nbw);
                exit(0);
        }
        else
        {
                nbr=read(fd[0],arr,sizeof(arr));
                arr[nbr]='\0';
                printf("Parent read %d bytes : %s\n",nbr,arr);
        }
        return 0;
}                                                   //END OF MAIN
```

OUTPUT:

```
Enter a string: Graphic Era
Child wrote 11 bytes
Parent read 11 bytes : Graphic Era
```

WEEK 6

Implementation of FIFO

FIFO:

FIFOs are sometimes called named pipes.Pipes can be used only between related processes but with FIFO unrelated data process can exchange data. We can create a FIFO using

- mkfifo function
- mknod function

The specification of the mode argument for the mkfifo function is the same as for the open function. Once we have used mkfifo to create a FIFO, we open it using open. Indeed, the normal file I/O functions (close, read, write, unlink, etc.) all work with FIFOs.

There are two uses for FIFOs.

- FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- FIFOs are used as rendezvous points in client server applications to pass data between the clients and the servers.

SYSTEM CALLS USED:

mkfifo() or mknod():

mkfifo() or mknod() is use to create the fifo file. Once we have used mkfifo to create fifo, we open it using open. Mknod() takes three argument filename,S_IFIFO|permission set and flag.

mknod("myfifo",S_IFIFO|0666,0);

PROGRAM:

FIFO (WRITER PROCESS):

```
#include<stdio.h>                                    //HEADER FILES
#include<string.h>
#include<sys/stat.h>
#include<fcntl.h>
```

```
intmain()                                        //MAIN FUNCTION
{
        int fd ,nbw;
        char str[100];
        mknod("myfifo",S_IFIFO|0666,0);
        printf("Writing for reader Process:\n\t");
        fd=open("myfifo",O_WRONLY);
        while(gets(str))
        {
                nbw=write(fd,str,strlen(str));
                        printf("Writer process write %d bytes: %s\n",nbw,str);
        }
        return 0;
}                                                //END OF MAIN
```

<u>FIFO (READER PROCESS):</u>

```
#include<stdio.h>                                //HEADER FILES
#include<string.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()                                       //MAIN FUNCTION
{
        int fd ,nbr; char
        arr[100];
        mknod("myfifo",S_IFIFO|0666,0);
        fd=open("myfifo",O_RDONLY);
        printf("If you got a writer process then type some data \n");

        do
        {
                nbr=read(fd,arr,sizeof(arr));
                arr[nbr]='\0';
                printf("Reader process read %d bytes: %s\n",nbr,arr);
        }while(nbr>0);

        return 0;
}                                        //END OF MAIN
```

<u>OUTPUT:</u>

TERMINAL 1:	TERMINAL 2:
Writing for reader Process: GRAPHIC ERA Writer process write 11 bytes: GRAPHIC ERA	If you got a writer process then type some data Reader process read 11 bytes: GRAPHIC ERA

LIMITATIONS

i)Both processes need to be active or online

ii)Pipe or FIFO do not distinguish between multiple write operations

WEEK 7

Implementation of MESSAGE QUEUE

MESSAGE QUEUE:

A message 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(). We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

When msgsnd() returns successfully, the msgid structure associated with the message queue is updated to indicate the process ID that made the call, the time that the call was made the call, and that one more message is on the queue. Messages are retrieved from a queue by msgrcv().

SYSTEM CALLS USED:

ftok():

ftok() is use to generate a unique key.

msgget():

msgget() either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.

msgsnd():

Data is placed on to a message queue by calling msgsnd(). The first argument to msgsnd() is queue identifier, returned by previous call to msgget(). The pointer argument points to a long integer followed by a data buffer for the actual message data.

msgrcv():

usingmsgrcv() messages are retrieved from a queue.

msgctl():

it performs various operations on a queue. Generally it is use to destroy message queue.

PROGRAM:

MESSAGE QUEUE (WRITER PROCESS):

```
#include<stdio.h>                                   //HEADER FILES
#include<string.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>

structmsgbuf                                        //STRUCTURE
{
        long mtype;
        char mtext[100];
}svarname;
int main()                                          //MAIN FUNCTION
{
        key_t key; int
        msgid ,c;
        key=ftok("progfile",'A');
        msgid=msgget(key,0666|IPC_CREAT);
        svarname.mtype=1;
        printf("\nEnter a string : ");
        gets(svarname.mtext);
        c=msgsnd(msgid,&svarname,strlen(svarname.mtext),0);
        printf("Sender wrote the text :\t %s \n",svarname.mtext);
        return(0);
}                                                   //END OF MAIN
```

MESSAGE QUEUE (READER PROCESS):

```
#include<stdio.h>                                   //HEADER FILES
#include<string.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
structmsgbuf                                        //STRUCTURE
{
        long mtype;
        char mtext[100];
}svarname;
int main()                                          //MAIN FUNCTION
```

```
{
        key_t key; int
        msgid ,c;
        key=ftok("progfile",'A');
        msgid=msgget(key,0666|IPC_CREAT);
        msgrcv(msgid,&svarname,sizeof(svarname),1,0);
        printf("Data Received is : \t %s \n",svarname.mtext);
        msgctl(msgid,IPC_RMID,NULL);
        return 0;
}                                                       //END OF MAIN
```

OUTPUT:

TERMINAL 1:	TERMINAL 2:
Enter a string : OPERATING SYSTEM Sender wrote the text : OPERATING SYSTEM	Data Received is : OPERATING SYSTEM

WEEK 8

Implementation of SHARED MEMORY

SHARED MEMORY:

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are requied[2 read and 2 write].So, shared memory provides a way by letting two or more processes share a memory segment.

With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED:

ftok():

ftok() is use to generate a unique key.

shmget():

intshmget(key_t,size_tsize,intshmflg);

upon successful completion, shmget() returns an identifier for the shared memory segment.

shmat():

Before you can use a shared memory segment, you have to attach yourself to it using shmat().

void *shmat(int shmid ,void *shmaddr ,int shmflg);

shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

shmdt():

When you're done with the shared memory segment, your program should detach itself from it using shmdt().

    intshmdt(void *shmaddr);

shmctl():

when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used.

    shmctl(int shmid,IPC_RMID,NULL);

PROGRAM:

SHARED MEMORY (WRITER PROCESS):

```
#include<stdio.h>                                   //HEADER FILES
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
int main()                                          //MAIN FUNCTION
{
        key_t   key;
        int   shmid;
        void *ptr;
        key=ftok("shmfile",'A');
        shmid=shmget(key,1024,0666|IPC_CREAT);
        ptr=shmat(shmid,(void *)0,0); printf("\nInput Data
        : ");
        gets(ptr);
        shmdt(ptr);
        return 0;
}                                                   //END OF MAIN
```

SHARED MEMORY (READER PROCESS):

```
#include<stdio.h>                                   //HEADER FILES
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
int main()                                          //MAIN FUNCTION
{
        key_t   key;
        int    shmid;
        void *ptr;
        key=ftok("srfile",'A');
        shmid=shmget(key,1024,0666|IPC_CREAT);
        ptr=shmat(shmid,(void *)0,0);
        printf("\nThe Data stored : %s\n",ptr);
        shmdt(ptr);
        shmctl(shmid,IPC_RMID,NULL); return(0);

}                                                   //END OF MAIN
```

OUTPUT:

TERMINAL 1:	TERMINAL 2:
Input Data : OPERATING SYSTEM	The Data stored : OPERATING SYSTEM

SHARED MEMORY (READER AND WRITER PROCESS TOGETHER):

```
#include<stdio.h>                                   //HEADER FILES
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
int main()                                          //MAIN FUNCTION
{
        key_t key;
        int shmid;
        void *ptr;
        key=ftok("srfile",'A');
        shmid=shmget(key,1024,0666|IPC_CREAT);
        ptr=shmat(shmid,(void *)0,0); printf("\nInput Data
        :");
        gets(ptr);
        printf("\nThe Data stored : %s\n",ptr);
        shmdt(ptr);
        shmctl(shmid,IPC_RMID,NULL); return(0);

}                                                   //END OF MAIN
```

OUTPUT:

```
Input Data : OPERATING SYSTEM
The Data stored : OPERATING SYSTEM
```

WEEK 09

SCHEDULING ALGORITHMS:

FIRST COME FIRST SERVED(FCFS)

By far the simplest, CPU-scheduling algorithm is the first come,first served(FCFS) algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting i/o.

PROGRAM:

USING POINTERS:

```
#include<stdio.h>                              //HEADER FILES
#include<malloc.h>
#include<string.h>
typedefstruct node                             //STRUCTURE
{
        char prss[3];
        int burst;
        int arrival;
        struct node *next;
}node;
node *front=NULL;                              //GLOBAL VARIABLES
node *rear=NULL;
void insert();                                 //FUNCTION DECLARATION
void display(int);
void main()                                    //MAIN FUNCTION
{
        int i ,n;
        printf("\nEnter number of processes : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)                       //LOOP
```

```
                insert();                               //FUNCTION CALL
        printf("\n\nExecuting processes : \n");
                display(n);                             //FUNCTION CALL
        printf("\n");
}                                                       //END OF MAIN

void insert()                                           //FUNCTION DEFINITION
{
        node *p; int
        b ,a; char
        str[3];
        p=(node*)malloc(sizeof(node)); //DYNAMIC MEMORY
ALLOCATION
        printf("\n\tEnter the process name : ");
        scanf("%s",p->prss);
        printf("\tEnter Burst time : ");
        scanf("%d",&b); printf("\tEnter
        arrival time : "); scanf("%d",&a);

        p->burst=b; p-
        >arrival=a; p-
        >next=NULL;
        if(front==NULL)
        {
                front=p;
                rear=p;
        }
        else
        {
                rear->next=p;
                rear=p;
        }
}
void display(int n)                                     //FUNCTION DEFINITION
{
        node *temp=front;
        intwttime=0,c=0;
        float turn=0.0;
        if(front!=NULL)
        {

                            printf("\n---------------------------------------------------------------
\n\t");
                while(temp!=NULL)
                {
                        printf("|\t%s\t",temp->prss);
                        temp=temp->next;
                }
                printf("|\n-----------------------------------------------------------
--\n\t");
                temp=front;
                while(temp!=NULL)
              {
printf(" \t%d\t ",temp->burst);
temp=temp->next;
              }
                            printf("\n-------------------------------------------------------------
--\n\t");
                temp=front;
                printf("0\t");
                while(temp!=NULL)
              {
                        wttime+=c;
                        turn+=c+temp->burst;
                        c=c+temp->burst;
                        printf(" \t%d\t ",c);
                        temp=temp->next;
              }
                            printf("\n-------------------------------------------------------------
--\n");
                printf("\n\nAveragewt time = %d ",wttime/n);
                printf("\nTurnaround time = %f\n",turn/n);
        }
}
```

PROGRAM:

USINGARRAYS:

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
char p[10][5],temp[5]; int
c=0,pt[10],i,j,n,temp1; float
bst=0.0,turn=0.0; clrscr();

printf("enter no of processes:");
scanf("%d",&n); for(i=0;i<n;i++)

{
printf("enter process%d name:\n",i+1);
scanf("%s",&p[i]);
printf("enter process time");
scanf("%d",&pt[i]);
}
printf("\n.................................................\n");
for(i=0;i<n;i++)
 {
                printf("|\t %s\t",p[i]);
 }
printf("|\n.................................................\n");
for(i=0;i<n;i++)
      {
        printf("\t\t%d",pt[i]);
      }
printf("\n.................................................\n");
printf("0");
for(i=0;i<n;i++)
      {
                bst+=c;
                turn+=c+pt[i];
                c=c+pt[i];
                printf("\t\t%d",c);
      }
printf("\nAverage time is %f: ",bst/n);
printf("\nTurn around time is %f", turn/n);
getch();
      }
```

<u>OUTPUT:</u>

```
Enter number of processes : 3

        Enter the process 1 name : P1
        Enter Burst time : 24
        Enter arrival time : 0

        Enter the process 2 name : P2
        Enter Burst time : 3
        Enter arrival time : 0

        Enter the process 3 name : P3
        Enter Burst time : 3
        Enter arrival time : 0
```

```
Executing processes :

------------------------------------------------------------------------------------------
         |        P1   |    P2   |    P3   |
------------------------------------------------------------------------------------------
                  24              3                3
------------------------------------------------------------------------------------------
         0                24              27               30


Average wt time = 17
Turnaround time = 27.000000
```

WEEK 10

SHORTEST-JOB-FIRST-SCHEDULING(SJFS)

A different approach to CPU scheduling is the shortest-job-first-scheduling(SJFS) algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next bursts of two processes are the same,FCFS is used to break the tie.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the long process. Consequently, the average waiting time decreases.

PROGRAM:

USING POINTERS:

```
#include<stdio.h>                                   //HEADER FILES
#include<malloc.h>
#include<string.h>
typedefstruct node                                  //STRUCTURE
{
        char prss[3];
        int burst;
        struct node *next;
}node;
node *front=NULL;                                   //GLOBAL VARIABLES
node *rear=NULL;
void insert();                                      //FUNCTION DECLARATION
void display(int);
void main()                                         //MAIN FUNCTION
{
        inti,n;
        printf("\nEnter number of processes : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)                            //LOOP
                insert();                           //FUNCTION CALL
        printf("\n\nExecuting processes : \n");
                display(n);                         //FUNCTION CALL
        printf("\n");
}                                                   //END OF MAIN

void insert()                                       //FUNCTION DEFINITION
```

```
{
	node *p,*temp;
	int b;
	p=(node*)malloc(sizeof(node));			//DYNAMIC MEMORY
ALLOCATION
	printf("\n\tEnter the process name : ");
	scanf("%s",p->prss);
	printf("\tEnter Burst time : ");
	scanf("%d",&b);
	p->burst=b;
	p->next=NULL;
	if(front==NULL)					//IF FIRST ELEMENT
	{
		front=p;
		rear=p;
	}
	else if( p->burst < front->burst)
	{
		p->next=front;
		front=p;
	}
	else if( p->burst > rear->burst)
	{
		rear->next=p;
		rear=p;
	}
	else
	{
		temp=front;
		while( p->burst > (temp->next)->burst )
			temp=temp->next;
		p->next=temp->next;
		temp->next=p;
	}
}

void display(int n)					//FUNCTION DEFINITION
{
	node *temp=front;
	int c=0;
	float turn=0.0,wttime=0.0;
	if(front!=NULL)
	{

				printf("\n--------------------------------------------------\n\t");
```

```
                while(temp!=NULL)
                {
                        printf("|\t%s\t",temp->prss);
                        temp=temp->next;
                }
                              printf("\n----------------------------------------------------------
--\n\t");
                temp=front;
                while(temp!=NULL)
                   {
                        printf(" \t%d\t ",temp->burst);
                        temp=temp->next;
                      }
                              printf("\n-----------------------------------------------------------
-\n\t");
                temp=front;
                printf("0\t");
                while(temp!=NULL)
                    {
                        wttime+=c;
                        turn+=c+temp->burst;
                        c=c+temp->burst;
                        printf(" \t%d\t ",c);
                        temp=temp->next;
              }
                              printf("\n-----------------------------------------------------------
------\n");
                printf("\n\nAveragewt time = %f ",wttime/n);
                printf("\nTurn around time = %f\n",turn/n);
        }
}
```

PROGRAM:

USINGARRAYS:

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
char p[10][5],temp[5]; int
```

```
c=0,pt[10],i,j,n,temp1; float
bst=0.0,turn=0.0; clrscr();
printf("enter no of processes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter process%d name:\n",i+1);
scanf("%s",&p[i]);
printf("enter process time");
scanf("%d",&pt[i]);
}
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(pt[i]>pt[j])
{
temp1=pt[i];
pt[i]=pt[j];
pt[j]=temp1;
strcpy(temp,p[i]);
strcpy(p[i],p[j]);
strcpy(p[j],temp);
}
}
}
printf("\n......................................................\n");
for(i=0;i<n;i++)
        {
                printf("|\t %s\t",p[i]);
        }
printf("|\n......................................................\n");
for(i=0;i<n;i++)
        {
          printf("\t\t%d",pt[i]);
        }
printf("\n......................................................\n");
printf("0");
for(i=0;i<n;i++)
        {
                bst+=c;
                turn+=c+pt[i];
                c=c+pt[i];
                printf("\t\t%d",c);
        }
```

```
printf("\nAverage time is %f: ",bst/n); printf("\nTurn around time is %f",
turn/n); getch();


        }
```

<u>OUTPUT:</u>

```
Enter number of processes : 3

          Enter the process 1 name : P1
          Enter Burst time : 24

          Enter the process 2 name : P2
          Enter Burst time : 2

          Enter the process 3 name : P3
          Enter Burst time : 3


Executing processes :

-----------------------------------------------------------------------------------------
          |        P2   |    P3   |    P1   |
-----------------------------------------------------------------------------------------
                    2                   3                   24
-----------------------------------------------------------------------------------------
          0                   2                   5                   29


Average wt time = 2.333333
Turnaround time = 12.000000
```

WEEK11

PRIORITY SCHEDULING

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority(p) is the inverse of the next CPU burst. The larger the CPU burst, the lower is the priority, and vice versa. Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.

Priority scheduling can be either preemptive or non-preemptive. A major problem with priority scheduling is indefinite blocking or starvation.

PROGRAM:

USINGPOINTERS:

```
#include<stdio.h>                               //HEADER FILES
#include<malloc.h>
#include<string.h>
typedefstruct node                              //STRUCTURE
{
        charprss[3];
        int burst;
        int priority;
        struct node *next;
}node;
node *front=NULL;                               //GLOBAL VARIABLES
node *rear=NULL;
void insert();                                  //FUNCTION DECLARATION
void display(int);
void main()                                     //MAIN FUNCTION
{
        int i,n;
        printf("\nEnter number of processes : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)                        //LOOP
                insert();                       //FUNCTION CALL
        printf("\n\nExecuting processes : \n");
                display(n);                     //FUNCTION CALL
        printf("\n");
}                                               //END OF MAIN
```

```
void insert()                                   //FUNCTION DEFINITION
{
        node *p,*temp;
        int b,pri;
        p=(node*)malloc(sizeof(node)); //DYNAMIC MEMORY ALLOCATION
        printf("\n\tEnter the process name : ");
        scanf("%s",p->prss);
        printf("\tEnter Burst time : ");
        scanf("%d",&b); printf("\tEnter
        Priority : "); scanf("%d",&pri); p-
        >burst=b;

        p->priority=pri;
        p->next=NULL;
        if(front==NULL)
        {
                front=p;
                rear=p;
        }
        else if(p->priority < front->priority)
        {
                p->next=front;
                front=p;
        }
        else if(p->priority > rear->priority)
        {
                rear->next=p;
                rear=p;
        }
        else
        {
                temp=front;
                while( p->priority > (temp->next)->priority )
                        temp=temp->next;
                p->next=temp->next;
                temp->next=p;
        }
}
void display(int n)                             //FUNCTION DEFINITION
{
        node *temp=front;
        int c=0;
        float turn=0.0,wttime=0.0;
        if(front!=NULL)
        {
```

```
                        printf("\n-----------------------------------------------------\n\t");
                while(temp!=NULL)
                {
                        printf("|\t%s\t",temp->prss);
                        temp=temp->next;
                }
                   printf("\n-----------------------------------------------------------------\n");
                temp=front;
                while(temp!=NULL)
              {
printf("\t%d\t ",temp->burst);
temp=temp->next;
              }
                   printf("\n-----------------------------------------------------------------\n\t");
                temp=front;
                printf("0\t");
                while(temp!=NULL)
              {
                        wttime+=c;
                        turn+=c+temp->burst;
                        c=c+temp->burst;
                        printf(" \t%d\t ",c);
                        temp=temp->next;
              }
                printf("\n-------------------------------------------------------------\n");
                printf("\n\nAveragewt time = %f ",wttime/n);
                printf("\nTurn around time = %f\n",turn/n);
        }
}
```

PROGRAM:

USINGARRAYS:

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
char p[10][5],temp[5];
int c=0,pt[10],pr[i],i,j,n,temp1;
float bst=0.0,turn=0.0;
```

```
clrscr();
printf("enter no of processes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter process%d name:\n",i+1);
scanf("%s",&p[i]);
printf("enter process time");
scanf("%d",&pt[i]);
printf(“\nenter the priority of process”);
scanf(“%d”,&pr[i]);
}
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(pr[i]>pr[j])
{
temp1=pt[i];
pt[i]=pt[j];
pt[j]=temp1;
t=pr[i];
pr[i]=pr[j];
pr[j]=t;
strcpy(temp,p[i]);
strcpy(p[i],p[j]);
strcpy(p[j],temp);
}
}
}
printf("\n....................................................\n");
for(i=0;i<n;i++)
        {
                        printf("|\t %s\t",p[i]);
        }
printf("|\n....................................................\n");
for(i=0;i<n;i++)
        {
            printf("\t\t%d",pt[i]);
        }
printf("\n....................................................\n");
printf("0");
for(i=0;i<n;i++)
        {
                        bst+=c;
                        turn+=c+pt[i];
```

```
                c=c+pt[i];
                printf("\t\t%d",c);
        }
printf("\nAverage time is %f: ",bst/n);
printf("\nTurn around time is %f", turn/n);
getch();
        }
```

OUTPUT:

```
Enter number of processes : 3

        Enter the process name : P1
        Enter Burst time : 24
        Enter Priority : 3

        Enter the process name : P2
        Enter Burst time : 3
        Enter Priority : 1

        Enter the process name : P3
        Enter Burst time : 2
        Enter Priority : 2


Executing processes :

-------------------------------------------------------------------------------------
        |       P2  |   P3  |   P1  |
-------------------------------------------------------------------------------------
                3               2               24
-------------------------------------------------------------------------------------
        0               3               5               29


Average wt time = 2.666667
Turnaround time = 12.333333
```

WEEK 12

FIFO PAGE REPLACEMENT

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 5
int  front=0,back=-1,cs=0,nf;
int f[MAX];
void enq(int x);
void deq(void);
void dis(void);
int  isfound(int);
void main()
{
        int  pf=0,rfs,rf[15],I;
        printf("\n FIFO page replacement");
        printf("\n Enter the size of reference string:");
        scanf("%d",&rfs);
        printf("\n Enter the reference string:");
        for(i=0;i<rfs;i++)
        {
                scanf("%d",&rf[i]);
        }
        printf("\n Enter the number of free frames:");
        scanf("%d",&nf);
       enq(rf[0]);
        pf=1;
        for(=0;i<rfs;i++)
        {
                if(!isfound(rf[i]))
                {
                        pf++;
                        if(cs==nf)
                        deq();
                        enq(rf[i]);
                }
                dis();
        }
        printf("\n No of page faults :%d",pf);
}

int isfound(int x)
{
        nt  I;
        for(i=0;i<cs;i++)
        if(f[i]==x)
                return 1;
        return 0;
}
void enq(int x)
{
        if(++back==nf)
                back=0;
        f[back]=x;
        cs++;
}
void dis()
{
        int  i;
        for(i=0;<cs;i++)
        printf("%d",f[i]);
        printf("\n");
}
void deq()
{
        Cs--;
        if(++front==nf)
        front=0;
        return;
}
```

WEEK 13

/*PAGE REPLACEMENT LRU */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
int fsize,ssize,f,frame[10],arrive[30],rstring[30];
int main()
{
int i,lfi,idx,cs=0,f,ls=0,pf=0,j=0,y,k,z=0,time=0;
int pagefound (int x);
void display();
int leastused();
int pagelocation(int x);
clrscr();
printf("\n\n\t\t LRU PAGE REPLACEMENT");
printf("\n\t\t --------------------");
printf("\n\n\t Enter the frame size:");
scanf("%d",&fsize);
printf("\n\t Enter the reference string size:");
scanf("%d",&ssize);
printf("\n\t Enter the reference string:");

for(i=0;i<ssize;i++)
scanf("%d",&rstring[i]);
for(k=0;k<fsize;k++)
{
frame[k]=-3;
arrive[k]=0;
}
for(i=0;i<ssize;i++)
{
y=pagefound(rstring[i]);
if(y==0)
{
pf++;
if(cs>=fsize)
{
lfi=leastused();
frame [lfi]=rstring[i];
arrive [lfi]=++time;
}
else if (cs<fsize)
{
frame[cs]=rstring[i];
arrive [cs]=++time;
}
}
else
{
idx=pagelocation(rstring[i]);
arrive [idx]=++time;
}
cs++;
display();
}
printf("\n Page fault=%d",pf);
}
int pagefound(int x)
{
int i,val=0;
for(i=0;i<fsize;i++)
{
if(x==frame[i])
{
val=1;
break;
}
}
return(val);
}
void display()
{
int i;
```

```
printf("\n");
for(i=0;i<fsize;i++)
{
if(frame[i]>=0)
{
printf("%d",frame[i]);
}
else
printf("\t");
}
}
int leastused()
{
int i,min=0,n=0;
for(i=1;i<fsize;i++)
{
if(arrive[i]<arrive[min])
{
min=i;
n++;
}
}
if(n==0)
return(0);
else
return(min);
}
int pagelocation(int pageno)
{
int i,flag=0;
for(i=0;i<fsize;i++)
{
if(frame[i]==pageno)
{
flag=1;
break;
}
}
return(i);
getch();
}
```

PAGE REPLACEMENT LRU
OUTPUT:

```
Enter the frame size:3
Enter the reference string  size:7
Enter  the reference string :1
2
3
1
4
3
5

1      1      1      1      1      5
       2      2      4      4      4
              3      3      3      3

Page fault:5
```

SPILL OVER

DEMONSTRATING of execl() WHERE CHILD PROCESS EXECUTES "ls" COMMAND AND PARENT PROCESS EXECUTES "date" COMMAND:

PROGRAM:

```
#include<stdio.h>                                   //HEADER FILES
#include<sys/wait.h>
#include<sys/types.h>
#include<stdlib.h>
#include<unistd.h>
int main()                                          //MAIN FUNCTION
{
        pid_t pid;
        int status;
        pid=fork();                //CALLING FORK TO CREATE A CHILD PROCESS
        if(pid==0)
        {
                printf("\nChild is executing Date command:\n");
                execl("/bin/date","date",NULL);
                exit(0);
        }
        else
        {
                wait(&status);
                printf("\nParent executing ls -l command:\n");
                execl("/bin/ls","ls","-l",NULL);
        }
        return 0;
}                                                   //END OF MAIN
```

OUTPUT:

```
Child is executing Date command
Mon Oct 3 10:07:56 EDI 2011 Parent
executing ls -l command total 12

-rw-r—r— 1 user user 371 2011-10-02 10:07 abc.c -rwxr-
xr-x 1 user user 7285 2011-10-02 10:07 xyz
```

IMPLEMENTATION OF COMMAND ls|wc USING PIPES

PROGRAM:

```
#include<stdlib.h>                                          //HEADER FILES

#include<unistd.h>

void main()                                         //MAIN FUNCTION
{
        pid_t pid;
        int fd[2];
        pipe(fd);
        pid=fork();                     //CALLING FORK TO CREATE A CHILD PROCESS
        if(pid != 0)
        {
                close(1);
                dup(fd[1]);
                close(fd[0]);
                execl("/bin/ls","ls",NULL);
        }
        else
        {
                close(0);
                dup(fd[0]);
                close(fd[1]);
                execl("/usr/bin/wc","wc",NULL);
                exit(0);
        }
}                                                   //END OF MAIN
```

OUTPUT:

```
47      47   288
```