Assignment No 01

AIM:
Assignment to learn and understand Shell Programming

Problem Statement
Write a program to handle students database with following options a) Create Database b) View Database c) Insert a record d) Delete a record e) modify a record f) Result of a particular student g) exit

THEORY:

Shell is an utility program which runs over the kernel. It acts as a command interpreter for the kernel and is the interface between user and the kernel. Thus it can be compared with command processor of MS-DOS which interprets, manages and co-ordinates the execution of commands by user.

To begin with the, kernel displays shell at prompt after authorized login. The input from the user decode the command line data and reaches for program. If the program is found, the shell retrieves it and submit it to the kernel to execute. Finally the kernel delivers the output. If the command is not found, it signals to the kernel to display "command not found" at the terminal. The shell accepts the kernel rely and in both cases, displays the next prompt. The cycle continues until it is terminated by Ctrl + D or logout. Login message is again displayed.

The /bin and /user/bin are the directories where all commands are located.

Types of shell :

1) **sh or Bourne Shell**: the original shell still used on UNIX systems. This is the basic shell, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.
2) **bash or Bourne Again shell**: the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. Commands that work in sh, also work in bash. However, the reverse is not always the case.
3) **csh or C shell**: the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.
4) **tcsh or Turbo C shell**: a superset of the common C shell, enhancing user friendliness and speed.
5) **ksh or the Korn shell**: sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.
6) **Restricted Shell (rsh) :** This is the restricted version of bash shell, It is used for guest login and in service installation where user must be restricted to work only in their own limited environments.

The file /etc/shells gives an overview of known shells on a Linux system. To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the PATH settings, since a shells is an executable files (program), the current shell activates it and it gets executed. A new prompt is usually shown, because each shell has its typical appearance.
A set of commands that has to be performed repeatedly is grouped into a batch file in DOS. Shell scripts are also similar to DOS batch files. Use any editor like vi or gedit to write shell script. .sh extension is used to identify shell script file.

$ vi sample.sh
#
My first shell script

#
clear
echo “Welcome to Shell Programming"

After writing shell script set execute permission for your script :
$ chmod +x sample.sh
After saving the above script, you can run the script :
$ ./sample.sh

clear is used to clear the screen and echo to print message or value of variables on screen.

if condition is used for decision making in shell script. test command or [expr] is used to see if an expression is true, and if it is true it return 0, otherwise returns nonzero for false.

Following script determine whether given argument number is positive.

```
$ vi sample.sh
if test $1 -gt 0
then
echo "$1 number is positive"
fi
```

The output is
$ ./sample.sh 5
5 number is positive

The various mathematical operators are used such as eq, ne, lt, le, gt and ge. The other operator used are : ! for Logical NOT, -a for AND, -o for OR. The above program can be modified as :

```
$ vi sample.sh
if [ $1 -gt 0 ]
then
echo "$1 number is positive"
else
        ------------------------
fi
```

Multi-level if else loop is as follows:

```
$ vi sample.sh
if [ $1 -gt 0 ]; then
        echo "$1 is positive"
elif [ $1 -lt 0 ] then
        echo "$1 is negative"
elif [ $1 -eq 0 ] then
        echo "$1 is zero"
else
        echo "Opps! $1 is not number, give number"
fi
```

For loop can be used as follows:

```
$ vi sample.sh
for i in 1 2 3 4 5
```

```
do
        echo "Welcome $i times"
done
```

$ vi sample.sh

```
for ((  i = 0 ;  i <= 5;  i++  ))
do
        echo "Welcome $i times"
done
```

The case construct is used for the execution of the shell script based on our choice. Some examples is as follows :

$ vi sample.sh

```
echo "Menu"
echo " 1. Your Current working directory"
echo "2. Today's date"
echo "3. List of users logged in"
echo "your Choice"
read choice
case $choice in
    1)      pwd;;
    2)      date;;
    3)      who;;
    *)  echo "invalid Choice"
esac
```

$ vi sample.sh

```
echo "Menu"
echo "1. Displays a long listing of file"
echo "2. Displays long listing of files including hidden files"
echo "3. Delete a file"
echo "Your Choice : "
read choice
case $choice in
        1)  ls –l ;;
        2)  ls –al;;
        3)  echo "Enter the name of file to be deleted"
            read file
            rm $file
            echo "File is deleted"
        *) echo "Invalid Choice"
esac
```

$ vi sample.sh (it uses break and while loop)

```
while true
do
echo "Menu"
echo "1. Displays a long listing of file"
echo "2. Displays long listing of files including hidden files"
echo "3. Delete a file"
echo "Your Choice :  (Press w to quit)"
read choice
case $choice in
        1)      ls –l ;;
```

```
        2)      ls –al;;
        3)      echo "Enter the name of file to be deleted"
                read file
                rm $file
                echo "File is deleted";;
        w)      break;;
        *)      echo "Invalid Choice"
esac
done
```

$ vi sample.sh (it uses continue and while loop)

```
ans="Y"
while [ $ans = y –o $ans = Y ]
do
echo "Menu"
echo "1. Displays a long listing of file"
echo "2. Displays long listing of files including hidden files"
echo "3. Delete a file"
echo "Your Choice :  "
read choice
case $choice in
        1)      ls –l ;;
        2)      ls –al;;
        3)      echo "Enter the name of file to be deleted"
                read file
                rm $file
                echo "File is deleted";;
        *)      echo "Invalid Choice"
esac
echo "Do You want to continue (y or Y)"
read ans
if [ $ans = y –o $ans = Y ]
then
        continue
else
        exit
fi
done
```

CONCLUSION –

Thus we have studied and implemented shell scripting language for various programs.

FAQ's – Shell Programming and introduction to operating system

1. Differentiate between
    a. DOS and Windows
    b. UNIX and Linux
    c. Windows and Linux
2. Specify at least five commands with description in UNIX?
3. Write a short note on
    a. Wild card patterns
    b. Redirection
    c. Pipes and filters
    d. File access permission.

4. Describe in detail what is shell with block diagram of UNIX?
5. What is the use of shell programming?

Assignment No 02 (A)

Process Control System Calls

AIM:

The demonstration of fork, execve and wait system calls along with zombie and orphan states.

1. Implement the C program in which main program accepts the integers to be sorted. Main program uses the fork system call to create a new process called a child process. Parent process sorts the integers using merge sort and waits for child process using wait system call to sort the integers using quick sort. Also demonstrate zombie and orphan states.
2. Implement the C program in which main program accepts an integer array. Main program uses the fork system call to create a new process called a child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve system call. The child process uses execve system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

OBJECTIVE:

This assignment covers the UNIX process control commonly called for process creation, program execution and process termination. Also covers process model, including process creation, process destruction, zombie and orphan processes.

THEORY:

Process in UNIX:

A process is the basic active entity in most operating-system models.

Process IDs

Each process in a Linux system is identified by its unique *process ID*, sometimes referred to as *pid*. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

When referring to process IDs in a C or C++ program, always use the pid_t typedef, which is defined in <sys/types.h>.A program can obtain the process ID of the process it's running in with the getpid() system call, and it can obtain the process ID of its parent process with the getppid() system call.

Creating Processes

Two common techniques are used for creating a new process.

- using system() function.
- using fork() system calls.

1. **Using system**

The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, system

creates a subprocess running the standard Bourne shell (/bin/sh)and hands the command to that shell for execution.

The system function returns the exit status of the shell command. If the shell itself cannot be run, system returns 127; if another error occurs, system returns –1.

2. **Using fork**

A process can create a new process by calling fork. The calling process becomes the parent, and the created process is called the child. The fork function copies the parent's memory image so that the new process receives a copy of the address space of the parent. Both processes continue at the instruction after the fork statement (executing in their respective memory images).

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

The fork function returns 0 to the child and returns the child's process ID to the parent. When fork fails, it returns –1.

The wait Function

When a process creates a child, both parent and child proceed with execution from the point of the fork. The parent can execute wait to block until the child finishes. The wait function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.

SYNOPSIS

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

pid_t wait(int *status);
```

If wait returns because the status of a child is reported, these functions return the process ID of that child. If an error occurs, these functions return –1

Example:

pid_t childpid;

childpid = **wait**(NULL);
if (childpid != -1)

printf("Waited for child with pid %ld\n", childpid);

Status values

The status argument of **wait** is a pointer to an integer variable. If it is not **NULL**, this function stores the **return status** of the child in this location. The child returns its status by calling exit, _exit or return from main.

A zero return value indicates EXIT_SUCCESS; any other value indicates EXIT_FAILURE.

POSIX specifies six macros for testing the child's return status. Each takes the status value returned by a child to **wait** as a parameter. Following are the two such macros:

SYNOPSIS

#include <sys/wait.h>

WIFEXITED(int stat_val)
WEXITSTATUS(int stat_val)

New program execution within the existing process (The exec Function)

The fork function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent. The exec family of functions provides a facility for overlaying the process image of the calling process with a new image. The traditional way to use the fork–exec combination is for the child to execute (with an exec function) the new program while the parent continues to execute the original code.

Assignment No 02 (B)
Process Control System Calls

SYNTAX

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg0, ... /*, char *(0) */);
int execle (const char *path, const char *arg0, ... /*, char *(0), char *const envp[] */);
```

- int execlp (const char *file, const char *arg0, ... /*, char *(0) */);
- int execv(const char *path, char *const argv[]);

- int execve (const char *path, char *const argv[], char *const envp[]);
- int execvp (const char *file, char *const argv[]);

exec() system call:
The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways. Within the exec family there are functions that vary slightly in their capabilities.
exec family:

1. execl() and execlp():
execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL.
e.g. execl("/bin/ls", "ls", "-l", NULL);

execlp(): It does same job except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly.
e.g. execlp("ls", "ls", "-l", NULL);

2. execv() and execvp():
execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string.
e.g. char *argv[] = ("ls", "-l", NULL);
execv("/bin/ls", argv);

execvp(): It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used.
e.g. execvp("ls", argv);

3. execve( ):
int execve(const char *filename, char *const argv[ ], char *const envp[ ]);
It executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form: argv is an array of argument strings passed to

the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

int main(int argc, char *argv[ ] , char *envp[ ])]

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

All exec functions return –1 if unsuccessful. In case of success these functions never return to the calling function.

Process Termination

Normally, a process terminates in one of two ways. Either the executing program calls the **exit()** function, or the program's main function **returns**. Each process has an exit code: a number that the process returns to its parent.The exit code is the argument passed to the exit function, or the value returned from main.

Zombie Processes

If a child process terminates while its parent is calling a wait function, the child process vanishes and its termination status is passed to its parent via the wait call. But what happens when a child process terminates and the parent is not calling wait? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, is becomes a zombie process.

A *zombie process* is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The wait functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls wait. If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes. If the child process finishes before the parent process calls wait, the child process becomes a zombie. When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

Orphan Process:

An Orphan Process is nearly the same thing which we see in real world. Orphan means someone whose parents are dead. The same way this is a process, whose parents are dead, that means parents are either terminated, killed or exited but the child process is still alive.

In Linux/Unix like operating systems, as soon as parents of any process are dead, re-parenting occurs, automatically. Re-parenting means processes whose parents are dead, means Orphaned processes, are immediately adopted by special process. Thing to notice here is that even after re-parenting, the process still remains Orphan as the parent which created the process is dead, Reasons for Orphan Processes:

A process can be orphaned either intentionally or unintentionally. Sometime a parent

process exits/terminates or crashes leaving the child process still running, and then they become orphans. Also, a process can be intentionally orphaned just to keep it running. For example when you need to run a job in the background which need any manual intervention and going to take long time, then you detach it from user session and leave it there. Same way, when you need to run a process in the background for infinite time, you need to do the same thing. Processes running in the background like this are known as daemon process.

Finding a Orphan Process:

It is very easy to spot a Orphan process. Orphan process is a user process, which is having init (process id â€“ 1) as parent. You can use this command in linux to find the Orphan processes.

```
# ps -elf | head -1; ps -elf | awk '{if ($5 == 1 && $3 != "root") {print $0}}' | head
```

This will show you all the orphan processes running in your system. The output from this command confirms that they are Orphan processes but does not mean that they are all useless, so confirm from some other source also before killing them.

Killing a Orphan Process:

As orphaned processes waste server resources, so it is not advised to have lots of orphan processes running into the system. To kill a orphan process is same as killing a normal process.

```
# kill -15 <PID>
```

If that does not work then simply use

```
# kill -9 <PID>
```

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

vfork: alternative of fork

create a new process when exec a new program.

Compare with fork:

- Creates new process without fully copying the address space of the parent.
- vfork guarantees that the child runs first, until the child calls exec or exit.
- When child calls either of these two functions(exit, exec), the parent resumes.

INPUT:

- An integer array with specified size.
- An integer array with specified size and number to search.

OUTPUT:

- Sorted array.
- Status of number to be searched.

FAQS:

bie process ?
n ?
ur system ?

rent and child process?
Block?

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

Example 1

Printing the Process ID

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

int main()
{
printf("The process ID is %d\n", (int) getpid());
printf("The parent process ID is %d\n", (int) getppid());
return 0;
}
```

Example 2

Using the system call

```
#include <stdlib.h>

int main()
{
int return_value;
return_value=system("ls –l /");
return return_value;
}
```

Example 3

Using fork to duplicate a program's process

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

int main()
{
pid_t child_pid;
printf("The main program process ID is %d\n", (int) getpid());
```

```
child_pid=fork();
if(child_pid!=0)  {
printf(“This is the parent process ID, with id %d\n”, (int) getpid());
printf(“The child process ID is %d\n”, (int) chi;d_pid);
}
else
printf(“This is the child process ID, with id %d\n”, (int) getpid());
return 0;
}
```

Example 4

Determining the exit status of a child.

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

void show_return_status(void)
{

    pid_t childpid; int
    status;

    childpid = wait(&status); if
    (childpid == -1)

        perror("Failed to wait for child");

    else if (WIFEXITED(status))

        printf("Child %ld terminated with return status %d\n", (long)childpid,
                WEXITSTATUS(status));

}
```

Example 5

A program that creates a child process to run ls -l.

```
#include <stdio.h> #include
<stdlib.h> #include
<unistd.h> #include
<sys/wait.h>

int main(void)
{
    pid_t childpid;

    childpid = fork();
```

```
    if (childpid == -1) { perror("Failed to
        fork"); return 1;

    }
    if (childpid == 0) {

                                    /* child code */
        execl("/bin/ls", "ls", "-l", NULL);
        perror("Child failed to exec ls"); return 1;

    }

    if (childpid != wait(NULL)) {

                                /* parent code */

        perror("Parent failed to wait due to signal or error"); return 1;

    }
    return 0;
}
```

Example 6
Making a zombie process

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

int main()
{
pid_t child_pid;
//create a child process
child_pid=fork();
if(child_pid>0)  {
//This is a parent process. Sleep for a minute
sleep(60)
}
else
{
//This is a child process. Exit immediately.
exit(0);
}
return 0;
}
```

Example 7
Demonstration of fork system call

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

int main()
{
pid_t pid;
char *msg;
int n;
printf("Program starts\n");
pid=fork();
switch(pid)
{
case -1:
printf("Fork error\n");
exit(-1);
case 0:
msg="This is the child process";
n=5;
break;
default:
msg="This is the parent process";
n=3;
break;
}
while(n>0)
{
puts(msg);
sleep(1);
n--;
}
return 0;
}
```

Example 8
Demo of multiprocess application using fork()system call

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

#define SIZE 1024
void do_child_proc(int pfd[2]);
void do_parent_proc(int pfd[2]);

int main()
{
int pfd[2];
```

```
int ret_val,nread;
pid_t pid;
ret_val=pipe(pfd);
if(ret_val==-1)
{
perror("pipe error\n");
exit(ret_val);
}
pid=fork();
switch(pid)
{
case -1:
printf("Fork error\n");
exit(pid);
case 0:
do_child_proc(pfd);
exit(0);
default:
do_parent_proc(pfd);
exit(pid);
}
wait(NULL);

return 0;
}

void do_child_proc(int pfd[2])
{
int nread;
char *buf=NULL;
printf("5\n");
close(pfd[1]);
while(nread=(read(pfd[0],buf,size))!=0)
printf("Child Read=%s\n",buf);
close(pfd[0]);
exit(0);
}
void do_parent_proc(int pfd[2])
 {
char ch;
char *buf=NULL;
close(pfd[0]);
while(ch=getchar()!='\n')  {
printf("7\n");
*buf=ch;
buff+;
}
*buf='\0';
write(pfd[1],buf,strlen(buf)+1);
close(pfd[1]);
}
```

ASSIGNMENT NO 03

Title: Implement CPU Scheduling Algorithms and Calculate Average Waiting Time for a) Preemptive Shortest Job Policy b) Round Robin Policy

Theory:

Scheduling of processes/work is done to finish the work on time. **CPU Scheduling** is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer. Whenever the CPU becomes idle, the operating system must select one of the processes in the line ready for launch. The selection process is done by a temporary (CPU) scheduler. The Scheduler selects between memory processes ready to launch and assigns the CPU to one of them.

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.

Characteristics of SJF:

- Shortest Job first has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.
- It is associated with each task as a unit of time to complete.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Advantages of Shortest Job first:

- As SJF reduces the average waiting time thus, it is better than the first come first serve scheduling algorithm./
- SJF is generally used for long term scheduling

Disadvantages of SJF:

- One of the demerit SJF has is starvation.
- Many times it becomes complicated to predict the length of the upcoming CPU request

Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That’s because it doesn’t need special hardware (for example, a timer) like preemptive scheduling.

When scheduling is Preemptive or Non-Preemptive?
To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

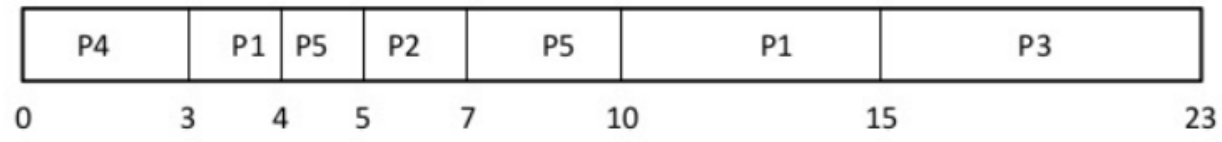
1. A process switches from the running to the waiting state.
2. Specific process switches from the running state to the ready state.
3. Specific process switches from the waiting state to the ready state.
4. Process finished its execution and terminated.

Only conditions 1 and 4 apply, the scheduling is called non- preemptive.
All other scheduling are preemptive.

Example: In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.
Consider the following five process:

Process ID	Arrival Time	Burst Time
P1	**6**	**2**
P2	**2**	**5**
P3	**8**	**1**
P4	**3**	**0**
P5	**4**	**4**

GANTT CHART



Wait time
P4= 0-0=0
P1= (3-2) + 6 =7
P2= 5-5 = 0
P5= 4-4+2 =2
P3= 15-1 = 14

Average Waiting Time = 0+7+0+2+14/5 = 23/5 =4.6

Round Robin Scheduling:
Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Characteristics of Round robin:

- It’s simple, easy to use, and starvation-free as all processes get the balanced CPU allocation.
- One of the most widely used methods in CPU scheduling as a core.
- It is considered preemptive as the processes are given to the CPU for a very limited time.

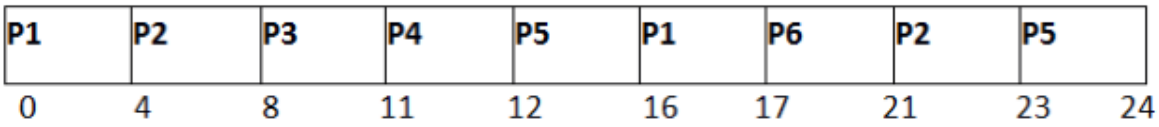
Advantages of Round robin:

- Round robin seems to be fair as every process gets an equal share of CPU.
- The newly created process is added to the end of the ready queue.

In the following example, there are six processes named as P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table. The time quantum of the system is 4 units.

Process ID	**Arrival Time**	**Burst Time**
1	**0**	**5**
2	**1**	**6**
3	**2**	**3**
4	**3**	**1**
5	**4**	**5**
6	**6**	**4**

According to the algorithm, we have to maintain the ready queue and the Gantt chart. The structure of both the data structures will be changed after every scheduling.
GANTT CHART:



1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

Avg Waiting Time = (12+16+6+8+15+11)/6 = 76/6 units

Conclusion: Preemptive Shortest Job First and Round Robin Policy are implemented successfully.

ASSIGNMENT NO: 4

Aim: Thread synchronization using counting semaphores and mutual exclusion using mutex.

OBJECTIVE: Implement C program to demonstrate producer-consumer problem with counting semaphores and mutex.

THEORY:

Semaphores:

An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.

The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.

Semaphores are the **OS tools** for **synchronization**. Two types:

1. **Binary Semaphore**.
2. **Counting Semaphore**.

Counting semaphore

The counting semaphores are free of the limitations of the binary semaphores. A counting semaphore comprises:

An integer variable, initialized to a value K (K>=0). During operation it can assume any value <= K, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

A counting semaphore can be implemented as follows:

```
typedef  struct Process
{
      int ProcessID;
    ------------------
    ------------------
    Process *Next;    /* Pointer to the next PCB in the queue/
};
```

```
typedef  struct Semaphore
{
       int count;
     Process *head  /* Pointer to the head of the queue */
     Process *tail;    /* Pointer to the tail of the queue/
};
Semaphore S;
```

Operation of a counting semaphore:

1. Let the initial value of the semaphore count be 1.

2. When semaphore count = 1, it implies that no process is executing in its critical section and no process is waiting in the semaphore queue.
3. When semaphore count = 0, it implies that one process is executing in its critical section but no process is waiting in the semaphore queue.
4. When semaphore count = N, it implies that one process is executing in its critical section and N processes are waiting in the semaphore queue.
5. When a process is waiting in semaphore queue, it is not performing any busy waiting. It is rather in a "waiting" or "blocked" state.
6. When a waiting process is selected for entry into its critical section, it is transferred from "Blocked" state to "ready" state.

The Producer/Consumer Problem

We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem. The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. We will look at a number of solutions to this problem to illustrate both the power and the pitfalls of semaphores. To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

```
producer:
while (true) {
   /* produce item v */;
   b[in] = v;
   in++;
}
```

```
consumer:
while (true) {
   while (in <= out)
        /* do nothing */;
   w = b[out];
   out++;
   /* consume item w */;
}
```

Figure illustrates the structure of buffer b. The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the

ASSIGNMENT NO:

4 (B)

AIM: To implement Dining Philosopher's problem using 'C' in Linux

OBJECTIVE: Implement the deadlock-free solution to Dining Philosophers problem to illustrate the problem of deadlock and/or starvation that can occur when many synchronized threads are competing for limited resources..

THEORY:

What is Deadlock?

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no interrupts condition is needed to prevent an otherwise deadlocked process from being awake**.**

Conditions for Deadlock

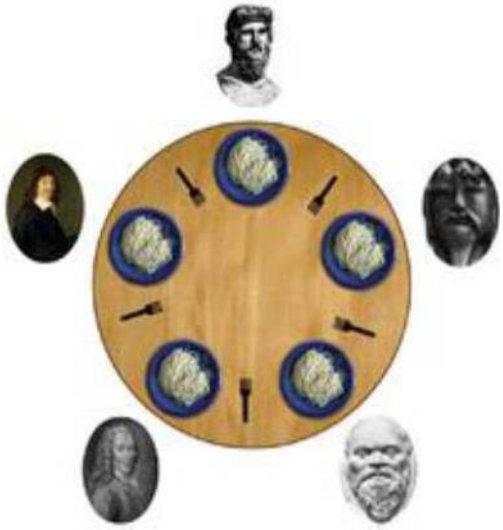
Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.
3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

In this problem, there are 5 philosophers present who spend their life in eating and thinking. Philosophers share a common circular table surrounded by 5 chairs, each belonging to 1 philosopher .In the center of the table. A bowl of slippery food(Noodles or rice) is present and across each philosopher a pair of chopstick is present. When a philosopher thinks, he does not interact with his colleague.

Whenever a philosopher gets hungry, he tries to pick up 2 chopsticks that are close to him. Philosopher may pick up one chopstick at a time. He cannot pick up a chopstick that is already in the hand of neighbor. When a hungry philosopher has both chopsticks at the same time, he start eating without releasing his chopstick and starts thinking again .The problem could be raised when all the philosopher try to keep the chopstick at the same time.

This may lead to deadlock situations. To synchronize all philosophers, semaphore chopsticks [5] are used as a variable where all the elements are first initialized to 1.The structure of philosophers is shown below;



SEMAPHORE CHOPSTICK [5]
Think: After eating

Eat: Hungry

Do

{

wait(chopstick[i]);

wait(chopstick[(i+1)%5]);

---

eat

signal(chopstick[i]);

signal(chopstick[(i+1)%5]);

---

think

}while(1);

1) The following three functions lock and unlock a mutex:

# include<pthread.h>
intpthread_mutex_lock(pthread_mutex_t * mptr);
intpthread_mutex_trylock(pthread_mutex_t * mptr);
intpthread_mutex_unlock(pthread_mutex_t * mptr);

2) The mutex or condition variable is initialized or destroyed with the following functions.

# include<pthread.h>
intpthread_mutex_init(pthread_mutex_t * mptr,constpthread_mutexattr_t * attr);
intpthread_mutex_destroy(pthread_mutex_t * mptr);

3) Functions used with their syntax :
Routines:

pthread_create (thread,attr,start_routine,arg) pthread_exit (status)

Creating Threads:

Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

pthread_create arguments:

a. thread: An opaque, unique identifier for the new thread returned by the subroutine.

b. attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.

c. start_routine: the C routine that the thread will execute once it is created.

d. arg: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.

When a program is started by exec, a single thread is created , called the initial thread or main thread . Additional threads are by pthread_create.

# include<pthread.h>
int pthread_create(pthread_t * tid,constpthread_attr_t * attr,void * (* func)void *),void* arg);

Thread Joining:

- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

- A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

- Two other synchronization methods, mutexes and condition variables, will be discussed later.

CONCLUSION:

Thus, we have implemented dining philosopher's problem using 'C' in Linux.

FAQ:

1. What is dead lock?
2. What are the necessary and sufficient conditions to occur deadlock?
3. What is deadlock avoidance and deadlock prevention techniques?

ASSIGNMENT NO 05

Title: Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

Theory:

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the **Banker's Algorithm** in detail. Also, we will solve problems based on the **Banker's Algorithm**. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an **operating system**. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

Disadvantages

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.

2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.
3. Each process has to know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work. Nedd[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock in a system:

Safety Algorithm

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

1. There are two vectors **Wok** and **Finish** of length m and n in a safety algorithm.

Initialize: Work = Available

Finish[i] = false; for I = 0, 1, 2, 3, 4… n - 1.

2. Check the availability status for each type of resources [i], such as:

Need[i] <= Work

Finish[i] == false

If the i does not exist, go to step 4.

3. Work = Work +Allocation(i) // to get new resource allocation

Finish[i] = true

Go to step 2 to check the status of resource availability for the next process.

4. If Finish[i] == true; it means that the system is safe for all processes.

Resource Request Algorithm

A resource request algorithm checks how a system will behave when a process makes each type of resource request in a system as a request matrix.

Let create a resource request array R[i] for each process P[i]. If the Resource $Request_i$ [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system.

1. When the number of **requested resources** of each type is less than the **Need** resources, go to step 2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

If Request(i) <= Need

Go to step 2;

2. And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

If Request(i) <= Available

Else Process P[i] must wait for the resource since it is not available for use.

3. When the requested resource is allocated to the process by changing state:

Available = Available – Request

Allocation(i) = Allocation(i) + Request (i)

$Need_i = Need_i - Request_i$

When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Conclusion: Deadlock Avoidance Algorithms is implemented successfully.

ASSIGNMENT NO 06

Title:
Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

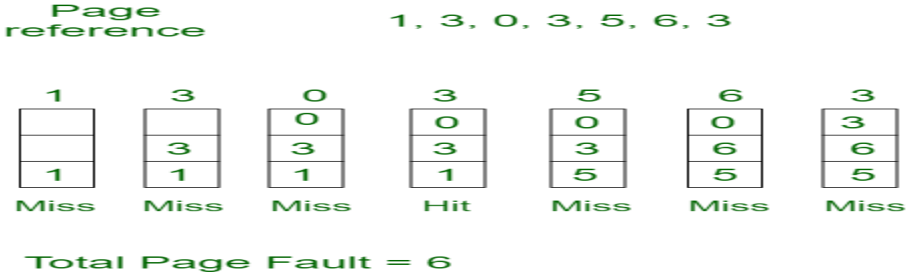
Theory:
In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in.

Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms:

1. First In First Out (FIFO): This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames.Find the number of page faults.



Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3**

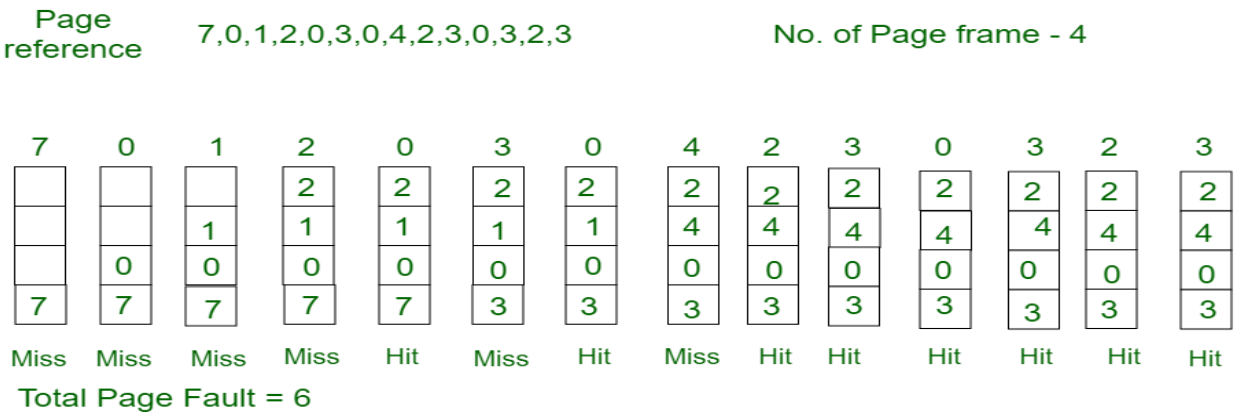
Page Faults.
when 3 comes, it is already in memory so —> **0 Page Faults.** Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault.** 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault.** Finally, when 3 come it is not available so it replaces 0 **1 page fault.**

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm.

For example, if we consider reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4, and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults.

2. Optimal Page replacement: In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example-2: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.



Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.** when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.** 0 is already there so —> **0 Page fault.** 4 will takes place of 1 —> **1 Page Fault.**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3. Least Recently Used: In this algorithm, page will be replaced which is least recently used.

Example-3: Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already their so —> **0 Page fault.** when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Conclusion: Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three. is implemented successfully.

ASSIGNMENT NO: 7 (A)

AIM: Inter process communication in Linux using FIFOs.

OBJECTIVES:
Implementation of Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

THEORY:
FIFOs

A first-in, first-out (FIFO) file is a pipe that has a name in the filesystem. Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called named pipes

You can make a FIFO using the mkfifo command. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-
1 samuel        users        0 Jan 16 14:04 /tmp/fifo
```

The first character of the output from ls is p, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
% cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
% cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window. Remove the FIFO with this line:

```
% rm /tmp/fifo
```

Creating a FIFO
Create a FIFO programmatically using the mkfifo function.The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions, and a pipe must have a reader and a writer, the permissions must include both read and write permissions. If the pipe cannot be created (for instance, if a file with that name already exists), mkfifo returns -1. Include<sys/types.h> and <sys/stat.h> if you call mkfifo.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions like

open, write, read, close or C library I/O functions (fopen, fprintf, fscanf, fclose, and soon) may be used.
For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
intfd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

CONCLUSION:
Thus, we studied inter process communication using FIFOs.

ASSIGNMENT NO:

7 (B)

AIM: Inter process communication in Linux using Shared Memory

Shared Memory

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.
SYSTEM CALL: shmget()
In order to create a new message queue, or access an existing queue, the shmget() system call is used.

SYSTEM CALL: shmget();

PROTOTYPE: int shmget (key_t key, int size, int shmflg);
RETURNS: shared memory segment identifier on success
-1 on error: errno = EINVAL (Invalid segment size specified)
EEXIST (Segment exists, cannot create)
EIDRM (Segment is marked for deletion, or was removed)
ENOENT (Segment does not exist)
EACCES (Permission denied)
ENOMEM (Not enough memory to create segment)
NOTES:

This particular call should almost seem like old news at this point. It is strikingly similar to the corresponding get calls for message queues and semaphore sets.

The first argument to shmget() is the key value (in our case returned by a call to ftok()). This key value is then compared to existing key values that exist within the kernel for other shared memory segments. At that point, the open or access operation is dependent upon the contents of the shmflg argument.

IPC_CREAT

Create the segment if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if segment already exists.

If IPC_CREAT is used alone, shmget() either returns the segment identifier for a newly created segment, or returns the identifier for a segment which exists with the same key value. If IPC_EXCL

is used along with IPC_CREAT, then either a new segment is created, or if the segment exists, the call fails with -1. IPC_EXCL is useless by itself, but when combined with IPC_CREAT, it can be used as a facility to guarantee that no existing segment is opened for access.

Once again, an optional octal mode may be OR'd into the mask.

Let's create a wrapper function for locating or creating a shared memory segment :

```
int open_segment( key_t keyval, int segsize )
{
      int     shmid;

      if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
      {
            return(-1);
      }

      return(shmid);
}
```

Note the use of the explicit permissions of 0660. This small function either returns a shared memory segment identifier (int), or -1 on error. The key value and requested segment size (in bytes) are passed as arguments.

Once a process has a valid IPC identifier for a given segment, the next step is for the process to attach or map the segment into its own addressing space.

SYSTEM CALL: shmat()

```
 SYSTEM CALL: shmat();

 PROTOTYPE: int shmat ( int shmid, char *shmaddr, int shmflg);
   RETURNS: address at which segment was attached to the process, or
         -1 on error: errno = EINVAL (Invalid IPC ID value or attach address passed)
                              ENOMEM (Not enough memory to attach segment)
                              EACCES (Permission denied)
 NOTES:
```

If the addr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method. An address can be specified, but is typically only used to facilitate proprietary hardware or to resolve conflicts with other apps. The SHM_RND flag can be OR'd into

the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).

In addition, if the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.

This call is perhaps the simplest to use. Consider this wrapper function, which is passed a valid IPC identifier for a segment, and returns the address that the segment was attached to:

```
char *attach_segment( int shmid )
{
      return(shmat(shmid, 0, 0));
}
```

Once a segment has been properly attached, and a process has a pointer to the start of that segment, reading and writing to the segment become as easy as simply referencing or dereferencing the pointer! Be careful not to lose the value of the original pointer! If this happens, you will have no way of accessing the base (start) of the segment.

SYSTEM CALL: shmctl()

```
 SYSTEM CALL: shmctl();
 PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
   RETURNS: 0 on success
         -1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
                      EFAULT (Address pointed to by buf is invalid with IPC_SET and
                           IPC_STAT commands)
                      EIDRM  (Segment was removed during retrieval)
                      EINVAL (shmqid invalid)
                      EPERM  (IPC_SET or IPC_RMID command was issued, but
                           calling process does not have write (alter)
                           access to the segment)
    NOTES:
```

This particular call is modeled directly after the *msgctl* call for message queues. In light of this fact, it won't be discussed in too much detail. Valid command values are:

IPC_STAT

Retrieves the shmid_ds structure for a segment, and stores it in the address of the buf argument

IPC_SET

Sets the value of the ipc_perm member of the shmid_ds structure for a segment. Takes the values from the buf argument.

IPC_RMID

Marks a segment for removal.

The IPC_RMID command doesn't actually remove a segment from the kernel. Rather, it marks the segment for removal. The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

To properly detach a shared memory segment, a process calls the *shmdt system call.*

SYSTEM CALL: shmdt()

SYSTEM CALL: shmdt();

PROTOTYPE: int shmdt (char *shmaddr);
RETURNS: -1 on error: errno = EINVAL (Invalid attach address passed)

After a shared memory segment is no longer needed by a process, it should be detached by calling this system call. As mentioned earlier, this is not the same as removing the segment from the kernel! After a detach is successful, the shm_nattch member of the associates shmid_ds structure is decremented by one. When this value reaches zero (0), the kernel will physically remove the segment.

Conclusion:

Thus we studied IPC mechanisms

ASSIGNMENT NO 08

Title:
Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

Theory:
Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.
Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

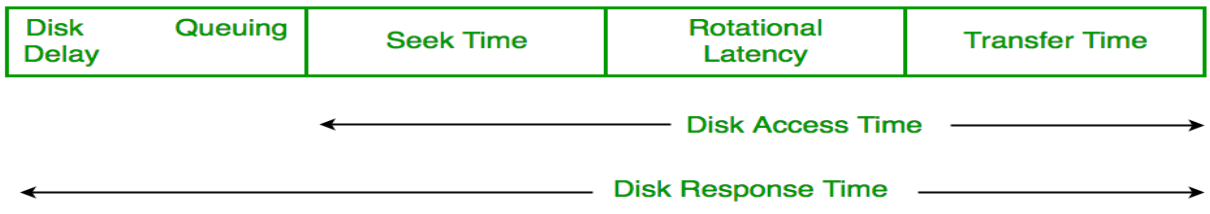
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

- **Disk Access Time:** Disk Access Time is:

  Disk Access Time = Seek Time +

  Rotational Latency +

  Transfer Time



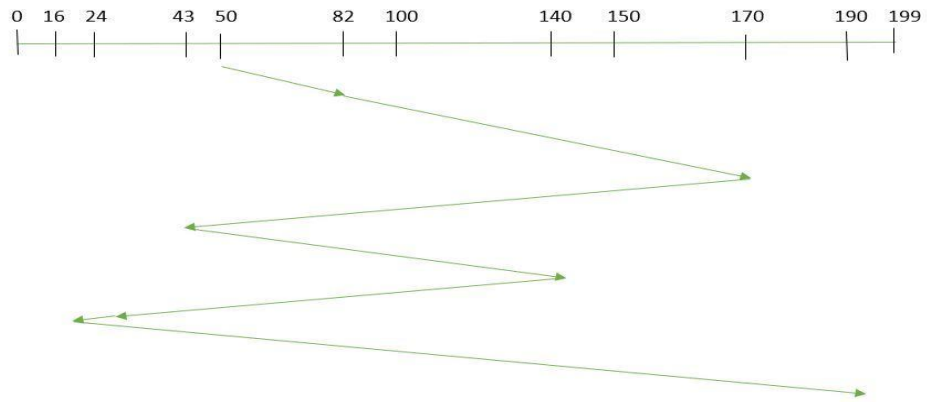
- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Example:

1. Suppose the order of request is- (82,170,43,140,24,16,190)
   And current position of Read/Write head is: 50



So, total seek time:
=(82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16)
=642

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

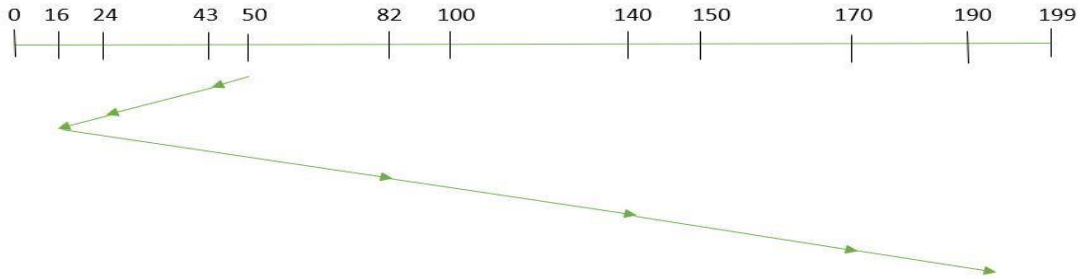
- Does not try to optimize seek time
- May not provide the best possible service

2. **SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue

and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.Let us understand this with the help of an example.

Example:

1. Suppose the order of request is- (82,170,43,140,24,16,190)
   And current position of Read/Write head is: 50



So, total seek time:

=(50-43)+(43-24)+(24-16)+(82-16)+(140-82)+(170-140)+(190-170)
=208

Advantages:

- Average Response Time decreases
- Throughput increases

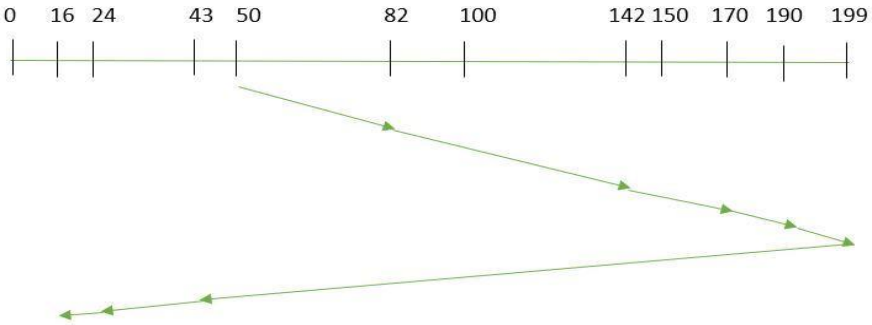
Disadvantages:

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favours only some requests

3. SCAN: In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



Therefore, the seek time is calculated as:

=(199-50)+(199-16)
=332

Advantages:

- High throughput
- Low variance of response time
- Average response time

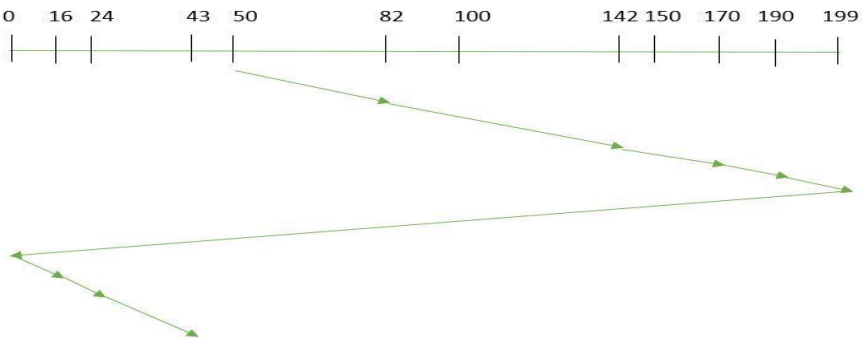
Disadvantages:

- Long waiting time for requests for locations just visited by disk arm

4. CSCAN: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.
These situations are avoided in *CSCAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



Seek time is calculated as:

=(199-50)+(199-0)+(43-0)
=391

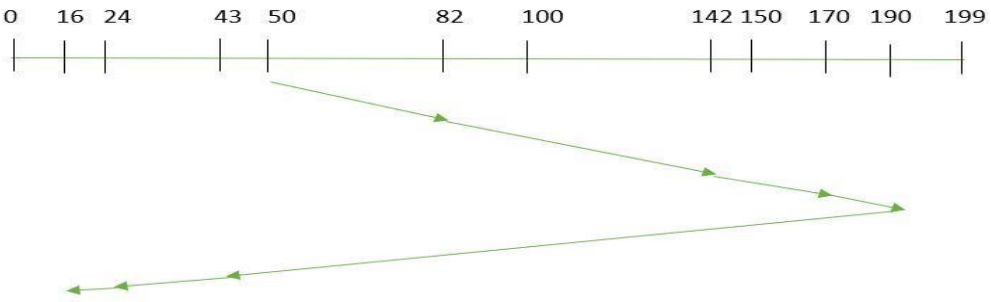
Advantages:

- Provides more uniform wait time compared to SCAN

5. LOOK: It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



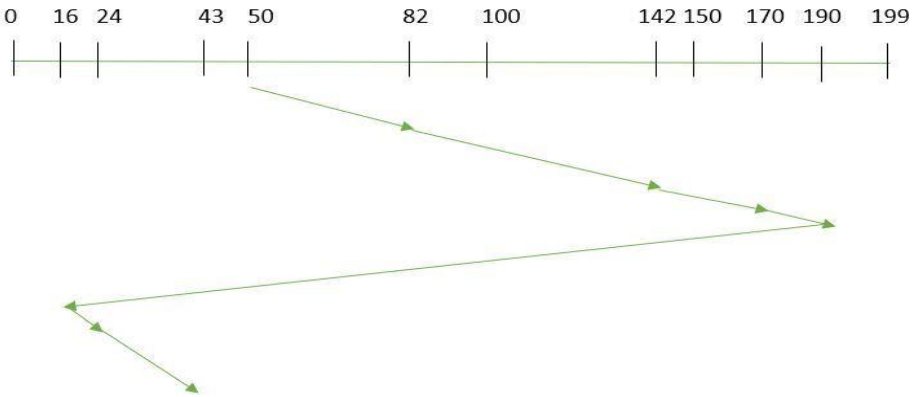
So, the seek time is calculated as:

1. =(190-50)+(190-16)
   =314

6. CLOOK: As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value"**



So, the seek time is calculated as:

=(190-50)+(190-16)+(43-16)
=341

Conclusion: Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle are implemented successfully.