NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE

(NAAC Accredited)

(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

Pampady, Thiruvilwamala (PO), Thrissur (DT), Kerala 680 588

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

CSL 204 OPERATING SYSTEMS LAB

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT THE DEPARTMENT

- Established in: 2002
- Course offered: B.Tech. in Computer Science and Engineering

  M. Tech. in Computer Science and Engineering

  M. Tech. in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Certified by ISO 9001-2015
- Affiliated to A P J Abdul Kalam Technological University, Kerala.

DEPARTMENT VISSION

To develop professionally ethical and socially responsible Mechatronics engineers to serve the humanity through quality professional education.

DEPARTMENT MISSION

1) The department is committed to impart the right blend of knowledge and quality education to create professionally ethical and socially responsible graduates.

2) The department is committed to impart the awareness to meet the current challenges in technology.

3) Establish state-of-the-art laboratories to promote practical knowledge of mechatronics to meet the needs of the society

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamwork and leadership qualities.

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOME

At the end of the course, the student should be able to

CO1	Illustrate the use of systems calls in Operating Systems. **(Cognitive knowledge: Understand)**
CO2	Implement Process Creation and Inter Process Communication in Operating Systems. **(Cognitive knowledge: Apply)**
CO3	Implement Fist Come First Served, Shortest Job First, Round Robin and Priority-based CPU Scheduling Algorithms. **(Cognitive knowledge: Apply)**
CO4	Illustrate the performance of First In First Out, Least Recently Used and Least Frequently Used Page Replacement Algorithms. **(Cognitive knowledge: Apply)**
CO5	Implement modules for Deadlock Detection and Deadlock Avoidance in Operating Systems. **(Cognitive knowledge: Apply)**
CO6	Implement modules for Storage Management and Disk Scheduling in Operating Systems. **(Cognitive knowledge: Apply)**

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	✓	✓	✓					✓		✓		✓
CO2	✓	✓	✓					✓		✓		✓
CO3	✓	✓	✓	✓				✓		✓		✓
CO4	✓	✓	✓	✓				✓		✓		✓
CO5	✓	✓	✓	✓				✓		✓		✓
CO6	✓	✓	✓	✓				✓		✓		✓

Abstract POs defined by National Board of Accreditation			
PO#	**Broad PO**	**PO#**	**Broad PO**
PO1	Engineering Knowledge	PO7	Environment and Sustainability
PO2	Problem Analysis	PO8	Ethics
PO3	Design/Development of solutions	PO9	Individual and team work
PO4	Conduct investigations of complex problems	PO10	Communication
PO5	Modern tool usage	PO11	Project Management and Finance
PO6	The Engineer and Society	PO12	Life long learning

Assessment Pattern:

Bloom's Category	Continuous Assessment Test (Internal Exam) Marks in percentage	End Semester Examination Marks in percentage
Remember	20	20
Understand	20	20
Apply	60	60
Analyse		
Evaluate		
Create		

PREPARATI ON FOR THE LABORATORY SESSION

GENERAL INSTRUCTIONS TO STUDENTS

1. Read carefully and understand the description of the experiment in the lab manual. You may go to the lab at an earlier date to look at the experimental facility and understand it better. Consult the appropriate references to be completely familiar with the concepts and hardware.

2. Make sure that your observation for previous week experiment is evaluated by the faculty member and you have transferred all the contents to your record before entering to the lab/workshop.

3. At the beginning of the class, if the faculty or the instructor finds that a student is not adequately prepared, they will be marked as absent and not be allowed to perform the experiment.

4. Bring necessary material needed (writing materials, graphs, calculators, etc.) to perform the required preliminary analysis. It is a good idea to do sample calculations and as much of the analysis as possible during the session. Faculty help will be available. Errors in the procedure may thus be easily detected and rectified.

5. Please actively participate in class and don't hesitate to ask questions. Please utilize the teaching assistants fully. To encourage you to be prepared and to read the lab manual before coming to the laboratory, unannounced questions may be asked at any time during the lab.

6. Carelessness in personal conduct or in handling equipment may result in serious injury to the individual or the equipment. Do not run near moving machinery/equipment. Always be on the alert for strange sounds. Guard against entangling clothes in moving parts of machinery.

7. Students must follow the proper dress code inside the laboratory. To protect clothing from dirt, wear a lab coat. Long hair should be tied back. Shoes covering the whole foot will have to be worn.

8. In performing the experiments, please proceed carefully to minimize any water spills, especially on the electric circuits and wire.

9. Maintain silence, order and discipline inside the lab. Don't use cell phones inside the laboratory.

10. Any injury no matter how small must be reported to the instructor immediately.

11. Check with faculty members one week before the experiment to make sure that you have the handout for that experiment and all the apparatus.

AFTER THE LABORATORY SESSION

1. Clean up your work area.

2. Check with the technician before you leave.

3. Make sure you understand what kind of report is to be prepared and due submission of record is next lab class.

4. Do sample calculations and some preliminary work to verify that the experiment was successful

MAKE-UPS AND LATE WORK

Students must participate in all laboratory exercises as scheduled. They must obtain permission from the faculty member for absence, which would be granted only under justifiable circumstances. In such an event, a student must make arrangements for a make-up laboratory, which will be scheduled when the time is available after completing one cycle. Late submission will be awarded less mark for record and internals and zero in worst cases.

LABORATORY POLICIES

1. Food, beverages & mobile phones are not allowed in the laboratory at any time.

2. Do not sit or place anything on instrument benches.

3. Organizing laboratory experiments requires the help of laboratory technicians and staff. Be punctual.

SYLLABUS

OPERATING SYSTEMS LAB

* mandatory

1. Basic Linux commands
2. Shell programming

   -Command syntax

   -Write simple functions with basic tests, loops, patterns
3. System calls of Linux operating system:*

   fork, exec, getpid, exit, wait, close, stat, opendir, readdir
4. Write programs using the I/O system calls of Linux operating system (open, read, write)
5. Implement programs for Inter Process Communication using Shared Memory *
6. Implement Semaphores*
7. Implementation of CPU scheduling algorithms. a) Round Robin b) SJF c) FCFS d) Priority *
8. Implementation of the Memory Allocation Methods for fixed partition*

   a) First Fit b) Worst Fit c) Best Fit
9. Implement l page replacement algorithms a) FIFO b) LRU c) LFU*
10. Implement the banker's algorithm for deadlock avoidance. *
11. Implementation of Deadlock detection algorithm
12. Simulate file allocation strategies.

    b) Sequential b) Indexed c) Linked
13. Simulate disk scheduling algorithms. *

    c) FCFS b)SCAN c) C-SCAN

OPERATING SYSTEMS LAB - PRACTICE QUESTIONS

1. Write a program to create a process in linux.

2. Write programs using the following system calls of Linux operating system:

fork, exec, getpid, exit, wait, close, stat, opendir, readdir

3. Write programs using the I/O system calls of Linux operating system (open, read, write)

Given the list of processes, their CPU burst times and arrival times, display/print the

Gantt chart for FCFS and SJF. For each of the scheduling policies, compute and print the

average waiting time and average turnaround time

5. Write a C program to simulate following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.

a)FCFS b) SJF c) Round Robin (pre-emptive) d) Priority

6. Write a C program to simulate following contiguous memory allocation techniques

a) Worst-fit b) Best-fit c) First-fit

7. Write a C program to simulate paging technique of memory management.

8. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

9. Write a C program to simulate disk scheduling algorithms

a) FCFS b) SCAN c) C-SCAN

10. Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) LFU

11. Write a C program to simulate producer-consumer problem using semaphores.

12. Write a program for file manipulation for display a file and directory in memory.

13. Write a program to simulate algorithm for deadlock prevention.

14. Write a C program to simulate following file allocation strategies.

a)Sequential b) Indexed c) Linked

INDEX

EXP NO	EXPERIMENT NAME	PAGE NO	MARKS	SIGN
1	SYSTEM CALLS OF LINUX OPERATING SYSTEM:	12		
2	INTER PROCESS COMMUNICATION USING SHARED MEMORY	16		
3	SEMAPHORES IMPLEMENTATION	23		
4	IMPLEMENTATION OF CPU SCHEDULING ALGORITHMS. A) FCFS B) SJF C) PRIORITY D) ROUND ROBIN	26		
5	IMPLEMENTATION OF THE MEMORY ALLOCATION METHODS FOR FIXED	43		
6	PAGE REPLACEMENT ALGORITHMS A) FIFO B) LRU C) LFU	52		
7	BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE.	63		
8	DISK SCHEDULING ALGORITHMS. A) FCFS B)SCAN	70		
FINAL VERIFICATION BY THE FACULTY				
TOTAL MARKS:				

INTERNAL EXAMINER **EXTERNAL EXAMINER**

EXPERIMENT – 1

GETTING USED TO LINUX BASIC COMMANDS, DIRECTORY STRUCTURE,

AIM:

To familiarize with Linux basic commands, directory structure, file and directory operations.

DESCRIPTION

Linux basic commands:

(a) cd: - It is used to change directory.

- The format of this command is:

cd <space> <directory_name> Eg: cd bin

- Here "/" indicates root directory and "~" indicates home directory.

-To move step by step backwards we use:

cd<space>..

(b) ls: - It is used to list all the files in the current directory.

-The hidden files in the directory can be listed using the command:

ls<space>-a

(c) clear: - It is used to clear the terminal screen.

(d) pwd: - It is used to get the full path to current directory.

(e) cal: - It is used to get the calendar of the current month.

-The whole calendar can be retrieved using: cal<space>-y

(f) date: - It is used to get the current time of the machine.

(g) mkdir: - It is used to make directory in the current location.

- The format is given by:

mkdir<space><directory_name>

Eg: mkdir newfile

(h) rmdir: - It is used to remove directory from the current location.

- The format is given by:

rmdir<space><directory name>

Eg: rmdir newfile

- If the directory is not present then it displays error as:

rmdir: cannot remove <directoryname>:No such file or directory

- If the directory is not empty, it displays the error as:

rmdir: failed to remove<directoryname>: Directory not empty

(i) touch: -It is used to create a file.
-The format is given by:
touch<space><filename>
Eg: touch hi

(j) rm: -It is used to remove a file.
-The format is given by:
rm<space><filename>
Eg: rm hi

(k) mv: -It is used to rename a file.
-The format is given by: mv<space><oldname><newname>
Eg: mv hi hello

(l) cp: -It is used to copy files between directories.
-The format is given by:
cp<space><source><destination>

Eg: cp hello /usr

(m) man: -It is used to get the manual of each of the commands.

-The format is given by:

man<space><command>

(n) whoami: -It is used to get the current user

(o)history: -It is used to display the history details of the terminal.

Directory structure:

The Unix operating system consists of a single file system. All the controlling directories is present in the main file system termed as the root directory of the operating system. The different directories in the root are as follows:

(1) bin: -This is used to store binary files.
-All the executable files will be present here.

(2) boot: -This is used to store all booting related files.

-The initial program to be loaded on to the ram and the kernel is situated in this directory.

(3) dev: -This is used to store all device descriptions.

-All the known devices will have a description and the newly installed devices will have an entry in this directory.

(4) etc: -This is used to store the configuration files of the system.

-It includes details like user login, IP address, hostname, etc.

(5) home: -Each user will have a directory here.

-User login is directed to this and all home directory files are here.

(6)lib/lib64/lib32: -This consists of all the system libraries.

(7) media: - Contents of external drives will be stored here.

(8) mnt: -All the mounted drives details will be stored here.

(9) opt: -This is an optional folder for third part programs.

(10) proc: -This is a special folder that resides in RAM.

-All the processes are here and process ids are displayed as sub-directories which contains its resources.

(11) lost+found: -This is an administrative level directory that contains recovery bits for the deleted files or applications.

(12) root: -This is another administrative level directory to store system administrative files.

(13) sbin: -This also contains system administrative files but can be accessed by all users.

(14) srv: -This contains server related files.

(15) sys: -This contains system related files like power related files.

(16) tmp: -This is used to store temporary files.

(17) usr: -This is used for extra programs like compilers.

(18) var: -This is used to store variable files like system logs.

VIVA QUESTIONS

1. What is the use of tmp
2. What is the use of dev
3. Command used to create a file

EXPERIMENT – 2

INTERPROCESS COMMUNICATION USING SHARED MEMORY

AIM:

Implement programs for Inter Process Communication using Shared Memory

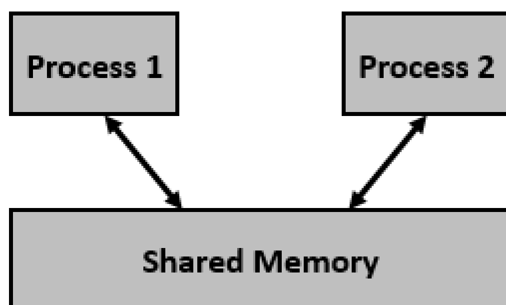
Shared memory is a memory shared between two or more processes. However, why do we need to share memory or some other means of communication?

To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques. As we are already aware, communication can be between related or unrelated processes.

Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.

In this chapter, we will know all about shared memory.



We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this −

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Attach the process to the already created shared memory segment (shmat())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg)
```

The above system call creates or allocates a System V shared memory segment. The arguments that need to be passed are as follows −

The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communiation. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

```
void * shmat(int shmid, const void *shmaddr, int shmflg)
```

The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, shmaddr, is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

The third argument, shmflg, specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

int shmdt(const void *shmaddr)
```

The above system call performs shared memory operation for System V shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is −

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/ipc.h>
#include <sys/shm.h>

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

The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are −

- **IPC_STAT** − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.
- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.
- **IPC_RMID** − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.
- **IPC_INFO** − Returns the information about the shared memory limits and parameters in the structure pointed by buf.
- **SHM_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other

operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Let us consider the following sample program.

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- The program performs writing into the shared memory by write process (shm_write.c) and reading from the shared memory by reading process (shm_read.c)
- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes 5 times the Alphabets from ‘A’ to ‘E’ each of 1023 bytes into the shared memory. Last byte signifies the end of buffer
- Read process would read from the shared memory and write to the standard output
- Reading and writing process actions are performed simultaneously
- After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)
- Reading process performs reading from the shared memory and displays on the output until it gets indication of write process completion (complete variable in struct shmseg)
- Performs reading and writing process for a few times for simplication and also in order to avoid infinite loops and complicating the program

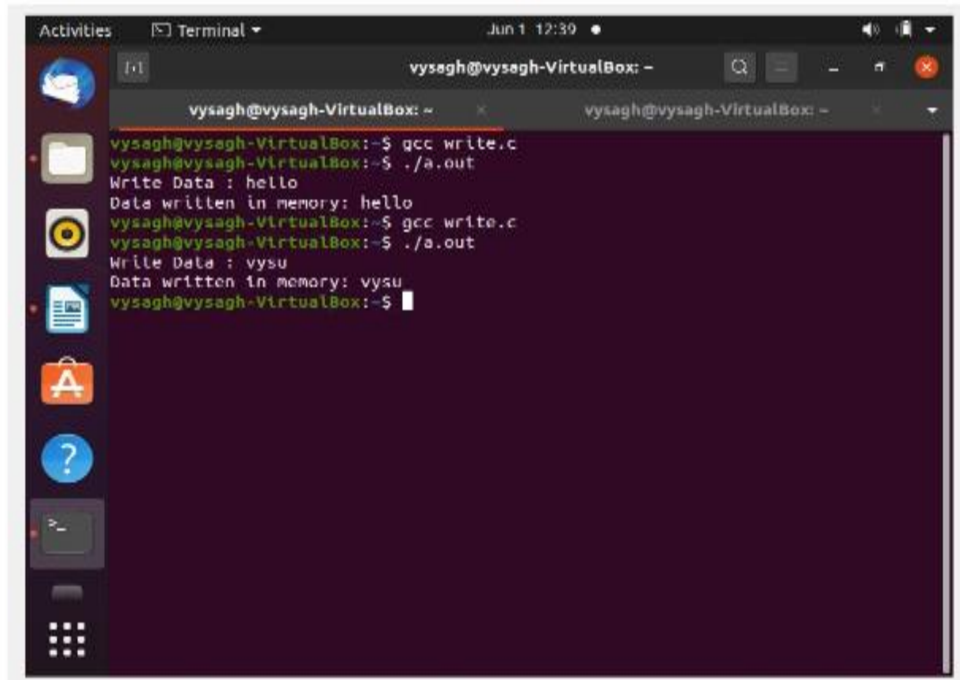
PROGRAM :

```
//Write program
#include &lt;sys/ipc.h&gt;
#include &lt;sys/shm.h&gt;
#include &lt;stdio.h&gt;
int main()
{
// ftok to generate unique key
key_t key = ftok(&quot;shmfile&quot;,65);
// shmget returns an identifier in shmid
```

```
int shmid = shmget(key,1024,0666|IPC_CREAT);
// shmat to attach to shared memory
char *str = (char*) shmat(shmid,(void*)0,0);
printf(&quot;Write Data : &quot;);
scanf(&quot;%s&quot;, str);
printf(&quot;Data written in memory: %s\n&quot;,str);
//detach from shared memory
shmdt(str);
return 0;
}

//Read program
#include &lt;sys/ipc.h&gt;
#include &lt;sys/shm.h&gt;
#include &lt;stdio.h&gt;
int main()
{
// ftok to generate unique key
key_t key = ftok(&quot;shmfile&quot;,65);
// shmget returns an identifier in shmid
int shmid = shmget(key,1024,0666|IPC_CREAT);
// shmat to attach to shared memory
char *str = (char*) shmat(shmid,(void*)0,0);
printf(&quot;Data read from memory: %s\n&quot;,str);
//detach from shared memory
shmdt(str);
// destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);
return 0;
}
```

Output:



VIVA QUESTIONS

1. Explain shared memory concept
2. What is readers writers problem
3. What are the methods used for inter process communication

EXPERIMENT – 3

SEMAPHORES IMPLEMENTATION

AIM:

Write a C program to implement the Producer consumer problem using semaphores

Semaphores in operating system

Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi-processing environment.
The two most common kinds of semaphores are counting semaphores and binary semaphores.
Semaphores are of two types:

1. **Binary Semaphore –** This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement solution of critical section problem with multiple processes.
2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

P and V are the two operations which can be used to access and change the value of semaphore variable

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.
3. The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.
4. A critical section is surrounded by both operations to implement process synchronization. See below image. Critical section of Process P is in between P and V operation.

PROGRAM:

```
#include<stdio.h>
void main()
{
        int buffer[10],bufsize = 5, in, out, pro, cons,
        choice; in=out=0;

        do{

            printf("\n1 --- Produce\t2 --- Consume\t3 ---
            Exit\n"); printf("Choice ?[1/2/3] : ");
            scanf("%d",&choic

            e); switch(choice){

                case 1: if((in+1)%bufsize == out)
                        printf("Buffer          is
                        full.\n");
                    else{
                        printf("Enter production value :
                        "); scanf("%d",&pro);
                        buffer[in] = pro;
                        in = (in + 1) % bufsize;
                    }
                    break;
                case 2: if(in == out)
                        printf("Buffer               is
                    empty.\n"); else{
                        cons = buffer[out];
                        printf("\nConsumed        Product        :
                        %d\n",cons); out = (out+1) % bufsize;
                    }
            }

        }while(choice!=3);
}
```

OUTPUT

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 1
Enter production value : 100

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 1
Enter production value : 200

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 1
Enter production value : 300

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 1
Enter production value : 400

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 1
Buffer is full.

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 500

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 1
Buffer is full.

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 2

Consumed Product : 100

1 --- Produce 2 --- Consume 3 --- Exit
Choice ?[1/2/3] : 3

VIVA QUESTIONS

1. What is semaphore
2. What are the types of semaphores
3. What is producer consumer problem

EXPERIMENT – 4

IMPLEMENTATION OF CPU SCHEDULING ALGORITHMS

AIM

Implementation of CPU scheduling algorithms.

a) Round Robin

b) SJF

c) FCFS

d)Priority

FCFS: First Come First Serve Scheduling

It is the simplest algorithm to implement.

The process with the minimal arrival time will get the CPU first.

 The lesser the arrival time, the sooner will the process gets the CPU.

 It is the non-pre-emptive type of scheduling.

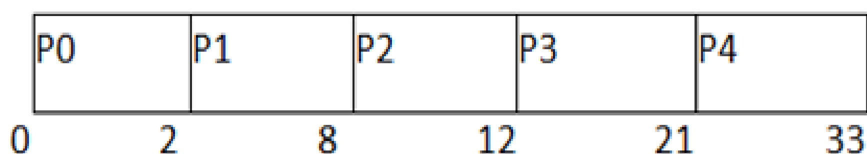
 The Turnaround time and the waiting time are calculated by using the following formula.

Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turnaround time - Burst Time

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4
3	3	9	21	18	9
4	4	12	33	29	17

Avg Waiting Time=31/5

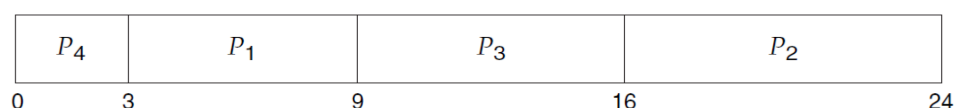


Shortest-Job-First Scheduling

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 CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next- CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.We use the term SJF because most people and textbooks use this term to refer to this type of scheduling. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9 milliseconds for process *P*3, and 0 milliseconds for process *P*4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

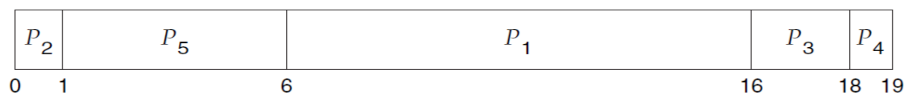
Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. Apriority is associated with each process, and the CPUis 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 thepriority (*p*) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority.

Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority. As an example, consider the following set of processes, assumed to have arrived at time 0 in the order *P*1, *P*2, · · ·, *P*5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Round-Robin Scheduling

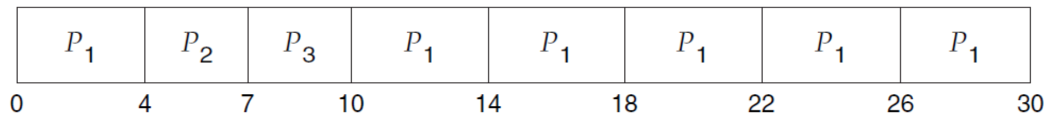
The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. *P*1 waits for 6 milliseconds (10 - 4), *P*2 waits for 4 milliseconds, and *P*3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66

PROGRAM

1. FIRST COME FIRST SERVE (FCFS)

```
#include<stdio.h>

void main()

{

int i=0,j=0,b[i],g[20],p[20],w[20],t[20],a[20],n=0,m;

float avgw=0,avgt=0;

printf("Enter the number of process : ");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("Process ID : ");

scanf("%d",&p[i]);

printf("Burst Time : ");

scanf("%d",&b[i]);

printf("Arrival Time: ");

scanf("%d",&a[i]);

}

int temp=0;

for(i=0;i<n-1;i++)

{

for(j=0;j<n-1;j++)

{

if(a[j]>a[j+1])

{

temp=a[j];
```

```
a[j]=a[j+1];
a[j+1]=temp;
temp=b[j];
b[j]=b[j+1];
b[j+1]=temp;
temp=p[j];
p[j]=p[j+1];
p[j+1]=temp;
}
}
}
g[0]=0;
for(i=0;i<=n;i++)
g[i+1]=g[i]+b[i];
for(i=0;i<n;i++)
{
t[i]=g[i+1]-a[i];
w[i]=t[i]-b[i];
avgw+=w[i];
avgt+=t[i];
}
avgw=avgw/n;
avgt=avgt/n;
printf("pid\tarrivalT\tBrustT\tCompletionT\tWaitingtime\tTurnaroundTi\n");
for(i=0;i<n;i++)
{
```

```
printf("%d\t%d\t%d\t%d\t\t%d\t\t\t%d\n",p[i],a[i],b[i],g[i+1],w[i],t[i]);

}

printf("\nAverage waiting time %f",avgw);

printf("\nAverage turnarround time %f",avgt);

}
```

Output

```
Enter the number of process : 5
Process ID : 1
Burst Time : 4
Arrival Time: 0
Process ID : 2
Burst Time : 3
Arrival Time: 1
Process ID : 3
Burst Time : 1
Arrival Time: 2
Process ID : 4
Burst Time : 2
Arrival Time: 3
Process ID : 5
Burst Time : 5
Arrival Time: 4
```

Pid	arrival	BrustT	CompletionT	Waitingtime	TurnaroundTi
1	0	4	4	0	4
2	1	3	7	3	6
3	2	1	8	5	6
4	3	2	10	5	7
5	4	5	15	6	11

2.SHORTEST JOB FIRST

```
#include<stdio.h>
void main()
{
        int i=0,j=0,p[i],b[i],g[20],w[20],t[20],a[20],n=0,m;
        int k=1,min=0,btime=0;
        float avgw=0,avgt=0;
        printf("Enter the number of process : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nProcess    id    :    ");
                scanf("%d",&p[i]);

                printf("Burst    Time    :    ");
                scanf("%d",&b[i]);


                printf("Arrival Time: ");
                scanf("%d",&a[i]);
        }

  //sort the jobs based on burst time.
        int temp=0;
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
```

```
                    temp=a[j];
                    a[j]=a[j+1];
                    a[j+1]=temp;

                    temp=b[j];
                    b[j]=b[j+1];
                    b[j+1]=temp;


                    temp=p[j];
                    p[j]=p[j+1];
                    p[j+1]=temp;
                }
            }
        }

        for(i=0;i<n;i++)
        {
            btime=btime+b[i];
            min=b[k];
            for(j=k;j<n;j++)
            {
                if(btime >= a[j] && b[j]<min)
                {
                    temp=a[j];
                    a[j]=a[j-1];
                    a[j-1]=temp;

                    temp=b[j];
                    b[j]=b[j-1];
                    b[j-1]=temp;
                    temp=p[j]

                }
            } k++;
```

```
        }

        g[0]=a[0];
        for(i=0;i<n;i++)
        {
                g[i+1]=g[i]+b[i];
                if(g[i]<a[i])
                        g[i]=a[i];
        }

        for(i=0;i<n;i++)
        {

                t[i]=g[i+1]-a[i];
                w[i]=t[i]-b[i];
                avgw+=w[i];
                avgt+=t[i];

        }
        avgw=avgw/n;
        avgt=avgt/n;
        printf("pid\tBrustTime\tGantChart\tWaiting time\t\tTurnarround Time\n");
        for(i=0;i<n;i++)
        {
                printf(" %d\t %d\t\t%d-%d\t\t%d\t\t\t%d\n",p[i],b[i],g[i],g[i+1],w[i],t[i]);
        }
        printf("\nAverage    waiting    time    %f",avgw);
        printf("\nAverage turnarround time %f\n",avgt);


}
```

OUTPUT

Enter the number of process : 5

Process id : 1
Burst Time : 7
Arrival Time: 0

Process id : 2
Burst Time : 5

Arrival Time: 1

Process id : 3 Burst
Time : 1 Arrival
Time: 2

Process id : 4 Burst
Time : 2 Arrival
Time: 3

Process id : 5 Burst
Time : 8 Arrival
Time:

pid	Brust Time	GantChart	Waiting time	Turnarround Time
8	7	0-7	0	7
3	1	7-8	5	6
4	2	8-10	5	7
2	5	10-15	9	14
5	8	15-23	11	19

Average waiting time 6.000000 Average turnaround time 10.600000

3.PRIORITY SCHEDULING

```
#include<stdio.h>
int main()
{
    int burst_time[20], process[20], waiting_time[20], turnaround_time[20], priority[20];
    int i, j, limit, sum = 0, position, temp;
    float average_wait_time, average_turnaround_time;
    printf("Enter Total Number of Processes:\t");
    scanf("%d", &limit);
    printf("\nEnter Burst Time and Priority For %d Processes\n", limit);
    for(i = 0; i < limit; i++)
    {
        printf("\nProcess[%d]\n", i + 1);
        printf("Process Burst Time:\t");
        scanf("%d", &burst_time[i]);
        printf("Process Priority:\t");
        scanf("%d", &priority[i]);
        process[i] = i + 1;
    }
    for(i = 0; i < limit; i++)
    {
        position = i;
        for(j = i + 1; j < limit; j++)
        {
            if(priority[j] < priority[position])
            {
                position = j;
            }
        }
        temp = priority[i];
```

```
            priority[i] = priority[position];
            priority[position]     =     temp;
            temp = burst_time[i];
            burst_time[i] = burst_time[position];
            burst_time[position] = temp;
            temp = process[i];
            process[i] = process[position];
            process[position] = temp;
        }
        waiting_time[0]    =    0;
        for(i = 1; i < limit; i++)
        {
            waiting_time[i]     =     0;
            for(j = 0; j < i; j++)
            {
                waiting_time[i] = waiting_time[i] + burst_time[j];
            }
            sum = sum + waiting_time[i];
        }
        average_wait_time = sum / limit;
        sum = 0;
        printf("\nProcess ID\t\tBurst Time\t Waiting Time\t Turnaround Time\n");
        for(i = 0; i < limit; i++)
        {
            turnaround_time[i] = burst_time[i] + waiting_time[i];
            sum = sum + turnaround_time[i];
         printf("\nProcess[%d]\t\t%d\t\t  %d\t\t  %d\n",  process[i],  burst_time[i],  waiting_time[i],
turnaround_time[i]);
        }
        average_turnaround_time = sum / limit;
        printf("\nAverage       Waiting       Time:\t%f",       average_wait_time);
        printf("\nAverage Turnaround Time:\t%f\n", average_turnaround_time);
        return 0;}
```

OUTPUT

Enter the number of process: 3

Process id : 1
Burst Time : 15
Priority: 3

Process id : 2

Burst Time : 10
Priority: 2

Process id : 3 Burst
Time : 90
Priority: 1

pid	Burst Time	Waiting time	Turnaround Time
3	90	0	90
2	10	90	100
1	15	100	115

Average waiting time 63.000000
Average turnaround time 101.000000

4.Round Robin (pre-emptive)

```
#include<stdio.h>
 int main()
{
      int i, limit, total = 0, x, counter = 0, time_quantum;
      int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
      float average_wait_time, average_turnaround_time;
      printf("\nEnter Total Number of Processes:\t");
      scanf("%d", &limit);
      x = limit;
      for(i = 0; i < limit; i++)
```

```
{
    printf("\nEnter Details of Process[%d]\n", i + 1);
    printf("Arrival Time:\t");
    scanf("%d", &arrival_time[i]);
    printf("Burst            Time:\t");
    scanf("%d",    &burst_time[i]);
    temp[i] = burst_time[i];
}
printf("\nEnter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
for(total = 0, i = 0; x != 0;)
{
    if(temp[i] <= time_quantum && temp[i] > 0)


    {
        total = total + temp[i];
        temp[i] = 0;
        counter = 1;
    }
    else if(temp[i] > 0)
    {
        temp[i] = temp[i] - time_quantum;
        total = total + time_quantum;
    }
    if(temp[i] == 0 && counter == 1)
    {
        x--;
        printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst_time[i], total - arrival_time[i],
          total - arrival_time[i] - burst_time[i]);
        wait_time = wait_time + total - arrival_time[i] - burst_time[i];
        turnaround_time = turnaround_time + total - arrival_time[i];
        counter = 0;
```

```
            }
            if(i == limit - 1)
            {
                i = 0;
            }
            else if(arrival_time[i + 1] <= total)
            {
                i++;
            }
            else
            {
                i = 0;
            }
        }
        average_wait_time = wait_time * 1.0 / limit;
        average_turnaround_time = turnaround_time * 1.0 / limit;
        printf("\n\nAverage Waiting Time:\t%f", average_wait_time);
        printf("\nAvg Turnaround Time:\t%f\n", average_turnaround_time);
        return 0;
}
```

OUTPUT

Enter the number of process : 3

Process id : 2
Burst Time : 3
Arrival Time:2

Process id : 3 Burst
Time : 2 Arrival
Time:3

pid	Burst Time	Waiting time	Turnarround Time
3	2	1	3
2	3	9	6

VIVA QUESTIONS

1. What are all the scheduling algorithms?
2. What is CPU Scheduler?
3. What is CPU utilization?
4. What is Throughput?
5. Explain Priority Scheduling algorithm?
6. Explain Round Robin?
7. Explain FCFS(First Come First Served)?

EXPERIMENT – 5

IMPLEMENTATION OF THE MEMORY ALLOCATION METHODS FOR FIXED PARTITION

AIM:

Implementation of the Memory Allocation Methods for fixed partition

a) First Fit

b) Worst Fit

c) Best Fit

Contiguous Memory Allocation Techniques

First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Best Fit

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Worst fit

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

PROGRAM

1. **FIRST FIT**

```
#include<stdio.h>
struct process
{
int ps;
int flag;
} p[50];
struct sizes
{
```

```
int size;
int alloc;
}
s[5];
int main()
{
int i=0,np=0,n=0,j=0;
printf("\n first fit");
printf("\n");
printf("enter the number of blocks \t");
scanf("%d",&n);
printf("\t\t\n enter the size for %d blocks\n",n);
for(i=0;i<n;i++)
{
printf("enter the size for %d block \t",i);
scanf("%d",&s[i].size);
}
printf("\n\t\t enter the number of process\t",i);
scanf("%d",&np);
printf("\n enter the size of %d processors !\t",np);
printf("/n");
for(i=0;i<np;i++)
{
printf("enter the size of process %d\t",i);
scanf("\n%d",&p[i].ps);
}
printf("\n\t\t Allocation of blocks using first fit is as follows\n");
printf("\n\t\t process \t process size\t blocks\n");
for(i=0;i<np;i++)
{
for(j=0;j<n;j++)
{
if(p[i].flag!=1)
{
if(p[i].flag!=1)
```

```
{
if(p[i].ps<=s[j].size)
{
if(!s[j].alloc)
{
p[i].flag=1;
s[j].alloc=1;
printf("\n\t\t %d\t\t\t%d\t%d\t",i,p[i].ps,s[j].size);
}
}
}
}
}
}
for(i=0;i<np;i++)
{
if(p[i].flag!=1)
printf("sorry !!!!!!!process %d must wait as there is no sufficient memory");
}
}
```

OUTPUT

First fit

Enter the number of blocks :5

Enter the size for 5 bocks

Enter the size for 0 block : 100

Enter the size for 1 block : 250

Enter the size for 2 block : 300

Enter the size for 3 block : 50

Enter the size for 4 block : 120

Enter the number of process: 3

Enter the size of process

Enter the size of process 1 40

Enter the size of process 2 200

Enter the size of process 3 300

Allocation of block using first fit is as follows

Process	**process size**	**block**
1	40	100
2	200	250
3	300	300

2. **WORST FIT**

```
#include<stdio.h>
int main()
{
    int fragments[10], blocks[10], files[10];
    int m, n, number_of_blocks, number_of_files, temp, top = 0;
    static int block_arr[10], file_arr[10];
    printf("\nEnter the Total Number of Blocks:\t");
    scanf("%d",&number_of_blocks);
    printf("Enter the Total Number of Files:\t");
    scanf("%d",&number_of_files);
    printf("\nEnter the Size of the Blocks:\n");
    for(m = 0; m < number_of_blocks; m++)
    {
        printf("Block No.[%d]:\t", m + 1);
        scanf("%d", &blocks[m]);
    }
```

```
printf("Enter the Size of the Files:\n");
for(m = 0; m < number_of_files; m++)
{
    printf("File No.[%d]:\t", m + 1);
    scanf("%d", &files[m]);
}
for(m = 0; m < number_of_files; m++)
{
    for(n = 0; n < number_of_blocks; n++)
    {
        if(block_arr[n] != 1)
        {
            temp = blocks[n] - files[m];
            if(temp >= 0)
            {
                if(top < temp)
                {
                    file_arr[m] = n;
                    top = temp;
                }
            }
        }
        fragments[m] = top;
        block_arr[file_arr[m]] = 1;
        top = 0;
    }
```

```
    }
    printf("\nFile Number\tFile Size\tBlock Number\tBlock Size\tFragment");
    for(m = 0; m < number_of_files; m++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", m, files[m], file_arr[m], blocks[file_arr[m]],
fragments[m]);
    }
    printf("\n");
    return 0;
}
```

OUTPUT

Enter total number of blocks : 4

Enter total number of files: 3

Enter the size of blocks

Block No.[1] : 50

Block No.[2] : 75

Block No.[3] : 100

Block No.[4] : 300

Enter the size of the files

File No.[1] : 20

File No.[2] : 100

File No.[3] : 40

File number	**File size**	**Block Number**	**Block Size**	**Fragment**
0	20	3	300	280
1	100	0	50	0
2	40	0	50	1 0

3.BEST FIT

```
#include<stdio.h>
#define MAX 20
int main()
{
int bsize[MAX],fsize[MAX],nb,nf;
int temp,low=10000;
static int bflag[MAX],fflag[MAX];
int i,j;
printf("\n enter the number of blocks");
scanf("%d",&nb);
for(i=1;i<=nb;i++)
{
printf("Enter the size of memory block % d",i);
scanf("%d", &bsize[i]);
}
printf("\n enter the number of files");
scanf("%d",&nf);
for(i=1;i<=nf;i++)
{
printf("\n enetr the size of  file %d",i);
scanf("%d",&fsize[i]);
}
for(i=1;i<=nf;i++)

{
for(j=1;j<=nb;j++)
{
if(bflag[j]!=1)
{
temp=bsize[j]-fsize[i];
if(temp>=0)
{
if(low>temp)
{
```

```
fflag[i]=j;
low=temp;
}
}
}}
bflag[fflag[i]]=1;
low=10000;
}
printf("\n  file no \t file.size\t block no \t block size");
for(i=1;i<=nf;i++)
printf("\n \n %d \t\t%d\t\t%d\t\t%d",i,fsize[i],fflag[i],bsize[fflag[i]]);
}
```

OUTPUT

Enter the number of blocks : 4

Enter the size of memory block 1: 40

Enter the size of memory block 1: 100

Enter the size of memory block 1: 250

Enter the size of memory block 1: 50

Enter the number of files : 3

Enter the size of file 1: 50

Enter the size of file 2: 20

Enter the size of file 3: 225

File no	File size	Block No	Block Size
1	50	4	50
2	20	1	40
23	225	3	250

VIVA QUESTIONS

1. What are memory allocation strategies
2. Explain best fit memory allocation strategy
3. Explain worst fit memory allocation strategy
4. Explain first fit memory allocation strategy

EXPERIMENT – 6

IMPLEMENT PAGE REPLACEMENT ALGORITHMS

AIM:

Implement l page replacement algorithms a) FIFO b) LRU c) LFU

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 with 3 page frames.Find number of page faults

Page reference 1, 3, 0, 3, 5, 6, 3

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Total Page Fault = 6

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**

2.Least Recently Used – (LRU)

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 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

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

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.

3. **LEAST FREQUENTLY USED – (LFU)**

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, with 4 page frame. Find number of page fault.

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

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

PROGRAM

FIFO (FIRST IN FIRST OUT)

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
printf("\n enter the number of pages:\n");
scanf("%d",&n);
printf("\n enter the page number:\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n enter the number of frames:\n"); scanf("%d",&no);
for(i=0;i<no;i++) frame[i]=-1;
j=0;
printf("ref string\t page frames\n");
for(i=1;i<=n;i++)
{
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
if(frame[k]==a[i]) avail=1;
if(avail==0)
{
frame[j]=a[i];
j=(j+1)%no;
count++;
for(k=0;k<no;k++)
```

```
printf("%d\t",frame[k]);
}
printf("\n");
}
printf("page fault is %d",count); getch();
return 0;
}
```

OUTPUT

```
Enter the number of frames :    3

Enter number of pages :         20

Enter page reference string

7   0   1   2   0   3   0   4   2   3   0   3   2   1

0   1   7   0   1

    7   -1   -1
    7    0   -1
    7    0    1
    2    0    1

    2    3    1
    2    3    0
    2    0    3
    4    3    0
    4    2    0
    4    2    3
    0    2    3


    0    1    3
    0    1    2


    7    1    2
    7    0    2
    7    0    1

Total page fault = 15
```

2.LEAST RECENTLY USED

```
#include<stdio.h>
int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;

for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}

int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos,
faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
    for(i = 0; i < no_of_pages; ++i){
     scanf("%d", &pages[i]);
    }

for(i = 0; i < no_of_frames; ++i){
     frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
     flag1 = flag2 = 0;

     for(j = 0; j < no_of_frames; ++j){
```

```
    if(frames[j] == pages[i]){
    counter++;
    time[j] = counter;
   flag1 = flag2 = 1;
   break;
   }
    }

    if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
    counter++;
    faults++;
    frames[j] = pages[i];
    time[j] = counter;
    flag2 = 1;
    break;
    }
    }
    }

    if(flag2 == 0){
    pos = findLRU(time, no_of_frames);
    counter++;
    faults++;
    frames[pos] = pages[i];
    time[pos] = counter;
    }

    printf("\n");

    for(j = 0; j < no_of_frames; ++j){
    printf("%d\t", frames[j]);
    }
}
```

```
printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

OUTPUT

```
Enter the number of frames :    3

Enter number of pages :         20

Enter page reference string

7   0   1   2   0   3   0   4   2   3   0   3   2   1

0   1   7   0   1

    7     -1    -1
    7      0    -1
    7      0     1
    2      0     1
    2      0     1
    2      0     3
    2      0     3
    4      0     3
    4      0     2
    4      3     3
    0      3     2
    0      3     2
    0      3     2
    1      3     2
    1      3     2
    1      0     2
    1      0     2
    1      0     7
    1      0     7
    1      0     7

Total page fault = 12
```

3. LEAST FREQUENTLY USED

```
#include<stdio.h>
int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos,
max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                flag1 = flag2 = 1;
                break;
            }
        }

        if(flag1 == 0){
            for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == -1){
```

```
                faults++;
                frames[j] = pages[i];
                flag2 = 1;
                break;
            }
        }
    }

    if(flag2 == 0){
     flag3 =0;

        for(j = 0; j < no_of_frames; ++j){
         temp[j] = -1;

         for(k = i + 1; k < no_of_pages; ++k){
         if(frames[j] == pages[k]){
         temp[j] = k;
         break;
         }
         }
        }

        for(j = 0; j < no_of_frames; ++j){
         if(temp[j] == -1){
         pos = j;
         flag3 = 1;
         break;
         }
        }

        if(flag3 ==0){
         max = temp[0];
         pos = 0;

         for(j = 1; j < no_of_frames; ++j){
```

```
            if(temp[j] > max){
            max = temp[j];
            pos = j;
            }
            }
           }
frames[pos] = pages[i];
faults++;
        }

        printf("\n");

        for(j = 0; j < no_of_frames; ++j){
           printf("%d\t", frames[j]);
        }
    }

    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

OUTPUT

Enter the number of frames :   3

Enter number of pages :   20

Enter page reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1

0 1 7 0 1

```
7    -1    -1
7     0    -1
7     0     1
2     0     1
2     0     1
2     0     3
2     4     3
2     4     3
```

```
2    4    3
2    0    3
2    0    3
2    0    3
2    0    1
2    0    1
2    0    1
2    0    1
7    0    1
7    0    1
7    0    1
```

Total page fault = 9

VIVA QUESTIONS

1. What is the purpose of page replacement algorithms
2. What are the page replacement algorithms
3. What is LFU
4. What is least recently used page replacement algorithm

EXPERIMENT – 7

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

AIM:

Implement the banker's algorithm for deadlock avoidance**.**

The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

For the Banker's algorithm to work, it needs to know three things:
1. How much of each resource each process could possibly request[MAX]
2. How much of each resource each process is currently holding[ALLOCATED]
3. How much of each resource the system currently has available[AVAILABLE]

Resources may be allocated to a process only if the amount of resources requested is less than or equal to the amount available; otherwise, the process waits until resources are available.The Banker's Algorithm derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. By using the Banker's algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated; otherwise the customer must wait until some other customer deposits enough.Basic data structures to be maintained to implement the

Banker's Algorithm:

Let n be the number of processes in the system and m be the number of resource types. Then we need the following data structures:

1. Available: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.
2. Max: An n×m matrix defines the maximum demand of each process. If Max[i,j] = k, then Pi may request at most k instances of resource type Rj.

3. Allocation: An n×m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.
4. Need: An n×m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then Pi may need k more instances of resource type Rj to complete the task.

PROGRAM

```
#include<stdio.h>
struct pro{
int all[10],max[10],need[10];
int flag;
};
int i,j,pno,r,nr,id,k=0,safe=0,exec,count=0,wait=0,max_err=0;
struct pro p[10];
int aval[10],seq[10];
void safeState()
{
while(count!=pno){
safe = 0;
for(i=0;i<pno;i++){
if(p[i].flag){
exec = r;
for(j=0;j<r;j++)
{
if(p[i].need[j]>aval[j]){
exec =0;
}
}
if(exec == r){
for(j=0;j<r;j++){
aval[j]+=p[i].all[j];
}
p[i].flag = 0;
seq[k++] = i;
safe = 1;
```

```
count++;
}
}
}
if(!safe)
{
printf("System is in Unsafe State\n");
break;
}
}
if(safe){

printf("\n\nSystem is in safestate \n");
printf("Safe State Sequence \n");
for(i=0;i<k;i++)
printf("P[%d] ",seq[i]);
printf("\n\n");
}
}
void reqRes(){
printf("\nRequest for new Resourses");
printf("\nProcess id ? ");
scanf("%d",&id);
printf("Enter new Request details ");
for(i=0;i<r;i++){
scanf("%d",&nr);
if( nr <= p[id].need[i])
{
if( nr <= aval[i]){
aval[i] -= nr;
p[id].all[i] += nr;
p[id].need[i] -= nr;
}
else
wait = 1;
}
else
```

```
max_err = 1;
}
if(!max_err && !wait)
safeState();
else if(max_err){
printf("\nProcess has exceeded its maximum usage \n");
}
else{
printf("\nProcess need to wait\n");
}
}
void main()
{
printf("Enter no of process ");
scanf("%d",&pno);
printf("Enter no. of resourses ");
scanf("%d",&r);
printf("Enter Available Resourse of each type ");
for(i=0;i<r;i++){

scanf("%d",&aval[i]);
}
printf("\n\n---Resourse Details---");
for(i=0;i<pno;i++){
printf("\nResourses for process %d\n",i);
printf("\nAllocation Matrix\n");
for(j=0;j<r;j++){
scanf("%d",&p[i].all[j]);
}
printf("Maximum Resourse Request \n");
for(j=0;j<r;j++){
scanf("%d",&p[i].max[j]);
}
p[i].flag = 1;
}
// Calcualting need
for(i=0;i<pno;i++){
```

```
for(j=0;j<r;j++){
p[i].need[j] = p[i].max[j] - p[i].all[j];
}
}
//Print Current Details
printf("\nProcess Details\n");
printf("Pid\t\tAllocattion\t\tMax\t\tNeed\n");
for(i=0;i<pno;i++)
{
printf("%d\t\t",i);
for(j=0;j<r;j++){
printf("%d ",p[i].all[j]);
}
printf("\t\t");
for(j=0;j<r;j++){
printf("%d ",p[i].max[j]);
}
printf("\t\t");
for(j=0;j<r;j++){
printf("%d ",p[i].need[j]);
}
printf("\n");
}
//Determine Current State in Safe State
safeState();
int ch=1;
do{

printf("Request new resourse ?[0/1] :");
scanf("%d",&ch);
if(ch)
reqRes();
}while(ch!=0);
//end:printf("\n");
}
```

OUTPUT

Enter no of process 5
Enter no. of resourses 3
Enter Available Resourse of each type 3
3
2
---Resourse Details---
Resourses for process 0

Allocation Matrix
0 1 0
Maximum Resourse Request
7 5 3

Resourses for process 1

Allocation Matrix
3 0 2
Maximum Resourse Request
3 2 2

Resourses for process 2

Allocation Matrix
3 0 2
Maximum Resourse Request
9 0 2

Resourses for process 3

Allocation Matrix
2 1 1
Maximum Resourse Request
2 2 2

Resourses for process 4

Allocation Matrix 0

0 2
Maximum Resource Request
4 3 3

Process Details

Pid	Allocation	Max	Need
0	0 1 0	7 5 3	7 4 3
1	3 0 2	3 2 2	0 2 0
2	3 0 2	9 0 2	6 0 0
3	2 1 1	2 2 2	0 1 1
4	0 0 2	4 3 3	4 3 1

System is in safe state

Safe State Sequence

P[1] P[2] P[3] P[4] P[0]

VIVA QUESTIONS

1. What is deadlock
2. What is deadlock avoidance
3. What is the purpose of banker’s algorithm

EXPERIMENT – 8

DISK SCHEDULING ALGORITHMS.

AIM

Simulate disk scheduling algorithms**.**

a) FCFS b) SCAN c) C-SCAN

First Come -First Serve (FCFS)

It is the simplest form of disk scheduling algorithms. The I/O requests are served or processes according to their arrival. The request arrives first will be accessed and served first. Since it follows the order of arrival, it causes the wild swings from the innermost to the outermost tracks of the disk and vice versa. The farther the location of the request being serviced by the read/write head from its current location, the higher the seek time will be.

Example:

Given the following track requests in the disk queue, compute for the Total Head Movement2 (THM) of the read/write head: 95, 180, 34, 119, 11, 123, 62, and 64.

Consider that the read/write head is positioned at location 50. Prior to this track location 199 was serviced. Show the total head movement for a 200 track disk (0-199).

Solution:

Total Head Movement Computation:

(THM) = (180-50) + (180-34) + (119-34) + (119-11) + (123-11) + (123-62) + (64-62)

= 130 + 146 + 85 + 108 + 112 + 61 + 2 (THM) = 644 tracks

Assuming a seek rate of 5 milliseconds is given, we compute for the seek time using the formula: Seek Time = THM * Seek rate = 644 * 5 ms

Seek Time = 3,220ms

There are some requests that are far from the current location of the R/W head which causes theaccess arm to travel from innermost to the outermost tracks of the disk or vice versa.In this example, it had a total of 644 tracks and a seek time of 3,220 milliseconds. Based on the result, this algorithm produced higher seek rate since it follows the arrival of the track requests.

SCAN Scheduling Algorithm

This algorithm is performed by moving the R/W head back-and-forth to the innermost and outermost track. As it scans the tracks from end to end, it process all the requests found in the direction it is headed. This will ensure that all track requests, whether in the outermost, middle or innermost location, will be traversed by the access arm thereby finding all the requests. This is also known as the Elevator algorithm. Using the same sets of example in
FCFS the solution are as follows:

(THM) = (50-0) + (180-0) = 50 + 180

(THM) = 230

Seek Time = THM * Seek rate = 230* 5ms

Seek Time = 1,150ms

This algorithm works like an elevator does. In the algorithm example, it scans down towards the nearest end and when it reached the bottom it scans up servicing the requests that it did not getgoing down. If a request comes in after it has been scanned, it will not be serviced until the process comes back down 8 or moves back up. This process moved a total of 230 tracks and a seek time of 1,150.

Circular SCAN (C-SCAN) Algorithm

This algorithm is a modified version of the SCAN algorithm. C-SCAN sweeps the disk from end-to-end, but as soon it reaches one of the end tracks it then moves to the other end track without servicing any requesting location. As soon as it reaches the other end track it then starts servicing and grants requests headed to its direction. This algorithm improves the unfair situation of the end tracks against the middle tracks. Using the same sets of example in FCFS the solution are as follows: alpha3 symbol (α) was used to represent the dash line. This return sweeps is sometimes given a numerical value which is included in the computation of the THM. As analogy, this can be compared with the carriage return lever of a typewriter. Once it is pulled to the right most direction, it resets the typing point to the leftmost margin of the report. A typist is not supposed to type during the movement of the carriage return lever because the line spacing is being adjusted. The frequent use of this lever consumes time, same with the time consumed when the R/W head is reset to its
starting position.

Assume that in this example, α has a value of 20ms, the computation would be as follows:

(THM) = (50-0) + (199-62) + α = 50 + 137 + 20

(THM) = 207 tracks

Seek Time = THM * Seek rate = 187 * 5ms

Seek Time = 935ms

The computation of the seek time excluded the alpha value because it is not an actual seek or search of a disk request but a reset of the access arm to the starting position.

PROGRAM

1. First Come -First Serve (FCFS)

```
#include<conio.h>
#include<stdio.h>
int main()
{
int i,j,sum=0,n;
int ar[20],tm[20];
int disk;
printf("enter number of location\t");
scanf("%d",&n);
printf("enter position of head\t");
scanf("%d",&disk);
printf("enter elements of disk queue\n");
for(i=0;i<n;i++)
{
scanf("%d",&ar[i]);
tm[i]=disk-ar[i];
if(tm[i]<0)
{
tm[i]=ar[i]-disk;
}
disk=ar[i];
sum=sum+tm[i];
}
/*for(i=0;i<n;i++)
{
printf("\n%d",tm[i]);
} */
printf("\nmovement of total cylinders %d",sum);
getch();
```

```
return 0;
}
```

OUTPUT

```
Enter number of requests
8
Enter initial head position
53
Enter the reqest sequence
95    180   34    119   11    123   62    64
```

Movement of total cylinders : 641

2. **SCAN Scheduling**

```
#include<stdio.h>

#include<stdlib.h>

int main()

{

   int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

   printf("Enter the number of Requests\n");

   scanf("%d",&n);

   printf("Enter the Requests sequence\n");

   for(i=0;i<n;i++)

    scanf("%d",&RQ[i]);

   printf("Enter initial head position\n");

   scanf("%d",&initial);

   printf("Enter total disk size\n");

   scanf("%d",&size);

   printf("Enter the head movement direction for high 1 and for low 0\n");

   scanf("%d",&move);


   // logic for Scan disk scheduling
```

```
    /*logic for sort the request array */
for(i=0;i<n;i++)
{
    for(j=0;j<n-i-1;j++)
    {
        if(RQ[j]>RQ[j+1])
        {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
        }

    }
}

int index;
for(i=0;i<n;i++)
{
    if(initial<RQ[i])
    {
        index=i;
        break;
    }
}

// if movement is towards high value
```

```
if(move==1)
{
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    //  last movement for max size
    TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
    initial = size-1;
    for(i=index-1;i>=0;i--)
    {
         TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];

    }
}
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    //  last movement for min size
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
    initial =0;
```

```
        for(i=index;i<n;i++)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];

        }
    }

    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

OUTPUT

Enter number of requests
8
Enter the reqest sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter movement direction for high 1 and for low 0
1
Total head movement is 337

3. Circular SCAN (C-SCAN)

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
     scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);

    // logic for C-Scan disk scheduling

        /*logic for sort the request array */
    for(i=0;i<n;i++)
    {
        for( j=0;j<n-i-1;j++)
        {
            if(RQ[j]>RQ[j+1])
```

```
            {
                int temp;
                temp=RQ[j];
                RQ[j]=RQ[j+1];
                RQ[j+1]=temp;
            }

        }
    }

    int index;
    for(i=0;i<n;i++)
    {
        if(initial<RQ[i])
        {
            index=i;
            break;
        }
    }

    // if movement is towards high value
    if(move==1)
    {
        for(i=index;i<n;i++)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
```

```
    //  last movement for max size
    TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
    /*movement max to min disk */
    TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
    initial=0;
    for( i=0;i<index;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
       TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
       initial=RQ[i];
    }
    //  last movement for min size
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
    /*movement min to max disk */
    TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
    initial =size-1;
    for(i=n-1;i>=index;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
```

```
        }
    }
    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

OUTPUT

```
Enter number of requests
8
Enter the reqest sequence
95   180      34        119      11        123      62        64
Enter initial head position
50
Enter total disk size
200
Enter movement direction for high 1 and for low 0
1
Total head movement is 382
```

VIVA QUESTIONS

1. What is disk scheduling
2. Differentiate SCAN and C SCAN
3. What is FCFS disk scheduling
4. How to calculate total head movements