Savitribai Phule Pune University
(Formerly University of Pune)

T.Y.B.Sc.(Computer Science)

With

Major: Computer Science

(Faculty of Science and Technology)

(For Colleges Affiliated to Savitribai Phule Pune University)

Choice Based Credit System (CBCS) Syllabus Under National Education Policy (NEP)

To be implemented from Academic Year 2026-2027

Title of the Course: B.Sc.(Computer Science)

Savitribai Phule Pune University
(Formerly University of Pune)

T.Y.B.Sc.(Computer Science) Semester V

WORKBOOK

Course Code: CS-305-MJ-P

(Practical Course based on CS-302-MJ-T)

WORKBOOK

T.Y.B.Sc. (Computer Science)

Subject Name- Operating Systems

Course Type: **Major Core** Course Code: **CS 305-MJ-P**

Course Title: Practical Course based on **CS-302-MJ-T**

SEMESTER V

Student Name:			
College:			
Roll No:		**Exam Seat No:**	
Year:		**Division:**	

BOARD OF STUDIES

1. Dr. Patil Ranjeet
2. Dr. Joshi vinayak
3. Dr. Wani Vilas
4. Dr. Mulay Prashant
5. Dr. Sardesai Anjali
6. Dr. Shelar Madhukar
7. Dr. Bharambe Manisha
8. Dr. Deshpande Madhuri
9. Dr. Kardile Vilas
10. Dr. Korpal Ritambhara
11. Dr. Gangurde Rajendra
12. Dr. Manza Ramesh
13. Dr. Bachav Archana
14. Dr. Bhat Shridhar

Co-ordinators

- **Dr. Prashant Mulay**

Annasaheb Magar College, Hadapsar , Pune.
Member, BOS Computer Science, Savitribai Phule Pune University

- **Dr. Sardesai Anjali**

Modern College of Arts, Science and Commerce, Shivajinagar, Pune.
Member, BOS Computer Science, Savitribai Phule Pune University

Editor

Dr. Amale B.B. Padmashri Vikhe Patil College of Arts, Science & Commerce Pravaranagar

Prepared by:

Ms.Shubhangi Ghule	Dr. D.Y. Patil Arts , Science & CommerceCollege Pimpri
Ms.Rohini S. Kapse	MVP Samaj;s KRT Arts, BH Commerce and AM Science (KTHM) College Nashik
Ms.Sunita N. Deore	MVP S.V.K.T College Devali Camp Nashik
Ms. Nirmal J.M.	Padmashri Vikhe Patil College of Arts, Science & Commerce Pravaranaga
Dr. Yogini D. Salunke	Karm. Abasaheb Alias N.M.Sonawane Arts, Commerce and Science College Satana
Ms. Priynka N. Kutwad	PVG's College of Science & Commerce ,Pune

Table of Contents for CS-305-MJ-P

Introduction

About the workbook

This workbook is intended to be used by T.Y.B.Sc (Computer Science) students for the Lab Course based on CS-302-MJ-T Operating Systems.

Operating System is core subject of computer science curriculum and hands-on laboratory experience is critical to the understanding of theoretical concepts studied as part of this course. Study of any programming language is incomplete without hands on experience of implementing solutions using programming paradigms and verifying them in the lab. This workbook provides rich set of problems covering the basic algorithms as well as numerous computing problems demonstrating the applicability and importance of various data structures and related algorithms.

The objectives of this book are

- Defining the scope of the course
- Bringing uniformity in the way the course is conducted across different colleges
- Continuous assessment of the course
- Bring variation and variety in experiments carried out by different students in a batch
- Providing ready reference for students while working in the lab
- Catering to the need of slow paced as well as fast paced learners

How to use this workbook?

The **Operating Systems** practical syllabus is divided into six assignments. Each assignment has problems divided into three sets A, B and C.

- Set A is used for implementing the basic algorithms or implementing data structure along with its basic operations. **Set A is mandatory.**
- Set B is used to demonstrate small variations on the implementations carried out in set A to improve its applicability. Depending on the time availability the students should be encouraged to complete set B.
- In Set C, Students should spend additional time either at home or in the Lab and solve these problems so that they get a deeper understanding of the subject.

Instructions to the students

Please read the following instructions carefully and follow them.

- Students are expected to carry workbook during every practical.
- Students should prepare oneself beforehand for the Assignment by reading the relevant material.

- Instructor will specify which problems to solve in the lab during the allotted slot & student should complete them and get verified by the instructor. However, student should spend additional hours in Lab and at home to cover as many problems as possible given in this work book.

- Students will be assessed for each exercise on a scale from 0 to 5
  - Not done 0
  - Incomplete 1
  - Late Complete 2
  - Needs improvement 3
  - Complete 4
  - Well Done 5

Instruction to the Practical In-Charge

- Explain the assignment and related concepts.
- Choose appropriate problems to be solved by students. Set A is mandatory. Choose problems from set B depending on time availability. Discuss set C with students and encourage them to solve the problems by spending additional time in lab or at home.
- Make sure that students follow the instruction as given above.
- You should evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
- The value should also be entered on assignment completion page of the respective Lab course.

Instructions to the Lab administrator and Exam guidelines

- You have to ensure appropriate hardware and software is made available to each student.
- Do not provide Internet facility in Computer Lab while examination
- Do not provide pen drive facility in Computer Lab while examination.

The operating system and software requirements are as given below:

- Operating system: Linux
- Editor: Any Linux based editor like vi, gedit etc.
- Compiler: cc or gcc

Assignment Completion Sheet

Practical Course based on CS – 302-MJ-T (Operating Systems)

Sr. No		Marks	Signature
1.	**Assignment No.1** Operations on processes :(Create a child process using fork()nd commands like exec(), execv() and execvp())		
2.	**Assignment No.2** Simulation of CPU Scheduling Algorithms – FCFS, SJF, Priority and Round Robin		
3.	**Assignment No.3** Simulation of demand paging using memory page replacement algorithms – FIFO, LRU, OPT, MFU		
4.	**Assignment No.4** Simulation of Banker's algorithm of deadlock avoidance in processes of operating system		
5.	**Assignment No.5** Simulation of File Allocation methods Contiguous allocation, Linked allocation, Indexed allocation		
6.	**Assignment No.6** Simulation of Disk Scheduling algorithms – FCFS, SSTF, SCAN, LOOK,C-LOOK		
Total out of 30			

This is to certify that **Mr/Ms.** _____

University Exam Seat Number _____have successfully and satisfactorily completed and submitted the course work for **T.Y.B.Sc.(CS)** Lab course CS-305-MJ-P Semester-V prescribed by Savitribai Phule Pune University during the academic year _____.

Instructor/Practical In charge **Head of Department**

Internal Examiner **External Examiner**

Assignment No. 1
Operations on Processes (Total Slots = 1)

Process: A process is basically a program in execution. We write our computer programs in a text file, during execution it is loaded in computer's memory and then it becomes a process. A process performs all the tasks mentioned in the program. A process can be divided into four sections ─ stack, heap, text and data.

Stack: The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap: This is dynamically allocated memory to a process during its run time.

Text: This includes all instructions specified in the program.

Data: This section contains the global and static variables.

In Linux operating system, new processes are created through fork() system call**.**

Fork() System Call:

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

fork() returns a zero to the newly created child process and returns a positive value (process ID of the child process) to the parent.

If fork() returns a negative value, the creation of a child process was unsuccessful.

The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

Let us take the following example:

```
int main()
{ printf("Before Forking");
  fork();
  printf("After Forking");
  return 0;
}
```

If the call to fork() is executed successfully, Linux will

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.

Output of above program:

Before Forking

After Forking

After Forking

Here printf() statement after fork() system call executed by parent as well as child process. Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

Consider one simpler example that distinguishes the parent from the child.

```
#include <stdio.h>
#include <sys/types.h>
void ChildProcess(); /* child process prototype */
void ParentProcess(); /* parent process prototype */
int main()
{  pid_t pid;
    pid = fork();
    if (pid == 0)
```

```
        ChildProcess();
    else
        ParentProcess();
    return 0;
}
void ChildProcess()
{ printf("I am child process..");
}
void ParentProcess()
{ printf("I am parent process..");
}
```

exec() system call

The exec() family of functions **replaces** the current process image with a new process image. It loads the program into the current process space and runs it from the entry point. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec.

The exec() family consists of following functions,

execl(): l is for the command line arguments passed a list to the function.

*int execl(const char *path, const char *arg, ...);*

execlp(): p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

*int execlp(const char *file, const char *arg, ...);*

execle(): It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.

*int execle(const char *path, const char *arg, ..., char * const envp[]);*

execv(): v is for the command line arguments. These are passed as an array of pointers to the function.

*int execv(const char *path, char *const argv[]);*

execlp() System Call:

execlp() system call is used after a fork() call by one of the two processes to replace the processes memory space with a new program. This call loads a binary file into memory and starts its execution. So two processes can be easily communicates with each other.

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
        int pid;
        pid = fork(); /* fork a child process */
        if (pid< 0) /* error occurred */
        { fprintf(stderr, "Fork Failed");
          return 1;
        }
        else if (pid == 0) /* child process */
                execlp("/bin/wc","wc",NULL);
            else /* parent process */
                { wait(NULL); /* parent will wait for the child to complete */
                  printf("Child Complete");
                }
        return 0;
}
```

fork() vs exec()

- Fork() starts a new process which is a copy of the one that calls it, while exec() replaces the current process image with another (different) one.
- Both parent and child processes are executed simultaneously in case of fork() while Control never returns to the original program unless there is an exec() error.

Wait() system call

The **wait**() system call suspends execution of the calling process until one of its children terminates.

wait(int &status);

waitpid() system call :

By using this system call it is possible for parent process to synchronize its execution with child process. Kernel will block the execution of a Process that calls waitpid system call if a some

child of that process is in execution. It returns immediately when child has terminated by return termination status of a child.

*int waitpid(int pid, int *status, int options);*

nice() System Call:

Using nice() system call, we can change the priority of the process in multi-tasking system. The new priority number is added to the already existing value.

int nice(int inc);

nice() adds *inc* to the nice value. A higher nice value means a lower priority. The range of the nice value is +19 (low priority) to -20 (high priority).

```
#include<stdio.h>
main()
{
        int pid, retnice;
        printf("press DEL to stop process \n");
        pid=fork();
        for(;;)
        {
        if(pid == 0)
        {
        retnice = nice (−5);
        print("child gets higher CPU priority %d \n", retnice);
        sleep(1);
        }
        else
        {
        retnice=nice(4);
        print("Parent gets lower CPU priority %d \n", retnice);
        sleep(1);
        }
        }
}
```

Orphan process

The child processes whose parent process has completed execution or terminated are called orphan process. Usually, a parent process waits for its child to terminate or finish their job and report to it after execution but if parent fails to do so its child results in the Orphan process. In most cases, the Orphan process is immediately adopted by the init process (a very first process of the system).

What is Shell?

Shell is an interface between user and operating system. It is the **command interpreter**, which accept the command name (program name) from user and executes that command/program. Shell mostly accepts the commands given by user from keyboard.

Shell gets started automatically when Operating system is successfully started.

How Shell Execute the command?

1. Accept the command from user.
2. Tokenize the different parts of command.
3. First part given on the given command line is always a command name.
4. **Creates (Forks)** a child process for executing the program associated with given command.
5. Once child process is created successfully then it **loads (exec)** the binary (executable) image of given program in child process area.
6. Once the child process is loaded with given program it will start its execution while shell is **waiting (wait)** for it (child) to complete the execution. Shell will wait until child finish its execution.
7. Once child finish the execution then Shell wakeup, display the command prompt again and accept the command and continue.

Example

$ cat f1.dat f2.dat f3.dat

This command line has four tokens- cat, f1.dat, f2.dat and f3.dat. First token is the command name which is to be executed. In linux operating system, some commands are internally coded and implemented by shell such as mkdir, rmdir, cd, pwd, ls, cat, grep, who etc.

Objective of this practical assignment is to simulate the shell which interprets all internal or predefined Linux commands

Program Logic

1) Main function will execute and display the command prompt as $
2) Accept the command at $ prompt from user.
3) Separate or tokenize the different parts of command line.
4) Check that first part is one of the extended commands or not (count, typeline, list, search).
5) If the command is extended command then call corresponding functions which is implementing that command
6) Otherwise fork a new process and then load (exec) a program in that newly created process and execute it. Make the shell to wait until command finish its execution.
7) Display the prompt and continue until given command is “q” to Quit.

Practical Assignments:

Set A

(1) Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

(2) Implement the C program to accept n integers to be sorted. Main function creates child process using fork system call. Parent process sorts the integers using bubble sort and waits for child process using wait system call. Child process sorts the integers using insertion sort.

Set B

(1) Write a C program to illustrate the concept of orphan process. Parent process creates a child and terminates before child has finished its task. So child process becomes orphan process. (Use fork(), sleep(), getpid(), getppid()).

(2) Implement the C program that accepts an integer array. Main function forks 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.

Set C

(1) Write a C program that behaves like a shell which displays the command prompt as $ or #. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command 'count' as

$ count c filename: It will display the number of characters in given file.

$ count w filename : It will display the number of words in given file.

$ count l filename : It will display the number of lines in given file.

(2) Write a C program that behaves like a shell which displays the command prompt as $ or #. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command 'search' as

$ search f filename pattern : It will search the first occurrence of pattern in the given file.

$ search a filename pattern: It will search all the occurrence of pattern in the given file.

$ search c filename pattern: It will count the number of occurrence of pattern in the given file.

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Instructor **Date of Completion____/____/_____**

Assignment: 2
CPU Scheduling (Total Slots = 3)

CPU Scheduling is one of the important tasks performed by the operating system. In multiprogramming operating systems many processes are loaded into memory for execution and these processes are sharing the CPU and other resources of computer system.

We have to implement various Short Term Scheduling algorithms as a part of this practical assignment.

Various CPU Scheduling algorithms are:

1) First Come First Serve (FCFS)

2) Shortest Job First (SJF)

3) Priority Scheduling

4) Round Robin Scheduling (RR)

These scheduling algorithms are further classified into 2 types as **Preemptive** and **Non-Preemptive**. FCFS scheduling is always Non-Preemptive while Round Robin is always Preemptive, while Shortest Job First and Priority Scheduling can be preemptive or non-preemptive.

The performance of various scheduling algorithms is compared on the basis of following criteria called as **scheduling criteria's** as:

1) CPU Utilization 2) Throughput 3) Turnaround Time 4) Waiting Time

5) Response Time

Data Structures

To simulate the working of various CPU scheduling algorithms following data structures is required.

1) Ready Queue - It represents the queue of processes which ready to execute but waiting for CPU to become available. Scheduling algorithm will select appropriate process from the ready queue and dispatch it for execution. For FCFS and Round Robin this queue is strictly operated as **First In First out queue**. But for Shortest Job First and Priority Scheduling it will operate as **Priority Queue**.

2) Process Control Block- It will maintain various details about the each process as processID, CPU-Burst, Arrival time, waiting time, completion time, execution time, turnaround time, etc. A structure can be used to define all these fields for processes and we can use array of structures of size n. (n is number of processes)

1) First Come First Serve Scheduling (FCFS):

In this algorithm the order in which process enters the ready queue, in same order they will execute on the CPU. This algorithm is simple to implement but it may be worst for several times.

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly said that first come first serve the algorithm which came first will be served first.

Note: Variables used in the

Algorithm wt - Waiting time

tat- Turnaround time

tatavg- Average

Turnaround time wtavg-

Average waiting time

qt- time quantum

rembt – remaining

burst time t- Current

time

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the burst

```
time wt[0] = wtavg = 0;
 tat[0] = tatavg = bt[0];
 for(i=1;i<n;i++)
    {
 wt[i] = wt[i-1] +bt[i-1];
 tat[i] = tat[i-1]
 +bt[i]; wtavg =
```

```
        wtavg + wt[i];
        tatavg = tatavg +
        tat[i];
  }
```

Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time

Step 5: for each process in the Ready Q calculate

a) Waiting time (n) = waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n) = waiting time (n) +Burst time (n)

Step 6: Calculate

a) Average waiting time = Total waiting Time / Number of process

b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

2) Shortest Job First (SJF):

In this algorithm the jobs will execute at the CPU according their next CPU-Burst time. The job in a ready queue which has shortest next CPU burst will execute next at the CPU. This algorithm can be preemptive or non-preemptive.

DESCRIPTION: (non-preemptive SJF)

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the

CPU burst time

```
        wt[0] = wtavg =
        0; tat[0] = tatavg
        = bt[0];
        for(i=1;i<n;i++)
            {
        wt[i] = wt[i-1] +bt[i-1];
        tat[i] = tat[i-1] +bt[i];
```



```
}
```

wtavg =
wt[i]/n;
tatavg =
tat[i]/n;

Step 4: Start the Ready Q according the shortest Burst time by sorting according

to lowest to highest burst time.

Step 5: Set the waiting time of the first process as _0'and its turnaround time as its

burst time.

Step 6: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue,

Calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time /Number of process

Step 9: Stop the process

3) Priority Scheduling (PS):

In this algorithm the job will execute according to their priority order. The job which has highest priority will execute first and the job which has least priority will execute last at the CPU. Priority scheduling can be preemptive or non-preemptive.

DESCRIPTION

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst Time

wt[0] = wtavg =
0; tat[0] = tatavg
= bt[0];

```
        for(i=1;i<n;i++)
           {
        wt[i] = wt[i-1] +bt[i-1];
        tat[i] = tat[i-1] +bt[i];

}
        wtavg =
        wt[i]/n;
        tatavg =
        tat[i]/n;
```

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround time

Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q

Step 8: for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate

c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process (Print the results in an order.)

Step10: Stop

4) Round Robin Scheduling (RR):

This algorithm is mostly used in time-sharing operating systems like Unix/Linux. This algorithm gives the fair change of execution to each process in system in rotation. In this algorithm each process is allowed to execute for some fixed time quantum. If process has the CPU-burst more than time quantum then it will execute for the given time quantum. But if it has CPU-burst less than the time quantum then it will execute for its CPU-burst time and then immediately release the CPU so that it can be assigned to other process. The advantage of this algorithm is that the average waiting time is less. This algorithm use/ ready queue and is strictly operated as First In First Out queue. This algorithm is intrinsically preemptive scheduling algorithm.

DESCRIPTION:

To calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check

whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst

```
time wt[0] = wtavg = 0;
tat [0] = tatavg =
bt[0]; for
(i=1;i<n;i++)
   {
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];

}
wtavg =
wt[i]/n;
tatavg =
tat[i]/n;
```

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

```
if (rembt[i] > qt)
        {

          t += qt

          rembt[i] -= qt
        }

        else
        {

          t = t +

          rembt[i];

          wt[i] = t -

          bt[i];
```

```
            rembt[i] = qt;
        }
    }
}
```

Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

c) Average waiting time = Total waiting Time / Number of process

d) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

Set A:

(i) Write the program to simulate FCFS CPU-scheduling. The arrival time and first CPU- burst for different n number of processes should be input to the algorithm. Assume that the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

(ii) Write the program to simulate Non-preemptive Shortest Job First (SJF) -scheduling. The arrival time and first CPU-burst for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

Set B:

i. Write the program to simulate Preemptive Shortest Job First (SJF) -scheduling. The arrival time and first CPU-burst for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

ii. Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first CPU-burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

Set C:

i. Write the program to simulate Preemptive Priority scheduling. The arrival time and first CPU-burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

ii. Write the program to simulate Round Robin (RR) scheduling. The arrival time and

first CPU-burst for different n number of processes should be input to the algorithm. Also give the time quantum as input. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Teacher/Instructor **Date of Completion_____/____/_____**

Assignment No. 3

Memory Management Methods (Total Slots = 2)

Introduction:

Memory Management Methods has following advantages:

1) System can execute the process of larger size than the size of available physical memory.

2) Reduced disk IO operations.

3) More numbers of processes can be loaded into memory.

4) Performance of multi-programming and multi-tasking can be improved.

In this memory management scheme instead of loading entire process into memory, only required portion of each process is loaded in memory for execution. Different parts of physical memory of each process are

brought into memory as and when required hence it is called as

Demand paging.

Each process is logically divided into number of pages. Each page is of same size.

Physical memory (RAM) is also divided into number of equal size frames. Generally, size of frame and a page is same. When process is to be loaded into memory its pages are loaded into available free frames. Memory management scheme maintains the list of free frames.

It also maintains the page table for each process. Page table keep track of various frames allocated to each process in physical memory. That is page table is used to map the logical pages of a process to frames in physical memory.

Memory Management Unit (MMU) of computer system is responsible to convert the logical address in process to physical memory in frame.

In demand paging memory management scheme no page of any process in brought into memory until it not really required. Whenever page is required (demanded) then only it is brought into memory.

Page Fault:

When process requests some page during execution and that page is not available in physical memory then it is called as Page Fault.

Whenever page fault occurs Operating system check really it is a page fault or it is an invalid memory reference. If page fault has occurred, then system try to load the corresponding page in available free frame.

Page Replacement:

When a page fault occurs, the operating system attempts to load the required page into one of the available

free frames in physical memory. If a free frame is available, the page is simply loaded into that frame. If no free frame is available, the system must replace one of the existing pages in memory with the required page. This process is known as page replacement. The frame selected for replacement is called the victim

frame.The decision of selecting a victim frame is made by the page replacement algorithm used in the demand paging memory management scheme. There are various page replacement algorithms, such as FIFO, Optimal, LRU, MRU, and MFU.

1. First in First out Page Replacement (FIFO)
2. Optimal Page Replacement (OPT)
3. Least Recently Used Page Replacement (LRU)
4. Most Frequently Used Page Replacement (MFU)

Input to Page Replacement Algorithm:

1. Reference String: It is the list of numbers which represent the various page numbers demanded by the process.
2. Number of Frames: It represents the number of frames in physical memory.

Ex. Reference String: 3, 6, 5, 7, 3, 5, 2, 5,7 ,3,2,4,2, 8, 3, 6

It means first process request the page number 3 then page number 6 then 5 and so on.

Data Structure:

1. Array of memory frames which is used to maintain which page is loaded in which frame. With each frame we may associate the counter or a time value whenever page is loaded in that frame or replaced.

2. Array of reference String.

3. Page Fault Count.

Output:

The output for each page replacement algorithm should display how each next page is loaded in which frame. Finally, it should display the total number of page faults.

FIFO:

- In this page replacement algorithm, the order in which pages are loaded in memory, in same order they are replaced.
- We can associate a simple counter value with each frame when a page is loaded in memory.
- Whenever page is loaded in free frame a next counter value is also set to it.
- When page is to be replaced, select that page which has least counter value.
- It is most simple page replacement algorithm.

Pseudocode: FIFO Page Replacement

```
BEGIN
  Read n (number of frames)
  Read reference_string
  Initialize all frames to EMPTY
  Initialize page_faults = 0
  Initialize pointer = 0
```

```
FOR each page in reference_string DO
    IF page is not present in frames THEN
            Replace frame[pointer] with page
             pointer = (pointer + 1) mod n
            page_faults = page_faults + 1
      ENDIF
    Display current frame status
ENDFOR
   Display total page faults
END
```

Optimal

- This algorithm looks into the future page demand of a process.
- Whenever page is to be replaced it will replace that page from the physical which will not be required longer time.
- To implement this algorithm whenever page is to be replaced, we compare the pages in physical memory with their future occurrence in reference string. The page in physical memory which will not be required for longest period of time will be selected as victim.

Pseudocode: Optimal Page Replacement (OPR)

```
BEGIN
 Read n (number of frames)
 Read reference_string
 Initialize all frames to EMPTY
 Initialize page_faults = 0

 FOR each page in reference_string DO
    IF page is not present in frames THEN
         FOR each page in frames DO
              Find next occurrence in reference_string
        ENDFOR
        Replace page whose next use is farthest in future
        page_faults = page_faults + 1
    ENDIF
    Display current frame status
 ENDFOR
 Display total page_faults
END
```

LRU:

- This algorithm replaces that page from physical memory which is used least recently.

- To implement this algorithm, we associate a next counter value or timer value with each frame/page in physical memory wherever it is loaded or referenced from physical
  memory. When page replacement is to be performed, it will replace that frame in physical memory which has smallest counter value.

Pseudocode: LRU Page Replacement

```
BEGIN
  Read n (number of frames)
  Read reference_string
  Initialize all frames to EMPTY
  Initialize usage_time for each frame
  Initialize page_faults = 0
  Initialize time = 0

  FOR each page in reference_string DO
       time = time + 1
      IF page is present in frames THEN
              Update usage_time of page to time
     ELSE
              Find frame with minimum usage_time
              Replace that frame with page
              Update usage_time
              page_faults = page_faults + 1
    ENDIF
     Display current frame status
  ENDFOR
  Display total page_faults
END
```

MFU:

- This algorithm replaces that page from physical memory which is used most frequently.
- This algorithm is bit complex to implement.
- To implement this algorithm, we have to maintain the history of each page that how many times it is used (frequency count) so far whenever it is loaded in physical memory or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory of which frequency count is greatest. If frequency count is same for two or more pages then it will apply FCFS.

Pseudocode: MFU Page Replacement

```
BEGIN
Read n (number of frames)
```

```
 Read reference_string
Initialize all frames to EMPTY
Initialize frequency_count for each frame to 0
Initialize page_faults = 0
FOR each page in reference_string DO
IF page is present in frames THEN
    Increase frequency_count of that page
ELSE IF there is an empty frame THEN
        Place page in empty frame
        Set frequency_count = 1
    ELSE
        Find frame with maximum frequency_count
    IF more than one frame has same maximum frequency THEN
        Select the oldest page among them (FCFS)
    ENDIF
    Replace selected frame with new page
  Set frequency_count = 1
 ENDIF
  page_faults = page_faults + 1
ENDIF
Display current frame status
ENDFOR
Display total page_faults
END
```

Set A

i. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string

Give input n as the number of memory frames.

Reference String: 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

1) Implement FIFO

2) Implement LRU

Set B

i. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string
Give input n as the number of memory frames.
Reference String: 2,5,2,8,5,4,1,2,3,2,6,1,2,5,9,8
1) Implement MFU
2) Implement Optimal

Set C

i. Write a simulation program to implement the following page replacement algorithms:
FIFO LRU MFU OPT. simulates all four algorithms for the same input, calculate the total number of page faults generated by each algorithm, and display a comparative analysis indicating the algorithm that performs best (i.e., produces the minimum number of page faults).
The program should accept n (number of memory frames) as input from the user
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Instructor/in charge Date of Completion ____/____/_____

Assignment No.4

Banker's Algorithm (Total Slots = 2)

Introduction:

This algorithm is related to handling deadlocks. There are three methods to handle deadlocks.

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection

Banker's algorithm is a technique used for **Deadlock Avoidance**. A deadlock avoidance algorithm dynamically examines the resource allocation state of system to the processes to ensure that a **circular-wait condition never exists.** The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes. A system is **safe** if the system can allocate resources to each process in some order and still avoid a deadlock. i.e. a system is in a safe state only if there exists a **safe sequence**. A sequence of processes $<P_1, P_2, \ldots, P_n>$ is a safe sequence for the current allocation state if, for each P_i , the resources that P_i can still request can be satisfied by currently available resources plus the resources held by all the P_j, with $j < i$.
Resource allocation graph is a technique used for deadlock avoidance. But the resource allocation graph algorithm is not applicable to the system with multiple instances of each resource type. Banker's algorithm is applicable to such a system.

Banker's algorithm: When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will keep the system in a safe state. If it will, then resources are allocated, otherwise the process must wait until some other process releases enough resources. This includes 2 algorithms

1) **Resource-request algorithm**
2) **Safety Algorithm.**

Safety Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work = Available
   Finish [*i*] = *0 for i* = 1, 2, 3, …, *n.*

2. Find and *i* such that both:
   (a) *Finish* [*i*] = 0
(b) *Need*$_i$ <=*Work*
   If no such *i* exists, go to step 4.
3. *Work=Work+Allo*
   cation$_i$
   Finish[*i*]=1
   go to step 2.
4. If *Finish* [*i*] == 1 for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process *Pi :*

Request = request vector for process P_i. If *Request*$_i$ [*j*] = *k* then process P_i wants *k* instances of resource type *Rj*.

1. If *Request*$_i$ <= *Need*$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If *Request*$_i$ <= *Available*, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   Available = Available -
   Request$_i$*; Allocation*$_i$ =
   Allocation$_i$ + *Request*$_i$; *Need*$_i$
   = Need$_i$ *– Request*$_i$*;*
   - *If safe -> the resources are allocated to Pi.*
   - *If unsafe -> Pi must wait, and the old resource-allocation state is restored*

The menu for the program should look like

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes, Allocation matrix, Maximum matrix
3. Get the avail value.
4. Find the need value.
5. Check whether it's possible to allocate.
6. If it is possible then the system is in safe state.

   Else system is not in safe state. Deadlock Occur
7. If the new request comes then check that the system is in safety.

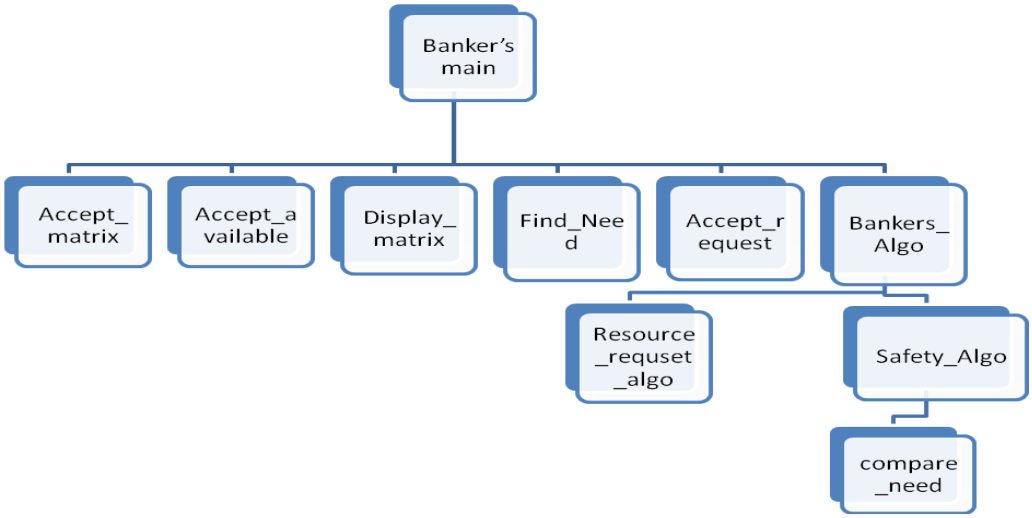
   or not allow the request.
8. stop the program.

Data Structure Design:

Let n be the number of processes in the system and m be the number of resource types.

Component	**Description**	**'c' code**
Available	A vector of length m indicates the number of available resources of each type	int Avail[10]
Max	An n x m matrix which defines the maximum demand of each process	int Max[10][10]
Allocation	An n x m matrix that defines the number of resources of each type currently allocated to each process	int Alloc[10[10]
Need	An n x m matrix indicates the remaining resource need of each process.	int Need[10][10]
Request	A vector of length m indicates the request made by a process Pi	int Request[10]
Finish	Vector of length n to check whether the process finished or not	int Finish[10] ={0}
Work	Vector of length m which is initialized to Available	int Work[10]
Safe	Vector of length n initialized to 0	Int Safe[10]= {0};

Control structure – The design is modular. The main module performs necessary Variable declarations, displays menu and accepts option from user. The structure diagram



is as follows.

Procedural Design – The following is a table of the guidelines for accepting inputs, calculated need and updating available, etc. as per Banker's Algorithm and implementation hints.

Procedure	Description	Programming hints
Bankers main	First accept number of processes and number of resources.	Printf("enter no. of processes & no. of resources "); scanf("%d%d", &n,&m);
Accept_matrix allocation,max	A generalized procedure which will accept all required matrices, like Allocation, Max depending upon which matrix is passed as parameter	printf("Enter the allocation matrix: "); for(i=0; i<n; i++) for(j=0; j<m; j++) scanf("%d",&allocation[i][j]); printf("\nEnter the max matrix: "); for(i=0; i<n; i++) for(j=0; j<m; j++) scanf("%d",&max[i][j]);
Accept_available	Accept single dimentional array/vector Available	printf("\nEnter the available matrix: "); for(i=0; i<m; i++) { scanf("%d",&available[i]); work[i]=available[i]; }
Display_matrix	Display Allocation, Max, Need	printf("\n*** Allocation Matrix ***\n"); for(i=0; i<n; i++) { for(j=0; j<m; j++) printf("\t%d",allocation[i][j]); printf("\n"); } printf("\n*** Max Matrix ***\n"); for(i=0; i<n; i++) { for(j=0; j<m; j++) printf("\t%d",max[i][j]); printf("\n"); } printf("\n*** Available Matrix ***\n"); for(i=0; i<m; i++) printf("\t%d",available[i]); printf("\n\n");

Find_Need	Calculate Need matrix	// calculating need matrix
		for(i=0; i<n; i++)
		for(j=0; j<m; j++)
		need[i][j]=max[i][j]-allocation[i][j];

Accept_Request	Accept the request for a process Pi	printf("\nIs there any request?[1/0] : ");
		scanf("%d",&f);
		if(f)
		{
		printf("\nEnter Process no : ");
		scanf("%d",&rpn);
		printf("\nEnter request : ");
		for(i=0; i<m; i++)
		scanf("%d",&request[i]);
		}

Bankers_algo	Invoke resource_request_algo and from it safety_algo	Bankers_algo()
		{
		resource_request_algo ();
		}

Safety-algo	Find if a safe sequence exists	// safety algorithm
		outer:
		for(i=0; i<n; i++)
		{
		if(finish[i]==0)
		{
		for(j=0; j<m; j++)
		{
		if(need[i][j] > work[j])
		flag=0;
		}
		if(flag==0)
		printf("\nProcess P%d must wait!\nResource not available...",i);
		else
		{
		printf("\nProcess P%d safe",i);
		finish[i]=1;
		sequence[++proc] = i;
		for(k=0; k<m; k++)
		
		work[k]=work[k]+allocation[i][k];
		}
		flag=1;
		}
		}
		
		for(i=0; i<n; i++)
		if(finish[i]==0)
		goto outer;
		
		printf("\n*** Safe Sequence ***\n");
		for(i=0; i<n; i++)
		
		printf("\tP%d",sequence[i]);
		printf("\n\n");

Resource_request_algo	Check if the given request can be immediately granted or not	// Checking Request for grantable or not
		for(i=0; i<m; i++)
		if(request[i]>need[rpn][i] | request[i]>available[i])
		{
		grant=0;
		break;
		}
		if(grant==1)

		{
		printf("\nRequest P%d can be granted immediately\n\n",rpn);
		for(i=0; i<m; i++)
		{
		available[i]=available[i]-request[i];
		allocation[rpn][i]=allocation[rpn][i]+request[i];
		need[rpn][i]=need[rpn][i]-request[i];
		work[i]=available[i];
		}
		for(i=0; i<n; i++)
		finish[i]=0;
		flag=1;
		proc=-1;
		
		// again call safety algorithm

SET-A

I) Add the following functionalities in your program

a) Accept process and Resources

b) Display Allocation, Max

c) Display the contents of need matrix

Process	Allocation			MAX		
	A	B	C	A	B	C
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

SET-B

I) Modify above program so as to include the following:

a) Accept Available and Display Available Matrix and Implement Safety Algorithm

SET-C

I) Consider a system with 'n' processes and 'm' resource types. Accept number of instances for every resource type. For each process accept the allocation and maximum requirement matrices. Write a program to display the contents of need matrix and to check if the given request of a process can be granted immediately or not. By using resource request Algorithm

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Teacher / Instructor **Date of Completion** _____/_____/_____

Assignment No. 5

File Allocation Methods (Total Slots = 3)

Introduction:

File allocation methods define how files are stored on secondary storage devices and how disk space is assigned to them in an operating system. An efficient file allocation method is essential for optimizing storage utilization, improving access speed, and ensuring reliable data retrieval. As files vary in size and access patterns, different allocation techniques are used to balance factors such as performance, flexibility, and space management. Understanding file allocation methods helps in analyzing how operating systems organize data on disks and handle file creation, access, and deletion effectively.

In file allocation methods, **sectors and tracks** are the basic physical units used to organize data on a disk. A **track** is a circular path on the surface of a magnetic disk, while each track is divided into smaller units called **sectors**, which store a fixed amount of data. During file allocation, files are assigned to one or more sectors located on specific tracks, and the operating system keeps track of these locations for efficient access. Proper arrangement of sectors and tracks plays a crucial role in reducing disk access time, as data stored on nearby tracks and contiguous sectors can be read faster. Understanding sectors and tracks helps explain how physical disk structure influences file allocation strategies and overall system performance.

Operating system maintains the details of each block. **Status of disk block** can be

1) Free
2) Allocated
3) Reserved
4) Bad

Objective of this practical assignment is to understand various file allocation methods and their implementation. When user accesses any file, the system has to locate all the disk blocks of that file on disk. Hence operating system has systematic way to maintain, allocate the blocks for each file and releasing the blocks of file when it is deleted.

File Allocation Method is a method that specifies how the blocks allocated to file are maintained so that file can be accessed properly. Most commonly used file allocation methods are

1) Contiguous File Allocation
2) Linked File Allocation
3) Indexed File Allocation

1) Contiguous File Allocation (Sequential File Allocation):

This is a file allocation method in an operating system where all the blocks of a file are stored in consecutive memory locations on the disk. When a file is created, the operating system allocates a continuous sequence of disk blocks sufficient to store the entire file. This method is simple to

implement and provides fast access to files because the disk head does not need to move frequently to read consecutive blocks. However, contiguous allocation has limitations such as **external fragmentation** and difficulty in handling file growth, as additional contiguous space may not be available. Despite these drawbacks, it is easy to understand and is often discussed to explain basic file storage concepts in operating systems for undergraduate studies.

The main drawbacks of **contiguous file allocation** in an operating system are:

- **External fragmentation**: Free disk space gets broken into small non-contiguous blocks, making it difficult to allocate large files.
- **Difficulty in file growth**: If a file needs to expand, additional contiguous space may not be available, requiring the file to be moved to a new location.
- **Wastage of disk space**: Extra space may be allocated to anticipate file growth, leading to unused disk blocks.
- **Inflexibility**: The size of the file must be known in advance, which is not always possible.
- **Inefficient space utilization over time**: Frequent file creation and deletion worsen fragmentation, reducing overall disk efficiency.

2) Linked File Allocation:

Linked file allocation is a file allocation method in an operating system in which the disk blocks of a file are stored at different locations and linked together using pointers. Each block contains a pointer to the next block of the file, forming a linked list structure. This method eliminates external fragmentation because files do not need to occupy contiguous disk space, and it allows files to grow easily by adding new blocks anywhere on the disk. However, linked allocation has drawbacks such as slower access time, especially for direct access, since the system must follow the pointers sequentially to reach a specific block. It also requires extra space to store pointers, making it slightly less efficient in terms of storage.

Directory maintains the file name along with starting block number and last block number of a file.

The main drawbacks of **Linked File Allocation** in an operating system are:

- **Slow access time**: Direct or random access is not supported efficiently because the system must traverse the file block by block to reach a specific location.
- **Pointer overhead**: Each disk block requires extra space to store a pointer to the next block, reducing effective storage capacity.
- **Reliability issues**: If a pointer is lost or corrupted, the remaining part of the file may become inaccessible.
- **Poor performance for large files**: Reading large files can be slow due to frequent disk head movements between non-contiguous blocks.
- **Complex disk access**: Sequential access requires following pointers, which can increase disk I/O operations and reduce overall performance.

3) Index File Allocation:

Indexed file allocation is a file allocation method in an operating system where all disk blocks of a file are linked through a special block called an **index block**. This index block contains a

list of pointers, each pointing to a disk block that stores a part of the file. The actual file data blocks can be located anywhere on the disk, which eliminates external fragmentation and allows files to grow easily. Indexed allocation supports both sequential and direct access efficiently because the system can directly refer to the required block through the index. However, it requires extra storage space for the index block and may be inefficient for very small files. This method is widely used in modern operating systems due to its flexibility and efficient file access.

Directory maintains the file name along with its index block number and number of entries in it.

The main drawbacks of **indexed file allocation** in an operating system are:

- **Extra storage overhead**: An index block is required for each file, which consumes additional disk space, especially for small files.
- **Limited file size**: The maximum file size is constrained by the number of pointers that can be stored in a single index block.
- **Additional disk access**: Accessing a file may require first reading the index block, increasing disk I/O operations.
- **Wastage for small files**: Small files may not fully utilize the index block, leading to inefficient space usage.
- **Complexity in management**: Maintaining and updating index blocks adds some overhead to file system management.

Data Structures:

1) Bit Vector:

In the file allocation method of an operating system, a bit vector/bitmap is a data structure used to keep track of free and allocated disk blocks efficiently. In a bitmap, each disk block is represented by a single bit, where a value of 0 typically indicates an allocated block and 1 indicates free block. This approach makes it easy for the operating system to quickly identify available space and manage disk allocation and deallocation. Bitmaps are compact, easy to implement, and allow fast searching for free blocks, especially when contiguous space is required. However, for very large disks, the bitmap itself may become large and require additional memory to store.

For Ex. It bit vector is:

00111010001101…….

It means that first 2 blocks are allocated, blocks 3,4,5 are free, 6^{th} is allocated, 7^{th} free and so on.

2) Directory:

Each directory entry maintains the filename along with certain details relevant to file allocation method. Directory list can be maintained statically or dynamically. For Contiguous allocation each directory entry contains filename, starting block number and length. Likewise for each file allocation method appropriate details are maintained in directory entry.

Implementation:

- A bit vector (array) of size n which is initialized randomly to 0 or 1.
- A menu with the options
  - Show Bit Vector
  - Create New File
  - Show Directory
  - Delete File
  - Exit
- When Create New File option is selected the program should ask the file name along with number of blocks to allocate to the file. According the free blocks should be searched by using the bit vector and allocated to the file. A new directory entry shall be added in directory list along with appropriate details.
- When Delete File option is selected the program should ask the file name. Release the blocks allocated to the file. Accordingly update the bit vector and remove the entry from directory.

SET-A

Write a program to simulate Sequential (Contiguous) file allocation method. Assume disk with n number of blocks. Give value of n as input. Write menu driven program with menu options as mentioned above and implement each option.

SET-B

Write a program to simulate Linked file allocation method. Assume disk with n number of blocks. Give value of n as input. Write menu driven program with menu options as mentioned above and implement each option.

SET-C

Write a program to simulate Indexed file allocation method. Assume disk with n number of blocks. Give value of n as input. Write menu driven program with menu options as mentioned above and implement each option.

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Teacher/Instructor **Date of Completion ____/____/_____**

Assignment No. 6

Disk Scheduling Algorithms (Total Slots = 1)

Introduction:

Disk scheduling is the process by which an operating system decides the order in which pending disk I/O requests are serviced. Since disk access time is much slower compared to CPU and main memory, efficient disk scheduling is essential to improve system performance. Disk drives need fast access time and large disk bandwidth. Disk access is slow mainly due to mechanical movements in traditional Hard Disk Drives (HDDs). When multiple processes request access to the disk simultaneously, the operating system maintains these requests in a queue. The disk scheduler selects the next request to minimize access time and improve throughput. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling.

The disk contains rotating platters and a movable read/write head. Access time depends on:

1. Seek Time – Time required moving the read/write head to the desired track.
2. Rotational Latency – Time waiting for the required sector to rotate under the head.
3. Transfer Time – Time to transfer data.

Among these, seek time is the largest component. Therefore, disk scheduling algorithms mainly aim to reduce seek time.

The simplest form of disk scheduling is, the First Come First Serve (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

Disk Scheduling Algorithm

A disk scheduling algorithm is a strategy implemented in the operating system that selects the next disk request to be processed from a queue of pending requests, with the objective of reducing head movement, minimizing waiting time, increasing throughput, and improving system efficiency.

First Come First Serve (FCFS) Algorithm

First Come First Serve (FCFS) is the simplest disk scheduling algorithm in which disk I/O requests are serviced in the exact order in which they arrive in the request queue.

Working of FCFS

1. When a disk request arrives, it is added to the end of the queue.
2. The disk head services requests one by one in arrival order.
3. No reordering or optimization is done.

4. The head moves from its current position to the requested track sequentially.

FCFS does not consider the distance between current head position and requested track.

Advantages of FCFS

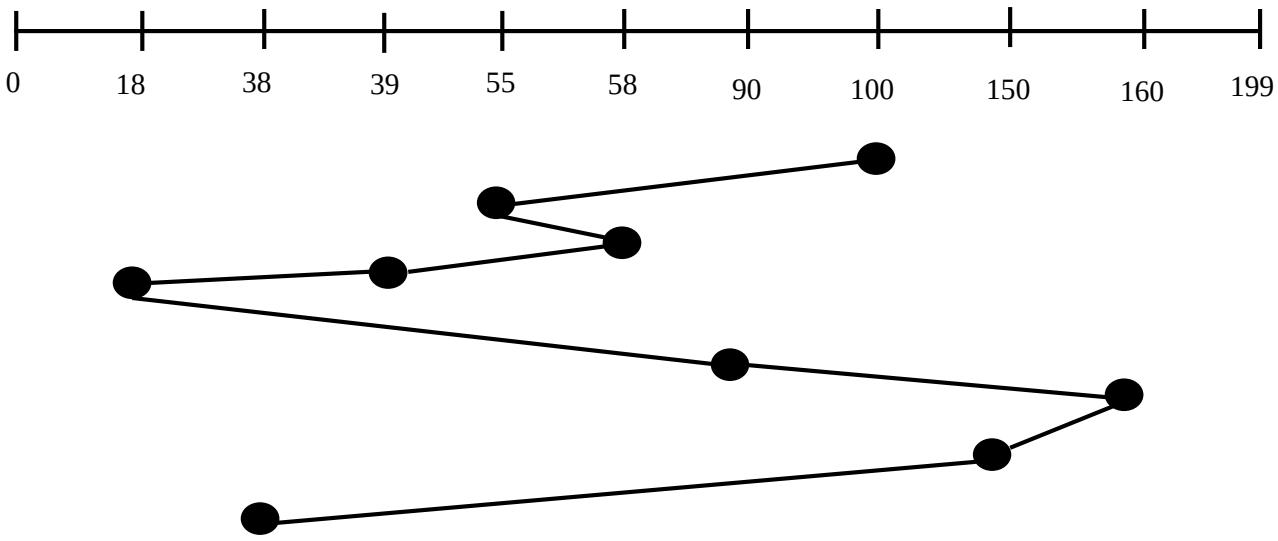
1. Simple to understand and implement.
2. Fair algorithm (no starvation).
3. Requests are handled in arrival order.
4. No complex calculations required.

Disadvantages of FCFS

1. High average seeks time.
2. Large head movement possible.
3. Poor throughput.
4. Not suitable for heavy load systems.
5. Can cause long waiting time (Convoy Effect).

Example

Consider a disk queue with requests for I/O to blocks on cylinders 55, 58, 39, 18, 90, 160, 150, 38. The FCFS scheduling algorithm is used. The head is initially at cylinder number 100. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is _____.



Order: 100 → 55 → 58 → 39 → 18 → 90 → 160 → 150 → 38

Total Head Movement = (100-55)+(58-55)+(58-39)+(39-18)+(90-18)+(160-90)+(160-150)+(150-38)

= 45 + 3 + 19 + 21 + 72 + 70 + 10 + 112

= 352 tracks

SSTF (Shortest Seek Time First) Disk Scheduling Algorithm

Shortest Seek Time First (SSTF) is a disk scheduling algorithm that selects the disk I/O request which is closest to the current position of the disk head, i.e. it services the request that requires the least head movement from the current position. The main goal is to reduce seek time by minimizing the movement of the disk head.

Working of SSTF

1. Start from the current head position.
2. Find the request that is nearest to the current head.
3. Service that requests.
4. Update the head position.
5. Repeat the process for the remaining requests.

Unlike FCFS, SSTF reorders the requests dynamically.

Advantages of SSTF

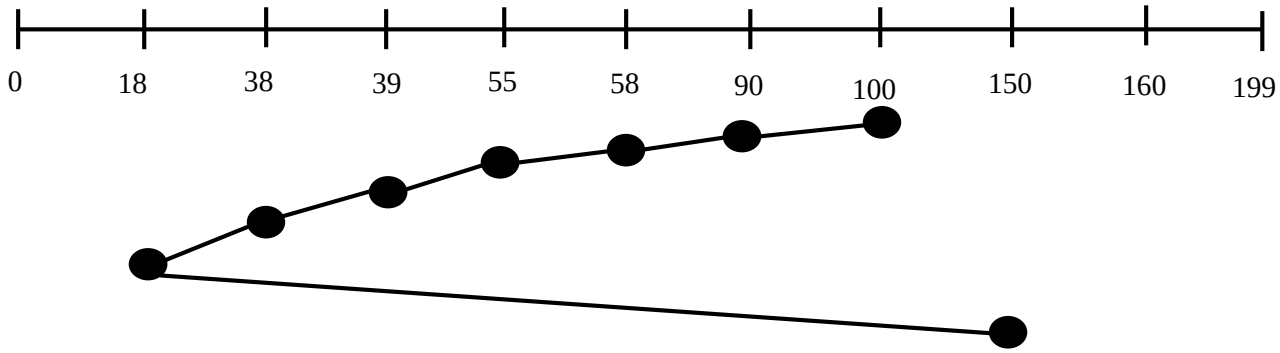
1. Reduces total seek time.
2. Better performance than FCFS.

Disadvantages of SSTF

1. Starvation may occur (far requests may wait indefinitely).
2. Not completely fair.
3. Slightly complex to implement.
4. Can cause overhead due to continuous distance calculation.

Example

Consider a disk queue with requests for I/O to blocks on cylinders 55, 58, 39, 18, 90, 160, 150, 38. The SSTF scheduling algorithm is used. The head is initially at cylinder number 100. The cylinders are numbered from 0 to 199. Total head movement (in number of cylinders) incurred while servicing these requests is ____.



Order: 100 → 90 → 58 → 55 → 39 → 38 → 18 → 150 → 160

Total Head Movement = (100-90)+(90-58)+(58-55)+(55-39)+(39-38)+(38-18)+(150-18)+(160-150)

= 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10

= 224

LOOK Disk Scheduling Algorithm

LOOK is an improved version of the SCAN (Elevator) disk scheduling algorithm. In SCAN, the disk head moves all the way to the end of the disk (0 or max cylinder), even if there are no requests there. In LOOK, the head Moves in one direction services all requests in that direction, goes only as far as the last request in that direction, then reverses direction. It does not go to the extreme end of the disk unless a request exists there. That's why it is called LOOK, because the head "looks" ahead and goes only as far as needed.

Working of LOOK

1. Sort all disk requests.
2. Choose the direction (left or right).
3. Service all requests in that direction.
4. Stop at the last request in that direction.
5. Reverse direction.
6. Continue servicing remaining requests.

Advantages of LOOK

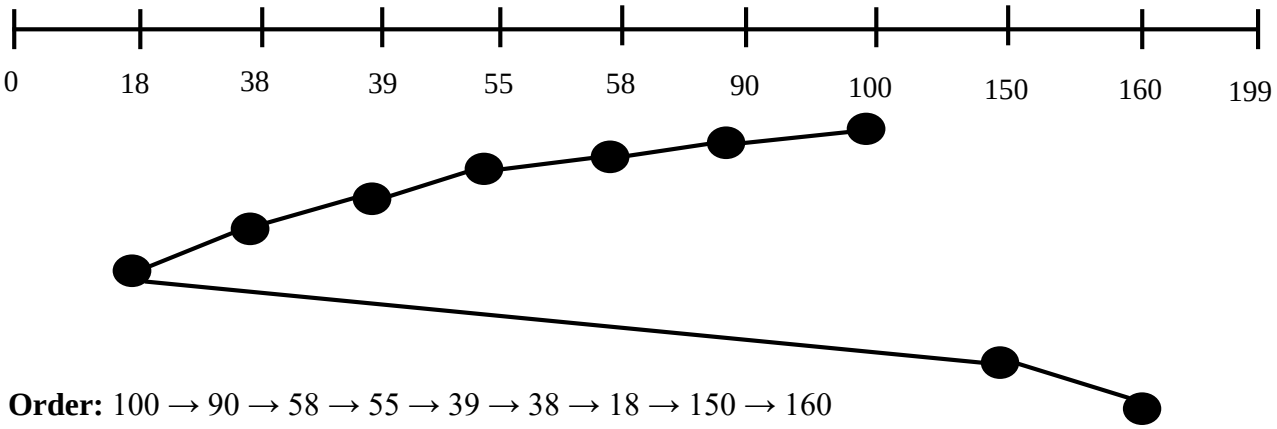
1. Reduces unnecessary movement
2. Lower seek time than SCAN
3. Better performance
4. Fair scheduling
5. No starvation

Disadvantages of LOOK

1. Slightly complex implementation
2. Not optimal like SSTF in some cases
3. Still not perfectly fair in heavy-load systems

Example

Consider a disk queue with requests for I/O to blocks on cylinders 55, 58, 39, 18, 90, 160, 150, 38. The LOOK scheduling algorithm is used. The head is initially at cylinder number 100. The cylinders are numbered from 0 to 199. Total head movement (in number of cylinders) incurred while servicing these requests is _____.



Order: 100 → 90 → 58 → 55 → 39 → 38 → 18 → 150 → 160

Total head movement = (100-90)+(90-58)+(58-55)+(55-39)+(39-38)+(38-18)+(150-18)+(160-150)

= 10+32+3+16+1+20+132+10

= 224

Comparison among all Disk Scheduling Algorithms:

Algorithm	Total Head Movement
FCFS	352
SSTF	224
LOOK	224

SET-A

i. Write an OS program to implement FCFS Disk Scheduling algorithm.
ii. Write an OS program to implement SSTF algorithm Disk Scheduling algorithm.

SET-B

i. Write an OS program to implement SCAN Disk Scheduling algorithm.
ii. Write an OS program to implement LOOK algorithm Disk Scheduling algorithm.

SET-C

i. Implement FCFS, SSTF, SCAN and LOOK Disk Scheduling algorithms in a single C program. Accept a request sequence, initial head position, and disk size as input. Generate a comparative report showing:

- Service order of requests
- Total seek operations
- Average seek time
- Best-performing algorithm

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Teacher/ Instructor **Date of Completion ____/____/_____**