



SAVITRIBAI PHULE PUNE UNIVERSITY
(Formerly University of Pune)

T. Y. B. SC. (COMPUTER SCIENCE)

CS-367 Laboratory Course on Operating Systems II Semester VI

Choice Based Credit System Syllabus to be implemented from
Academic Year 2021-2022

Name of Student			
Name of College			
Roll No.		Exam Seat No.	
Academic Year		Division	

Certificate

This is to certify that Mr./Ms -----

----- University Exam Seat Number

----- has successfully completed the assignment for the Lab Course I
(Operating System-II) during the Academic Year -----

His /Her performance was Very good / good / satisfactory /poor.

Head

Teacher In charge

Dept. of Computer Science

Internal Examiner

External Examiner

Co-Ordinators:**Dr. Prashant Muley**

Member Board of Study, Computer Science SPPU Pune

Dr. Manisha Bharambe , MES Abasaheb Garware College

Member Board of Study, Computer Science SPPU Pune

Prepared by:

Name of Teacher	Name of College
Dr. Manisha Bharambe	Vice-Principal, MES Abasaheb Garware College, Pune
Dr. Shelar M.N.	Associate Professor Head, Dept. of Computer Science, K.T.H.M. College Nasik.
Mrs. Chitra Alvani	Assistant Prof. Dept. of Computer Science, Kaveri College, Pune
Dr. Amale B.B.	Assistant Professor, Dept. of Computer Science Padmashri Vikhe Patil College of Arts, Science And Commerce Pravaranagar

Table of contents

Sr. No.	Contents	Page No.
1.	Introduction	5-7
2.	Assignment Completion Sheet	8-8
3.	Simulation of Banker's Algorithm of Deadlock Avoidance in processes of operating system	9-16
4.	Simulation of File Allocation Methods and free space management	17-19
5.	Simulation of Disk Scheduling Algorithms	20-26
6.	Assignment based on distributed and mobile OS	27-34

Introduction

About the workbook

This workbook is intended to be used by T. Y. B. Sc (Computer Science) students for the Practical Course based on CS-357 (Operating Systems-II) in Semester VI.

Operating System is an important 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.

About the Work Book Objectives –

1. Defining clearly the scope of the course
2. Bringing uniformity in the way course is conducted across different Colleges.
3. Continuous assessment of the students.
4. Bring variation and variety in experiments carried out by different students in a batch
5. Providing ready references for students while working in the lab.
6. Catering to the need of slow paced as well as fast paced learners

How to use this book?

This book is mandatory for the completion of the laboratory course. It is a Measure of the performance of the student in the laboratory for the entire duration of the course.

The Operating Systems-II practical syllabus is divided into four assignments. Each assignment has problems divided into three Slots

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 before hand for the Assignment by reading the relevant material.
- Teacher/Instructor will specify which problems to solve in the lab during the allotted slot and 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 in around ten minutes using white board if required or by demonstrating the software.
- Choose appropriate problems to be solved by students. Slot I is mandatory. Choose problems from Slot II depending on time availability. Discuss Slot III 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

Sr. No.		Assignment Name	Marks (out 5)	Signature
a	1.	Simulation of Banker's Algorithm of Deadlock Avoidance in processes of operating system		
	2.	Simulation of File Allocation Methods and free space management		
	3.	Simulation of Disk Scheduling Algorithms		
	4.	Assignment based on distributed and mobile OS		
Total out of 40				
b. Conduct Quiz at the time of Submission				
a. Total out of 10				
b. Total out of 5				
c. Total (Out of 15) (a + b)				

This is to certify that Mr. /Ms _____
 University Exam Seat Number _____ has successfully completed the course
 work for Lab Course I and has scored _____ Marks out of 15.

Head

Teacher/Instructor In charge

Dept. of Computer Science

Assignment No. 1:

Banker's Algorithm

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 $\langle P_1, P_2, \dots, P_n \rangle$ 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 $i = 1, 2, 3, \dots, n$.
2. Find and i such that both:
 - (a) $Finish[i] = 0$
 - (b) $Need_i \leq Work$If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$
 $Finish[i] = 1$
 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 P_i :

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

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, 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 \Rightarrow the resources are allocated to P_i .
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

The menu for the program should look like

1. Accept Allocation
2. Accept Max
3. Calculate Need
4. Accept Available
5. Display Matrices
6. Accept Request and Apply Banker's Algorithm
7. Exit

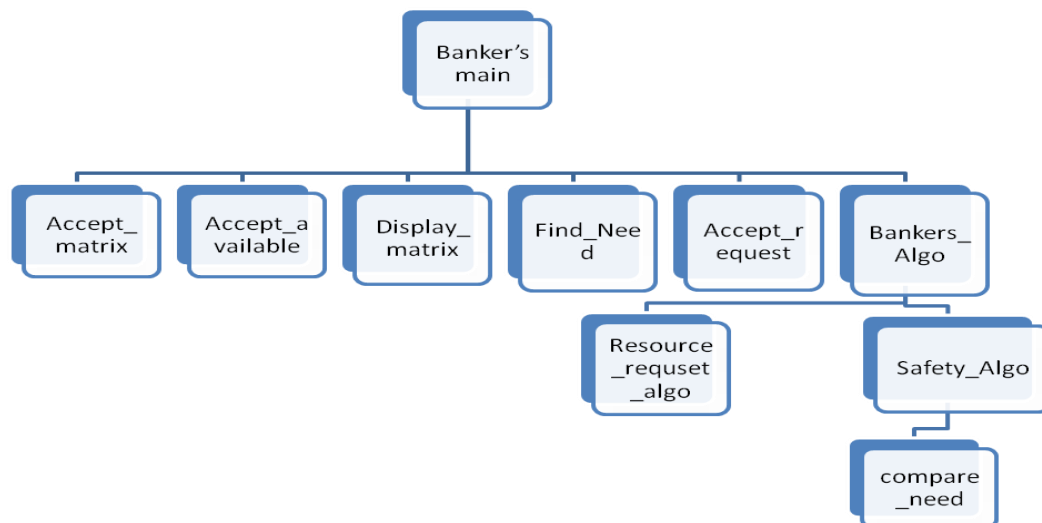
Choose option by specifying corresponding integer

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

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. In a loop Print options Read an option Branch depending on option	Printf(“enter no. of processes & no. of resources “); scanf(“%d%d”, &n,&m); While (option != 9) { printoptions(); scanf(“%d”,&option); switch(option) {case 1: } }
Accept_matrix (int A[][10])	A generalized procedure which will accept all required matrices, like Allocation, Max depending upon which matrix is passed as parameter	Void accept_matrix(int A[][10]) { int I,j; for (i=0; i<n;i++) for (j=0; j<m;j++) scanf(“%d” ,&A[i][j]) ; }
Accept_available	Accept single dimensional array/vector Available	Void Accept_available() { int I; for (i=0; i<m; i++) scanf(“%d” ,&Avail[i]) ; }

Display_matrix	Display Allocation, Max, Need	<pre> Void Display_matrix() {.... Printf (\n\tAllocation\t\tMax\t\tNeed\n"); for (i=0; i<n;i++) { for (j=0; j<m;j++) printf("%d",Allocl[i][j]) ; printf("\t); for (j=0; j<m;j++) printf("%d",Maxil[i][j]) ; printf("\t); for (j=0; j<m;j++) printf("%d",Need[i][j]) ; printf("\t); } Printf("\n Available\n"); for (j=0; j<m;j++) printf("%d",Avail[i][j]) ; printf("\t); } </pre>
Find_Need	Calculate Need matrix	<pre> Void Find_Need() { for (i=0; i<n;i++) for (j=0; j<m;j++) <i>Need [i,j] = Max[i,j] – Allocation [i,j];</i> } </pre>
Accept_Request	Accept the request for a process Pi	<pre> Void Accept_Request() { int i; Printf("\nenter process no:"); Scanf("%d",&proc); for (i=0; i<m; i++) scanf("%d" ,&Requestl[i]) ; } </pre>
Bankers_algo	Invoke resource_request_algo and from it safety_algo	<pre> Bankers_algo() { resource_request_algo (); } </pre>
Safety-algo	Find if a safe sequence exists	<pre> Safety_algo() { int over =0, i, j, k,l=0, flag; //initialize work = available for (i=0; i<m; i++) </pre>

		<pre> work[i] = Avail[i]; while (!over) { // check for any not finished process for (i=0; i<n; i++) { if (Finish[i] ==0) { flag =0; pno=compare_need(i); if(pno>-1) break; } } if (i ==n) { printf("System is unsafe"); exit(1); } if (i < n && pno>=0) { for (k=0; k<m; k++) work[k] += Alloc[pno][k]; Finish[pno] = 1; Safe[l++] = pno; //add process in safe sequence if (l >= n) // if all processes over { // print safe sequence printf("\n Safe sequence is :"); for(l=0;l<n; l++) printf("P%d\t",Safe[l]); over = 1; }} } } </pre>
Int compare_need(int t p)	Checks if need of process p is <= work or not. If condition true returns process no else returns -1	Int compare_need(int p) { int i,j, flag=0;

		<pre> for (j=0;j< nor; j++) { if (Need[p][j] > work[j]) { flag =1; break; } } if(flag==0) return p; return -1; } </pre>
Resource_request_algo	Check if the given request can be immediately granted or not	<pre> Resource_request_algo() { ... // check if Requesti <= Need; for (i=0; i<m; i++) { if Request[i] > Need[proc][i] { printf("error.. process exceeds its Max demand"); exit(1); } } // check if Reuest [i] <= Available for (i=0; i<m; i++) { if Request[i] > Avail[i] { printf("Process must wait , resources not available"); exit(1); } } // pretend to have allocated requested recources for (i=0; i<m; i++) { Avail[i] = Avail[i] - Request[i]; Alloc[proc][i] = Alloc[proc][i] + Request[i]; Need[proc][i] = Need[proc][i] - Request[i]; } //run Safety algorithm to check whether safe sequence exists or not Safety_algo(); } </pre>

Slot 1

- I) Add the following functionalities in your program
- Accept Available
 - Display Allocation, Max
 - Display the contents of need matrix
 - Display Available

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

Slot 2

- I) Modify above program so as to include the following:
- Accept Request for a process
 - Resource request algorithm
 - Safety algorithm

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.

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.2:

File Allocation Methods

Introduction:

Files are managed by the operating system on secondary storage devices. Hard disk drive is most commonly used storage device.

Physically hard disk is divided into number **sectors** also called as **blocks**. When a file is to be created the free blocks on the disk are allocated to the file.

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

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

Purpose 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

Contiguous File Allocation:

- This file allocation method is also called as Sequential file allocation.
- In this method blocks allocated the file are contiguous. That is if file is of n blocks then all the n blocks are adjacent to one another.
- Advantage of this method is that the file can access sequentially as well as randomly.
- Directory maintains the file name along with starting block number and length (that is number of blocks allocated to file).
- Drawback of this method is that there can be external fragmentation problem and file cannot grow efficiently.

Linked File Allocation:

- In this method n blocks file are maintained as chain (linked list) of n blocks. First block of file contains the address or link to second block of file. Second block of file contains the address or link to third of file as so on. Last block of file points to null. That any free block on the storage device can be allocated to the file.
- Advantage of this method is that the file can grow or shrink dynamically and there is no external fragmentation.
- Directory maintains the file name along with starting block number and last block number of a file.
- Drawback of this method is that the file can be access sequentially only. Random file accessing is not possible and wastage of memory for pointer in each block of a file.

Index File Allocation:

- This method overcomes the drawbacks of Sequential File Allocation method as well as Linked Allocation Method.
- In this method a special Index Block is maintained for each file. This block contains the list of block numbers allocated to the file. When file is to be accessed the index block will give the list of blocks allocated to that file.
- Hence in this method file can be accessed sequentially as well as randomly.
- Each available free block on disk can be allocated to any file on demand and that number is added to the index block of that file.
- Directory maintains the file name along with its index block number and number of entries in it.
- Drawback of this method is the overhead of maintaining index block.

Data Structures:

1) Bit Vector :

It is used to maintain the free space on disk. It is a vector of size n (where n is number of blocks on disk) with values 0 or 1. These values mark the status of block as whether it is free or allocated. For Ex. It bit vector is:

00111010001101.....

It means that first 2 blocks are allocated, block 3,4,5 are free, 6th is allocated, 7th free, 8th to 10th are not free, 11th and 12th are 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.

Slot-1

- i. 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 driver program with menu options as mentioned above and implement each option.

Slot- II

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

Slot- III

- i. Write a program to simulate Indexed file allocation method. Assume disk with n number of blocks. Give value of n as input. Write menu driver 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 Instructor

Date of Completion ____/____/____

Assignment No.3:

Disk Scheduling Algorithms

Introduction:

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. 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 simplest form of disk scheduling is, of course, the first - come, first - served (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

- The algorithms used for disk scheduling are called as disk scheduling algorithms.
- The purpose of disk scheduling algorithms is to reduce the total seek time.

First Come First Serve (FCFS)

FCFS is the simplest disk scheduling algorithm. As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

Algorithm:

Advantages-

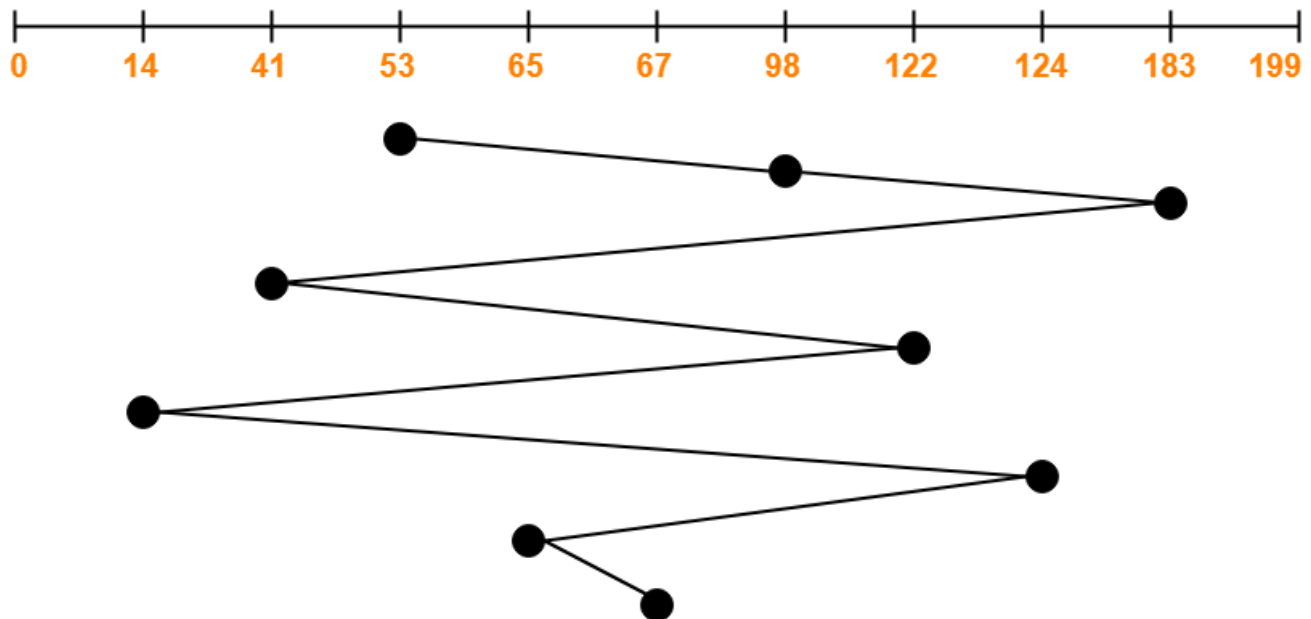
- It is simple, easy to understand and implement.
- It does not cause starvation to any request.

Disadvantages-

- It results in increased total seek time.
- It is inefficient.

Example

a) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The FCFS scheduling algorithm is used. The head is initially at cylinder number 53. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is _____.



$$\begin{aligned}
 &\text{Total head movements incurred while servicing these requests} \\
 &= (98 - 53) + (183 - 98) + (183 - 41) + (122 - 41) + (122 - 14) + (124 - 14) + (124 - 65) + (67 - 65) \\
 &= 45 + 85 + 142 + 81 + 108 + 110 + 59 + 2 \\
 &= 632
 \end{aligned}$$

SSTF Disk Scheduling Algorithm-

SSTF stands for **Shortest Seek Time First**.

- This algorithm services that request next which requires least number of head movements from its current position regardless of the direction.
- It breaks the tie in the direction of head movement

Advantages-

It reduces the total seek time as compared to FCFS.

- It provides increased throughput.
- It provides less average response time and waiting time.

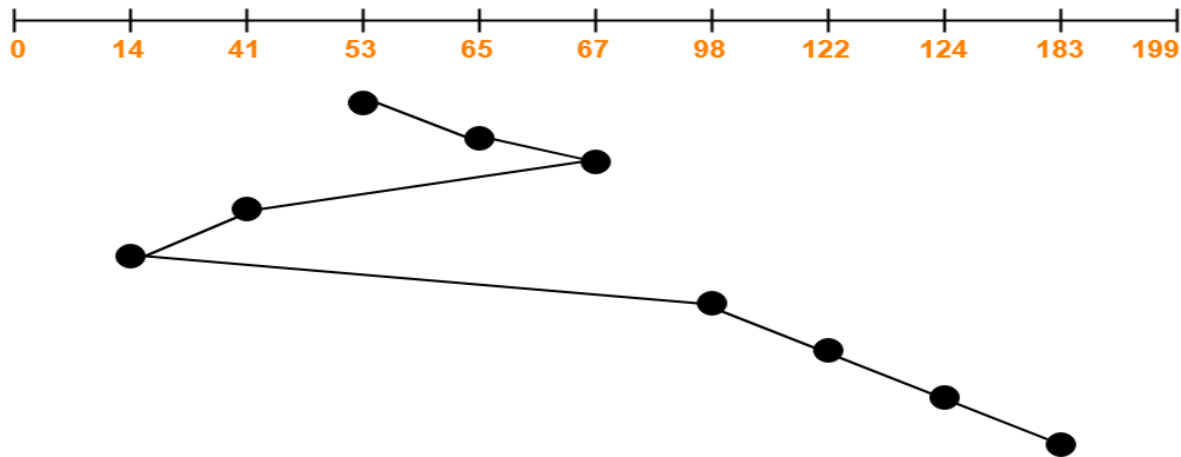
Disadvantages-

There is an overhead of finding out the closest request.

- The requests which are far from the head might starve for the CPU.
- It provides high variance in response time and waiting time.
- Switching the direction of head frequently slows down the algorithm.

Example

b) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SSTF scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is _____.



Total head movements incurred while servicing these requests

$$\begin{aligned} &= (65 - 53) + (67 - 65) + (67 - 41) + (41 - 14) + (98 - 14) + (122 - 98) + (124 - 122) + (183 - 124) \\ &= 12 + 2 + 26 + 27 + 84 + 24 + 2 + 59 \\ &= 236 \end{aligned}$$

SCAN DISK SCHEDULING ALGORITHM

Given an array of disk track numbers and initial head position, our aim is to find the total number of seek operations done to access all the requested tracks if SCAN disk scheduling algorithm is used.

SCAN (Elevator) algorithm

In SCAN disk scheduling algorithm, head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reach the other end. Then the direction

of the head is reversed and the process continues as head continuously scan back and forth to access the disk. So, this algorithm works as an elevator and hence also known as the elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages of SCAN (Elevator) algorithm

- This algorithm is simple and easy to understand.
- SCAN algorithm has no starvation. 3. This algorithm is better than FCFS Scheduling algorithm.

Disadvantages of SCAN (Elevator) algorithm

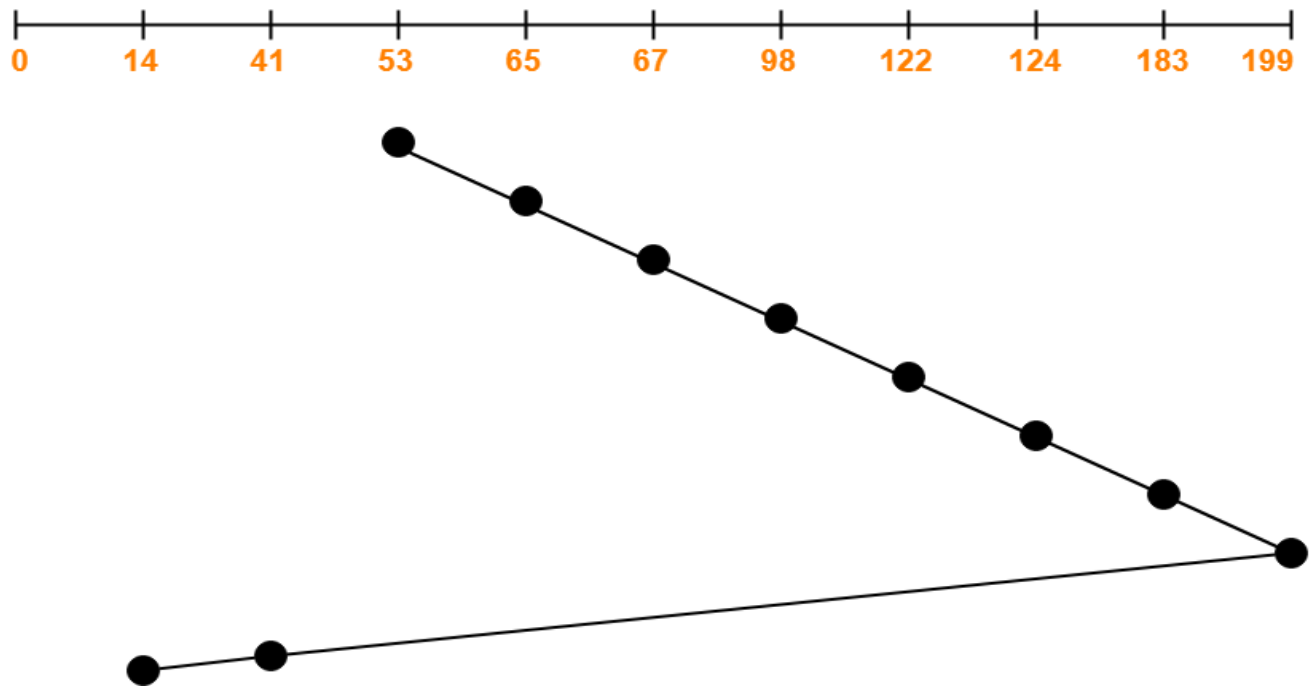
- More complex algorithm to implement.
- This algorithm is not fair because it cause long waiting time for the cylinders just visited by the head.
- It causes the head to move till the end of the disk in this way the requests arriving ahead of the arm position would get immediate service but some other requests that arrive behind the arm position will have to wait for the request to complete.

ALGORITHM

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seeks count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

Example

Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SCAN scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is _____.



Total head movements incurred while servicing these requests

$$= (65 - 53) + (67 - 65) + (98 - 67) + (122 - 98) + (124 - 122) + (183 - 124) + (199 - 183) + (199 - 41) + (41 - 14)$$

$$= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 158 + 27$$

$$= 331$$

C-SCAN DISK SCHEDULING ALGORITHM

Description:

- Circular-SCAN Algorithm is an improved version of the SCAN Algorithm.
- Head starts from one end of the disk and move towards the other end servicing all the requests in between.
- After reaching the other end, head reverses its direction.
- It then returns to the starting end without servicing any request in between.
- The same process repeats.

Advantages-

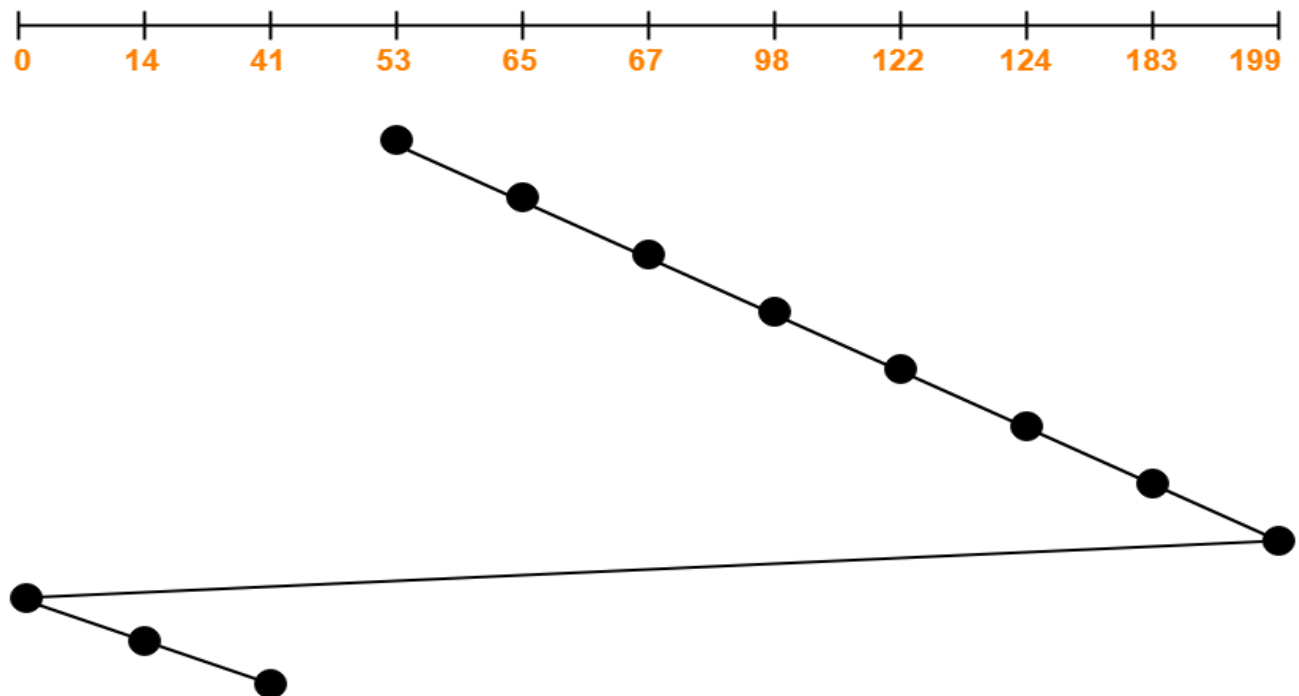
- The waiting time for the cylinders just visited by the head is reduced as compared to the SCAN Algorithm.
- It provides uniform waiting time.
- It provides better response time.

Disadvantages-

- It causes more seek movements as compared to SCAN Algorithm.
- It causes the head to move till the end of the disk even if there are no requests to be serviced.

Example.

Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The C-SCAN scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is _____.



Total head movements incurred while servicing these requests

$$= (65 - 53) + (67 - 65) + (98 - 67) + (122 - 98) + (124 - 122) + (183 - 124) + (199 - 183) + (199 - 0) + (14 - 0) + (41 - 14)$$

$$= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 199 + 14 + 27$$

$$= 386$$

Slot-I

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

Slot-II

- i. Write an OS program to implement SCAN Disk Scheduling algorithm.
- ii. Write an OS program to implement C-SCAN algorithm Disk Scheduling 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. 4:

Distributed and mobile OS

Introduction:

Distributed Computing

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

Let us say about Google Web Server, from users perspective while they submit the searched query, they assume google web server as a single system. However, behind the curtain, google has built a lot of servers which is distributed (geographically and computationally) to give us the result within few seconds.

Advantages of Distributed Computing?

- Highly efficient
- Scalability
- Less tolerant to failures
- High Availability

What is MPI?

Message Passing Interface (MPI) is a standardized and portable **message-passing** system developed for distributed and parallel computing. MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented. As a result, hardware vendors can build upon this collection of standard low-level routines to create higher-level routines for the distributed-memory communication environment supplied with their parallel machines.

MPI gives user the flexibility of calling set of routines from **C, C++, FORTRAN, and C #, Java or Python**. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs)

Link for MPI Cluster Setup:

<https://medium.com/mpi-cluster-setup/mpi-clusters-within-a-lan-77168e0191b1>

MPI Installation

The machines which requires to be used to run MPI program, need to be connected using NIS and NFS. MPI can be installed by downloading the tar file from <https://www.mpich.org/> or using command

```
yum install mpich*
```

Hello world code examples

```
#include <mpi.h>

#include <stdio.h>

int main(int argc, char** argv) {

    // Initialize the MPI environment

    MPI_Init(NULL, NULL);

    // Get the number of processes

    int world_size;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process

    int world_rank;

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor

    char processor_name[MPI_MAX_PROCESSOR_NAME];

    int name_len;

    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message

    printf("Hello world from processor %s, rank %d out of %d processors\n",

           processor_name, world_rank, world_size);

    // Finalize the MPI environment.

    MPI_Finalize();

}
```

You will notice that the first step to building an MPI program is including the MPI header files with `#include <mpi.h>`.

After this, the MPI environment must be initialized with:

```
MPI_Init(int* argc, char*** argv)
```

During `MPI_Init`, all of MPI's global and internal variables are constructed.

For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process.

After `MPI_Init`, there are two main functions that are called. These two functions are used in almost every single MPI program that you will write.

```
MPI_Comm_size(MPI_Comm communicator, int* size)
```

`MPI_Comm_size` returns the size of a communicator. In our example, `MPI_COMM_WORLD` (which is constructed for us by MPI) encloses all of the processes in the job, so this call should return the amount of processes that were requested for the job.

```
MPI_Comm_rank(MPI_Comm communicator, int* rank)
```

`MPI_Comm_rank` returns the rank of a process in a communicator. Each process inside of a communicator is assigned an incremental rank starting from zero. The ranks of the processes are primarily used for identification purposes when sending and receiving messages.

A miscellaneous and less-used function in this program is:

```
MPI_Get_processor_name(char* name, int* name_length)
```

`MPI_Get_processor_name` obtains the actual name of the processor on which the process is executing. The final call in this program is:

```
MPI_Finalize()
```

`MPI_Finalize` is used to clean up the MPI environment. No more MPI calls can be made after this one.

Compilation of MPI program

execute the following command to compile the program having MPI function calls

```
$ mpicc <name of .c file>
```

Once compilation is successful. Execute the program as follows

\$mpitun -np <no. of processors><name of executable file>

\$mpirun -np 2 ./a.out ## np attribute initializes the MPI environment to let the program know how many processors are available to execute the program

Basic MPI data type

MPI data type	C data type
MPI_INT	signed integer
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char
MPI_PACKED	user-defined

The syntax of MPI_Send is

```
int MPI_Send(void* msg_buf_p                /* input */,
             int msg_size                   /* input */,
             MPI_Datatype msg_type         /* input */,
             int dest                       /* input */,
             int tag                        /* input */,
             MPI_Comm communicator /* input */);
```

The first three arguments determine the contents of the message; the last three determine the destination.

MPI_Recv has a similar (complementary) syntax to MPI_Send , but with a few important differences.

```
int MPI_Recv( void* msg_buf_p                /* out */,
             int buf_size                    /* in */,
             MPI_Datatype buf_type           /* in */,
             int src                         /* in */,
             int tag                         /* in */,
             MPI_Comm communicator /* in */);
```

```

int source                /* in */,

int tag                   /* in */,

MPI_Comm communicator    /* in */

MPI_Status status_p      /* out */);

```

Again, the first three arguments specify the memory available (buffer) for receiving the message.

The next three identify the message.

Both tag and communicator must match those sent by the sending process.

MPI program to send and receive data on processors

```

#include <stdio.h>
#include <string.h> /* For strlen */
#include <mpi.h> /* For MPI functions, etc */
const int MAX_STRING = 100;
int main(void) {
    char    greeting[MAX_STRING]; /* String storing message */
    int     comm_sz;              /* Number of processes */
    int     my_rank;              /* My process rank */
    int i;
    /* Start up MPI */
    MPI_Init(NULL, NULL);
    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    /* Get my rank among all the processes */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank != 0) /* Code to be executed on slave machines */
    {
        /* Create message */
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        /* Send message to process 0 */
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0,
0,MPI_COMM_WORLD);
    }
}

```

```

else
{
    /* Print my message */
    printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
    for (i = 1; i < comm_sz; i++) {
        /* Receive message from process q */
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, i, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /* Print message from process i */
        printf("%s \n", greeting);
    }
}

/* Shut down MPI */
MPI_Finalize();
return 0;
}/* main end*/

```

MPI Collective Communication Routines

MPI provides the following collective communication routines:

- MPI_Barrier synchronization across all processes.
- MPI_Broadcast from one process to all other processes.
- MPI_Reduce: Global reduction operations such as sum, min, max, or user-defined reductions.
- MPI_Gather : Gather data from all processes to one process.
- MPI_Scatter : Scatter data from one process to all processes.
- Advanced operations in which all processes receive the same result from a gather, scatter, or reduction.
- There is also a vector variant of most collective operations where messages can have different sizes.

Execute MPI program on different hosts

You specify the available hosts to MPI in three ways:

- Through the batch scheduler in your resource management software.

- By using a hostfile with the `--hostfile` option. The hostfile is a text file that contains the names of hosts, the number of available slots on each host, and the maximum slots on each host.
- By using the `--host` option. Use this option to specify which hosts to include or exclude.

Specifying Hosts By Using a Hostfile

The hostfile lists each node, the available number of slots, and the maximum number of slots on that node. For example, the following listing shows a simple hostfile:

```
node0
node1 slots=2
node2 slots=4 max_slots=4
node3 slots=4 max_slots=20
```

In this example file, node0 is a single-processor machine. node1 has two slots. node2 and node3 both have 4 slots, but the values of slots and max_slots are the same (4) on node2. This disallows the processors on node2 from being oversubscribed. The four slots on node3 can be oversubscribed, up to a maximum of 20 processes.

When you use this hostfile with the `--nooversubscribe` option mpirun assumes that the value of max_slots for each node in the hostfile is the same as the value of slots for each node. It overrides the values for max_slots set in the hostfile.

MPI assumes that the maximum number of slots you can specify is equal to infinity, unless explicitly specified. Resource managers also do not specify the maximum number of available slots.

Specifying Hosts By Using the `--host` Option

You can use the `--host` option to mpirun to specify the hosts you want to use on the command line in a comma-delimited list. For example, the following command directs mpirun to run a program called a.out on hosts a, b, and c:

```
% mpirun -np 3 --host a,b,c a.out
```

MPI assumes that the default number of slots on each host is one, unless you explicitly specify otherwise.

To Specify Multiple Slots Using the `--host` Option

To specify multiple slots with the `--host` option for each host repeat the host name on the

command line for each slot you want to use. For example:

```
% mpirun -host node1,node1,node2,node2 ...
```

Slot-I

- i. Write an MPI program to calculates sum of randomly generated 1000 numbers (stored in array) on a cluster

Slot-II

- ii. Write an MPI program to find the min and max number from randomly generated 1000 numbers (stored in array) on a cluster (Hint: Use MPI_Reduce)

Slot-III

- iii. Write a review paper on comparative study of different mobile operating system.

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 ____/____/____