OPERATING SYSTEMS LABORATORY

Course Code: CSE309

Semester: V

Lab Manual

2024

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur-613 401
School of Computing

Course Objective:

This course will help the learner to gain pragmatic knowledge on the different modules of operating system by simulating them and analysing their performance differences.

Course Learning Outcomes:

Upon successful completion of this course, the learner will be able to

- Develop a program to create parent/child or concurrent process and establish communication between them using pipe or IPC methods
- Analyze CPU and thread scheduling algorithms by simulating them and providing the CPU schedule with sample test data and calculate performance related metrics
- Implement mutual exclusion based on semaphores to prevent concurrent issues under classic problems of concurrency
- Demonstrate deadlock avoidance and detection strategies
- Simulate memory management schemes and disk scheduling schemes

List of Experiments:

1. a. Program to create of a child process using fork, trace the process ids and establish communication between parent and child
   b. Program to create multiple children and establish communication between siblings and parent using pipes
2. a. Program to implement IPC using Shared Memory System Call.
   b. Program to implement IPC with the help of Message Queues.
3. a. Programs for the simulation of uni-processor scheduling algorithms and analyze their performances.
   b. Simulate CPU Scheduling with CPU and I/O Burst time and analyse the performance.
4. Program to experiment multi-processor scheduling.
5. Program for the simulation of thread scheduling approaches.
6. Program to implement peterson's algorithm for enforcing mutual exclusion.
7. a. Program to demonstrate for producer-consumer problem using semaphore.
   b. Program to apply semaphore for tackling reader-writer problem.
8. Program to apply banker's algorithm for deadlock avoidance.
9. Program to implement deadlock detection algorithm.
10. Program to implement dining philosopher's problem without causing deadlocks.

11. a. Program to simulate page replacement algorithms and to compute number of page faults
    b. Program to simulate address translation under paging.
12. Program to implement disk scheduling algorithms.

Additional Experiments:

13. Program to simulate dynamic partitioning and buddy system.
14. Program to demonstrate file allocation techniques.

Observation

Exercise No. 1.a&b: **Creation of a child process and communication between parent and children**

Objectives

To create a child process using fork system call and use pipe for interaction between parent and child

Pre-requisite:

Knowledge of parent-child process, fork and pipe commands.

Theory or Concept

Fork () - creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

Syntax: #include <unistd.h>
pid_t fork(void);

Return Values:

- On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set to indicate the error
-

Pipe() - creates a pipe, a unidirectional data channel that can be used for interprocess communication.The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

Procedure / Algorithm

- Develop the parent process with code for calls to fork and pipe
- Child process created as a result of fork()
- Write a message into pipe under parent part of the code
- Suspend parent process to invoke child
- Reads the message from the pipe under the child part of the code
- Parent and child terminates

Sample Input:

Output:

Observation

Exercise No. 2a IPC Using Shared Memory System Call

Objectives

To implement IPC using shared memory concept with the help of the library functions available.

Prerequisite

- Knowledge of IPC, Shared memory functions, their syntaxes and functionalities

Shared Memory:

- Shared memory is a memory shared between two or more processes. Each process has its own address space;

Procedure

❑Create the sender process and receiver process

❑Create a shared memory making using the appropriate function

❑Sender pushes its message into shared memory

❑Receiver retrieves the message and displays it to the user

Sample Input :

Output:

Observation

Exercise No. 2b IPC using Message Queue

Objectives

To implement IPC using message queues with the help of the library functions available

Prerequisite

Knowledge of IPC, Message queues, syntax and functionalities

Theory

- A **message queue** is an inter-process communication (IPC) mechanism that allows processes to exchange data in the form of messages between two processes
- It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed.

Procedure

❑Create the sender process and receiver process

❑Create a message queue using the appropriate functions

❑Sender pushes its message into message queue using msgsnd

❑Receiver retrieves the message using msgrcv and show it to the user

Sample Input

Output

Observation

Exercise No. 3a. Simulation of CPU Scheduling algorithms

Objectives

Simulation of preemptive and non-preemptive CPU Scheduling algorithms

Prerequisite

Knowledge of scheduling algorithms

Procedure

❑ Input the number of processes to be scheduled

❑ Input the arrival time and CPU burst time of each process

❑ Calculate the turn around time and the waiting time of the processes based on the following scheduling methods:

- First Come First Serve (FCFS), Shortest Job First (SJF),
- Round Robin(RR), Preemptive SJF or Shortest Remaining Time(SRT)
- Feedback queue scheduling

❑Compare the mean turn around time and choose the algorithm providing the best result.

Sample Input

Process	Arrival time	Burst time
P1	0	5
P2	2	3
P3	4	8

Output : Gant chatt to be displayed :

(Pno,starttime,endtime),(pno,starttime,endtime),…..idle(starttime,endtime)

e.g: (p1,0,4),(02,4,8),idle(8,9)……

FCFS- CT: P1 – 5 ; P2 – 8 ; P3 -16

SJF: P1 – 5 ; P2 – 8 ; P3 -16

RR:TQ=2: Execution order :

P1	P2	P3	P1	P2	P3	P1	P3	P3

SRT : P1 P2 P3 P2 P1

Feedback queue : Queues:

- Q1: High priority, quantum = 2 ms
- Q2: Lower priority, quantum = 4 ms

Execution Order:

- **P1** (Q1: 0 ms to 2 ms), remaining burst = 6 ms, moved to Q2
- **P2** (Q1: 2 ms to 4 ms), remaining burst = 2 ms
- **P3** (Q1: 4 ms to 5 ms), remaining burst = 0 ms (P3 completes)
- **P2** (Q1: 5 ms to 7 ms), remaining burst = 0 ms (P2 completes)
- **P1** (Q2: 7 ms to 11 ms), remaining burst = 2 ms
- **P1** (Q2: 11 ms to 13 ms), remaining burst = 0 ms (P1 completes)

Output performance measures

Throughput
Response time,
Turnaround time
Waiting time,
Average of all the above times

Observation

Exercise No. 3b. Simulation of CPU Scheduling with IO

Objectives

Simulation of CPU Scheduling algorithm with IO burst time

Prerequisite

Knowledge of scheduling algorithms

Procedure

❑ Input the number of processes to be scheduled

❑ Input the arrival time, CPU burst time1, IO burst time and CPU burst time2 of each process

❑ Calculate the turn around time and the waiting time of the processes based on the FCFS

Sample Input

- Processes: P1, P2, P3
- Arrival Times: P1 = 0 ms, P2 = 1 ms, P3 = 2 ms
- CPU and I/O Burst Times:
  - **P1**: CPU1 = 4 ms, I/O = 3 ms, CPU2 = 2 ms
  - **P2**: CPU1 = 3 ms, I/O = 4 ms, CPU2 = 1 ms
  - **P3**: CPU1 = 5 ms, I/O = 2 ms, CPU2 = 3 ms

OUTPUT:

- P1 (CPU1: 0 ms to 4 ms), goes for I/O
- P2 (CPU1: 4 ms to 7 ms), goes for I/O
- P3 (CPU1: 7 ms to 12 ms), goes for I/O
- P1 (CPU2: 12 ms to 14 ms), completes
- P2 (CPU2: 14 ms to 15 ms), completes
- P3 (CPU2: 15 ms to 18 ms), completes

Observation

Exercise No. 4   Simulate Thread Scheduling

Write a program for Thread Scheduling

Objectives

To Simulate multi-threading using pthread

Prerequisite / Tools Required (optional):

Knowledge of thread concepts and pthread functions

Procedure / Algorithm

- Create user level threads using pthreads
- Associate different function to each thread
- Assign priorities to the threads
- Schedule them on LWP by requesting the kernel

Sample Input

Output

Observation

Exercise No. 5   Simulate Peterson's Algorithm

Write a program to simulate Peterson's Algorithm

Objectives

To Simulate Peterson's algorithms for mutual exclusion

Prerequisite / Tools Required (optional)

Knowledge of critical-sections, mutual exclusion, bounded waiting, Synchronization and producer-consumer problem

Procedure / Algorithm

- Two Process Solution by sharing two variables
- Turn, Flag[2]
- Turn - Indicates whose turn is to enter the critical section (CS)
- Flag[] - Array used to indicate if a process is ready to enter CS
- flag[i]= true implies process Pi is ready

Sample Input

Output

Observation

Exercise No. 6a. Simulating Producer-Consumer problem

Write a program for simulating Producer-Consumer Problem

Objectives

Implementation of Producer-Consumer problem using bounded and unbounded variations

Prerequisite / Tools Required (optional):

Knowledge of Concurrency, Mutual exclusion, Synchronization and producer-consumer problem

Procedure / Algorithm

- ❑ Implement producer-consumer program with producers and consumers simulated as threads.
- ❑ Employ necessary semaphores for bounded and unbounded implementations
- ❑ Run the program to allow the producer and consumer share the buffer by synchronizing themselves through mutual exclusion

Sample Input

Output

Observation

Exercise No. 6b. Simulating Reader-Writer problem

Write a program to simulating Reader-Writer Problem

Objectives

To write a code to solve the readers writer's problem based on reader priority and writer priority solution

Prerequisite / Tools Required (optional)

Knowledge of Concurrency, Mutual exclusion, Synchronization

and Reader writer problem

Procedure / Algorithm

- ❑ Create a reader process
- ❑ Create a writer process
- ❑ Implement necessary semaphores
- ❑ Implement the programs giving reader priority and writer priority

Sample Input

Output

Observation

Exercise No. 7: Banker's Algorithm for Deadlock Avoidance

Write a program to simulate banker's algorithm for deadlock avoidance

Objectives

The primary goal of the Banker's Algorithm is to prevent deadlock while allocation of resources to processes, where multiple processes compete for limited resources (such as CPU time, memory, or devices).

Theory or Concept

The algorithm ensures that the system remains in a safe state during resource allocation.

A safe state allows all processes to complete their execution without getting stuck due to resource unavailability.

Procedure / Algorithm

The Banker's Algorithm keeps track of:

Available resources: The number of available instances of each resource type.

Maximum demand: The maximum resources each process may request.

Current allocation: Resources currently allocated to each process.

Remaining need: Resources still needed by each process.

Data Structures Needed:

Available: A 1-dimensional array indicating the number of available resources for each resource type.

Max: A 2-dimensional array defining the maximum demand of each process for each resource type.

Allocation: A 2-dimensional array specifying the currently allocated resources for each process.

Need: A 2-dimensional array indicating the remaining resource need for each process.

The vector $Allocation_i$ specifies the resources currently allocated to process P_i; the vector $Need_i$ specifies the additional resources that process P_i may still request to complete its task

Safety Algorithm

Safety algorithm checks whether or not the system is in a safe state.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize

Work =Available and Finish[i]=false for $i = 0, 1, ..., n - 1$.

2. Find an index i such that both

   a. Finish[i]==false

   b. $Need_i \leq Work$

   If no such i exists, go to step 4.

3. *Work=Work+Allocation*$_i$

   Finish[i]=true

   Goto step 2.

4. If *Finish[i]==true* for all *i*, then the system is in a safe state.

Resource-Request Algorithm:

Resource request algorithm determines whether requests can be safely granted.

Let *Request*$_i$ be the request vector for process P_i. If *Request*$_i$ *[j]* == *k*, then process P_i wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi,* the following actions are taken:

Step 1: If *Request*$_i$ ≤*Need*$_i$, go to step 2. Otherwise,raise an error condition, since

the process has exceeded its maximum claim.

Step 2. If *Request*$_i$ ≤ *Available*, go to step 3. Otherwise, P_i must wait, since the

resources are not available.

Step 3. Have the system pretend to have allocated the requested resources to

process *Pi* by modifying the state as follows:

$Available = Available – Request_i;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i – Request_i;$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for *request*$_i$, and the old resource-allocation state is restored.

Sample Input / Output

Resource type A has 10 instances, B has 5 instances, and C has 7 instances to the maximum. Suppose that, at time T_0, the following is the current state of the system:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The system is currently in a safe state. Indeed, the sequence <P1, P3, P4, P2, P0> satisfies the safety criteria.

Then check whether the following request can be granted immediately

Request 1: $Request_1 = (1,0,2)$

Assume following request has been granted:

New state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Check whether the new state is safe or not by executing safety algorithm.

The system is in safe state. print the safe state sequence <P1, P3, P4, P0, P2>

Request 2: $Request_4=(3,3,0)$ cannot be granted

Request 3: $Request_2=(0,2,0)$ results in unsafe state

Output:

Observation

Exercise No. 8 Deadlock Detection Algorithm

Write a program to simulate the deadlock detection algorithm

Objectives

To examine the state of the system to determine whether a deadlock has occurred and to perform recovery action to break the deadlock and restore normal operations.

Theory or Concept

The algorithm checks the allocation of resources to various processes. If it detects that the system is in a deadlock (where processes are blocked, waiting for each other to release resources), it triggers recovery mechanisms. The detection algorithm investigates every possible allocation sequence for the processes that remain to be completed.

Procedure / Algorithm

Allocation and Request vectors are referred as $Allocation_i$ and $Request_i$.

1. Initialize the following data structures:
   - *Work*: A vector of length *m* (number of resource types). Set it equal to the *Available* resources.
   - *Finish*: A vector of length *n* (number of processes). Initialize all elements to false.
2. Check Processes:
   - For each process *i* (from *0 to n-1):*
     - If *Request[i]* is zero (meaning the process doesn't need any more resources), set *Finish[i]* to true.
     - Otherwise, set *Finish[i]* to false.
3. Find a Suitable Process:
   - Search for an index *i* such that:
     - *Finish[i]* is false.
     - *Request[i]* ≤ Work.
4. Deadlock Detection:

   If no such *i* exists (i.e., all processes are either finished or their requests can't be satisfied), go to step 5.

   Otherwise, proceed to step 6.
5. No Suitable Process Found:

   If no suitable process was found in step 3, the system is in a deadlock state.

   Take appropriate actions to recover from the deadlock (e.g., terminate processes, release resources, etc.)
6. Resource Allocation:

If a suitable process was found in step 3:

- Allocate resources to process *i*.
- $Work = Work + Allocation_i$ (update work by adding the allocated resources).
- Mark process *i* as finished (*Finish[i] = true*).

7. Repeat:
   - Repeat steps 3 to 6 until either all processes are finished or a deadlock is detected.
   - If *Finish[i]==false* for some i, $0 \le i < n$, then the system is in a deadlocked state. Moreover, if *Finish[i]==false*, then process P_i is deadlocked.

Sample Input /Output:

- Number of processes (n): 5
- Number of resource types (m): 3 each with 10, 5 and 7 instances
- Allocation matrix

  [0, 1, 0]

  [2, 0, 0]

  [3, 0, 2]

  [2, 1, 1]

  [0, 0, 2]

- Request matrix
  [7, 5, 3]
  [3, 2, 2]
  [9, 0, 2]
  [2, 2, 2]
  [4, 3, 3]
- Available resources: [3, 3, 2]
  Safe sequence no deadlock
  - P2 -> P4 -> P1 -> P3
  - P2 -> P4 -> P3 -> P1
  - P4 -> P2 -> P1 -> P3
  - P4 -> P2 -> P3 -> P1

Observations

Exercise No. 9: Dining Philosopher's problem

Write a program to simulate Dining Philosopher's problem

Objectives

To design a solution that ensures the philosophers can safely share the forks and avoid deadlock or starvation (goes hungry).

Theory or Concept

Each philosopher must alternate between thinking and eating. Eating is possible only when two forks are available. It is used to handle synchronization and concurrency in process execution related to resource sharing and mutual exclusion.

Procedure / Algorithm

Step1: Represent each chopstick with a semaphore.

Step2: A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore.

Step3: A philosopher releases her chopsticks by executing the signal () operation on the appropriate semaphores.

When each philosopher tries to grab her right chopstick, philosopher will be delayed forever (deadlock) to handle the situation allow a philosopher to pick up her chopsticks only if both chopsticks are available (the philosopher must pick them up in a critical section).

```
#define N            5              /* number of philosophers */
#define LEFT         (i+N−1)%N      /* number of i's left neighbor */
#define RIGHT        (i+1)%N        /* number of i's right neighbor */
#define THINKING     0              /* philosopher is thinking */
#define HUNGRY       1              /* philosopher is trying to get forks */
#define EATING       2              /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

```
void take_forks(int i)                /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                   /* enter critical region */
      state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
      test(i);                        /* try to acquire 2 forks */
      up(&mutex);                     /* exit critical region */
      down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                     /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                   /* enter critical region */
      state[i] = THINKING;            /* philosopher has finished eating */
      test(LEFT);                     /* see if left neighbor can now eat */
      test(RIGHT);                    /* see if right neighbor can now eat */
      up(&mutex);                     /* exit critical region */
}

void test(i)                          /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Sample Input /Output

Interleaving operation of eating and thinking will be displayed for different philosophers

The solution guarantees that no two neighbours are eating simultaneously

Observation

Exercise No. 10 Program to implement dining philosopher's problem without causing deadlocks.

Objectives:

To implement a C program to simulate the concept of Dining-philosophers problem.

Pre-requisite\ Tools Required:

Knowledge of Concurrency, Deadlock and Starvation

Theory or Concept:

The dining-philosophers problem is considered a classic synchronization problem. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

Procedure / Algorithm:

Create philosopher process

- Declare semaphore for mutual exclusion and left & right forks
- Implement function for obtaining fork
- Implement function for releasing fork
- Implement function for testing blocked philosophers

Sample Input:

DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry : 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5

Output

1. One can eat at a time
2. Two can eat at a time
3. Exit Enter your choice: 1

Allow one philosopher to eat at any time
P 3 is granted to eat
P 3 is waiting
P 5 is waiting
P 0 is waiting
P 5 is granted to eat
P 5 is waiting
P 0 is waiting
P 0 is granted to eat
P 0 is waiting……

Observation

Exercise 11.a Program to simulate page replacement algorithms and to compute number of page faults

Objectives:

To simulate page replacement algorithms.

Prerequisite/ Tools Required:

Knowledge of Paging concepts, replacement algorithms

Theory or Concept:

Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.

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

LRU-In this algorithm page will be replaced which is least recently used

OPTIMAL- In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms.

Procedure \ Algorithm:

1. Start the process
2. Read number of pages n
3. Read number of pages no
4. Read page numbers into an array a[i]
5. Initialize avail[i]=0 .to check page hit
6. Replace the page with circular queue, while re-placing check page availability in the frame Place avail[i]=1 if page is placed in theframe Count page faults
7. Print the results.

calculate the page faults under each approach:

- FIFO
- Least Recently Used
- Optimal page replacement

8. Stop the process

Sample Input

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

Output

Number of page faults: 9 , No. Of page hits :   Hit ratio :   Miss ratio:

Observation

Exercise No. 11. b. Program to simulate address translation under paging.

Objectives:

To simulate the address translation from logical to physical address under paging

Prerequisite:

Knowledge of Pages, Frames, Memory partitioning

Theory / Concept:

The address translation in paging in OS is an address space that is the range of valid addresses available in a program or process memory. It is the memory space accessible to a program or process. The memory can be physical or virtual and is used for storing data and executing instructions.

Procedure /Algorithm:

Get the range of physical and logical addresses.

- Get the page size.
- Get the page number of the data.
- Construct page table by mapping logical address to physical address.
- Search page number in page table and locate the base address.
- Calculate the physical address of the data.

Sample Input:

Enter the memory size – 1000Enter
The page size -- 100
The no. of pages available in memory are 10
Enter number of processes -- 3
Enter no. of pages required for p[1]-- 4
Enter page table for p[1] --- 8 6 9 5
Enter no. of pages required for p[2]-- 5
Enter page table for p[2] --- 1 4 5 7 3
Enter no. of pages required for p[3]--
Memory is Full
5
Enter Logical Address to find Physical Address Enter process no. and page number and offset
--
2
3
60

Output

The Physical Address is -- 760

Observation

Exercise No. 12. Program to implement disk scheduling algorithms.

Objectives:

To Simulate of disk scheduling techniques.

Prerequisite / Tools Required:

Knowledge of disk scheduling algorithms.

Theory/ Concept:

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

Procedure \ 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 us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

- Calculate the seek time as per the algorithms listed below:
- First-in-First-Out
- Shortest Seek Time First
- Scan
- C-look

Sample Input :

Enter the max range of disk
200
Enter the size of queue request
8
Enter the queue of disk positions to be read
90 120 35 122 38 128 65 68
Enter the initial head position
50

Disk head moves from 50 to 90 with seek
40
Disk head moves from 90 to 120 with seek
30
Disk head moves from 120 to 35 with seek
85
Disk head moves from 35 to 122 with seek
87
Disk head moves from 122 to 38 with seek
84
Disk head moves from 38 to 128 with seek
90
Disk head moves from 128 to 65 with seek
63
Disk head moves from 65 to 68 with seek
3

Output / Performance Measures:

Total seek time is 482
Average seek time is 60.250000

Observation