

GREENFIELD NATIONAL COLLEGE

**Bafal- Sitapaila, Kathmandu**

Affiliated to [TU]

**Lab Report**

**of**

**“Operating System”**

**Submitted To: Submitted By:**

Binod Gyawali

Lecturer/ Faculty Member

BCA Department

Name: Manish karki

Semester: IV

Program: Bachelor of Computer Application

**TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| **S.N.** | **TTTLE** | **SIGNATURE** |
| 1. | Write a C program for implementing process creation and termination. |  |
| 2. | Write a C program for implementing thread creation and termination. |  |
| 3. | Write a C program for Implementing Process Scheduling Algorithm.   1. FCFS (First Come First Serve) 2. SJF (Shortest Job First) 3. SRTF (Shortest Remaining Time First) 4. RR (Round-Robbin CPU Scheduling Algorithm). 5. Priority Scheduling. Also compute TAT and WT time of each algorithm. |  |
| 4. | To implement deadlock detection Algorithms. (Bankers Algorithms) |  |
| 5. | To Implement Memory Allocation Techniques.  a. First Fit b. Best Fit c. Worst Fit |  |
| 6. | To implement Page Replacement Algorithms   1. FIFO 2. LRU 3. Optimal Page Replacement Algorithms |  |
| 7. | To implement Disk Scheduling Algorithm  a. FCFS b. SSTF c. SCAN  d. C-SCAN e. LOOK f. C-LOOK |  |

# Lab : 1 Title :

**A C program for implementing Process Creation and Termination. Theory :**

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. Processes are organized hierarchically. Each process has a parent process, which explicitly arranged to create it. The processes created by a given parent are called its child processes. A child inherits many of its attributes from the parent process.

Processes are created with the fork() system call (so the operation of creating a new process is sometimes called forking a process). The child process created by fork is a copy of the original parent process, except that it has its own process ID.

When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action. A process that is waiting for its parent to accept its return code is called a zombie process. If a parent dies before its child, the child (orphan process) is automatically adopted by the original “init” process whose PID is 1.

# Program :

#include <stdio.h>

#include <unistd.h> /\* contains fork prototype \*/ int main(void)

{

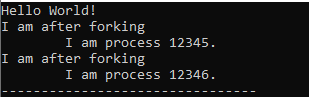
printf("Hello World!\n"); fork( );

printf("I am after forking\n");

printf("\tI am process %d.\n", getpid( ));

}

# Output :

****

**Lab : 2 Title :**

**A C program for implementing thread creation and termination.**

# Theory :

Threads are lightweight processes that enable concurrent execution of code within the same program. In C, threads are commonly implemented using the POSIX threads (pthreads) library.. Threads share the same memory space, making them efficient for tasks that require communication or shared data, but they also require careful synchronization to avoid race conditions.

To create a thread, the pthread\_create() function is used. Once a thread is created, it begins executing the specified function concurrently with the main program. The pthread\_self() function can be used to retrieve the ID of the current thread, enabling easy identification of threads in a multi-threaded environment.

# Program :

#include <stdio.h> #include <stdlib.h> #include <pthread.h>

void \*thread\_function(void \*arg) {

printf("Hello from the thread! Thread ID: %ld\n", pthread\_self()); printf("Thread is exiting...\n");

pthread\_exit(NULL); // Exit the thread

}

int main() {

pthread\_t thread\_id; int ret;

ret = pthread\_create(&thread\_id, NULL, thread\_function, NULL); if (ret != 0) {

perror("Thread creation failed"); exit(1);

}

printf("Thread created successfully. Main thread ID: %ld\n", pthread\_self()); ret = pthread\_join(thread\_id, NULL);

if (ret != 0) {

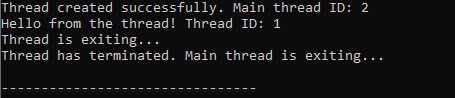
perror("Thread join failed"); exit(1);

}

printf("Thread has terminated. Main thread is exiting...\n"); return 0;

}

# Output :

****

**Lab : 3 Title :**

**A C program for implementing Process Scheduling Algorithm**

# Objectives :

* To Simulate First Come First Serve (FCFS) scheduling algorithm in C
* To Simulate Shortest Job First (SJF) scheduling algorithm in C
* To Simulate Shortest Remaining Time First (SRTF) scheduling algorithm in C
* To Simulate Round Robin (RR) scheduling algorithm in C
* To Simulate Priority scheduling algorithm in C

# Theory :

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 says first come first serve the algorithm which came first will be served first.

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.

To calculate Shortest Remaining Time First (SRTF) scheduling, first gather the arrival times and burst times for all processes. Check which processes have arrived and selecting the one with the shortest remaining time to execute; if a new process arrives with a shorter remaining time than the currently running process, preempt the current process. As each process completes (when its remaining time reaches zero), update its completion time and calculate waiting and turnaround times.

To calculate Round Robin scheduling, 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.

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.

**Program 1:FCFS**

#include <stdio.h> int main() {

int n, bt[20], wt[20], tat[20], avwt = 0, avtat = 0, i, j; printf("Enter total number of processes (maximum 20): "); scanf("%d", &n);

printf("\nEnter Process Burst Time:\n"); for(i = 0; i < n; i++) {

printf("P[%d]: ", i + 1);

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

}

wt[0] = 0;

for(i = 1; i < n; i++) { wt[i] = 0;

for(j = 0; j < i; j++) { wt[i] += bt[j];

}

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n"); for(i = 0; i < n; i++) {

tat[i] = bt[i] + wt[i]; avwt += wt[i]; avtat += tat[i];

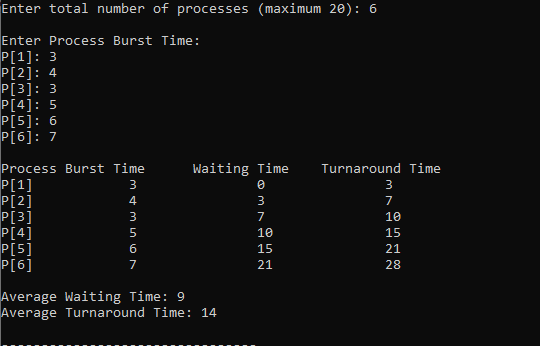
printf("P[%d]\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);

}

avwt /= n; avtat /= n;

printf("\nAverage Waiting Time: %d", avwt); printf("\nAverage Turnaround Time: %d\n", avtat); return 0; }

**Output : 1**

****

**Program 2: SJF** #include <stdio.h> int main() {

int n, bt[20], p[20], wt[20] = {0}, tat[20], i, j, total\_wt = 0, total\_tat = 0; printf("''Shortest Job First'' \n");

printf("Enter number of processes:"); scanf("%d", &n);

printf("\nEnter Burst Time:\n"); for (i = 0; i < n; i++) {

printf("P%d: ", i + 1);

scanf("%d", &bt[i]); p[i] = i + 1;

}

for (i = 0; i < n - 1; i++) { for (j = i + 1; j < n; j++) { if (bt[i] > bt[j]) {

int temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;

temp = p[i]; p[i] = p[j]; p[j] = temp;

}

}

}

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

tat[i] = bt[i] + wt[i]; total\_wt += wt[i]; total\_tat += tat[i];

}

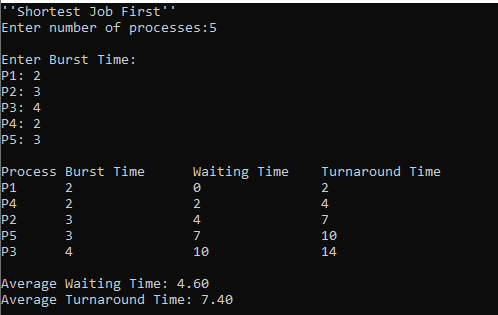
printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

for (i = 0; i < n; i++) printf("P%d\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]); printf("\nAverage Waiting Time: %.2f", (float)total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", (float)total\_tat / n); return 0;

}

# Output : 2

****

**Program 3 : SRTF** #include <stdio.h> #include <limits.h> int main() {

int n, bt[20], rt[20], wt[20] = {0}, tat[20]; int time = 0, completed = 0, smallest;

int i, min\_rt;

float total\_wt = 0, total\_tat = 0;

printf("''''Shortest Remaining Time First'''' \n"); printf("Enter number of processes: ");

scanf("%d", &n);

printf("\nEnter Burst Time for each process:\n"); for (i = 0; i < n; i++) {

printf("P%d: ", i + 1);

scanf("%d", &bt[i]); rt[i] = bt[i];

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n"); while (completed < n) {

smallest = -1;

min\_rt = INT\_MAX; for (i = 0; i < n; i++) {

if (rt[i] > 0 && rt[i] < min\_rt) { min\_rt = rt[i];

smallest = i;

}

}

if (smallest == -1) break; rt[smallest]--;

time++;

if (rt[smallest] == 0) { completed++;

tat[smallest] = time;

wt[smallest] = tat[smallest] - bt[smallest]; total\_wt += wt[smallest];

total\_tat += tat[smallest];

printf("P%d\t%d\t\t%d\t\t%d\n", smallest + 1, bt[smallest], wt[smallest], tat[smallest]);

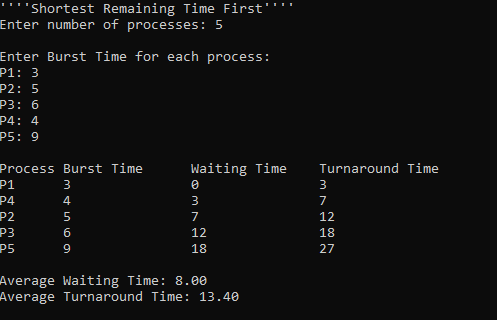
}

}

printf("\nAverage Waiting Time: %.2f", total\_wt / n); printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n); return 0;

}

# Output : 3

****

**Program 4 : Round Robin**

#include <stdio.h> int main() {

int n, tq, bt[20], rt[20], wt[20] = {0}, tat[20] = {0}; int time = 0, completed = 0, i;

float total\_wt = 0, total\_tat = 0;

printf("''''Round Robin'''' \n"); printf("Enter number of processes: "); scanf("%d", &n);

printf("Enter Time Quantum: "); scanf("%d", &tq);

printf("\nEnter Burst Time for each process:\n"); for (i = 0; i < n; i++) {

printf("P%d: ", i + 1);

scanf("%d", &bt[i]); rt[i] = bt[i];

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n"); while (completed < n) {

for (i = 0; i < n; i++) { if (rt[i] > 0) {

int exec\_time = (rt[i] > tq) ? tq : rt[i]; time += exec\_time;

rt[i] -= exec\_time; if (rt[i] == 0) {

completed++; tat[i] = time;

wt[i] = tat[i] - bt[i];

total\_wt += wt[i]; total\_tat += tat[i];

printf("P%d\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);

}

}

}

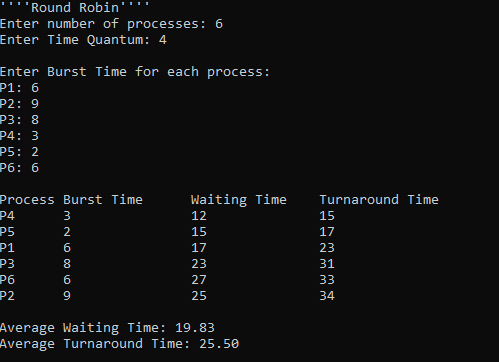
}

printf("\nAverage Waiting Time: %.2f", total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n); return 0;

}

# Output : 4

****

**Program 5 : Priority Schdeuling**

#include <stdio.h> int main() {

int n, i, j, bt[10], p[10], pr[10], wt[10] = {0}, tat[10] = {0}, temp; float total\_wt = 0, total\_tat = 0;

printf("''''Priority Scheduling'''' \n"); printf("Enter number of processes: "); scanf("%d", &n);

printf("Enter Burst Time and Priority for each process:\n"); for (i = 0; i < n; i++) {

printf("P%d Burst Time: ", i + 1); scanf("%d", &bt[i]);

printf("P%d Priority: ", i + 1); scanf("%d", &pr[i]);

p[i] = i + 1;

}

for (i = 0; i < n - 1; i++) { for (j = i + 1; j < n; j++) { if (pr[i] > pr[j]) {

temp = pr[i]; pr[i] = pr[j]; pr[j] = temp;

temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;

temp = p[i]; p[i] = p[j]; p[j] = temp;

}

}

}

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

tat[i] = wt[i] + bt[i]; total\_wt += wt[i];

total\_tat += tat[i];

}

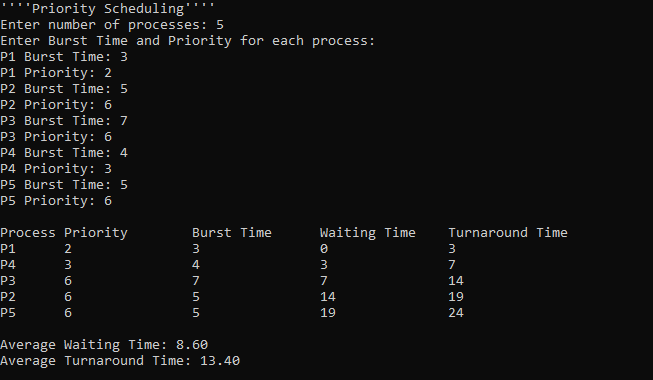
printf("\nProcess\tPriority\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (i = 0; i < n; i++)

printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i], pr[i], bt[i], wt[i], tat[i]); printf("\nAverage Waiting Time: %.2f", total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n); return 0;

}

# Output : 5

****

**Lab : 4 Title :**

**Implementation of deadlock avoidance algorithms.(Bankers Algoritm)**

# Theory :

Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare

the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the userrequest a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

* n-Number of process, m-number of resource types
* Available: Available[j]=k, k – instance of resource type Rj is available. Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
* Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj Need:

If Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]- Allocation[I, j];

*Safety Algorithm*

* Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
* Find an i such that both
  1. Finish[i]=False
  2. Need<=Work If no such I exists go to step 4.
* work= work + Allocation, Finish[i] =True;
* if Finish[1]=True for all I, then the system is in safe state. Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

* if Request<=Need I go to step 2. Otherwise raise an error condition.
* if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
* 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 +Request I;
* Need i=Need i- Request I;
* If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state isrestored.

# Program :

# #include <stdio.h>

# #include <stdbool.h>

# #define MAX\_PROCESSES 10

# #define MAX\_RESOURCES 10

# int n, m, available[MAX\_RESOURCES], max[MAX\_PROCESSES][MAX\_RESOURCES], allocation[MAX\_PROCESSES][MAX\_RESOURCES], need[MAX\_PROCESSES][MAX\_RESOURCES];

# void calculateNeed() {

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

# for (int j = 0; j < m; j++)

# need[i][j] = max[i][j] - allocation[i][j];

# }

# bool isSafe() {

# int work[MAX\_RESOURCES], finish[MAX\_PROCESSES] = {0}, safeSequence[MAX\_PROCESSES], index = 0;

# for (int i = 0; i < m; i++) work[i] = available[i];

# for (int k = 0; k < n; k++) {

# bool found = false;

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

# if (!finish[i]) {

# int j;

# for (j = 0; j < m && need[i][j] <= work[j]; j++);

# if (j == m) {

# for (int y = 0; y < m; y++) work[y] += allocation[i][y];

# safeSequence[index++] = i;

# finish[i] = found = 1;

# }

# }

# }

# if (!found) break;

# }

# if (index < n) {

# printf("The system is in an unsafe state!\n");

# return false;

# }

# printf("The system is in a safe state.\nSafe sequence is: ");

# for (int i = 0; i < index; i++) printf("P%d ", safeSequence[i]);

# printf("\n");

# return true;

# }

# int main() {

# printf("Enter the number of processes (max %d): ", MAX\_PROCESSES);

# scanf("%d", &n);

# printf("Enter the number of resource types (max %d): ", MAX\_RESOURCES);

# scanf("%d", &m);

# printf("Enter the available resources: \n");

# for (int i = 0; i < m; i++) scanf("%d", &available[i]);

# printf("Enter the maximum resource matrix (Max): \n");

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

# printf("For process P%d: ", i);

# for (int j = 0; j < m; j++) scanf("%d", &max[i][j]);

# }

# printf("Enter the allocation resource matrix (Allocation): \n");

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

# printf("For process P%d: ", i);

# for (int j = 0; j < m; j++) scanf("%d", &allocation[i][j]);

# }

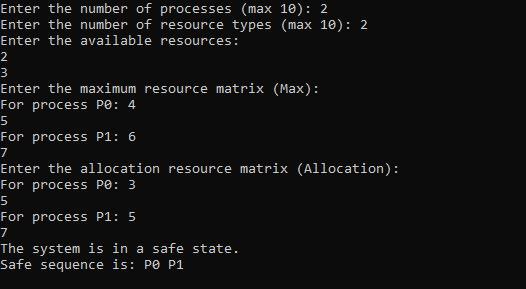
# calculateNeed();

# isSafe();

# return 0;

# }

# Output :

****

**Lab : 5 Title :**

**Implementation of Memory Allocation Techniques.**

# Theory :

One of the simplest methods for memory allocation is to divide memory into several fixed- sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

# Program 1 : First Fit

#include <stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n) { int allocation[n];

int i, j;

for (i = 0; i < n; i++) { allocation[i] = -1;

}

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

for (j = 0; j < m; j++) {

if (blockSize[j] >= processSize[i]) { allocation[i] = j;

blockSize[j] -= processSize[i]; break;

}

}

}

printf("\nProcess No.\tProcess Size\tBlock No.\n"); for (i = 0; i < n; i++) {

printf("%d\t\t %d\t\t", i + 1, processSize[i]); if (allocation[i] != -1)

printf(" %d\n", allocation[i] + 1); else

printf("Need to wait\n");

}

}

int main() { int m, n, i;

printf("Enter the number of memory blocks: "); scanf("%d", &m);

int blockSize[m];

printf("Enter the sizes of the memory blocks in order: "); for (i = 0; i < m; i++) {

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

}

printf("Enter the number of processes: "); scanf("%d", &n);

int processSize[n];

printf("Enter the sizes of the processes in order: "); for (i = 0; i < n; i++) {

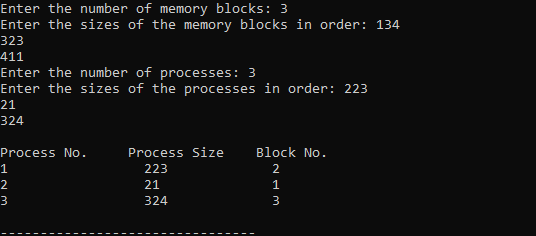
scanf("%d", &processSize[i]);

}

firstFit(blockSize, m, processSize, n); return 0;

}

**Output : 1**

****

**Program 2 : Best Fit**

#include <stdio.h>

void bestFit(int blockSize[], int m, int processSize[], int n) { int allocation[10];

int i, j;

for (i = 0; i < n; i++) { allocation[i] = -1;

}

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

for (j = 0; j < m; j++) {

if (blockSize[j] >= processSize[i]) {

if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) { bestIdx = j;

}

}

}

if (bestIdx != -1) { allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

printf("\nProcess No.\tProcess Size\tBlock No.\n"); for (i = 0; i < n; i++) {

printf("%d\t\t%d\t\t", i + 1, processSize[i]); if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Need to wait\n");

}

}

}

int main() { int m, n, i;

printf("Enter the number of memory blocks: "); scanf("%d", &m);

int blockSize[10];

printf("Enter the sizes of the memory blocks in order: "); for (i = 0; i < m; i++) {

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

}

printf("Enter the number of processes: "); scanf("%d", &n);

int processSize[10];

printf("Enter the sizes of the processes in order: "); for (i = 0; i < n; i++) {

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

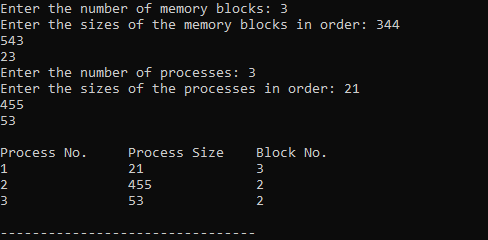
}

bestFit(blockSize, m, processSize, n);

return 0;

}

# Output : 2

****

**Program 3 : Worst Fit** #include <stdio.h> #include <string.h>

void worstFit(int blockSize[], int m, int processSize[], int n) { int allocation[n];

memset(allocation, -1, sizeof(allocation)); for (int i = 0; i < n; i++) {

int wstIdx = -1;

for (int j = 0; j < m; j++) {

if (blockSize[j] >= processSize[i]) {

if (wstIdx == -1 || blockSize[wstIdx] < blockSize[j]) { wstIdx = j;

}

}

}

if (wstIdx != -1) { allocation[i] = wstIdx;

blockSize[wstIdx] -= processSize[i];

}

}

printf("\nProcess No.\tProcess Size\tBlock no.\n"); for (int i = 0; i < n; i++) {

printf(" %d\t\t%d\t\t", i + 1, processSize[i]); if (allocation[i] != -1)

printf("%d", allocation[i] + 1); else

printf("Not Allocated"); printf("\n");

}

}

int main() {

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

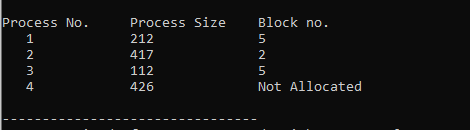
int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]); worstFit(blockSize, m, processSize, n);

return 0;

}

# Output : 3

****

**Lab : 6 Title :**

**Implementation of Page Replacement Algorithms.**

# Theory :

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.

**Program 1 : FIFO** #include <stdio.h int main() {

int capacity, n, page\_faults = 0, page\_hits = 0, front = 0; printf("Enter the number of frames: ");

scanf("%d", &capacity);

printf("Enter the number of page requests: "); scanf("%d", &n);

int frames[100]; int pages[100]; int filled = 0;

int i, j;

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

}

printf("Enter the page reference string: ");

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

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

}

for (i = 0; i < n; i++) { int page = pages[i]; int found = 0;

for (j = 0; j < filled; j++) { if (frames[j] == page) {

found = 1; break;

}

}

printf("Page %2d: ", page); if (found) {

page\_hits++;

} else {

if (filled < capacity) { frames[filled] = page; filled++;

} else {

frames[front] = page;

front = (front + 1) % capacity;

}

page\_faults++;

}

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

printf(" - "); else

printf("%3d ", frames[j]);

}

if (found) {

printf("[ H ]\n");

} else {

printf("[ F ]\n");

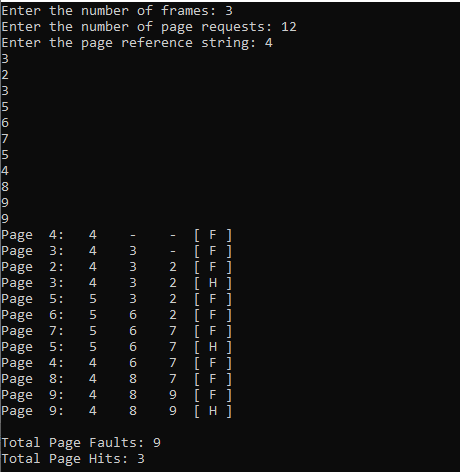
}

}

printf("\nTotal Page Faults: %d\n", page\_faults); printf("Total Page Hits: %d\n", page\_hits); return 0;

}

# Output :

****

**Program 2 : LRU** #include <stdio.h> struct Page {

int value;

int frequency; int last\_used;

};

int main() {

int capacity, n, page\_faults = 0, page\_hits = 0, time = 0; printf("Enter the number of frames: ");

scanf("%d", &capacity);

printf("Enter the number of page requests: "); scanf("%d", &n);

struct Page frames[100]; int pages[100];

int i;

for (i = 0; i < capacity; i++) { frames[i].value = -1;

frames[i].frequency = 0;

frames[i].last\_used = 0;

}

printf("Enter the page reference string: "); for (i = 0; i < n; i++) {

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

}

for (i = 0; i < n; i++) { int page = pages[i]; int found = 0;

time++; int j;

for (j = 0; j < capacity; j++) {

if (frames[j].value == page) { found = 1;

frames[j].frequency++;

frames[j].last\_used = time; break;

}

}

printf("Page %2d: ", page); if (found) {

page\_hits++;

} else {

page\_faults++;

int min\_freq = frames[0].frequency, replace\_idx = 0; for (j = 1; j < capacity; j++) {

if (frames[j].frequency < min\_freq ||

(frames[j].frequency == min\_freq && frames[j].last\_used < frames[replace\_idx].last\_used)) {

min\_freq = frames[j].frequency; replace\_idx = j;

}

}

frames[replace\_idx].value = page; frames[replace\_idx].frequency = 1; frames[replace\_idx].last\_used = time;

}

for (j = 0; j < capacity; j++) { if (frames[j].value == -1)

printf(" - "); else

printf("%3d ", frames[j].value);

}

if (found) {

printf("[ H ]\n");

} else {

printf("[ F ]\n");

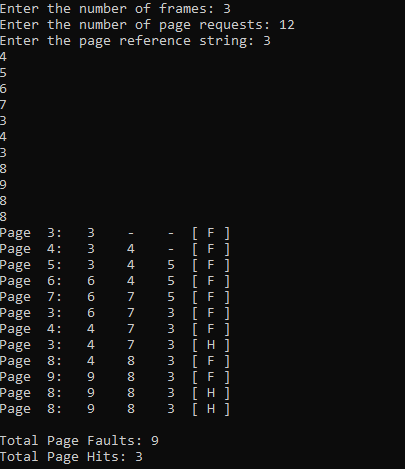
}

}

printf("\nTotal Page Faults: %d\n", page\_faults); printf("Total Page Hits: %d\n", page\_hits); return 0;

}

# Output : 2

****

**Program 3 : OPR**

#include <stdio.h>

int find\_farthest(int pages[], int frames[], int n, int current\_index, int capacity) { int farthest\_index = -1;

int farthest\_distance = -1; int i, j;

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

for (j = current\_index + 1; j < n; j++) { if (frames[i] == pages[j]) {

if (j > farthest\_distance) { farthest\_distance = j; farthest\_index = i;

}

break;

}

}

if (j == n) { return i;

}

}

return (farthest\_index == -1) ? 0 : farthest\_index;

}

int main() {

int capacity, n, page\_faults = 0, page\_hits = 0; int filled = 0;

printf("Enter the number of frames: "); scanf("%d", &capacity);

printf("Enter the number of page requests: "); scanf("%d", &n);

int frames[100]; int pages[100]; int i;

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

}

printf("Enter the page reference string: "); for (i = 0; i < n; i++) {

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

}

for (i = 0; i < n; i++) { int page = pages[i]; int found = 0;

int j;

for (j = 0; j < filled; j++) { if (frames[j] == page) {

found = 1; break;

}

}

printf("Page %2d: ", page); if (found) {

page\_hits++;

} else {

page\_faults++;

if (filled < capacity) { frames[filled] = page; filled++;

} else {

int farthest\_index = find\_farthest(pages, frames, n, i, capacity); frames[farthest\_index] = page;

}

}

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

printf(" - "); else

printf("%3d ", frames[j]);

}

if (found) {

printf("[ H ]\n");

} else {

printf("[ F ]\n");

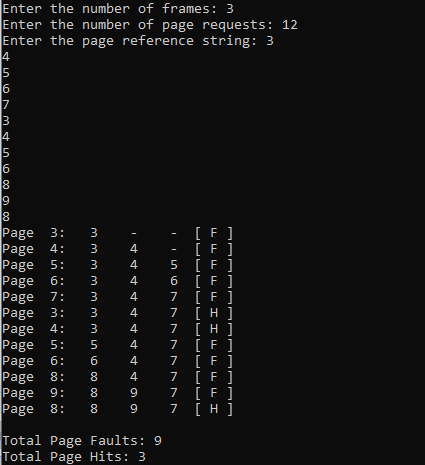
}

}

printf("\nTotal Page Faults: %d\n", page\_faults); printf("Total Page Hits: %d\n", page\_hits); return 0;

}

# Output : 3

****

**Lab : 7 Title :**

**Implementation of Disk Schduling Algorithms**

# Theory :

The FCFS (First-Come, First-Served) disk scheduling algorithm is a straightforward and simple method used in operating systems to manage input/output (I/O) requests from processes to access disk blocks. This algorithm processes I/O requests in the order they arrive in the queue, without any reordering or prioritization. When a process generates an I/O request, it is added to the end of the queue, and the operating system services the requests in the same order.

Shortest Seek Time First, is a disk scheduling algorithm that prioritizes servicing I/O requests based on the shortest seek time from the current position of the disk head. This algorithm improves upon the First-Come, First-Served (FCFS) method by selecting requests that minimize the total movement of the disk arm, thereby reducing average wait times and improving overall system performance. When a new I/O request is generated, SSTF calculates the seek time for each pending request and selects the one that requires the least movement to service next.

The SCAN disk scheduling algorithm, also known as the Elevator algorithm, is a method used in operating systems to manage input/output (I/O) requests from processes to access disk blocks. This algorithm is designed to improve disk system efficiency by reducing the seek time, which is the time it takes for the disk arm to move to the track where the data is stored.

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.

The LOOK algorithm is an enhancement of the SCAN algorithm. In LOOK, the disk arm moves in one direction (either inward or outward) and serves all requests in that direction until there are no more requests to process. Once it reaches the last request in that direction, it reverses direction and services requests on its way back.

C-LOOK (Circular LOOK) is a variation of the LOOK algorithm that restricts the movement of the disk arm to one direction only. After servicing all requests in its current direction, instead of reversing direction, the arm jumps back to the first request in that same direction and continues servicing.

**Program 1:FCFS** #include <stdio.h> #include <stdlib.h>

void FCFS(int requests[], int n, int head) {

int total\_head\_movement = 0; int i;

printf("Initial Head Position: %d\n", head); printf("Disk Head Movement Sequence:\n"); for (i = 0; i < n; i++) {

printf("%d -> %d\n", head, requests[i]); int movement = abs(requests[i] - head); total\_head\_movement += movement; head = requests[i];

}

printf("Total Distance Covered in Tracks: %d\n", total\_head\_movement);

}

int main() {

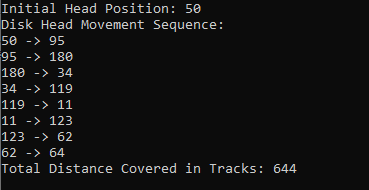
int requests[] = {95, 180, 34, 119, 11, 123, 62, 64};

int n = sizeof(requests) / sizeof(requests[0]); int head = 50;

FCFS(requests, n, head); return 0;

}

# Output :

****

**Program 2:SSTF** #include <stdio.h> #include <stdlib.h>

void SSTF(int requests[], int n, int head) { int total\_head\_movement = 0;

int completed[100] = {0}; int current\_index, i;

printf("Initial Head Position: %d\n", head); printf("Disk Head Movement Sequence:\n"); for (int count = 0; count < n; count++) {

int min\_distance = 10000; int closest\_index = -1;

for (i = 0; i < n; i++) { if (!completed[i]) {

int distance = abs(requests[i] - head); if (distance < min\_distance) {

min\_distance = distance; closest\_index = i;

}

}

}

if (closest\_index != -1) {

printf("%d -> %d\n", head, requests[closest\_index]); total\_head\_movement += min\_distance;

head = requests[closest\_index]; completed[closest\_index] = 1;

}

}

printf("Total Distance Covered in Tracks: %d\n", total\_head\_movement);

}

int main() {

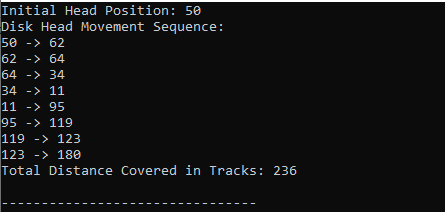
int requests[] = {95, 180, 34, 119, 11, 123, 62, 64};

int n = sizeof(requests) / sizeof(requests[0]); int head = 50;

SSTF(requests, n, head); return 0;

}

# Output :

****

**Program 3:SCAN** #include <stdio.h> #include <stdlib.h>

void SCAN(int requests[], int n, int head, int direction) { int total\_head\_movement = 0;

int current\_index, i;

printf("Initial Head Position: %d\n", head); printf("Disk Head Movement Sequence:\n"); for (i = 0; i < n; i++) {

for (int j = i + 1; j < n; j++) {

if (requests[i] > requests[j]) { int temp = requests[i]; requests[i] = requests[j];

requests[j] = temp;

}

}

}

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

if (requests[i] >= head) { current\_index = i; break;

}

}

if (direction == 1) {

for (i = current\_index; i < n; i++) {

printf("%d -> %d\n", head, requests[i]);

total\_head\_movement += abs(requests[i] - head); head = requests[i];

}

printf("%d -> %d\n", head, 199);

total\_head\_movement += abs(199 - head); head = 199;

for (i = n - 1; i >= 0; i--) {

printf("%d -> %d\n", head, requests[i]);

total\_head\_movement += abs(requests[i] - head); head = requests[i];

}

} else {

for (i = current\_index - 1; i >= 0; i--) {

printf("%d -> %d\n", head, requests[i]);

total\_head\_movement += abs(requests[i] - head); head = requests[i];

}

printf("%d -> %d\n", head, 0);

total\_head\_movement += abs(0 - head); head = 0;

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

printf("%d -> %d\n", head, requests[i]);

total\_head\_movement += abs(requests[i] - head); head = requests[i];

}

}

printf("Total Distance Covered in Tracks: %d\n", total\_head\_movement);

}

int main() {

int requests[] = {95, 180, 34, 119, 11, 123, 62, 64};

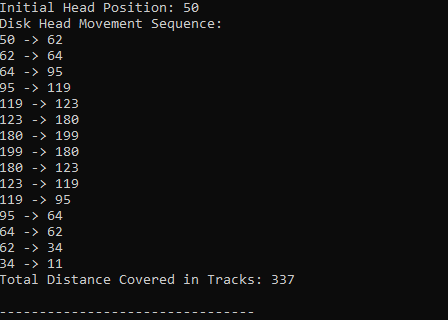
int n = sizeof(requests) / sizeof(requests[0]); int head = 50;

int direction = 1;

SCAN(requests, n, head, direction); return 0;

}

# Output :

****

**Program 4:C-SCAN** #include <stdio.h> #include <stdlib.h>

void C\_SCAN\_Top\_Bottom(int requests[], int n, int head, int disk\_size)

{

int total\_head\_movement = 0; int sorted\_requests[n + 2];

int i, j;

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

sorted\_requests[i] = requests[i]; sorted\_requests[n] = 0;

sorted\_requests[n + 1] = disk\_size - 1; for (i = 0; i < n + 2; i++)

{

for (j = i + 1; j < n + 2; j++)

{

if (sorted\_requests[i] > sorted\_requests[j])

{

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j]; sorted\_requests[j] = temp;

}

}

}

printf("Initial Head Position: %d\n", head);

printf("Disk Head Movement Sequence (Top-Bottom C-SCAN Approach):\n"); int start\_index = 0;

for (i = 0; i < n + 2; i++)

{

if (sorted\_requests[i] >= head)

{

start\_index = i; break;

}

}

for (i = start\_index; i < n + 2; i++)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

if (head != 0)

{

printf("%d -> 0 (Jump)\n", head);

total\_head\_movement += abs(head - 0); head = 0;

}

for (i = 0; i < start\_index; i++)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

printf("Total Distance Covered in Tracks (Top-Bottom C-SCAN): %d\n", total\_head\_movement);

}

void C\_SCAN\_Bottom\_Top(int requests[], int n, int head, int disk\_size)

{

int total\_head\_movement = 0;

int sorted\_requests[n + 2]; int i, j;

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

sorted\_requests[i] = requests[i]; sorted\_requests[n] = 0;

sorted\_requests[n + 1] = disk\_size - 1; for (i = 0; i < n + 2; i++)

{

for (j = i + 1; j < n + 2; j++)

{

if (sorted\_requests[i] > sorted\_requests[j])

{

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j]; sorted\_requests[j] = temp;

}

}

}

printf("\nInitial Head Position: %d\n", head);

printf("Disk Head Movement Sequence (Bottom-Top C-SCAN Approach):\n"); int start\_index = 0;

for (i = 0; i < n + 2; i++)

{

if (sorted\_requests[i] >= head)

{

start\_index = i; break;

}

}

for (i = start\_index - 1; i >= 0; i--)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

if (head != disk\_size - 1)

{

printf("%d -> %d (Jump)\n", head, disk\_size - 1);

total\_head\_movement += abs(head - (disk\_size - 1)); head = disk\_size - 1;

}

for (i = n + 1; i >= start\_index; i--)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

printf("Total Distance Covered in Tracks (Bottom-Top C-SCAN): %d\n", total\_head\_movement);

}

int main()

{

int requests[] = {95, 180, 34, 119, 11, 123, 62, 64};

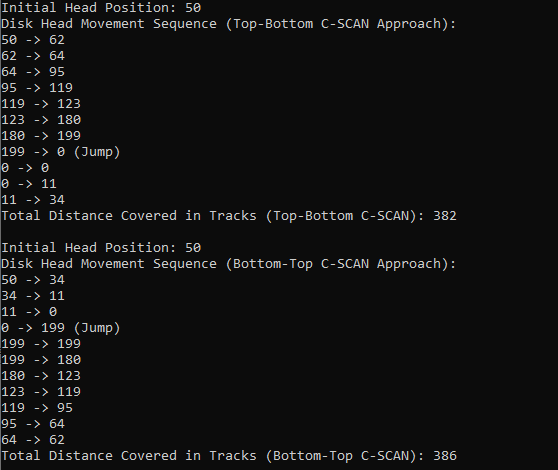
int n = sizeof(requests) / sizeof(requests[0]); int head = 50;

int disk\_size = 200;

C\_SCAN\_Top\_Bottom(requests, n, head, disk\_size); C\_SCAN\_Bottom\_Top(requests, n, head, disk\_size); return 0;

}

# Output :

****

**Program 5:LOOK** #include <stdio.h> #include <stdlib.h>

void LOOK\_Top\_Bottom(int requests[], int n, int head)

{

int total\_head\_movement = 0; int sorted\_requests[n];

int i, j;

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

sorted\_requests[i] = requests[i]; for (i = 0; i < n; i++)

{

for (j = i + 1; j < n; j++)

{

if (sorted\_requests[i] > sorted\_requests[j])

{

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j]; sorted\_requests[j] = temp;

}

}

}

printf("Initial Head Position: %d\n", head);

printf("Disk Head Movement Sequence (Top-Bottom LOOK Approach):\n"); int start\_index = 0;

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

{

if (sorted\_requests[i] >= head)

{

start\_index = i; break;

}

}

for (i = start\_index; i < n; i++)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

for (i = start\_index - 1; i >= 0; i--)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

printf("Total Distance Covered in Tracks (Top-Bottom LOOK): %d\n", total\_head\_movement);

}

void LOOK\_Bottom\_Top(int requests[], int n, int head)

{

int total\_head\_movement = 0; int sorted\_requests[n];

int i, j;

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

sorted\_requests[i] = requests[i]; for (i = 0; i < n; i++)

{

for (j = i + 1; j < n; j++)

{

if (sorted\_requests[i] > sorted\_requests[j])

{

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j]; sorted\_requests[j] = temp;

}

}

}

printf("\nInitial Head Position: %d\n", head);

printf("Disk Head Movement Sequence (Bottom-Top LOOK Approach):\n"); int start\_index = 0;

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

{

if (sorted\_requests[i] >= head)

{

start\_index = i; break;

}

}

for (i = start\_index - 1; i >= 0; i--)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

for (i = start\_index; i < n; i++)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

printf("Total Distance Covered in Tracks (Bottom-Top LOOK): %d\n", total\_head\_movement);

}

int main()

{

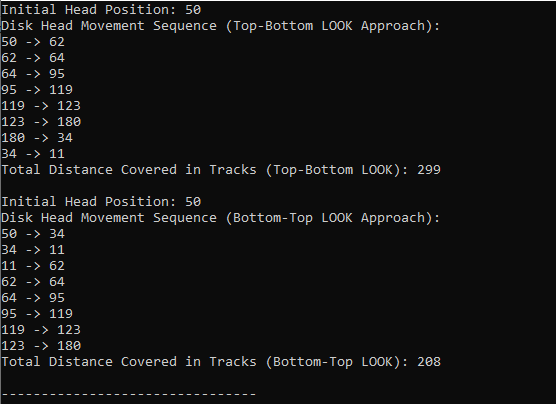
int requests[] = {95, 180, 34, 119, 11, 123, 62, 64};

int n = sizeof(requests) / sizeof(requests[0]); int head = 50;

LOOK\_Top\_Bottom(requests, n, head); LOOK\_Bottom\_Top(requests, n, head); return 0;

}

# Output :

****

**Program 6: C-LOOK** #include <stdio.h> #include <stdlib.h>

void C\_LOOK\_Top\_Bottom(int requests[], int n, int head)

{

int total\_head\_movement = 0; int sorted\_requests[n];

int i, j;

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

sorted\_requests[i] = requests[i]; for (i = 0; i < n; i++)

{

for (j = i + 1; j < n; j++)

{

if (sorted\_requests[i] > sorted\_requests[j])

{

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j]; sorted\_requests[j] = temp;

}

}

}

printf("Initial Head Position: %d\n", head);

printf("Disk Head Movement Sequence (Top-Bottom C-LOOK Approach):\n"); int start\_index = 0;

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

{

if (sorted\_requests[i] >= head)

{

start\_index = i; break;

}

}

for (i = start\_index; i < n; i++)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

if (start\_index > 0)

{

printf("%d -> %d (Jump)\n", head, sorted\_requests[0]);

total\_head\_movement += abs(head - sorted\_requests[0]);

head = sorted\_requests[0]; for (i = 0; i < start\_index; i++)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

}

printf("Total Distance Covered in Tracks (Top-Bottom C-LOOK): %d\n", total\_head\_movement);

}

void C\_LOOK\_Bottom\_Top(int requests[], int n, int head)

{

int total\_head\_movement = 0; int sorted\_requests[n];

int i, j;

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

sorted\_requests[i] = requests[i]; for (i = 0; i < n; i++)

{

for (j = i + 1; j < n; j++)

{

if (sorted\_requests[i] > sorted\_requests[j])

{

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j]; sorted\_requests[j] = temp;

}

}

}

printf("\nInitial Head Position: %d\n", head);

printf("Disk Head Movement Sequence (Bottom-Top C-LOOK Approach):\n"); int start\_index = 0;

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

{

if (sorted\_requests[i] >= head)

{

start\_index = i; break;

}

}

for (i = start\_index - 1; i >= 0; i--)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

if (start\_index < n)

{

printf("%d -> %d (Jump)\n", head, sorted\_requests[n - 1]);

total\_head\_movement += abs(head - sorted\_requests[n - 1]); head = sorted\_requests[n - 1];

for (i = n - 2; i >= start\_index; i--)

{

printf("%d -> %d\n", head, sorted\_requests[i]);

total\_head\_movement += abs(sorted\_requests[i] - head); head = sorted\_requests[i];

}

}

printf("Total Distance Covered in Tracks (Bottom-Top C-LOOK): %d\n", total\_head\_movement);

}

int main()

{

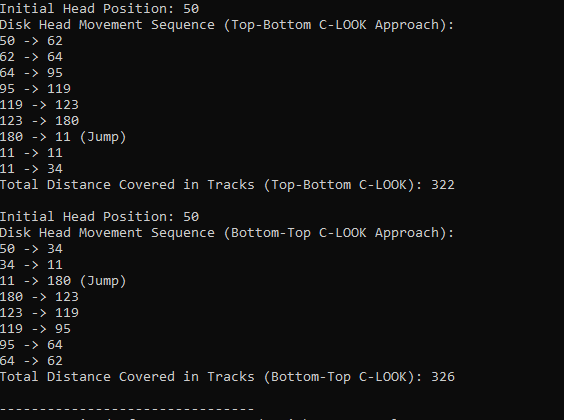
int requests[] = {95, 180, 34, 119, 11, 123, 62, 64};

int n = sizeof(requests) / sizeof(requests[0]); int head = 50;

C\_LOOK\_Top\_Bottom(requests, n, head); C\_LOOK\_Bottom\_Top(requests, n, head); return 0;

}

**Output :**

****