OSPractical1b

#!/bin/bash

# File to store address book records

address\_book="address\_book.txt"

# Function to create address book

create\_address\_book() {

if [ ! -f "$address\_book" ]; then

touch "$address\_book"

echo "Address book created."

else

echo "Address book already exists."

fi

}

# Function to view address book

view\_address\_book() {

if [ -f "$address\_book" ]; then

cat "$address\_book"

else

echo "Address book does not exist."

fi

}

# Function to insert a record

insert\_record() {

echo "Enter Name:"

read name

echo "Enter Phone Number:"

read phone

echo "Enter Email:"

read email

echo "$name, $phone, $email" >> "$address\_book"

echo "Record added successfully."

}

# Function to delete a record

delete\_record() {

echo "Enter the name to delete:"

read name

grep -v "$name" "$address\_book" > temp.txt && mv temp.txt "$address\_book"

echo "Record deleted successfully."

}

# Function to modify a record

modify\_record() {

echo "Enter the name to modify:"

read name

echo "Enter the new phone number:"

read phone

echo "Enter the new email:"

read email

sed -i "/$name/c\\$name, $phone, $email" "$address\_book"

echo "Record modified successfully."

}

# Main menu

while true; do

echo "Address Book Menu:"

echo "1. Create Address Book"

echo "2. View Address Book"

echo "3. Insert a Record"

echo "4. Delete a Record"

echo "5. Modify a Record"

echo "6. Exit"

echo "Choose an option (1-6):"

read option

case $option in

1) create\_address\_book ;;

2) view\_address\_book ;;

3) insert\_record ;;

4) delete\_record ;;

5) modify\_record ;;

6) exit 0 ;;

\*) echo "Invalid option, please choose again." ;;

esac

done

OSPractical2a

Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX 100

// Function to perform Bubble Sort

void bubbleSort(int arr[], int n) {

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

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

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Function to perform Selection Sort

void selectionSort(int arr[], int n) {

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

int minIdx = i;

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

if (arr[j] < arr[minIdx]) {

minIdx = j;

}

}

int temp = arr[minIdx];

arr[minIdx] = arr[i];

arr[i] = temp;

}

}

// Function to print an array

void printArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n;

printf("Enter number of integers: ");

scanf("%d", &n);

int arr[MAX];

printf("Enter the integers:\n");

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

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

}

pid\_t pid = fork(); // Create a child process

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

}

if (pid > 0) {

// Parent process

printf("Parent Process: Sorting using Bubble Sort\n");

bubbleSort(arr, n); // Parent sorts using Bubble Sort

printf("Parent Process: Sorted array:\n");

printArray(arr, n);

// Wait for the child process to finish

wait(NULL);

} else {

// Child process

printf("Child Process: Sorting using Selection Sort\n");

selectionSort(arr, n); // Child sorts using Selection Sort

printf("Child Process: Sorted array:\n");

printArray(arr, n);

// Child process ends without waiting for parent (causing zombie state)

exit(0); // Exit child process

}

return 0;

}

OSPractical2b

Implement the C program in which main program accepts an array. Main program uses the FORK system

call to create a new process called a child process. Parent process sorts an array and passes the sorted array

to child process through the command line arguments of EXECVE system call. The child process uses EXECVE

system call to load new program which display array in reverse order.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

void bubble\_sort(int arr[], int n) {

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

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

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int arr[] = {23, 1, 45, 12, 67, 34, 89}; // Sample array

int n = sizeof(arr) / sizeof(arr[0]);

// Sort the array

bubble\_sort(arr, n);

// Create an array of strings for command-line arguments

char \*args[n + 2]; // +2 for the program name and NULL terminator

// Convert array elements to strings for passing to execve

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

args[i + 1] = (char \*)malloc(10 \* sizeof(char)); // Allocate memory for each number

sprintf(args[i + 1], "%d", arr[i]);

}

args[0] = "./child\_program"; // Name of the child program

args[n + 1] = NULL; // NULL terminator for execve arguments

pid\_t pid = fork(); // Create child process

if (pid == 0) {

// In child process, execute the child program

execve(args[0], args, NULL); // Pass the sorted array to the child program

perror("execve failed"); // If execve fails

} else if (pid > 0) {

// In parent process, wait for child process to complete

wait(NULL);

} else {

perror("fork failed");

return 1;

}

// Free allocated memory for arguments

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

free(args[i]);

}

return 0;

}

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

if (argc <= 1) {

printf("No arguments passed.\n");

return 1;

}

// Display the array in reverse order

printf("Array in reverse order: ");

for (int i = argc - 1; i > 0; i--) {

printf("%s ", argv[i]);

}

printf("\n");

return 0;}

OSPractical3

Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

#include <stdio.h>

#define MAX 100

// Structure to store process info

typedef struct {

int pid;

int arrival;

int burst;

int remaining;

int completion;

int waiting;

int turnaround;

} Process;

// Function for SJF Preemptive Scheduling

void SJF\_Preemptive(Process p[], int n) {

int time = 0, completed = 0, shortest, min\_remaining = 1e9;

int i;

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

printf("\n--- SJF (Preemptive) Scheduling ---\n");

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

p[i].remaining = p[i].burst;

}

while (completed != n) {

shortest = -1;

min\_remaining = 1e9;

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

if (p[i].arrival <= time && p[i].remaining > 0 && p[i].remaining < min\_remaining) {

min\_remaining = p[i].remaining;

shortest = i;

}

}

if (shortest == -1) {

time++;

continue;

}

p[shortest].remaining--;

time++;

if (p[shortest].remaining == 0) {

completed++;

p[shortest].completion = time;

p[shortest].turnaround = p[shortest].completion - p[shortest].arrival;

p[shortest].waiting = p[shortest].turnaround - p[shortest].burst;

avg\_wt += p[shortest].waiting;

avg\_tat += p[shortest].turnaround;

}

}

printf("PID\tArrival\tBurst\tCompletion\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst, p[i].completion, p[i].turnaround, p[i].waiting);

}

printf("Average Turnaround Time = %.2f\n", avg\_tat / n);

printf("Average Waiting Time = %.2f\n", avg\_wt / n);

}

// Function for Round Robin Scheduling with Arrival Time

void RoundRobin(Process p[], int n, int quantum) {

int time = 0, i, done;

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

int queue[MAX], front = 0, rear = 0, visited[MAX] = {0};

printf("\n--- Round Robin Scheduling ---\n");

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

p[i].remaining = p[i].burst;

visited[i] = 0;

}

queue[rear++] = 0;

visited[0] = 1;

while (front != rear) {

int idx = queue[front++];

if (p[idx].arrival > time) {

time = p[idx].arrival;

}

int exec\_time = (p[idx].remaining < quantum) ? p[idx].remaining : quantum;

p[idx].remaining -= exec\_time;

time += exec\_time;

// Check for newly arrived processes

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

if (!visited[i] && p[i].arrival <= time) {

queue[rear++] = i;

visited[i] = 1;

}

}

if (p[idx].remaining > 0) {

queue[rear++] = idx;

} else {

p[idx].completion = time;

p[idx].turnaround = p[idx].completion - p[idx].arrival;

p[idx].waiting = p[idx].turnaround - p[idx].burst;

avg\_wt += p[idx].waiting;

avg\_tat += p[idx].turnaround;

}

}

printf("PID\tArrival\tBurst\tCompletion\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst, p[i].completion, p[i].turnaround, p[i].waiting);

}

printf("Average Turnaround Time = %.2f\n", avg\_tat / n);

printf("Average Waiting Time = %.2f\n", avg\_wt / n);

}

// Main

int main() {

Process p[MAX];

int n, i, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

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

p[i].pid = i + 1;

printf("Enter arrival time and burst time for process %d: ", i + 1);

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

}

SJF\_Preemptive(p, n);

printf("\nEnter time quantum for Round Robin: ");

scanf("%d", &quantum);

// Resetting the data before reusing

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

p[i].completion = p[i].waiting = p[i].turnaround = 0;

p[i].remaining = p[i].burst;

}

RoundRobin(p, n, quantum);

return 0;

}

OSPractical4a

Thread synchronization using counting semaphores. Application to demonstrate: producer-consumer problem with counting semaphores and mutex.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define SIZE 5 // Buffer size

#define NUM\_ITEMS 10 // Number of items to produce/consume

int buffer[SIZE];

int in = 0, out = 0;

sem\_t empty, full;

pthread\_mutex\_t mutex;

// Producer function

void\* producer(void\* arg) {

for (int i = 0; i < NUM\_ITEMS; i++) {

int item = rand() % 100;

sem\_wait(&empty); // Decrease empty count

pthread\_mutex\_lock(&mutex); // Enter critical section

buffer[in] = item;

printf("Produced: %d at index %d\n", item, in);

in = (in + 1) % SIZE;

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Increase full count

sleep(1); // Simulate time to produce

}

return NULL;

}

// Consumer function

void\* consumer(void\* arg) {

for (int i = 0; i < NUM\_ITEMS; i++) {

sem\_wait(&full); // Decrease full count

pthread\_mutex\_lock(&mutex); // Enter critical section

int item = buffer[out];

printf("Consumed: %d from index %d\n", item, out);

out = (out + 1) % SIZE;

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&empty); // Increase empty count

sleep(2); // Simulate time to consume

}

return NULL;

}

int main() {

pthread\_t prodThread, consThread;

sem\_init(&empty, 0, SIZE);

sem\_init(&full, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

pthread\_create(&prodThread, NULL, producer, NULL);

pthread\_create(&consThread, NULL, consumer, NULL);

pthread\_join(prodThread, NULL);

pthread\_join(consThread, NULL);

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

OSPractical4b

Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader-Writer problem with reader priority.

#include <iostream>

#include <pthread.h>

#include <unistd.h> // for sleep

using namespace std;

pthread\_mutex\_t mutexReadCount;

pthread\_mutex\_t resource;

int readCount = 0;

void\* reader(void\* arg) {

int id = \*((int\*)arg);

// Entry section for readers

pthread\_mutex\_lock(&mutexReadCount);

readCount++;

if (readCount == 1) {

// First reader locks the resource so writers can't access

pthread\_mutex\_lock(&resource);

}

pthread\_mutex\_unlock(&mutexReadCount);

// Critical Section

cout << "Reader " << id << " is reading.\n";

sleep(1); // simulate reading

// Exit section for readers

pthread\_mutex\_lock(&mutexReadCount);

readCount--;

if (readCount == 0) {

// Last reader unlocks the resource for writers

pthread\_mutex\_unlock(&resource);

}

pthread\_mutex\_unlock(&mutexReadCount);

pthread\_exit(NULL);

}

void\* writer(void\* arg) {

int id = \*((int\*)arg);

// Writers directly lock the resource (mutual exclusion)

pthread\_mutex\_lock(&resource);

cout << "Writer " << id << " is writing.\n";

sleep(2); // simulate writing

pthread\_mutex\_unlock(&resource);

pthread\_exit(NULL);

}

int main() {

pthread\_t r[5], w[2];

int readerIds[5] = {1, 2, 3, 4, 5};

int writerIds[2] = {1, 2};

pthread\_mutex\_init(&mutexReadCount, NULL);

pthread\_mutex\_init(&resource, NULL);

// Create reader and writer threads

for (int i = 0; i < 2; i++) {

pthread\_create(&w[i], NULL, writer, &writerIds[i]);

}

for (int i = 0; i < 5; i++) {

pthread\_create(&r[i], NULL, reader, &readerIds[i]);

}

// Join threads

for (int i = 0; i < 2; i++) {

pthread\_join(w[i], NULL);

}

for (int i = 0; i < 5; i++) {

pthread\_join(r[i], NULL);

}

pthread\_mutex\_destroy(&mutexReadCount);

pthread\_mutex\_destroy(&resource);

return 0;

}

OSPractical5

Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int main() {

int n, m; // n = number of processes, m = number of resources

int alloc[MAX\_PROCESSES][MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

int avail[MAX\_RESOURCES];

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resources: ");

scanf("%d", &m);

printf("Enter allocation matrix:\n");

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

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

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

printf("Enter maximum matrix:\n");

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

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

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

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

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

scanf("%d", &avail[j]);

// Calculate need matrix = max - alloc

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

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

need[i][j] = max[i][j] - alloc[i][j];

bool finish[MAX\_PROCESSES] = {false};

int safeSequence[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

for (int i = 0; i < m; i++)

work[i] = avail[i];

int count = 0;

while (count < n) {

bool found = false;

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

if (!finish[i]) {

bool canAllocate = true;

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

if (need[i][j] > work[j]) {

canAllocate = false;

break;

}

}

if (canAllocate) {

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

work[j] += alloc[i][j];

safeSequence[count++] = i;

finish[i] = true;

found = true;

}

}

}

if (!found) {

printf("System is not in a safe state.\n");

return 0;

}

}

printf("System is in a safe state.\nSafe sequence is: ");

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

printf("P%d ", safeSequence[i]);

printf("\n");

return 0;

}

OSPractical6

Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

// Function to check if page is in frames

int isInFrames(int frames[], int frameSize, int page) {

for (int i = 0; i < frameSize; i++) {

if (frames[i] == page) return 1;

}

return 0;

}

// Function to find index of the least recently used page

int findLRU(int time[], int frameSize) {

int min = time[0], pos = 0;

for (int i = 1; i < frameSize; i++) {

if (time[i] < min) {

min = time[i];

pos = i;

}

}

return pos;

}

// Function to find index of the page that will not be used for the longest time

int findOptimal(int pages[], int frames[], int frameSize, int n, int index) {

int pos = -1, farthest = index;

for (int i = 0; i < frameSize; i++) {

int j;

for (j = index; j < n; j++) {

if (frames[i] == pages[j]) {

if (j > farthest) {

farthest = j;

pos = i;

}

break;

}

}

if (j == n) return i;

}

return (pos == -1) ? 0 : pos;

}

void FCFS(int pages[], int n, int frameSize) {

int frames[frameSize], index = 0, pageFaults = 0;

for (int i = 0; i < frameSize; i++) frames[i] = -1;

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

if (!isInFrames(frames, frameSize, pages[i])) {

frames[index] = pages[i];

index = (index + 1) % frameSize;

pageFaults++;

}

}

printf("\nFCFS - Page Faults: %d\n", pageFaults);

}

void LRU(int pages[], int n, int frameSize) {

int frames[frameSize], time[frameSize];

int counter = 0, pageFaults = 0;

for (int i = 0; i < frameSize; i++) {

frames[i] = -1;

time[i] = 0;

}

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

if (isInFrames(frames, frameSize, pages[i])) {

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

if (frames[j] == pages[i])

time[j] = ++counter;

}

} else {

int pos = -1;

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

if (frames[j] == -1) {

pos = j;

break;

}

}

if (pos == -1) pos = findLRU(time, frameSize);

frames[pos] = pages[i];

time[pos] = ++counter;

pageFaults++;

}

}

printf("LRU - Page Faults: %d\n", pageFaults);

}

void Optimal(int pages[], int n, int frameSize) {

int frames[frameSize], pageFaults = 0;

for (int i = 0; i < frameSize; i++) frames[i] = -1;

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

if (!isInFrames(frames, frameSize, pages[i])) {

int pos = -1;

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

if (frames[j] == -1) {

pos = j;

break;

}

}

if (pos == -1) pos = findOptimal(pages, frames, frameSize, n, i + 1);

frames[pos] = pages[i];

pageFaults++;

}

}

printf("Optimal - Page Faults: %d\n", pageFaults);

}

int main() {

int n, frameSize, pages[MAX];

printf("Enter number of pages: ");

scanf("%d", &n);

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

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

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

}

printf("Enter frame size (min 3): ");

scanf("%d", &frameSize);

if (frameSize < 3) {

printf("Frame size must be at least 3!\n");

return 1;

}

FCFS(pages, n, frameSize);

LRU(pages, n, frameSize);

Optimal(pages, n, frameSize);

return 0;

}

OSPractical7a

FIFOs: Full duplex communication between two independent processes. First process accepts sentences

and writes on one pipe to be read by second process and second process counts number of characters,

number of words and number of lines in accepted sentences, writes this output in a text file and writes the

contents of the file on second pipe to be read by first process and displays on standard output.

// process1.c

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define FIFO1 "fifo1"

#define FIFO2 "fifo2"

int main() {

char sentence[1024], result[1024];

// Create FIFOs if they don't exist

mkfifo(FIFO1, 0666);

mkfifo(FIFO2, 0666);

// Get input from user

printf("Enter a sentence: ");

fgets(sentence, sizeof(sentence), stdin);

// Write sentence to FIFO1

int fd1 = open(FIFO1, O\_WRONLY);

write(fd1, sentence, strlen(sentence) + 1);

close(fd1);

// Read result from FIFO2

int fd2 = open(FIFO2, O\_RDONLY);

read(fd2, result, sizeof(result));

close(fd2);

printf("\nReceived from Process 2:\n%s", result);

return 0;

}

// process2.c

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define FIFO1 "fifo1"

#define FIFO2 "fifo2"

#define FILENAME "output.txt"

int main() {

char buffer[1024];

int chars = 0, words = 0, lines = 0;

// Open FIFO1 to read

int fd1 = open(FIFO1, O\_RDONLY);

read(fd1, buffer, sizeof(buffer));

close(fd1);

// Analyze the content

for (int i = 0; buffer[i] != '\0'; i++) {

chars++;

if (buffer[i] == ' ' || buffer[i] == '\t' || buffer[i] == '\n') {

words++;

}

if (buffer[i] == '\n') {

lines++;

}

}

// If there's at least one word

if (chars > 0)

words++;

if (strchr(buffer, '\n') == NULL)

lines++;

// Write output to a file

FILE \*fp = fopen(FILENAME, "w");

fprintf(fp, "Characters: %d\nWords: %d\nLines: %d\n", chars, words, lines);

fclose(fp);

// Read from file and write to FIFO2

char output[1024];

fp = fopen(FILENAME, "r");

fread(output, sizeof(char), sizeof(output), fp);

fclose(fp);

int fd2 = open(FIFO2, O\_WRONLY);

write(fd2, output, strlen(output) + 1);

close(fd2);

return 0;

}

OSPractical7b

b. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024 // Size of shared memory

int main() {

key\_t key = ftok("shmfile", 65); // Generate unique key

int shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT); // Create shared memory

if (shmid == -1) {

perror("shmget failed");

exit(1);

}

char \*str = (char \*) shmat(shmid, (void \*)0, 0); // Attach to shared memory

if (str == (char \*)(-1)) {

perror("shmat failed");

exit(1);

}

printf("Enter a message to write to shared memory: ");

fgets(str, SHM\_SIZE, stdin); // Write input to shared memory

printf("Message written to shared memory.\n");

shmdt(str); // Detach from shared memory

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024 // Size of shared memory

int main() {

key\_t key = ftok("shmfile", 65); // Generate same key

int shmid = shmget(key, SHM\_SIZE, 0666); // Locate shared memory

if (shmid == -1) {

perror("shmget failed");

exit(1);

}

char \*str = (char \*) shmat(shmid, (void \*)0, 0); // Attach to shared memory

if (str == (char \*)(-1)) {

perror("shmat failed");

exit(1);

}

printf("Message read from shared memory: %s\n", str); // Read message

shmdt(str); // Detach from shared memory

shmctl(shmid, IPC\_RMID, NULL); // Mark for deletion

return 0;

}

OSPractical8

Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAX 100

void sort(int arr[], int n) {

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

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

if(arr[j] > arr[j+1]) {

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

// SSTF Algorithm

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

int total\_seek = 0, completed = 0;

int visited[MAX] = {0};

int current = head;

printf("\nSSTF Disk Scheduling:\nOrder: ");

while(completed < n) {

int min = 1e9, index = -1;

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

if(!visited[i] && abs(current - requests[i]) < min) {

min = abs(current - requests[i]);

index = i;

}

}

visited[index] = 1;

total\_seek += abs(current - requests[index]);

current = requests[index];

printf("%d ", current);

completed++;

}

printf("\nTotal Seek Time: %d\n", total\_seek);

}

// SCAN Algorithm

void SCAN(int requests[], int n, int head, int disk\_size) {

int total\_seek = 0;

int all[MAX];

int size = n + 1;

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

all[i] = requests[i];

all[n] = head;

sort(all, size);

int pos;

for(pos = 0; pos < size; pos++)

if(all[pos] == head)

break;

printf("\nSCAN Disk Scheduling:\nOrder: ");

// Moving towards higher track numbers

for(int i = pos; i < size; i++) {

printf("%d ", all[i]);

if(i != pos)

total\_seek += abs(all[i] - all[i - 1]);

}

// Reversing direction to 0 (start of disk)

total\_seek += abs(all[size - 1] - 0);

printf("0 "); // assuming spindle is at 0

for(int i = pos - 1; i >= 0; i--) {

printf("%d ", all[i]);

if(i != pos - 1)

total\_seek += abs(all[i + 1] - all[i]);

}

printf("\nTotal Seek Time: %d\n", total\_seek);

}

// C-LOOK Algorithm

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

int total\_seek = 0;

int all[MAX];

int size = n + 1;

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

all[i] = requests[i];

all[n] = head;

sort(all, size);

int pos;

for(pos = 0; pos < size; pos++)

if(all[pos] == head)

break;

printf("\nC-LOOK Disk Scheduling:\nOrder: ");

// Move toward higher tracks

for(int i = pos; i < size; i++) {

printf("%d ", all[i]);

if(i != pos)

total\_seek += abs(all[i] - all[i - 1]);

}

// Jump to the lowest request

total\_seek += abs(all[size - 1] - all[0]);

printf("%d ", all[0]);

for(int i = 1; i < pos; i++) {

printf("%d ", all[i]);

total\_seek += abs(all[i] - all[i - 1]);

}

printf("\nTotal Seek Time: %d\n", total\_seek);

}

int main() {

int n, head, disk\_size;

int requests[MAX];

printf("Enter number of disk requests: ");

scanf("%d", &n);

printf("Enter disk requests:\n");

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

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

printf("Enter initial head position: ");

scanf("%d", &head);

printf("Enter total disk size (e.g., 200): ");

scanf("%d", &disk\_size);

SSTF(requests, n, head);

SCAN(requests, n, head, disk\_size);

CLOOK(requests, n, head);

return 0;

}