[1]//fork

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid;

printf("Before fork()\n");

pid = fork();

if (pid == 0) {

printf("Child process: PID = %d\n", getpid());

} else if (pid > 0) {

printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);

} else {

printf("Fork failed\n");

return 1;

}

printf("After fork()\n");

return 0;

}

[2]//sum odd num

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define ARRAY\_SIZE 10

int main() {

int arr[ARRAY\_SIZE];

int sum\_even = 0, sum\_odd = 0;

printf("Enter %d elements for the array:\n", ARRAY\_SIZE);

for (int i = 0; i < ARRAY\_SIZE; i++) {

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

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

}

pid\_t pid = fork();

if (pid == 0) {

for (int i = 0; i < ARRAY\_SIZE; i++) {

if (arr[i] % 2 == 0)

sum\_even += arr[i];

}

printf("Child process: Sum of even numbers = %d\n", sum\_even);

} else if (pid > 0) {

for (int i = 0; i < ARRAY\_SIZE; i++) {

if (arr[i] % 2 != 0)

sum\_odd += arr[i];

}

printf("Parent process: Sum of odd numbers = %d\n", sum\_odd);

} else {

printf("Fork failed\n");

return 1;

}

return 0;}

[3]//zombie

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

printf("Child process created, PID = %d\n", getpid());

exit(EXIT\_SUCCESS);

} else {

printf("Parent process, PID = %d\n", getpid());

sleep(10);

} return 0;}

[4]//orphan

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

printf("Child process created, PID = %d\n", getpid());

sleep(10);

printf("Child process exiting\n");

} else {

printf("Parent process, PID = %d\n", getpid());

printf("Parent process is terminating\n");

} return 0;}

[5]//FCFS #include <stdio.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

};

void calculateWaitingTime(struct Process processes[], int n, int waiting\_time[]) {

waiting\_time[0] = 0;

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

waiting\_time[i] = waiting\_time[i - 1] + processes[i - 1].burst\_time;

}

}

void calculateTurnaroundTime(struct Process processes[], int n, int waiting\_time[], int turnaround\_time[]) {

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

turnaround\_time[i] = processes[i].burst\_time + waiting\_time[i];

}

}

void calculateAverageTimes(struct Process processes[], int n) {

int waiting\_time[n], turnaround\_time[n];

calculateWaitingTime(processes, n, waiting\_time);

calculateTurnaroundTime(processes, n, waiting\_time, turnaround\_time);

float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

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

avg\_waiting\_time += waiting\_time[i];

avg\_turnaround\_time += turnaround\_time[i];

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

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

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

printf("%d\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burst\_time, waiting\_time[i], turnaround\_time[i]);

}

printf("Average Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

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

processes[i].pid = i + 1;

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

scanf("%d", &processes[i].arrival\_time);

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

scanf("%d", &processes[i].burst\_time);

}

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

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

if (processes[j].arrival\_time > processes[j + 1].arrival\_time) {

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp; }} }

calculateAverageTimes(processes, n);

return 0;

}

[6]//SRTF

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define MAX\_PROCESSES 10

typedef struct {

int pid; // Process ID

int burst\_time; // Burst time

int remaining\_time; // Remaining burst time

} Process;

void srtf\_schedule(Process processes[], int n) {

int current\_time = 0;

int completed = 0;

int shortest = INT\_MAX;

int shortest\_index = -1;

printf("SRTF Schedule:\n");

printf("Time\tProcess\n");

while (completed < n) {

shortest = INT\_MAX;

shortest\_index = -1;

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

if (processes[i].remaining\_time > 0 && processes[i].remaining\_time < shortest && processes[i].burst\_time > 0 && processes[i].pid != -1) {

shortest = processes[i].remaining\_time;

shortest\_index = i; }}

if (shortest\_index == -1) {

current\_time++;

continue;

}

processes[shortest\_index].remaining\_time--;

current\_time++;

if (processes[shortest\_index].remaining\_time == 0) {

printf("%d-%d\tP%d\n", current\_time - 1, current\_time, processes[shortest\_index].pid);

completed++;

processes[shortest\_index].pid = -1; } }}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

if (n > MAX\_PROCESSES || n <= 0) {

printf("Invalid number of processes. Maximum allowed: %d\n", MAX\_PROCESSES);

return 1;

}

Process processes[MAX\_PROCESSES];

printf("Enter burst times for each process:\n");

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

processes[i].pid = i + 1;

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

scanf("%d", &processes[i].burst\_time);

processes[i].remaining\_time = processes[i].burst\_time;}

srtf\_schedule(processes, n);

return 0;

}

[7]//non prem SJFS

#include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct {

int process\_id;

int burst\_time;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sjf\_nonpreemptive(Process processes[], int n) {

int waiting\_time[MAX\_PROCESSES], turnaround\_time[MAX\_PROCESSES];

float average\_waiting\_time = 0, average\_turnaround\_time = 0;

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

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

if (processes[j].burst\_time > processes[j + 1].burst\_time) {

swap(&processes[j], &processes[j + 1]);

}

}

}

waiting\_time[0] = 0;

turnaround\_time[0] = processes[0].burst\_time;

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

waiting\_time[i] = waiting\_time[i - 1] + processes[i - 1].burst\_time;

turnaround\_time[i] = waiting\_time[i] + processes[i].burst\_time;

average\_waiting\_time += waiting\_time[i];

average\_turnaround\_time += turnaround\_time[i];

}

average\_waiting\_time /= n;

average\_turnaround\_time /= n;

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

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

printf("%d\t%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].burst\_time,

waiting\_time[i], turnaround\_time[i]);

}

printf("\nAverage Waiting Time: %.2f\n", average\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", average\_turnaround\_time);

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter burst time for each process:\n");

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

processes[i].process\_id = i + 1;

printf("Burst time for process %d: ", i + 1);

scanf("%d", &processes[i].burst\_time);

}

sjf\_nonpreemptive(processes, n);

return 0;

}

[8]//non prem prior

#include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct {

int process\_id;

int priority;

int burst\_time;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void priority\_nonpreemptive(Process processes[], int n) {

int waiting\_time[MAX\_PROCESSES], turnaround\_time[MAX\_PROCESSES];

float average\_waiting\_time = 0, average\_turnaround\_time = 0;

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

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

if (processes[j].priority < processes[j + 1].priority) {

swap(&processes[j], &processes[j + 1]);

}

}

}

waiting\_time[0] = 0;

turnaround\_time[0] = processes[0].burst\_time;

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

waiting\_time[i] = waiting\_time[i - 1] + processes[i - 1].burst\_time;

turnaround\_time[i] = waiting\_time[i] + processes[i].burst\_time;

average\_waiting\_time += waiting\_time[i];

average\_turnaround\_time += turnaround\_time[i];

}

average\_waiting\_time /= n;

average\_turnaround\_time /= n;

// Display the schedule and timing information

printf("Process\tPriority\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].priority,

processes[i].burst\_time, waiting\_time[i], turnaround\_time[i]);

}

printf("\nAverage Waiting Time: %.2f\n", average\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", average\_turnaround\_time);

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter priority and burst time for each process:\n");

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

processes[i].process\_id = i + 1;

printf("Priority for process %d: ", i + 1);

scanf("%d", &processes[i].priority);

printf("Burst time for process %d: ", i + 1);

scanf("%d", &processes[i].burst\_time);

}

priority\_nonpreemptive(processes, n);

return 0;

}

[9]

#include <stdio.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int available[MAX\_RESOURCES];

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

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

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

int isSafe(int processes, int resources) {

int work[MAX\_RESOURCES];

int finish[MAX\_PROCESSES] = {0};

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

work[i] = available[i];

}

int count = 0;

while (count < processes) {

int found = 0;

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

if (!finish[i]) {

int j;

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

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

break;

}

}

if (j == resources) {

for (int k = 0; k < resources; k++) {

work[k] += allocated[i][k];

}

finish[i] = 1;

found = 1;

count++;

}

}

}

if (!found) {

return 0;

}

}

return 1;

}

void requestResources(int processes, int resources, int process, int request[]) {

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

if (request[i] > need[process][i] || request[i] > available[i]) {

printf("Invalid request. The request exceeds the maximum need or available

resources.\n");

return;

}

}

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

allocated[process][i] += request[i];

available[i] -= request[i];

need[process][i] -= request[i];

}

if (isSafe(processes, resources)) {

printf("Request granted. The system is in a safe state.\n");

} else {

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

allocated[process][i] -= request[i];

available[i] += request[i];

need[process][i] += request[i];

}

printf("Request denied. The system would be in an unsafe state.\n");

}

}

void releaseResources(int resources, int process, int release[]) {

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

if (release[i] > allocated[process][i]) {

printf("Invalid release. The release exceeds the allocated resources.\n");

return;

}

}

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

allocated[process][i] -= release[i];

available[i] += release[i];

}

printf("Resources released. The system is in a safe state.\n");

}

int main() {

int processes, resources;

printf("Enter the number of processes: ");

scanf("%d", &processes);

printf("Enter the number of resources: ");

scanf("%d", &resources);

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

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

printf("Process %d: ", i);

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

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

}

}

printf("Enter the allocated resources matrix:\n");

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

printf("Process %d: ", i);

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

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

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

}

}

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

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

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

}

if (isSafe(processes, resources)) {

printf("The initial state is safe.\n");

} else {

printf("The initial state is unsafe.\n");

return 1;

}

// Demonstrate resource request and release

int process, request[MAX\_RESOURCES], release[MAX\_RESOURCES];

printf("Enter the process number requesting resources: ");

scanf("%d", &process);

printf("Enter the resource request (e.g., R1 R2 ...): ");

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

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

}

requestResources(processes, resources, process, request);

printf("Enter the process number releasing resources: ");

scanf("%d", &process);

printf("Enter the resource release (e.g., R1 R2 ...): ");

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

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

}

releaseResources(resources, process, release);

return 0;}

[10] #include <stdio.h>

#define MAX\_FRAMES 3

void initializeFrames(int frames[MAX\_FRAMES]) {

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1; // -1 indicates an empty frame

}

}

void printFrames(int frames[MAX\_FRAMES]) {

printf("Frames: ");

for (int i = 0; i < MAX\_FRAMES; i++) {

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

printf("[ ] ");

} else {

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

}

}

printf("\n");

}

int isPageInFrames(int frames[MAX\_FRAMES], int page) {

for (int i = 0; i < MAX\_FRAMES; i++) {

if (frames[i] == page) {

return 1; // Page is already in frames

}

}

return 0; // Page is not in frames

}

void fifoPageReplacement(int frames[MAX\_FRAMES], int page, int \*nextFrameIndex) {

frames[\*nextFrameIndex] = page;

\*nextFrameIndex = (\*nextFrameIndex + 1) % MAX\_FRAMES;}

int main() {

int frames[MAX\_FRAMES];

initializeFrames(frames);

int pageSequence[] = {0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4};

int pageSequenceSize = sizeof(pageSequence) / sizeof(pageSequence[0]);

int pageFaults = 0;

int nextFrameIndex = 0;

printf("Page Replacement using FIFO:\n");

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

int currentPage = pageSequence[i];

if (!isPageInFrames(frames, currentPage)) {

printf("Page %d caused a page fault. ", currentPage);

fifoPageReplacement(frames, currentPage, &nextFrameIndex);

pageFaults++;

} else {

printf("Page %d is already in memory. ", currentPage);}

printFrames(frames);}

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

return 0;

}

[11] #include <stdio.h>

#define MAX\_FRAMES 3

void initializeFrames(int frames[MAX\_FRAMES]) {

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1; // -1 indicates an empty frame

}

}

void printFrames(int frames[MAX\_FRAMES]) {

printf("Frames: ");

for (int i = 0; i < MAX\_FRAMES; i++) {

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

printf("[ ] ");

} else {

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

}

}

printf("\n");

}

int isPageInFrames(int frames[MAX\_FRAMES], int page) {

for (int i = 0; i < MAX\_FRAMES; i++) {

if (frames[i] == page) {

return 1; // Page is already in frames

}

}

return 0; // Page is not in frames

}

int getLRUPage(int pageOrder[MAX\_FRAMES]) {

return pageOrder[MAX\_FRAMES - 1];

}

void updatePageOrder(int pageOrder[MAX\_FRAMES], int currentPage) {

// Move the current page to the front of the page order

for (int i = 0; i < MAX\_FRAMES; i++) {

if (pageOrder[i] == currentPage) {

for (int j = i; j > 0; j--) {

pageOrder[j] = pageOrder[j - 1];

}

pageOrder[0] = currentPage;

break;

}

}

}

void lruPageReplacement(int frames[MAX\_FRAMES], int pageOrder[MAX\_FRAMES], int

page) {

int leastRecentlyUsedPage = getLRUPage(pageOrder);

for (int i = 0; i < MAX\_FRAMES; i++) {

if (frames[i] == leastRecentlyUsedPage) {

frames[i] = page;

break;

}

}

updatePageOrder(pageOrder, page);

}

int main() {

int frames[MAX\_FRAMES];

initializeFrames(frames);

int pageSequence[] = {0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4};

int pageSequenceSize = sizeof(pageSequence) / sizeof(pageSequence[0]);

int pageOrder[MAX\_FRAMES];

for (int i = 0; i < MAX\_FRAMES; i++) {

pageOrder[i] = -1; // Initialize page order

}

int pageFaults = 0;

printf("Page Replacement using LRU:\n");

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

int currentPage = pageSequence[i];

if (!isPageInFrames(frames, currentPage)) {

printf("Page %d caused a page fault. ", currentPage);

if (pageFaults < MAX\_FRAMES) {

frames[pageFaults] = currentPage;

pageOrder[pageFaults] = currentPage;

} else {

lruPageReplacement(frames, pageOrder, currentPage);

}

pageFaults++;

} else {

printf("Page %d is already in memory. ", currentPage);

updatePageOrder(pageOrder, currentPage);

}

printFrames(frames);

}

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

return 0;

}

[12] #include <stdio.h>

#include <stdlib.h>

void calculateSeekTime(int requestSequence[], int numRequests, int initialHeadPosition) {

int seekTime = 0;

int currentHeadPosition = initialHeadPosition;

printf("Seek Sequence: %d", currentHeadPosition);

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

int distance = abs(requestSequence[i] - currentHeadPosition);

seekTime += distance;

currentHeadPosition = requestSequence[i];

printf(" -> %d", currentHeadPosition);

}

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

}

int main() {

int numRequests;

int initialHeadPosition;

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

scanf("%d", &numRequests);

int \*requestSequence = (int \*)malloc(numRequests \* sizeof(int));

if (requestSequence == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

return 1; // Exit with an error code

}

printf("Enter the request sequence:\n");

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

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

}

printf("Enter the initial head position: ");

scanf("%d", &initialHeadPosition);

calculateSeekTime(requestSequence, numRequests, initialHeadPosition);

free(requestSequence);

return 0;

}

[13] #include <stdio.h>

#include <stdlib.h>

#include <limits.h> // Include the header file for INT\_MAX

// Function to calculate seek time using SSTF algorithm

void calculateSeekTime(int requestSequence[], int numRequests, int initialHeadPosition) {

int seekTime = 0;

int currentHeadPosition = initialHeadPosition;

int visited[numRequests];

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

visited[i] = 0; // Initialize all requests as not visited

}

printf("Seek Sequence: %d", currentHeadPosition);

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

int minDistance = INT\_MAX; // Use INT\_MAX from <limits.h>

int nextRequest = -1;

// Find the request with the shortest seek time

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

if (!visited[j]) {

int distance = abs(requestSequence[j] - currentHeadPosition);

if (distance < minDistance) {

minDistance = distance;

nextRequest = j;

}

}

}

visited[nextRequest] = 1; // Mark the request as visited

seekTime += minDistance;

currentHeadPosition = requestSequence[nextRequest];

printf(" -> %d", currentHeadPosition);

}

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

}

int main() {

int numRequests;

int initialHeadPosition;

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

scanf("%d", &numRequests);

int \*requestSequence = (int \*)malloc(numRequests \* sizeof(int));

if (requestSequence == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

return 1; // Exit with an error code

}

printf("Enter the request sequence:\n");

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

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

}

printf("Enter the initial head position: ");

scanf("%d", &initialHeadPosition);

calculateSeekTime(requestSequence, numRequests, initialHeadPosition);

free(requestSequence);

return 0;}

[14] #include <stdio.h>

#include <stdlib.h>

void calculateSeekTime(int requestSequence[], int numRequests, int initialHeadPosition, int

direction) {

int seekTime = 0;

int currentHeadPosition = initialHeadPosition;

printf("Seek Sequence: %d", currentHeadPosition);

if (direction == 1) { // Move towards higher cylinder numbers

// Go to the end of the disk

for (int i = currentHeadPosition; i <= 199; i++) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

// Go to the beginning of the disk

for (int i = 199; i >= 0; i--) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

} else { // Move towards lower cylinder numbers

// Go to the beginning of the disk

for (int i = currentHeadPosition; i >= 0; i--) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

// Go to the end of the disk

for (int i = 0; i <= 199; i++) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

}

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

}

int main() {

int numRequests;

int initialHeadPosition;

int direction;

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

scanf("%d", &numRequests);

int \*requestSequence = (int \*)malloc(numRequests \* sizeof(int));

if (requestSequence == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

return 1; // Exit with an error code

}

printf("Enter the request sequence:\n");

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

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

}

printf("Enter the initial head position: ");

scanf("%d", &initialHeadPosition);

printf("Enter the direction (1 for towards higher cylinders, 0 for towards lower cylinders):

");

scanf("%d", &direction);

calculateSeekTime(requestSequence, numRequests, initialHeadPosition, direction);

free(requestSequence);

return 0;

}

[15] #include <stdio.h>

#include <stdlib.h>

void calculateSeekTime(int requestSequence[], int numRequests, int initialHeadPosition, int

direction) {

int seekTime = 0;

int currentHeadPosition = initialHeadPosition;

printf("Seek Sequence: %d", currentHeadPosition);

if (direction == 1) { // Move towards higher cylinder numbers

// Go to the end of the disk

for (int i = currentHeadPosition; i <= 199; i++) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

// Go to the beginning of the disk

for (int i = 199; i >= 0; i--) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

} else { // Move towards lower cylinder numbers

// Go to the beginning of the disk

for (int i = currentHeadPosition; i >= 0; i--) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

// Go to the end of the disk

for (int i = 0; i <= 199; i++) {

printf(" -> %d", i);

seekTime += abs(i - currentHeadPosition);

currentHeadPosition = i;

}

}

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

}

int main() {

int numRequests;

int initialHeadPosition;

int direction;

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

scanf("%d", &numRequests);

int \*requestSequence = (int \*)malloc(numRequests \* sizeof(int));

if (requestSequence == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

return 1; // Exit with an error code

}

printf("Enter the request sequence:\n");

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

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

}

printf("Enter the initial head position: ");

scanf("%d", &initialHeadPosition);

printf("Enter the direction (1 for towards higher cylinders, 0 for towards lower cylinders):

");

scanf("%d", &direction);

calculateSeekTime(requestSequence, numRequests, initialHeadPosition, direction);

free(requestSequence);

return 0;

}