

**Lab no: 1**

**Title: Basic Linux Commands**

**Theory:**

Linux is a Unix-Like operating system. All the Linux/Unix commands are run in the terminal provided by the Linux system. This terminal is just like command prompt of Windows OS. Linux/Unix commands are case-sensitive. Linux is a popular open-source operating system that offers a powerful command-line interface (CLI) for interacting with the system. Understanding basic Linux commands is essential for efficient system administration and software development tasks

1. Create a directory named linux\_practical



1. Move into the newly created directory



1. Print the current working directory to verify you’re in the right place



4. Create some sample files



5. List the files in the directory



6. Display the contents of file1.txt



7. Copy file1.txt to file3.txt



8. Rename file3.txt to newfile.txt



9. Remove file2.txt



10. Move out of the linxu\_practical directory





**Lab no: 2**

**Title: Write a program to implement First Come First Service (FCFS) Page Replacement Algorithm.**



**Theory:**

**First Come First Service (FCFS) Page Replacement Algorithm:**

FCFS is one of the simplest page replacement algorithms used in operating systems. In FCFS, the page that has been in memory the longest is the one replaced first when a page fault occurs. This algorithm works on the principle of "first in, first out," similar to a queue

Advantages:

* Simple to understand and implement.
* Requires minimal resources due to its straightforward logic and low computational complexity.

Disadvantages:

* May lead to Belady's anomaly, where increasing the number of page frames can unexpectedly increase the number of page faults.
* Typically has a lower page hit rate compared to more sophisticated algorithms.

**Algorithm:**

1. Initialize an array called frames to represent the page frames, initially empty.
2. Iterate over each page request sequentially.
3. For each page request: a. Check if the page is already present in any of the frames. b. If the page is not found in any frame, it's a page fault.
   * Increment the page fault counter.
   * If the frames are full, replace the oldest page in frames with the current page.
   * If there is space available in frames, add the current page to an empty frame.
4. Continue this process until all page requests are processed.
5. Return the total number of page faults encountered during the simulation.

**Source Code:**

#include <stdio.h>

// Function to implement First Come First Serve (FCFS) page replacement algorithm

int fcfs(int page\_requests[], int num\_requests, int frame\_size) {

int page\_faults = 0;

int frames[frame\_size];

int i, j, k;

// Initializing frames with -1 indicating an empty frame

for (i = 0; i < frame\_size; i++)

frames[i] = -1;

j = 0;

// Iterating over each page request

for (i = 0; i < num\_requests; i++) {

int page = page\_requests[i];

int found = 0;

// Checking if the page is already present in any of the frames

for (k = 0; k < frame\_size; k++) {

if (frames[k] == page) {

found = 1;

break;

}

}

// If the page is not found in any frame, it's a page fault

if (!found) {

page\_faults++;

frames[j] = page; // Replace the page at the current frame position

j = (j + 1) % frame\_size; // Increment the frame position using circular fashion

}

}

return page\_faults; // Return the total number of page faults

}

int main() {

int frame\_size, num\_requests;

// Asking the user to input the frame size

printf("Enter the frame size: ");

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

// Asking the user to input the number of page requests

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

scanf("%d", &num\_requests);

int page\_requests[num\_requests];

// Asking the user to input each page request individually

printf("Enter the page requests:\n");

for (int i = 0; i < num\_requests; i++) {

scanf("%d", &page\_requests[i]);

}

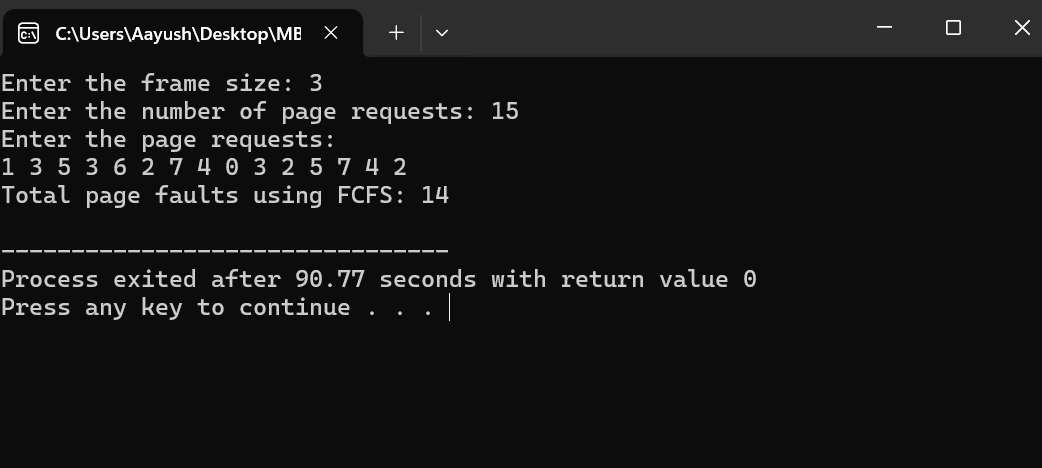
// Calling the FCFS function and printing the total number of page faults

printf("Total page faults using FCFS: %d\n", fcfs(page\_requests, num\_requests, frame\_size));

return 0;

}

**Output:**





**Lab no: 3**

**Title: Write a program to simulate LRU page replacement algorithm for user input no of reference string and no of page frame.**

**Theory:**

**Least Recently Used (LRU) Page Replacement Algorithm:**

LRU is a page replacement algorithm that replaces the least recently used page when a page fault occurs. The idea behind LRU is to track the usage of pages and replace the page that has not been accessed for the longest period of time. LRU assumes that pages that have been accessed recently are more likely to be accessed again in the near future.

Advantages:

* Generally performs well in practice, especially for programs with good temporal locality.
* Tends to have a higher hit rate compared to simpler algorithms like FIFO (First-In-First-Out).

Disadvantages:

* Requires additional bookkeeping to keep track of the usage history of pages.
* Implementation can be complex and resource-intensive, especially for large memory sizes.

**Algorithm**:

1. Initialize an array called frames to represent the page frames, initially empty.
2. Initialize another array called last\_used to keep track of the time each page in frames was last used. Initialize all values to a large number (e.g., infinity) to indicate that pages have not been used yet.
3. Iterate over each page request sequentially.
4. For each page request: a. Check if the page is already present in any of the frames. b. If the page is found in a frame:
   * Update the last\_used array for the corresponding frame to the current time. c. If the page is not found in any frame (page fault):
   * Increment the page fault counter.
   * Find the least recently used page in frames (i.e., the page with the smallest value in the last\_used array).
   * Replace the least recently used page with the current page.
   * Update the last\_used array for the replaced page with the current time.
5. Continue this process until all page requests are processed.
6. Return the total number of page faults encountered during the simulation.

**Source Code:**

#include <stdio.h>

#include <limits.h>

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

int findLRUIndex(int frames[], int num\_frames, int last\_used[]) {

int min\_index = 0;

int min\_value = INT\_MAX;

for (int i = 0; i < num\_frames; i++) {

if (last\_used[i] < min\_value) {

min\_value = last\_used[i];

min\_index = i;

}

}

return min\_index;

}

// Function to simulate the LRU (Least Recently Used) page replacement algorithm

int lru(int page\_requests[], int num\_requests, int frame\_size) {

int page\_faults = 0;

int frames[frame\_size];

int last\_used[frame\_size];

// Initialize frames with -1 indicating an empty frame

for (int i = 0; i < frame\_size; i++) {

frames[i] = -1;

last\_used[i] = INT\_MAX; // Initialize last used time to infinity

}

// Iterate over each page request

for (int i = 0; i < num\_requests; i++) {

int page = page\_requests[i];

int found = 0;

// Check if the page is already present in any of the frames

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == page) {

found = 1;

last\_used[j] = i; // Update the last used time for the page

break;

}

}

// If the page is not found in any frame, it's a page fault

if (!found) {

page\_faults++;

int lru\_index = findLRUIndex(frames, frame\_size, last\_used);

frames[lru\_index] = page; // Replace the least recently used page

last\_used[lru\_index] = i; // Update the last used time for the page

}

}

return page\_faults; // Return the total number of page faults

}

int main() {

int frame\_size, num\_requests;

// Asking the user to input the frame size

printf("Enter the frame size: ");

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

// Asking the user to input the number of page requests

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

scanf("%d", &num\_requests);

int page\_requests[num\_requests];

// Asking the user to input each page request individually

printf("Enter the page requests:\n");

for (int i = 0; i < num\_requests; i++) {

scanf("%d", &page\_requests[i]);

}

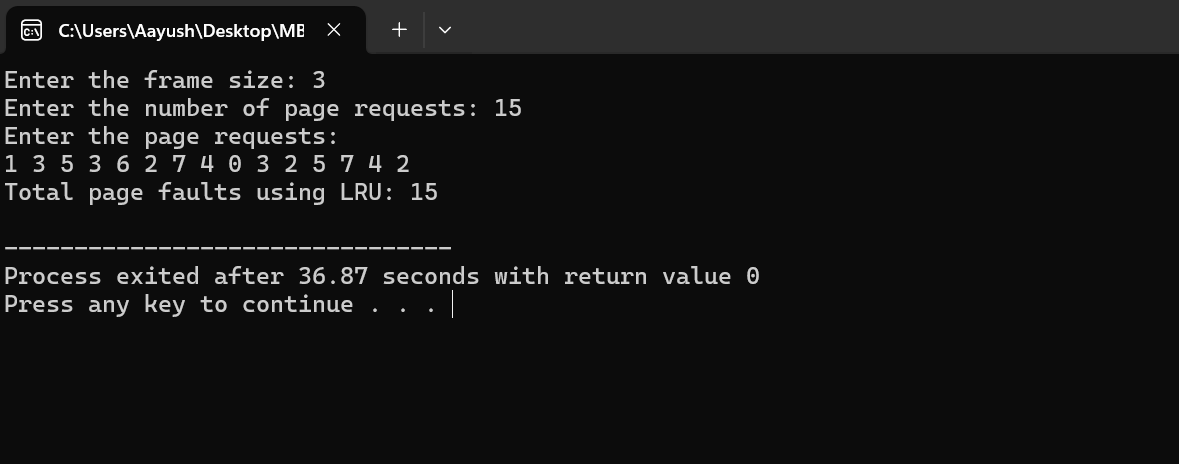
// Calling the LRU function and printing the total number of page faults

printf("Total page faults using LRU: %d\n", lru(page\_requests, num\_requests, frame\_size));

return 0;

}

**Output:**





**Lab no: 4**

**Title: Write a program to simulate Optimal page replacement algorithm for user input no of reference string and no of page frame.**

**Theory:**

**Optimal Page Replacement Algorithm:**

The Optimal algorithm, also known as the OPT algorithm or MIN algorithm, is an optimal page replacement algorithm. It replaces the page that will not be used for the longest time in the future. The idea behind the Optimal algorithm is to predict which page will be accessed furthest into the future and replace it when a page fault occurs.

Advantages:

* The Optimal algorithm provides the lowest possible page fault rate among all page replacement algorithms.
* It serves as a benchmark for comparing the performance of other algorithms.

Disadvantages:

* It is not practical for implementation in real operating systems due to its requirement of future knowledge about page accesses.
* The implementation of this algorithm requires scanning the future page requests, which may not be feasible in many scenarios.

Algorithm:

1. Initialize an array called frames to represent the page frames, initially empty.
2. Iterate over each page request sequentially.
3. For each page request: a. Check if the page is already present in any of the frames. b. If the page is found in a frame, no action is needed. c. If the page is not found in any frame (page fault):
   * Increment the page fault counter.
   * Find the page in frames that will not be used for the longest time in the future (i.e., the optimal page for replacement).
   * Replace the optimal page with the current page.
4. Continue this process until all page requests are processed.
5. Return the total number of page faults encountered during the simulation.

**Source Code:**

#include <stdio.h>

#include <stdbool.h>

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

int findOptimalIndex(int page\_requests[], int num\_requests, int frames[], int num\_frames, int index) {

int res = -1, farthest = index;

for (int i = 0; i < num\_frames; i++) {

int j;

for (j = index; j < num\_requests; j++) {

if (frames[i] == page\_requests[j]) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

// If the page is not found in future requests, it's the optimal page for replacement

if (j == num\_requests)

return i;

}

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

}

// Function to simulate the Optimal page replacement algorithm

int optimal(int page\_requests[], int num\_requests, int frame\_size) {

int page\_faults = 0;

int frames[frame\_size];

bool page\_present[frame\_size];

// Initialize frames with -1 indicating an empty frame

for (int i = 0; i < frame\_size; i++) {

frames[i] = -1;

page\_present[i] = false;

}

// Iterate over each page request

for (int i = 0; i < num\_requests; i++) {

int page = page\_requests[i];

int found = 0;

// Check if the page is already present in any of the frames

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

// If the page is not found in any frame, it's a page fault

if (!found) {

page\_faults++;

// Find the index of the page that will not be used for the longest time in the future

int index = findOptimalIndex(page\_requests, num\_requests, frames, frame\_size, i + 1);

frames[index] = page; // Replace the optimal page

}

}

return page\_faults; // Return the total number of page faults

}

int main() {

int frame\_size, num\_requests;

// Asking the user to input the frame size

printf("Enter the frame size: ");

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

// Asking the user to input the number of page requests

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

scanf("%d", &num\_requests);

int page\_requests[num\_requests];

// Asking the user to input each page request individually

printf("Enter the page requests:\n");

for (int i = 0; i < num\_requests; i++) {

scanf("%d", &page\_requests[i]);

}

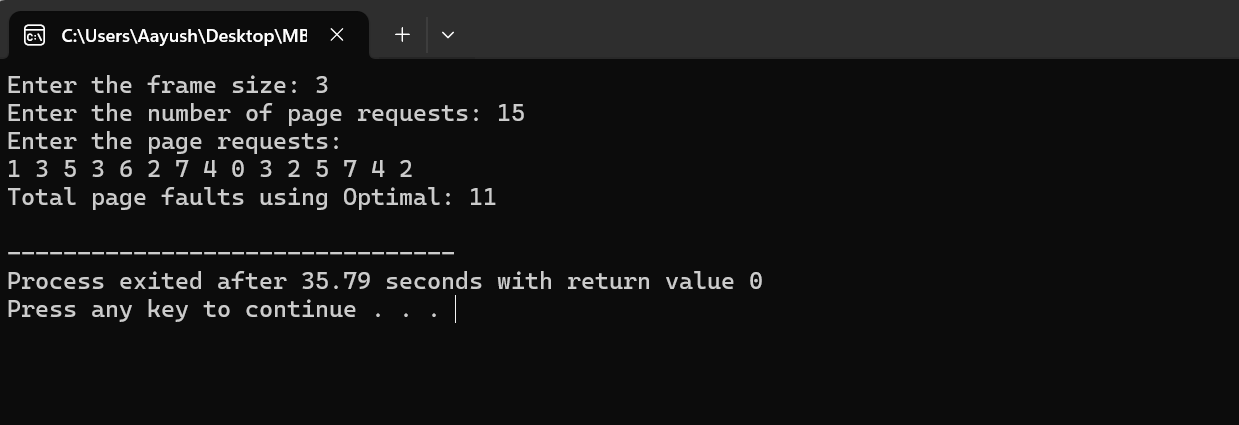
// Calling the optimal function and printing the total number of page faults

printf("Total page faults using Optimal: %d\n", optimal(page\_requests, num\_requests, frame\_size));

return 0;

}

**Output:**





**Lab no: 5**

**Title: Write a program to simulate FCFS process scheduling algorithm to calculate average turnaround time and waiting time for user input no. of process and its corresponding burst time.**

**Theory:**

**FCFS (First Come First Serve) Process Scheduling Algorithm:**

FCFS is the simplest scheduling algorithm in which the processes are executed in the order they arrive in the ready queue. In this algorithm, the process that arrives first is executed first, and so on. It is a non-preemptive scheduling algorithm, meaning once a process starts its execution, it continues until it completes or goes into the waiting state.

Advantages:

* Simple and easy to understand.
* Fairness is ensured as the processes are scheduled in the order they arrive.

Disadvantages:

* May lead to high average waiting time, especially if long processes arrive first (convoy effect).
* Not suitable for time-sensitive or real-time systems.

**Algorithm**:

1. Start with an empty ready queue.
2. Whenever a process arrives, it is added to the end of the ready queue.
3. If the CPU is idle, select the process at the front of the ready queue for execution.
4. Execute the selected process until it completes its CPU burst.
5. If the process completes its CPU burst, remove it from the system.
6. Repeat steps 3-5 until all processes have completed execution.

To calculate the average turnaround time and waiting time using FCFS:

1. Calculate the waiting time for each process. The waiting time for the first process is always 0. For subsequent processes, the waiting time is the sum of the burst times of the previous processes.
2. Calculate the turnaround time for each process. Turnaround time is the sum of waiting time and burst time for each process.
3. Calculate the average waiting time and average turnaround time by summing up the waiting time and turnaround time of all processes, respectively, and dividing by the total number of processes

**Source Code:**

#include <stdio.h>

// Function to calculate average turnaround time and waiting time for FCFS scheduling algorithm

void calculateAverageTimes(int burst\_time[], int num\_processes) {

int waiting\_time[num\_processes], turnaround\_time[num\_processes];

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

// Waiting time for the first process is always 0

waiting\_time[0] = 0;

// Calculate waiting time for each process

for (int i = 1; i < num\_processes; i++) {

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

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

}

// Calculate turnaround time for each process

for (int i = 0; i < num\_processes; i++) {

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

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

}

// Calculate average waiting time and average turnaround time

float avg\_waiting\_time = total\_waiting\_time / num\_processes;

float avg\_turnaround\_time = total\_turnaround\_time / num\_processes;

// Print the results

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

for (int i = 0; i < num\_processes; i++) {

printf("%d\t%d\t\t%d\t\t%d\n", i + 1, burst\_time[i], waiting\_time[i], turnaround\_time[i]);

}

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

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

}

int main() {

int num\_processes;

// Getting the number of processes from the user

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

scanf("%d", &num\_processes);

int burst\_time[num\_processes];

// Getting burst times for each process from the user

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

for (int i = 0; i < num\_processes; i++) {

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

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

}

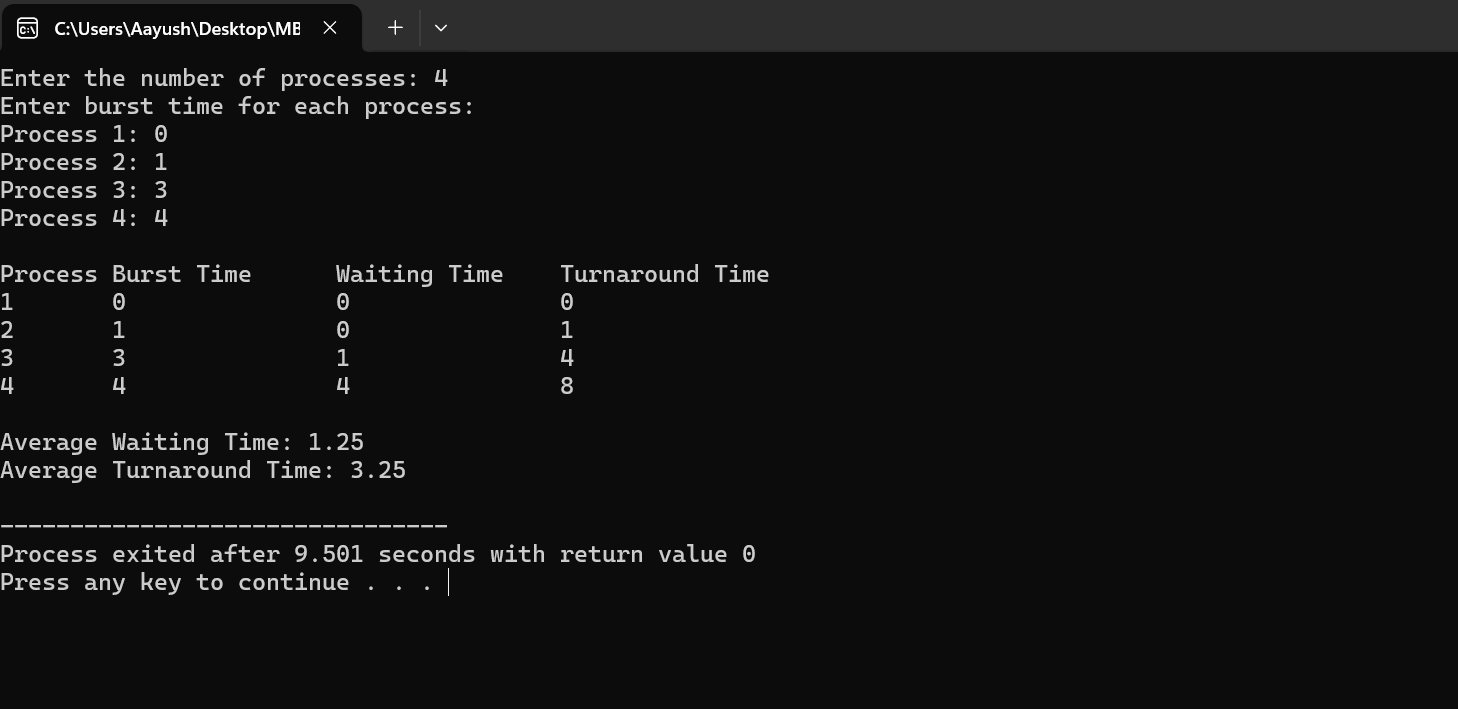
// Calculating and printing average waiting time and average turnaround time

calculateAverageTimes(burst\_time, num\_processes);

return 0;

}

**Output:**





**Lab no: 6**

**Title: Write a program to simulate SJF process scheduling algorithm to calculate average turnaround time and waiting time for user input no of process and its corresponding burst time.**

**Theory:**

**Shortest Job First (SJF) Process Scheduling Algorithm:**

SJF is a non-preemptive scheduling algorithm that selects the process with the shortest burst time to execute first. In SJF scheduling, the process with the smallest burst time among the ready processes is selected for execution. If multiple processes have the same shortest burst time, they are scheduled in the order they arrive in the ready queue. SJF is also known as Shortest Job Next (SJN) or Shortest Process Next (SPN).

Advantages:

* Minimizes average waiting time and turnaround time compared to other scheduling algorithms.
* Efficient for processes with varying burst times, as it prioritizes shorter tasks.

Disadvantages:

* Requires knowledge of the burst times of all processes in advance, which may not always be available in real-time systems.
* May lead to starvation of longer processes if a steady stream of short processes arrives continuously.

**Algorithm**:

1. Start with an empty ready queue.
2. Whenever a process arrives, it is added to the end of the ready queue.
3. Select the process with the shortest burst time from the ready queue for execution.
4. Execute the selected process until it completes its CPU burst.
5. If the process completes its CPU burst, remove it from the system.
6. Repeat steps 3-5 until all processes have completed execution.

To calculate the average turnaround time and waiting time using SJF:

1. Sort the processes based on their burst times in ascending order.
2. Calculate the completion time for each process, which is the sum of the burst times of all preceding processes and the burst time of the current process.
3. Calculate the waiting time for each process, which is the difference between the completion time and the burst time of the process.
4. Calculate the turnaround time for each process, which is the sum of the waiting time and the burst time of the process.
5. Calculate the average waiting time and average turnaround time by summing up the waiting time and turnaround time of all processes, respectively, and dividing by the total number of processes.

**Source Code:**

#include <stdio.h>

// Function to calculate average turnaround time and waiting time for SJF scheduling algorithm

void calculateAverageTimes(int burst\_time[], int num\_processes) {

int waiting\_time[num\_processes], turnaround\_time[num\_processes], completion\_time[num\_processes];

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

// Calculate completion time for each process

completion\_time[0] = burst\_time[0];

for (int i = 1; i < num\_processes; i++) {

completion\_time[i] = completion\_time[i - 1] + burst\_time[i];

}

// Calculate waiting time for each process

for (int i = 0; i < num\_processes; i++) {

waiting\_time[i] = completion\_time[i] - burst\_time[i];

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

}

// Calculate turnaround time for each process

for (int i = 0; i < num\_processes; i++) {

turnaround\_time[i] = completion\_time[i];

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

}

// Calculate average waiting time and average turnaround time

float avg\_waiting\_time = total\_waiting\_time / num\_processes;

float avg\_turnaround\_time = total\_turnaround\_time / num\_processes;

// Print the results

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

for (int i = 0; i < num\_processes; i++) {

printf("%d\t%d\t\t%d\t\t%d\n", i + 1, burst\_time[i], waiting\_time[i], turnaround\_time[i]);

}

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

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

}

int main() {

int num\_processes;

// Getting the number of processes from the user

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

scanf("%d", &num\_processes);

int burst\_time[num\_processes];

// Getting burst times for each process from the user

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

for (int i = 0; i < num\_processes; i++) {

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

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

}

// Sorting burst times in ascending order using bubble sort

for (int i = 0; i < num\_processes - 1; i++) {

for (int j = 0; j < num\_processes - i - 1; j++) {

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

// Swap burst times

int temp = burst\_time[j];

burst\_time[j] = burst\_time[j + 1];

burst\_time[j + 1] = temp;

}

}

}

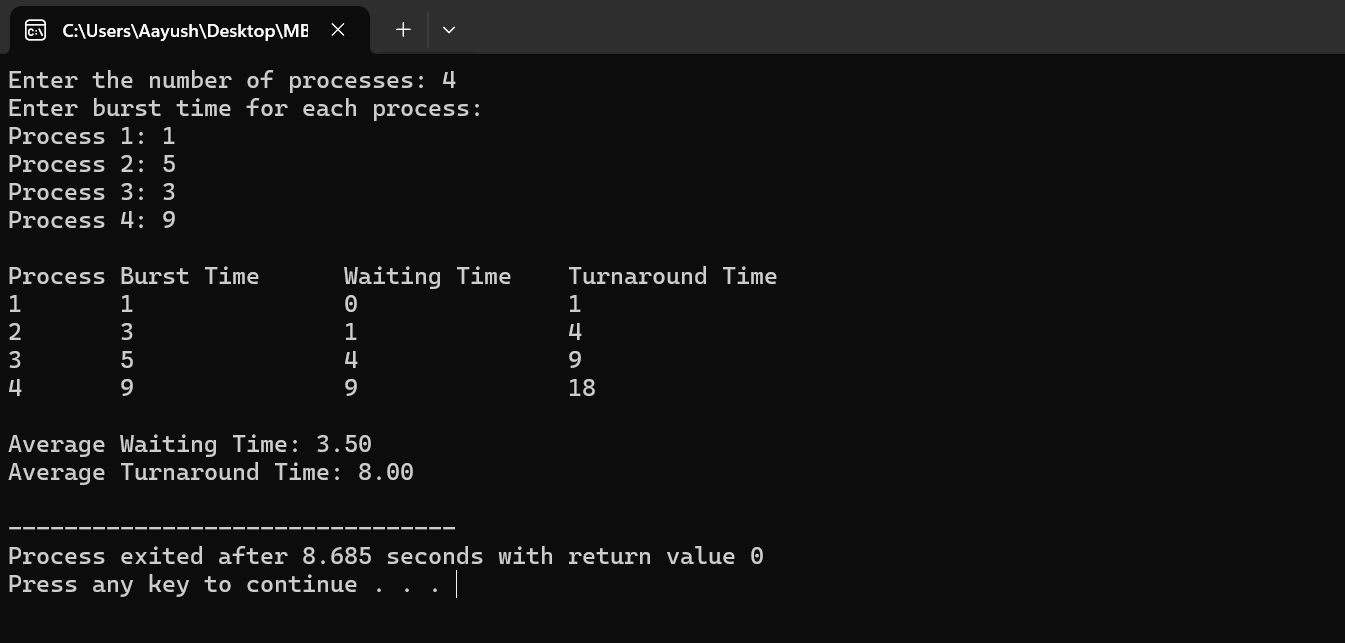
// Calculating and printing average waiting time and average turnaround time

calculateAverageTimes(burst\_time, num\_processes);

return 0;

}

**Output:**





**Lab no: 7**

**Title: Write a program to simulate RR process scheduling algorithm to calculate average turnaround time and waiting time for user input no of process, its corresponding burst time and time quantum.**

**Theory:**

**Round Robin (RR) Process Scheduling Algorithm:**

Round Robin is a preemptive scheduling algorithm that assigns a fixed time unit, known as a time quantum, to each process in the ready queue. It is commonly used in time-sharing systems where each process is allocated a small unit of CPU time, and the CPU is switched among processes in a circular order. When a process's time quantum expires, it is preempted, and the CPU is assigned to the next process in the ready queue.

Advantages:

* Fairness: Each process gets an equal share of CPU time.
* Suitable for time-sharing systems and interactive applications.

Disadvantages:

* May lead to high context-switching overhead, especially with small time quanta.
* May not be efficient for long-running processes.

**Algorithm**:

1. Start with an empty ready queue.
2. Whenever a process arrives, it is added to the end of the ready queue.
3. Select the process at the front of the ready queue for execution.
4. Execute the process for a fixed time unit known as the time quantum.
5. If the process completes its CPU burst within the time quantum, remove it from the system.
6. If the process does not complete its CPU burst within the time quantum, preempt it and move it to the end of the ready queue.
7. Repeat steps 3-6 until all processes have completed execution.

To calculate the average turnaround time and waiting time using RR:

1. Initialize remaining time, waiting time, and turnaround time arrays for each process.
2. Iterate through the processes in a round-robin manner, executing each process for the time quantum.
3. Update the remaining time for each process and adjust waiting time accordingly.
4. If a process completes its CPU burst within the time quantum, calculate its turnaround time.
5. Repeat the process until all processes have completed execution.
6. Calculate the total waiting time and total turnaround time for all processes.
7. Calculate the average waiting time and average turnaround time by dividing the total waiting time and total turnaround time by the number of processes, respectively.

**Source Code:**

#include <stdio.h>

#define MAX\_PROCESSES 10

// Function to simulate the Round Robin (RR) process scheduling algorithm

void roundRobin(int burst\_time[], int num\_processes, int time\_quantum) {

int remaining\_time[MAX\_PROCESSES];

int waiting\_time[MAX\_PROCESSES] = {0};

int turnaround\_time[MAX\_PROCESSES] = {0};

// Initialize remaining time with burst times

for (int i = 0; i < num\_processes; i++) {

remaining\_time[i] = burst\_time[i];

}

int current\_time = 0; // Current time in the simulation

int completed\_processes = 0; // Number of completed processes

// Simulate RR scheduling until all processes are completed

while (completed\_processes < num\_processes) {

// Traverse each process in a round-robin manner

for (int i = 0; i < num\_processes; i++) {

if (remaining\_time[i] > 0) {

// Execute the process for the time quantum or remaining time, whichever is smaller

int execution\_time = (remaining\_time[i] < time\_quantum) ? remaining\_time[i] : time\_quantum;

// Update current time and remaining time for the process

current\_time += execution\_time;

remaining\_time[i] -= execution\_time;

// Update waiting time for other processes

for (int j = 0; j < num\_processes; j++) {

if (j != i && remaining\_time[j] > 0) {

waiting\_time[j] += execution\_time;

}

}

// If process is completed, calculate turnaround time

if (remaining\_time[i] == 0) {

turnaround\_time[i] = current\_time;

completed\_processes++;

}

}

}

}

// Calculate total waiting time and turnaround time

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

for (int i = 0; i < num\_processes; i++) {

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

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

}

// Calculate average waiting time and average turnaround time

float avg\_waiting\_time = total\_waiting\_time / num\_processes;

float avg\_turnaround\_time = total\_turnaround\_time / num\_processes;

// Print the results

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

for (int i = 0; i < num\_processes; i++) {

printf("%d\t%d\t\t%d\t\t%d\n", i + 1, burst\_time[i], waiting\_time[i], turnaround\_time[i]);

}

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

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

}

int main() {

int num\_processes, time\_quantum;

// Getting the number of processes from the user

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

scanf("%d", &num\_processes);

// Getting the burst times for each process from the user

int burst\_time[MAX\_PROCESSES];

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

for (int i = 0; i < num\_processes; i++) {

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

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

}

// Getting the time quantum from the user

printf("Enter the time quantum: ");

scanf("%d", &time\_quantum);

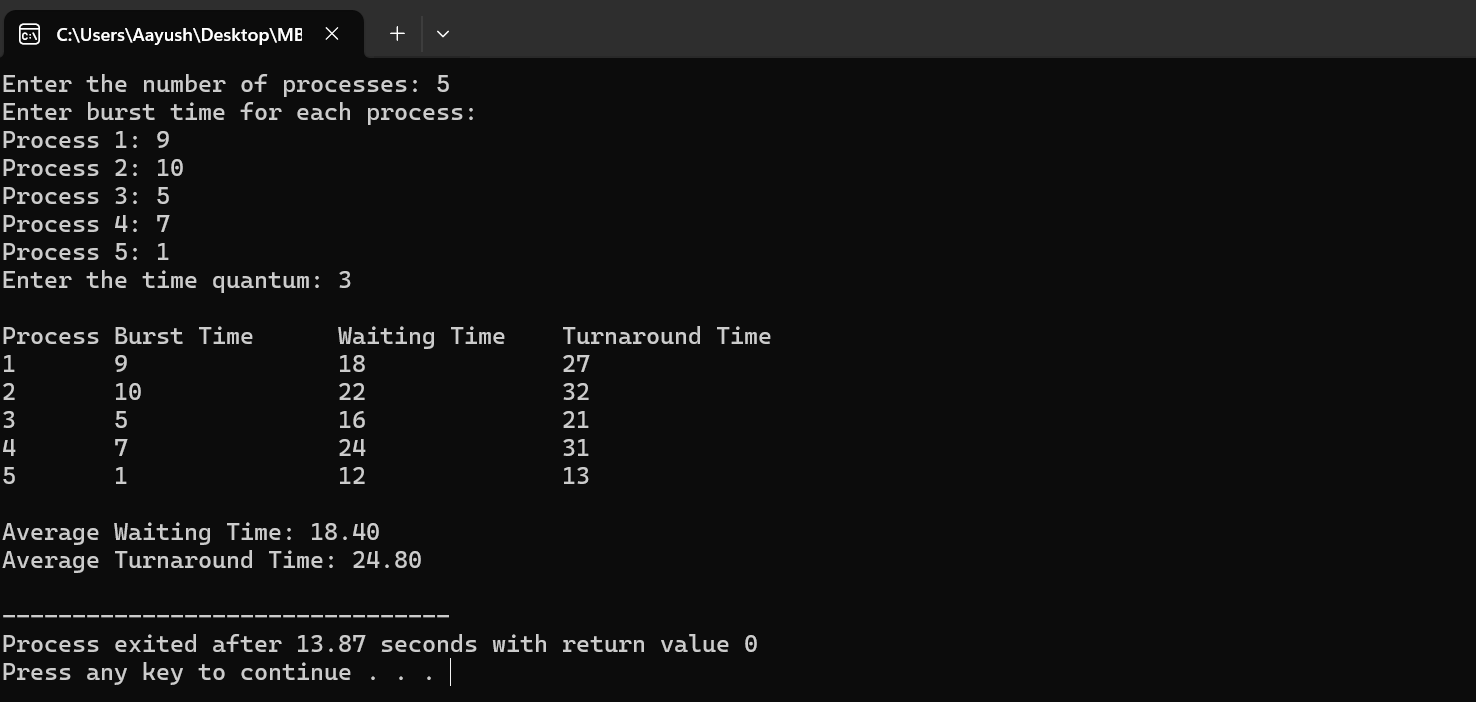
// Calling the Round Robin function to simulate the algorithm and calculate average turnaround time and waiting time

roundRobin(burst\_time, num\_processes, time\_quantum);

return 0;

}

**Output:**





**Lab no: 8**

**Title: Write a program to find the seek time in Disk scheduling algorithm using FCFS algorithm for user input pending request and head position.**

**Theory:**

**Disk Scheduling Algorithms:**

Disk scheduling algorithms are used to optimize the movement of the disk arm and reduce the total seek time, which is the time taken by the disk arm to move between disk requests. One common disk scheduling algorithm is the First Come First Serve (FCFS) algorithm.

**Disk Scheduling using FCFS Algorithm:**

FCFS (First Come First Serve) is a simple disk scheduling algorithm where the disk requests are served in the order they arrive. The requests are executed one by one, starting from the request closest to the current head position and moving towards the outermost track or innermost track based on the direction of movement.

Advantages:

* Simple and easy to implement.
* Fairness is ensured as requests are served in the order they arrive.

Disadvantages:

* May result in longer average seek time, especially when there are widely scattered requests.
* Not suitable for systems with strict response time requirements.

Algorithm:

1. Start with the current head position.
2. Traverse the pending disk requests in the order they arrived.
3. For each request, calculate the seek time as the absolute difference between the current head position and the position of the request on the disk.
4. Update the total seek time by adding the seek time calculated in step 3.
5. Update the current head position to the position of the request.
6. Repeat steps 3-5 for all pending disk requests.
7. After processing all requests, the total seek time represents the time taken by the disk arm to service all requests using the FCFS algorithm.

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Function to calculate the total seek time using FCFS algorithm

int calculateSeekTimeFCFS(int requests[], int num\_requests, int head) {

int seek\_count = 0;

int current\_track, distance;

// Traverse the pending requests in the order they arrived

for (int i = 0; i < num\_requests; i++) {

current\_track = requests[i];

// Calculate seek time as the absolute difference between current head position and request

distance = abs(head - current\_track);

// Increase the total seek time

seek\_count += distance;

// Update the head position to the current request

head = current\_track;

}

return seek\_count;

}

int main() {

int num\_requests, head\_position;

// Getting the number of pending requests from the user

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

scanf("%d", &num\_requests);

// Getting the pending requests from the user

int requests[num\_requests];

printf("Enter the pending requests:\n");

for (int i = 0; i < num\_requests; i++) {

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

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

}

// Getting the current head position from the user

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

scanf("%d", &head\_position);

// Calculating the total seek time using FCFS algorithm

int total\_seek\_time = calculateSeekTimeFCFS(requests, num\_requests, head\_position);

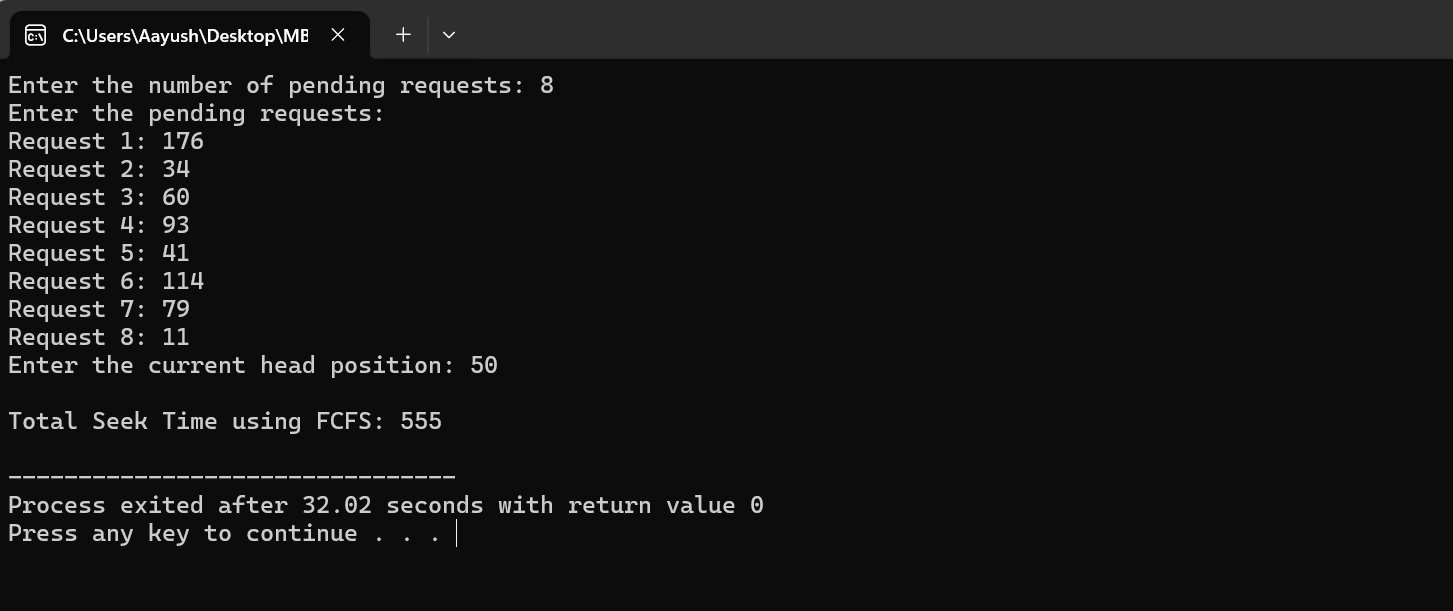
// Printing the total seek time

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

return 0;

}

**Output:**

****



**Lab no: 9**

**Title: Write a program to find the seek time in Disk scheduling algorithm using SCAN algorithm for user input pending request and head position.**

**Theory:**

**Disk Scheduling using SCAN Algorithm:**

SCAN, also known as the Elevator Algorithm, is a disk scheduling algorithm designed to reduce seek time by systematically scanning the disk in one direction until the end and then reversing direction to scan back to the starting point. The head moves from one end of the disk to the other, servicing all requests in its path, and then reverses direction when it reaches the end. This algorithm helps in minimizing seek time by optimizing the movement of the disk arm and ensuring fairness in servicing disk requests.

Advantages:

* High throughput
* Low variance of response time
* Average response time

Disadvantages:

* Long waiting time for requests for locations just visited by disk arm

**Algorithm**:

1. Start with the current head position and determine the direction of movement (right or left).
2. Sort the pending requests in ascending order to facilitate movement in the specified direction.
3. Traverse the requests in the specified direction until reaching the end of the disk or until all requests have been serviced.
4. For each request encountered, calculate the seek time as the absolute difference between the current head position and the request position.
5. Update the total seek time by summing up the seek times for all requests serviced.
6. Reverse the direction of movement when reaching the end of the disk and continue servicing requests in the opposite direction.
7. Repeat steps 3-6 until all pending requests have been serviced.

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Function to calculate the total seek time using SCAN algorithm

int calculateSeekTimeSCAN(int requests[], int num\_requests, int head, char direction) {

int seek\_count = 0;

int current\_track = head;

int i, j;

int min\_track = 0;

int max\_track = 199;

int found = 0;

// Sort the requests in ascending order

for (i = 0; i < num\_requests - 1; i++) {

for (j = 0; j < num\_requests - i - 1; j++) {

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

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// SCAN algorithm for the right direction

if (direction == 'r') {

// Move right until the end of the disk

for (i = 0; i < num\_requests; i++) {

if (requests[i] >= head) {

found = 1;

break;

}

}

// Calculate seek time for right direction

if (found == 1) {

for (j = i; j < num\_requests; j++) {

seek\_count += abs(requests[j] - current\_track);

current\_track = requests[j];

}

}

// Go to the end of the disk

seek\_count += abs(max\_track - current\_track);

// Go back to the starting position

current\_track = max\_track;

// Continue moving left from the starting position

for (j = num\_requests - 1; j >= 0; j--) {

seek\_count += abs(requests[j] - current\_track);

current\_track = requests[j];

}

}

// SCAN algorithm for the left direction

else if (direction == 'l') {

// Move left until the start of the disk

for (i = num\_requests - 1; i >= 0; i--) {

if (requests[i] <= head) {

found = 1;

break;

}

}

// Calculate seek time for left direction

if (found == 1) {

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

seek\_count += abs(requests[j] - current\_track);

current\_track = requests[j];

}

}

// Go to the beginning of the disk

seek\_count += abs(min\_track - current\_track);

// Go back to the starting position

current\_track = min\_track;

// Continue moving right from the starting position

for (j = 0; j < num\_requests; j++) {

seek\_count += abs(requests[j] - current\_track);

current\_track = requests[j];

}

}

return seek\_count;

}

int main() {

int num\_requests, head\_position;

char direction;

// Getting the number of pending requests from the user

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

scanf("%d", &num\_requests);

// Getting the pending requests from the user

int requests[num\_requests];

printf("Enter the pending requests:\n");

for (int i = 0; i < num\_requests; i++) {

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

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

}

// Getting the current head position from the user

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

scanf("%d", &head\_position);

// Getting the direction of movement from the user (r = right, l = left)

printf("Enter the direction of movement (r for right, l for left): ");

scanf(" %c", &direction);

// Calculating the total seek time using SCAN algorithm

int total\_seek\_time = calculateSeekTimeSCAN(requests, num\_requests, head\_position, direction);

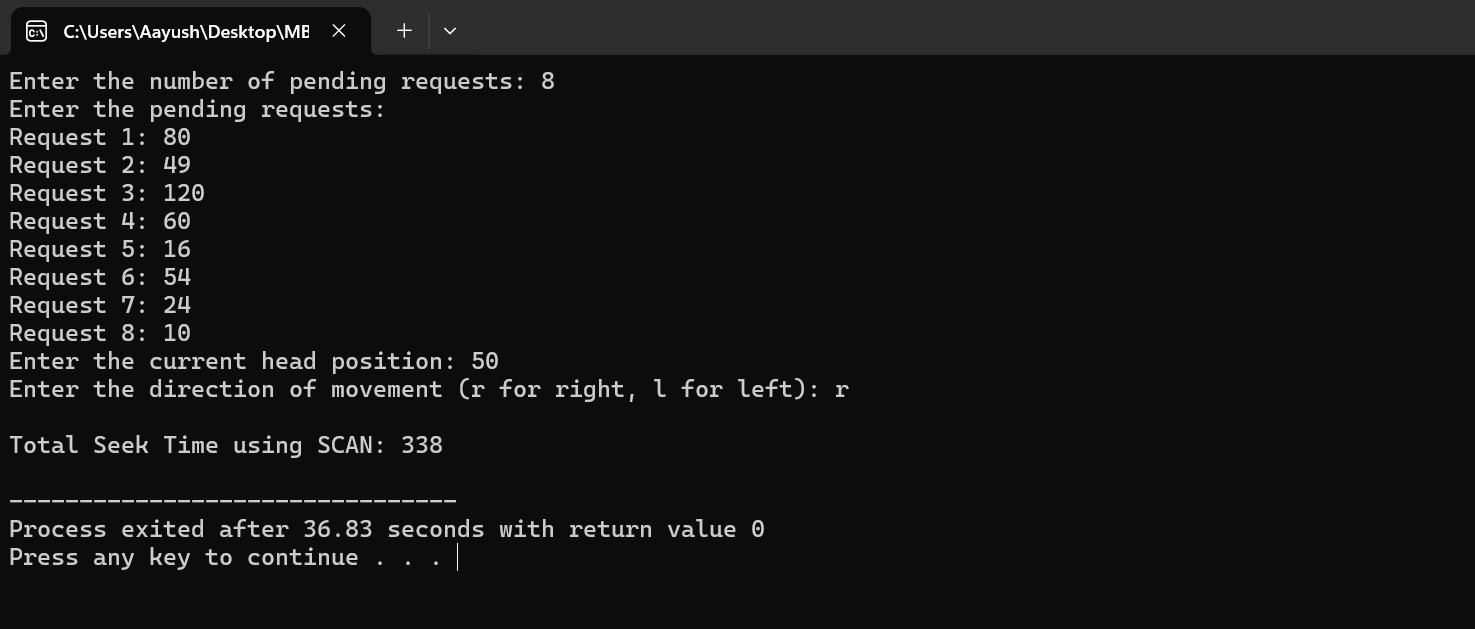
// Printing the total seek time

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

return 0;

}

**Output:**





**Lab no: 10**

**Title: Write a program to find the seek time in Disk scheduling algorithm using LOOK algorithm for user input pending request and head position.**

**Theory:**

**Disk Scheduling using LOOK Algorithm:**

LOOK is a disk scheduling algorithm that is similar to the SCAN algorithm but with a slight variation. LOOK algorithm services pending requests in one direction until the end of the disk or until there are no more requests in that direction, and then reverses its direction and services requests in the opposite direction. The head moves from one end of the disk to the other, servicing all requests in its path, and then reverses direction when it reaches the end or when there are no more requests in the current direction. Advantages:

1. LOOK algorithm efficiently reduces seek time by servicing requests in a continuous manner without unnecessary movement over empty areas of the disk.
2. It is simpler to implement compared to some other disk scheduling algorithms.

Disadvantages:

1. LOOK may cause delays for requests located towards the middle of the disk if there is a heavy load of requests concentrated in one direction.
2. It does not consider the specific positioning of requests on the disk, potentially resulting in inefficient disk access patterns.

**Algorithm:**

1. Start with the current head position and determine the direction of movement (right or left).
2. Sort the pending requests in ascending order to facilitate movement in the specified direction.
3. Traverse the requests in the specified direction until reaching the end of the disk or until there are no more requests in that direction.
4. For each request encountered, calculate the seek time as the absolute difference between the current head position and the request position.
5. Update the total seek time by summing up the seek times for all requests serviced.
6. Reverse the direction of movement when reaching the end of the disk or when there are no more requests in the current direction.
7. Continue servicing requests in the opposite direction until there are no more requests to be serviced.

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Function to calculate the total seek time using LOOK algorithm

int calculateSeekTimeLOOK(int requests[], int num\_requests, int head, char direction) {

int seek\_count = 0;

int current\_track = head;

int i, j;

int min\_track = 0;

int max\_track = 199;

int found = 0;

// Sort the requests in ascending order

for (i = 0; i < num\_requests - 1; i++) {

for (j = 0; j < num\_requests - i - 1; j++) {

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

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// LOOK algorithm for the right direction

if (direction == 'r') {

// Move right until the end of the disk

for (i = 0; i < num\_requests; i++) {

if (requests[i] >= head) {

found = 1;

break;

}

}

// Calculate seek time for right direction

if (found == 1) {

for (j = i; j < num\_requests; j++) {

seek\_count += abs(requests[j] - current\_track);

current\_track = requests[j];

}

}

}

// LOOK algorithm for the left direction

else if (direction == 'l') {

// Move left until the start of the disk

for (i = num\_requests - 1; i >= 0; i--) {

if (requests[i] <= head) {

found = 1;

break;

}

}

// Calculate seek time for left direction

if (found == 1) {

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

seek\_count += abs(requests[j] - current\_track);

current\_track = requests[j];

}

}

}

return seek\_count;

}

int main() {

int num\_requests, head\_position;

char direction;

// Getting the number of pending requests from the user

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

scanf("%d", &num\_requests);

// Getting the pending requests from the user

int requests[num\_requests];

printf("Enter the pending requests:\n");

for (int i = 0; i < num\_requests; i++) {

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

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

}

// Getting the current head position from the user

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

scanf("%d", &head\_position);

// Getting the direction of movement from the user (r = right, l = left)

printf("Enter the direction of movement (r for right, l for left): ");

scanf(" %c", &direction);

// Calculating the total seek time using LOOK algorithm

int total\_seek\_time = calculateSeekTimeLOOK(requests, num\_requests, head\_position, direction);

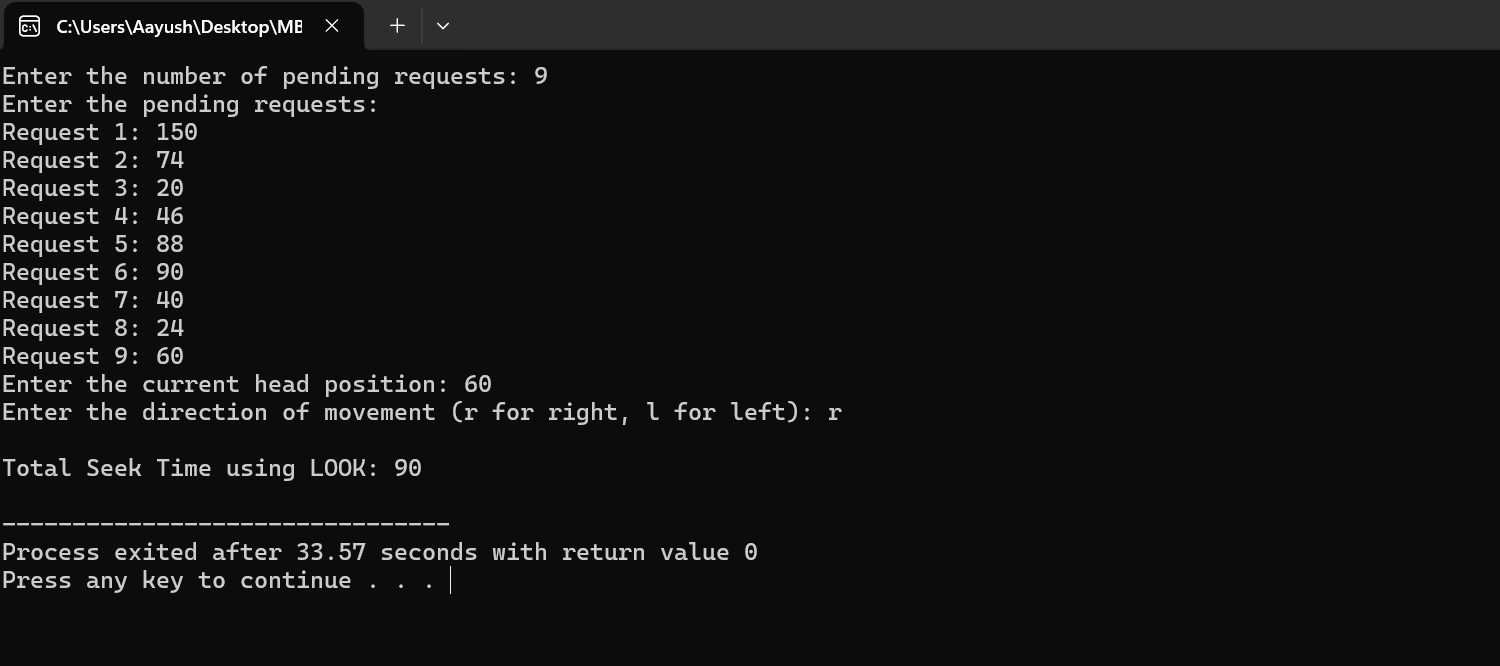
// Printing the total seek time

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

return 0;

}

**Output:**





**Lab no: 11**

**Title: Write a program to simulate Best fit memory allocation strategy for user input no and size of free blocks and arriving process**

**Theory:**

**Best Fit Memory Allocation Strategy:**

Best Fit is a memory allocation strategy used in operating systems to allocate memory blocks to processes. In this strategy, the operating system selects the memory block that best fits the size of the process being allocated. The goal is to minimize wasted memory by selecting the smallest memory block that can accommodate the process.

Advantages:

1. Best Fit minimizes wasted memory by selecting the smallest memory block that can accommodate the process, thereby reducing fragmentation.
2. It provides better utilization of memory compared to other allocation strategies like First Fit or Worst Fit.

Disadvantages:

1. Best Fit may lead to fragmentation, especially if there are many small free blocks scattered throughout memory that cannot be efficiently utilized by larger processes.
2. The search for the best fit block among all available memory blocks can be time-consuming, especially as the number of processes and memory blocks increases.

**Algorithm:**

1. Input the number of memory blocks and their sizes, as well as the number of processes and their sizes.
2. Iterate through each process.
3. For each process, find the memory block that best fits its size among the available memory blocks.
4. Allocate the process to the selected memory block and update the size of the memory block.
5. Repeat steps 3-4 for each process.
6. Print the allocation details, showing which process is allocated to which memory block.
7. If no suitable memory block is found for a process, mark it as "Not Allocated".

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Function to allocate memory using Best Fit strategy

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

int allocation[n];

int i, j;

// Initialize all allocations to -1, indicating no allocation

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

allocation[i] = -1;

// Iterate through each process

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

// Find the best fit block for the current process

int bestFitIdx = -1;

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

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

if (bestFitIdx == -1 || blockSize[j] < blockSize[bestFitIdx])

bestFitIdx = j;

}

}

// Allocate the current process to the best fit block if found

if (bestFitIdx != -1) {

allocation[i] = bestFitIdx;

blockSize[bestFitIdx] -= processSize[i];

}

}

// Print the allocation details

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("Not Allocated\n");

}

}

int main() {

int m, n; // Number of memory blocks and processes

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

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

scanf("%d", &n);

int blockSize[m], processSize[n]; // Arrays to store memory block sizes and process sizes

printf("Enter the size of memory blocks:\n");

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

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

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

}

printf("Enter the size of processes:\n");

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

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

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

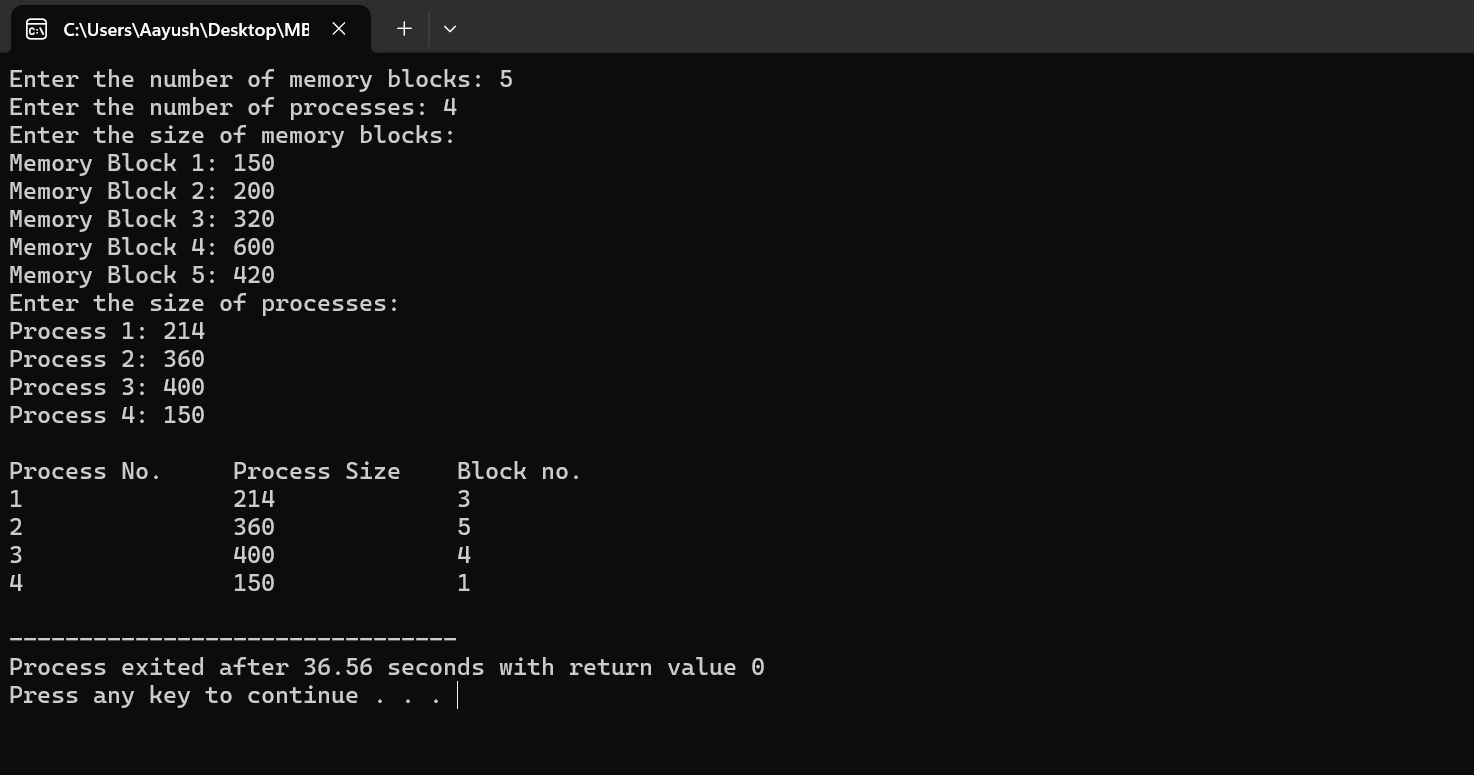
}

bestFit(blockSize, m, processSize, n); // Call the Best Fit function

return 0;

}

**Output:**





**Lab no: 12**

**Title: Write a program to simulate Worst fit memory allocation strategy for user input no and size of free blocks and arriving process.**

**Theory:**

**Worst Fit Memory Allocation Strategy:**

Worst Fit is a memory allocation strategy used in operating systems to allocate memory blocks to processes. In this strategy, the OS selects the largest memory block available that can accommodate the process being allocated. The goal is to minimize fragmentation by using larger memory blocks that are less likely to leave behind small unusable fragments.

Advantages:

1. Worst Fit reduces fragmentation by selecting larger memory blocks that are less likely to leave behind small unusable fragments, thereby improving memory utilization.
2. It provides better utilization of memory compared to other allocation strategies like First Fit or Best Fit in scenarios where large processes are more common.

Disadvantages:

1. Worst Fit may lead to inefficient utilization of memory if smaller memory blocks are continuously allocated, leaving behind large unused chunks of memory.
2. The search for the largest memory block among all available memory blocks can be time-consuming, especially as the number of processes and memory blocks increases.

**Algorithm:**

1. Input the number of memory blocks and their sizes, as well as the number of processes and their sizes.
2. Iterate through each process.
3. For each process, find the memory block that has the largest size among the available memory blocks.
4. Allocate the process to the selected memory block and update the size of the memory block.
5. Repeat steps 3-4 for each process.
6. Print the allocation details, showing which process is allocated to which memory block.
7. If no suitable memory block is found for a process, mark it as "Not Allocated".

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Function to allocate memory using Worst Fit strategy

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

int allocation[n];

int i, j;

// Initialize all allocations to -1, indicating no allocation

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

allocation[i] = -1;

// Iterate through each process

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

// Find the worst fit block for the current process

int worstFitIdx = -1;

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

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

if (worstFitIdx == -1 || blockSize[j] > blockSize[worstFitIdx])

worstFitIdx = j;

}

}

// Allocate the current process to the worst fit block if found

if (worstFitIdx != -1) {

allocation[i] = worstFitIdx;

blockSize[worstFitIdx] -= processSize[i];

}

}

// Print the allocation details

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("Not Allocated\n");

}

}

int main() {

int m, n; // Number of memory blocks and processes

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

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

scanf("%d", &n);

int blockSize[m], processSize[n]; // Arrays to store memory block sizes and process sizes

printf("Enter the size of memory blocks:\n");

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

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

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

}

printf("Enter the size of processes:\n");

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

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

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

}

worstFit(blockSize, m, processSize, n); // Call the Worst Fit function

return 0;

}

**Output:**

