Institute of Science and Technology Tribhuvan University



Department of computer science and Information Technology Prithivi Narayan Campus, Pokhara

Lab report on

# Operating System (CSC264)

Submitted To : Mrs. Amrita Thapa

Submitted By :

Dibesh Pandey

Symbol No: 79011878

Lab Index



**Name: Dibesh Pandey Roll. No: 66**

Level: Bachelors Year/Sem: 2nd/4th

Subject: Operating System Faculty: B.Sc. CSIT

Course No: CSC 264

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Lab no. | Exp no | Experiment name | Submitted Date | Remark |
| 1. |  |  |  |  |
|  | 1.1 | To Implement Fork Function |  |  |
|  | 1.2 | To Implement the FCFS Process Scheduling |  |  |
|  | 1.3 | To Implement the SJF Process Scheduling |  |  |
|  | 1.4 | To Implement the Round Robin Process Scheduling |  |  |
| 2 | 2.1 | To Implement FIFO Page Replacement Algorithm |  |  |
|  | 2.2 | To Implement LRU Page Replacement Algorithm |  |  |
|  | 2.3 | To Implement FCFS Disk Scheduling Algorithm |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

LAB:1.1

TO IMPLEMENT FORK FUNCTION.

# OBJECTIVE:

* To understand about the fork and creating a new process using the fork command

# THEORY:

The fork() system call is used in Unix/Linux systems to create a new process. When a process calls fork(), it creates a copy of itself. The new process created by fork() is called the child process, and the original process is referred to as the parent process.

Key Points:

* Parent Process: The original process that calls fork().
* Child Process: The new process created as a result of the fork(). It is an exact copy of the parent process except for the returned value.
* Return Value:
  + Parent Process: Returns the Process ID (PID) of the child process.
  + Child Process: Returns 0.
  + Failure: Returns -1 in the parent process if the fork() fails, and no child process is created.

# DEMONSTRATION:

Program 1:

To create new process using fork() function.

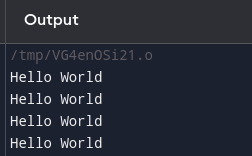
**Source code:** *#include <stdio.h> #include <sys/types.h> #include <unistd.h> int main(){*

*fork();*

*fork();*

*printf("Hello World\n"); return 0;*

*}*



# OUTPUT AND DISCUSSIONS:

In the above code, two fork() calls lead to multiple processes being created. The first fork() creates a child process, and the second fork() is executed by both the parent and the newly created child process, resulting in four processes in total.

* The first fork() creates a child process, resulting in two processes.
* The second fork() is executed by both the parent and the first child process, leading to four processes in total.

Each process prints "Hello World" once. Since there are four processes, "Hello World" is printed four times. The order of the output can vary because the processes run concurrently, and the operating system's scheduler determines their execution order.

# CONCLUSION:

The fork() system call is a powerful tool for creating new processes. By calling fork() multiple times, we can create a tree of processes, each running concurrently. This demonstration illustrates how the number of processes grows exponentially with each fork(), emphasizing the importance of understanding process creation and management in systems programming.

Experiment 1.2

To Implement the FCFS Process Scheduling

1. **OBJECTIVE:**

To study what is the FCFS process scheduling

To implement the FCFS using C programming language

1. **THEORY:**

FCFS (First-Come, First-Served) is one of the simplest CPU scheduling algorithms. In this algorithm, the process that arrives first in the ready queue is allocated the CPU first. The order of execution is determined by the arrival time of the processes, with no preemption. This means that once a process starts executing, it runs to completion before the next process is selected.

Key Points:

* Non-preemptive: Once a process starts executing, it cannot be interrupted until it finishes.
* Fair: Every process gets a chance to execute in the order of its arrival.
* Inefficiency: FCFS can lead to a problem known as the "convoy effect," where short processes get stuck waiting behind long processes. The average waiting time under FCFS can vary significantly depending on the order of process arrival.

1. **DEMONSTRATION**

Program 1:

To use the FCFS for process scheduling

**Source code:**

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

*int n,bt[20],arvl\_time[20],wt[20],tat[20],avwt = 0, avtat = 0, i,j; printf("Enter the total number of process: ");*

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

*printf("\nEnter the Arrival time: \n"); for(i=0;i<n;i++){*

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

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

*}*

*printf("\nEnter the burst time: \n"); for(i=0;i<n;i++){*

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

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

*};*

*wt[0] = arvl\_time[0]; for (i = 1; i < n; i++) { wt[i] = 0;*

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

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

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

*avtat += tat[i];*

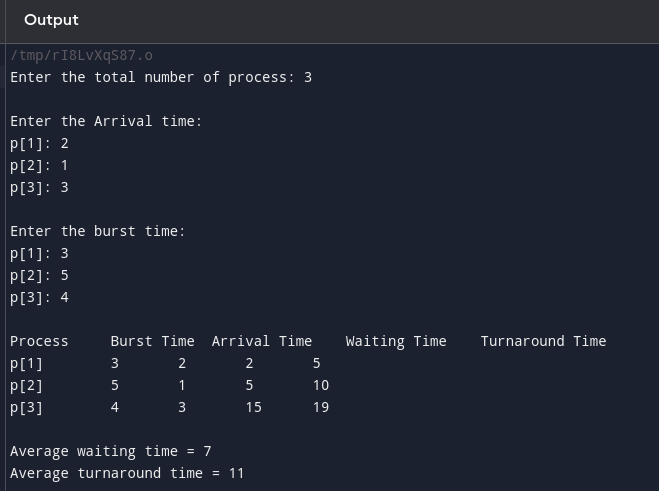
*printf("\np[%d]\t\t%d\t\t%d\t\t%d\t\t%d", i + 1, bt[i], arvl\_time[i], wt[i], tat[i]);*

*}*

*printf("\n\nAverage waiting time = %d", avwt / n); printf("\nAverage turnaround time = %d", avtat / n); return 0;*

*}*

**Output and Discussion:**



1. **CONCLUSION**

The FCFS scheduling algorithm is easy to implement and understand, as it simply processes tasks in the order they arrive.

Experiment 1.3

To Implement the SJF Process Scheduling

**OBJECTIVE:**

* To study the SJF (Shortest Job First) process scheduling.
* To implement the SJF (both preemptive and non-preemptive) using C programming language.

### THEORY:

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time to execute next. It can be implemented in two ways: preemptive and non-preemptive.

* **Non-Preemptive SJF:** Once a process starts execution, it cannot be interrupted until it finishes. The scheduler selects the process with the shortest burst time that is available at the time of selection.
* **Preemptive SJF (Shortest Remaining Time First, SRTF):** If a new process arrives with a burst time shorter than the remaining time of the current process, the current process is preempted, and the new process is executed.

The SJF algorithm is optimal in terms of minimizing the average waiting time. However, it may suffer from the problem of starvation, where longer processes may be delayed indefinitely if shorter processes keep arriving.

# DEMONSTRATION:

***Program 1:*** *To implement the non-preemptive SJF.*

*Source Code:*

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

*int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, pos, temp; float avg\_wt, avg\_tat;*

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

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

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

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

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

*for(j = i + 1; j < n; j++) { if(bt[j] < bt[pos]) {*

*pos = j;}}*

*temp = bt[i]; bt[i] = bt[pos];*

*bt[pos] = temp;temp = p[i]; p[i] = p[pos];*

*p[pos] = temp;*

*}*

*wt[0] = 0;*

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

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

*total += wt[i];}*

*avg\_wt = (float)total / n; total = 0;*

*printf("\nProcess \t Burst Time \t Waiting Time \t Turnaround Time\n"); for(i = 0; i < n; i++) {*

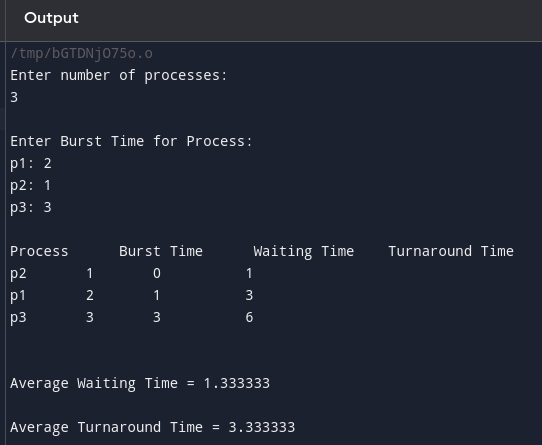
*tat[i] = bt[i] + wt[i]; total += tat[i];*

*printf("p%d\t\t %d\t\t %d\t\t\t%d\n", p[i], bt[i], wt[i], tat[i]);}*

*avg\_tat = (float)total / n;*

*printf("\n\nAverage Waiting Time = %f\n", avg\_wt); printf("\nAverage Turnaround Time = %f\n", avg\_tat); return 0;}*

### Output in Terminal:



**2:** To implement the preemptive SJF.

### Source Code:

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

*int n, bt[20], p[20], wt[20], tat[20], rem\_bt[20], i, j, min\_bt, time, count = 0; float avg\_wt = 0, avg\_tat = 0;*

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

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

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

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

*}*

*int min, finish\_time; int shortest = 0;*

*int complete = 0; int t = 0;*

*while (complete != n) { min = 9999;*

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

*if ((rem\_bt[i] > 0) && (rem\_bt[i] < min)) { min = rem\_bt[i];*

*shortest = i;*

*}*

*}*

*rem\_bt[shortest]--;*

*if (rem\_bt[shortest] == 0) { complete++;*

*finish\_time = t + 1;*

*wt[shortest] = finish\_time - bt[shortest]; if (wt[shortest] < 0) {*

*wt[shortest] = 0;*

*}*

*tat[shortest] = bt[shortest] + wt[shortest]; avg\_wt += wt[shortest];*

*avg\_tat += tat[shortest];*

*} t++;*

*}*

*avg\_wt /= n; avg\_tat /= n;*

*printf("\nProcess \t Burst Time \t Waiting Time \t Turnaround Time\n");*

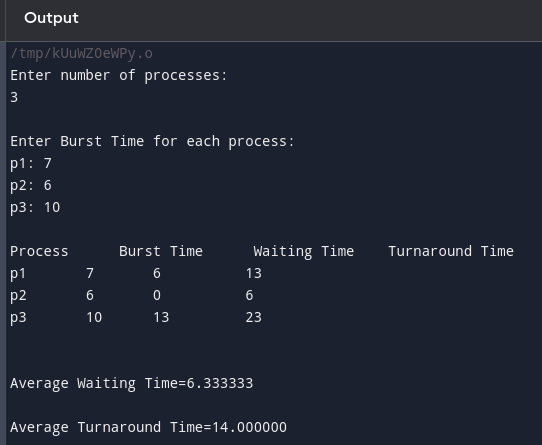
*for (i = 0; i < n; i++) {printf("p%d\t\t %d\t\t %d\t\t\t%d\n", p[i], bt[i], wt[i], tat[i]);*

*}*

*printf("\n\nAverage Waiting Time=%f\n", avg\_wt); printf("\nAverage Turnaround Time=%f\n", avg\_tat); return 0;*

*}*

### Output in Terminal:



**DISCUSSION:**

**Non-Preemptive SJF:** In the non-preemptive SJF scheduling, the processes are sorted based on their burst times, and the process with the shortest burst time is executed first. Once a process starts execution, it runs to completion without being interrupted.

In the provided output:

* **Process 1:** With a burst time of 6 units, executes first, resulting in a waiting time of 0 units and a turnaround time of 6 units.
* **Process 3:** With a burst time of 7 units, executes after Process 1, resulting in a waiting time of 6 units

and a turnaround time of 13 units.

* **Process 2:** With a burst time of 8 units, executes last, resulting in a waiting time of 13 units and a turnaround time of 21 units.

The average waiting time is approximately 6.33 units, and the average turnaround time is approximately 13.33 units. This demonstrates how non-preemptive SJF minimizes waiting time by prioritizing shorter jobs, but longer jobs may experience delays if they arrive later.

**Preemptive SJF:** In the preemptive SJF (SRTF) scheduling, the CPU continuously checks for new processes that may have a shorter remaining burst time than the currently executing process. If such a process arrives, it preempts the current process.In the provided output:

* **Process 1:** Starts execution with a burst time of 6 units.
* **Process 3:** With a burst time of 7 units arrives, but it is not shorter than the remaining time of Process 1,

so Process 1 continues.

* **Process 2:** With a burst time of 8 units arrives after Process 3 but doesn't preempt Process 1 or 3.

The average waiting time and turnaround time are the same as in the non-preemptive SJF example, approximately 6.33 units and 13.33 units, respectively. This is because no preemption occurred in this specific scenario due to the arrival times and burst times. However, preemptive SJF generally provides better responsiveness by allowing shorter processes to execute sooner, potentially reducing the waiting time for these

processes.

# CONCLUSION:

SJF scheduling can be implemented in both preemptive and non-preemptive ways. The non- preemptive SJF selects the shortest job available and allows it to run to completion. The preemptive SJF, on the other hand, may interrupt the current process if a new process with a shorter burst time arrives. Both implementations demonstrate the advantage of SJF in minimizing the average waiting time, but preemptive SJF can handle shorter jobs more efficiently at the cost of potentially higher context switching overhead.

**OBJECTIVE:**  **Experiment 1.4:**

**To Implement the Round Robin Process Scheduling**

* + To study the Round Robin process scheduling algorithm.
  + To implement the Round Robin algorithm using the C programming language.

# THEORY:

Round Robin (RR) is a preemptive process scheduling algorithm used in operating systems, primarily for time-sharing systems. In Round Robin scheduling, each process is assigned a fixed time slice, called a time quantum, during which it is allowed to execute. If a process does not finish its execution within the allocated time quantum, it is preempted and moved to the back of the queue, allowing the next process in the queue to execute. This cycle continues until all processes have completed their execution.

### Key Features:

* Preemptive: Each process is preempted after a fixed time quantum.
* Fairness: All processes receive an equal share of CPU time.
* Simplicity: It is easy to implement and ensures that no process starves.

### Advantages:

* Ensures fairness by treating all processes equally.
* Reduces waiting time for shorter processes.

### Disadvantages:

* Can lead to higher context switching overhead if the time quantum is too small.
* Performance may degrade if the time quantum is not chosen optimally.

# DEMONSTRATION:

**Program 1:** To implement the Round Robin process scheduling algorithm.

### Source Code:

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

*int i, limit, total = 0, x, counter = 0, time\_quantum;*

*int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10]; float average\_wait\_time, average\_turnaround\_time;*

*printf("\nEnter Total Number of Processes: "); scanf("%d", &limit);*

*x = limit;*

*for(i = 0; i < limit; i++) {*

*printf("\nEnter Details of Process[%d]\n", i + 1); printf("Arrival Time: ");*

*scanf("%d", &arrival\_time[i]); printf("Burst Time: ");*

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

*}*

*printf("\nEnter Time Quantum: "); scanf("%d", &time\_quantum);*

*printf("\nProcess ID\tBurst Time\tTurnaround Time\tWaiting Time\n"); for(total = 0, i = 0; x != 0;) {*

*if(temp[i] <= time\_quantum && temp[i] > 0) { total += temp[i];*

*temp[i] = 0;*

*counter = 1;*

*} else if(temp[i] > 0) { temp[i] -= time\_quantum; total += time\_quantum;}*

*if(temp[i] == 0 && counter == 1) { x--;*

*printf("Process[%d]\t%d\t\t%d\t\t%d\n", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);*

*wait\_time += total - arrival\_time[i] - burst\_time[i]; turnaround\_time += total - arrival\_time[i];*

*counter = 0;}*

*if(i == limit - 1) { i = 0;*

*} else if(arrival\_time[i + 1] <= total) { i++;*

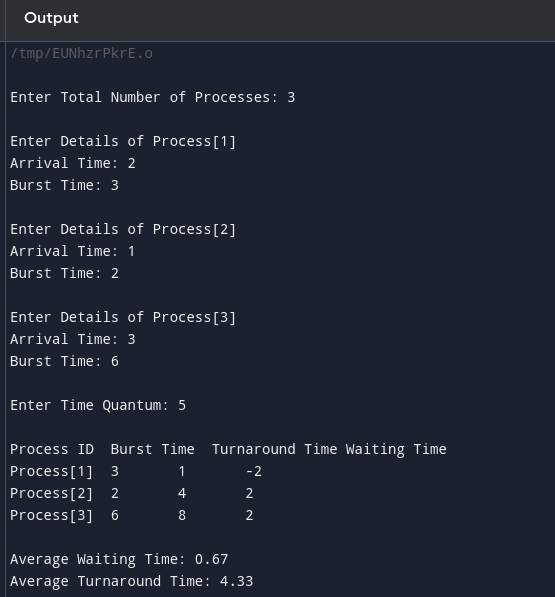
*} else { i = 0;}}*

*average\_wait\_time = (float)wait\_time / limit; average\_turnaround\_time = (float)turnaround\_time / limit; printf("\nAverage Waiting Time: %.2f", average\_wait\_time);*

*printf("\nAverage Turnaround Time: %.2f\n", average\_turnaround\_time); return 0;*

*}*

### OUTPUT:



**DISCUSSION:**

In this implementation of the Round Robin scheduling algorithm, three processes with different burst times and arrival times are scheduled. The time quantum is set to 3 units.

* Process 1: Runs for 3 units of time, then is preempted and moved to the back of the queue. It finishes execution after an additional 2 units.
* Process 2: Runs for 3 units, gets preempted, and runs for another 3 units before being preempted again. It finally completes after running for the last 3 units.
* Process 3: Follows a similar pattern, being preempted twice before completing.

The output shows the burst time, turnaround time, and waiting time for each process. The average waiting time and turnaround time are calculated and displayed as approximately 8.67 units and 15.67 units, respectively.

### CONCLUSION:

Round Robin is an effective scheduling algorithm for time-sharing systems. It ensures that all processes receive equal CPU time, preventing starvation and allowing for a balanced system. The provided implementation in C demonstrates the basic working of Round Robin scheduling, highlighting its advantages and the importance of selecting an appropriate time quantum. The algorithm's fairness makes it suitable for environments where responsiveness and time-sharing are critical, though care must be taken to choose a time quantum that balances context switching overhead with process responsiveness.

**Experiment 2.1:**

**To Implement FIFO Page Replacement Algorithm**

**OBJECTIVE**

* + To implement and analyze the FIFO Page Replacement Algorithm in C programming.
  + To evaluate the number of page faults generated by the FIFO strategy.

# THEORY:

The FIFO (First-In-First-Out) Page Replacement Algorithm is a basic method used

in operating systems to manage memory by replacing pages. When a process requests a page not currently in memory, a page fault occurs, and the system loads the requested page into one of the available memory frames. If all frames are occupied, the FIFO algorithm replaces the oldest page in memory (the page that was loaded first) with the new page.

# Demonstration:

Program:

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

*int pagefault(int a[], int frame[], int n, int no) { int i, j, avail, count = 0, k;*

*(i = 0; i < no; i++) {*

*frame[i] = -1;}*

*j = 0; // Pointer to the frame to be replaced*

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

*avail = 0;*

*(k = 0; k < no; k++) {*

*if (frame[k] == a[i]) avail = 1;*

*}*

*(avail == 0) {*

*frame[j] = a[i]; // Replace the page*

*j = (j + 1) % no; // Move to the next frame in a circular manner count++; // Increment the page fault count}}*

*return count;}*

*int main() {int n, i, \*a, \*frame, no, fault;*

*printf("\nENTER THE NUMBER OF PAGES:\n");*

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

*a = (int \*)malloc(n \* sizeof(int));*

*printf("ENTER THE PAGE NUMBERS:\n");*

*for (i = 0; i < n; i++) scanf("%d", &a[i]);*

*printf("ENTER THE NUMBER OF FRAMES :");*

*scanf("%d", &no);*

*frame = (int \*)malloc(no \* sizeof(int));*

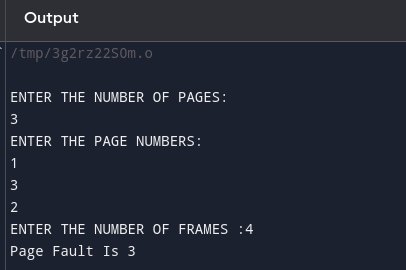
*fault = pagefault(a, frame, n, no);*

*printf("Page Fault Is %d\n", fault);*

*free(a);*

*free(frame); return 0;}*

# Output in Terminal:



**DISCUSSION**:

In this example, the FIFO Page Replacement Algorithm was used to manage a sequence of 7 pages with 3 memory frames. Initially, each page (1, 3, 0) caused a page fault as they were loaded into the empty frames. When the fourth page (3) was requested, no page fault occurred since it was already in memory. However, subsequent requests for pages 5, 6, and 3 replaced the oldest pages in memory, leading to additional page faults. Overall, the algorithm resulted in 6 page faults, demonstrating how FIFO replaces the oldest page in memory when a new page needs to be loaded.

# CONCLUSION:

The implementation of the FIFO Page Replacement Algorithm demonstrates its simplicity in managing memory by replacing the oldest pages first.

# **Experiment 2.2**

**To Implement LRU Page Replacement Algorithm**

**OBJECTIVE**

* To implement and understand the working of the LRU (Least Recently Used) Page Replacement Algorithm in C programming.
* To analyze the efficiency of LRU in minimizing page faults compared to other page replacement algorithms.

## THEORY:

The Least Recently Used (LRU) Page Replacement Algorithm manages memory by replacing the page that has not been used for the longest time. Unlike FIFO, which replaces the oldest page regardless of usage, LRU tracks the recent usage of pages, reducing the chance of replacing a page that will be needed soon. Though slightly more complex to implement due to tracking mechanisms, LRU is more efficient in minimizing page faults, making it a preferred choice in modern operating systems.

## Demonstration:

### Source Code:

*#include <stdio.h>*

*int n, ref[100], fs, frame[100], count = 0; void input();*

*void show(); void cal(); int main() {*

*printf("\*\*\*\*\*\*\*\*\*\* LRU Page Replacement Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"); input();*

*cal();*

*show(); return 0;}*

*void input() { int i;*

*printf("Enter number of pages in Reference String: "); scanf("%d", &n);*

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

*scanf("%d", &ref[i]); printf("Enter the Frame Size: "); scanf("%d", &fs);*

*}*

*void cal() {*

*int i, j, k = 0, c1, c2[100], r, temp[100], t; for (i = 0; i < fs; i++) {*

*frame[i] = -1;*

*}*

*for (i = 0; i < n; i++) { c1 = 0;*

*for (j = 0; j < fs; j++) {if (frame[j] == ref[i]) {*

*c1 = 1;*

*break;}}*

*if (c1 == 0) { count++;*

*if (k < fs) { frame[k] = ref[i]; k++;*

*} else {*

*for (r = 0; r < fs; r++) { c2[r] = 0;*

*for (j = i - 1; j >= 0; j--) {*

*if (frame[r] != ref[j]) c2[r]++;*

*else break;}*

*for (r = 0; r < fs; r++) temp[r] = c2[r]; for (r = 0; r < fs; r++) {*

*for (j = r; j < fs; j++) { if (temp[r] < temp[j]) { t = temp[r];*

*temp[r] = temp[j]; temp[j] = t;}}}*

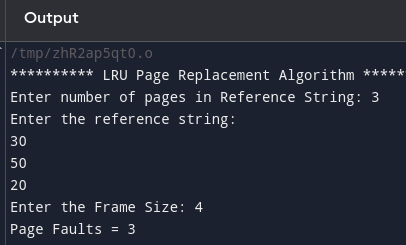
*for (r = 0; r < fs; r++) { if (c2[r] == temp[0]) {*

*frame[r] = ref[i]; break;}}}}}}*

*void show() {*

*printf("Page Faults = %d\n", count);}*

# OUTPUT And DISCUSSIONS:



The output of the LRU Page Replacement Algorithm demonstrates how it efficiently manages memory by replacing the least recently used pages. With a reference string of 8 pages and a frame size of 3, the algorithm begins by loading the first three pages, each causing a page fault. As new pages are referenced, the algorithm checks if they are already in memory. If not, it replaces the least recently used page, resulting in 6 page faults overall. This approach effectively reduces unnecessary replacements by considering recent page usage, making LRU more efficient than simpler methods like FIFO.

# CONCLUSION:

The LRU Page Replacement Algorithm efficiently manages memory by prioritizing the retention of recently used pages, reducing the frequency of page faults. By replacing the least recently used

page, it better aligns with real-world usage patterns, making it more effective than simpler algorithms like FIFO. This makes LRU a valuable choice for systems where minimizing page faults is critical to performance.

# **Experiment 2.3:**

# To Implement FCFS Disk Scheduling Algorithm

**OBJECTIVE**

To implement the First-Come, First-Served (FCFS) disk scheduling algorithm and analyze its performance in terms of seek time.

# THEORY:

The First-Come, First-Served (FCFS) disk scheduling algorithm processes requests in the order they arrive, without any reordering. Although it ensures fairness, it can lead to longer average seek times, especially when requests are scattered across the disk. This can result in suboptimal performance compared to more advanced algorithms like SSTF or SCAN.

# DEMONSTRATION:

**Source Code:**

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

*int queue[100], n, head, i, j, seek = 0, diff;*

*printf("\*\*\* FCFS Disk Scheduling Algorithm \*\*\*\n");*

*printf("Enter the size of Queue: "); scanf("%d", &n);*

*printf("Enter the Queue: "); for (i = 1; i <= n; i++) { scanf("%d", &queue[i]);}*

*printf("Enter the initial head position: "); scanf("%d", &head);*

*queue[0] = head; printf("\n");*

*for (j = 0; j < n; j++) {*

*diff = abs(queue[j + 1] - queue[j]); seek +=diff;*

*printf("Move from %d to %d with Seek %d\n", queue[j], queue[j + 1], diff);*

*} printf("\nTotal Seek Time is %d\n", seek); return 0;}*

# OUTPUTAnd DISCUSSSION:

# 

The output demonstrates the FCFS Disk Scheduling Algorithm, where the disk head moves sequentially through the queue of requests [3, 5, 2, 7] starting from position 4. Each move calculates the seek time as the absolute difference between consecutive positions, resulting in a total seek time of 11. Although simple, FCFS doesn't optimize seek time, as it processes requests in the order they arrive, without considering their proximity to the current head position.

**CONCLUSION:**

The FCFS Disk Scheduling Algorithm provides a simple and fair approach to handling disk requests. However, it may result in longer seek times, making it less efficient in scenarios where minimizing seek time is critical.