Dehi Technological University

Lab file

Operating system - CO -204

Submitted By:

Animesh Gupta (2K22/CO/60)

Submitted to:

Prof.Rajni Jindal

S. No.	Experiments	Page No.	Sign
1)	Introduction to Linux – Write a program to print Hello World		
2)	Write a program to implement Prims Algorithm using Disjoint Sets		
3)	Write a program to implement Shortest Job First (SJF) job scheduling algorithm.		
4)	Write a program to implement Shortest Remaining Time First (SRTF) job scheduling algorithm.		
5)	Write a program to implement First Come First Serve (FCFS) scheduling algorithm.		
6)	Write a program to implement Round Robin algorithm.		
7)	Write a program to implement priority scheduling algorithm.		
8)	Write a program to implement the Peterson's Solution.		
9)	Write a program to implement Banker's algorithm.		
10)	Write a program to implement Dekker's algorithm using Semaphore		
11)	Write a program to implement Reader and Writer Problem using Semaphore		
12)	Write a program to implement Optimal page replacement algorithm.		
13)	Write a program to implement Least Recently Used (LRU) page replacement algorithm.		
14)	Write a program to implement First In First Out (FIFO) page replacement algorithm.		

EXPERIMENT-1

Aim:-

Introduction to Linux – Write a program to print Hello World

Theory:-

Linux is a Unix-like operating system that was initially created by Linus Torvalds and first released on September 17, 1991. It has since become one of the most prominent examples of open-source software development and free software, as its underlying source code is freely available to the public and can be modified, distributed, and used by anyone.

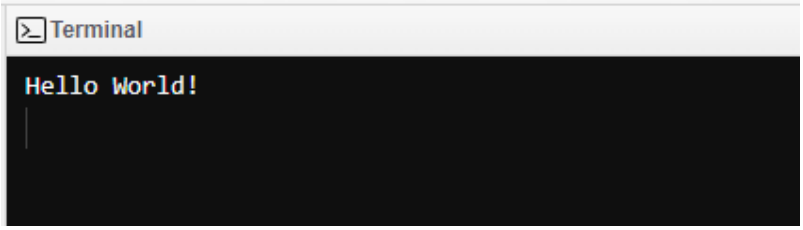
Some Linux Commands are:-

a) sudo: allows a user to execute a command with root/administrator privileges.
b) pwd: prints the current working directory.
c) cd: changes the current working directory.
d) ls: lists the files and directories in the current directory.
e) cat: displays the contents of a file on the terminal.
f) cp: copies a file or directory from one location to another.
g) mv: moves or renames a file or directory.
h) mkdir: creates a new directory.
i) rmdir: removes an empty directory.
j) rm: removes a file or directory (use with caution!).
k) touch: creates an empty file or updates the modification time of an existing file.
l) diff: compares two files and shows the differences.
m) tar: creates or extracts a compressed archive of files and directories.
n) find: searches for files and directories based on certain criteria.
o) grep: searches for a pattern or text string in a file or output.
p) df: displays the available disk space on the file system.
q) du: displays the disk usage of files and directories.
r) head: displays the first 10 lines of a file.
s) tail: displays the last 10 lines of a file.

Code:-

```
//To print "Hello World" in Linux, you can use the echo command in the terminal:

// Create a file named hello.sh and add the following code:

echo "Hello World"

// Save the file and give it execute permission

chmod +x hello.sh


//Run the script

./hello.sh
```

Output:-



EXPERIMENT-2

Aim:-

Write a program to implement Prims Algorithm using Disjoint Sets

Theory:-

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. The algorithm starts with an arbitrary node and grows the MST by adding the shortest edge that connects the tree to a vertex not yet in the tree. This process is repeated until all vertices are included in the MST. A disjoint set data structure is used to keep track of partitions or disjoint sets of elements. It provides two main operations:

- Union: Merge two sets.
- Find: Determine which set an element belongs to.

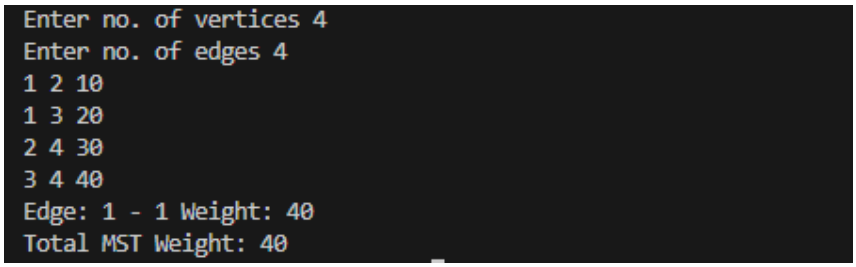
Code:-

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 10;

int parent[MAXN], rank_arr[MAXN];

int find(int x) {

   if (parent[x] == x) return x;


return parent[x] = find(parent[x]);

}

void union_sets(int x, int y) {

   x = find(x);

   y = find(y);
```

```
  if (x == y) return;
   if (rank_arr[x] < rank_arr[y]) swap(x, y);
   parent[y] = x;

 if (rank_arr[x] == rank_arr[y]) rank_arr[x]++;
}
int main() {
   int n, m;
   cout<<"Enter no. of vertices ";
   cin >>n;
   cout<<"Enter no. of edges ";
   cin>>m;
   for (int i = 1; i <= n; i++) {
      parent[i] = i;
      rank_arr[i] = 1;
   }
   int u, v, w;
   for (int i = 1; i <= m; i++) {
      cin >> u >> v >> w;
      union_sets(u, v);
   }
   int mst_weight = 0;
   for (int i = 1; i <= n; i++) {

  if (parent[i] == i) {
         mst_weight += w;
         cout << "Edge: " << i << " - " << parent[i] << " Weight: " << w << endl;
```

```
        }
    }
    cout << "Total MST Weight: " << mst_weight << endl;
    return 0;
}
```

Output:-



EXPERIMENT-3

Aim:-

Write a program to implement Shortest Job First (SJF) job scheduling algorithm.

Theory:-

The Shortest Job First (SJF) scheduling algorithm is a non-preemptive, preemptive, or priority scheduling algorithm where the process with the smallest execution time is selected for execution next. The main idea behind SJF scheduling is to minimize the average waiting time of processes. In SJF, once a process starts executing, it runs to completion. Processes are sorted based on their burst times (execution times). The process with the shortest burst time is selected for execution first.

Code:-

```
#include <iostream>

#include <algorithm>

#include <climits>

using namespace std;


struct Process {

        int processID; // Process ID


int burstTime; // Burst Time

        int arrivalTime; // Arrival Time

};


bool compareArrivalTime(Process a, Process b) {
```

```
    return a.arrivalTime < b.arrivalTime;
}

void findWaitingTime(Process proc[], int n, int waitingTime[]) {
    int remainingTime[n];

    for (int i = 0; i < n; i++)
        remainingTime[i] = proc[i].burstTime;

    int complete = 0, currentTime = 0, minBurst = INT_MAX;
    int shortestIndex = 0, finishTime;
    bool check = false;

    while (complete != n) {

        for (int j = 0; j < n; j++) {
            if ((proc[j].arrivalTime <= currentTime) &&
            (remainingTime[j] < minBurst) && remainingTime[j] > 0) {
                minBurst = remainingTime[j];
                shortestIndex = j;
                check = true;
            }
        }

        if (check == false) {
            currentTime++;
            continue;
        }
```

```
                remainingTime[shortestIndex]--;

                minBurst = remainingTime[shortestIndex];
                if (minBurst == 0)
minBurst = INT_MAX;
                if (remainingTime[shortestIndex] == 0) {
                        complete++;
                        check = false;

                        finishTime = currentTime + 1;

                        waitingTime[shortestIndex] = finishTime -
                                        proc[shortestIndex].burstTime -
                                        proc[shortestIndex].arrivalTime;

                        if (waitingTime[shortestIndex] < 0)
                                waitingTime[shortestIndex] = 0;
                }
                currentTime++;
        }
}

void findTurnAroundTime(Process proc[], int n, int waitingTime[], int turnAroundTime[]) {
        for (int i = 0; i < n; i++)
```

```
        turnAroundTime[i] = proc[i].burstTime + waitingTime[i];
}

void findAverageTime(Process proc[], int n) {
        int waitingTime[n], turnAroundTime[n], totalWaitingTime = 0, totalTurnaroundTime
= 0;

        findWaitingTime(proc, n, waitingTime);

        findTurnAroundTime(proc, n, waitingTime, turnAroundTime);

        cout << " P\t\t"
                << "BT\t\t"
                << "WT\t\t"
                << "TAT\t\t\n";

        for (int i = 0; i < n; i++) {
                totalWaitingTime = totalWaitingTime + waitingTime[i];
                totalTurnaroundTime = totalTurnaroundTime + turnAroundTime[i];
                cout << " " << proc[i].processID << "\t\t"
                        << proc[i].burstTime << "\t\t " << waitingTime[i]
                        << "\t\t " << turnAroundTime[i] << endl;
        }

        cout << "\nAverage waiting time = "
                << (float)totalWaitingTime / (float)n;
        cout << "\nAverage turn around time = "
                << (float)totalTurnaroundTime / (float)n;
```

```
}

int main() {
        Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },
                                        { 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
        int n = sizeof(proc) / sizeof(proc[0]);

        sort(proc, proc + n, compareArrivalTime);

        findAverageTime(proc, n);
        return 0;
}
```

OUTPUT:

```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o 1_sjf 1_sjf.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./1_sjf
 P              BT              WT              TAT
 4              3               0               3
 3              8               14              22
 1              6               7               13
 5              4               2               6
 2              2               0               2

Average waiting time = 4.6
Average turn around time = 9.2%
(base) animeshgupta@Animeshs-MacBook-Air OS Lab %
```

EXPERIMENT-4

Aim:-

Write a program to implement Shortest Remaining Time First (SRTF) job scheduling algorithm

Theory :

The Shortest Remaining Time First (SRTF) scheduling algorithm is a preemptive version of the Shortest Job First (SJF) scheduling algorithm. In SRTF, the process with the shortest remaining burst time is selected for execution. If a new process arrives with a shorter burst time than the currently executing process, the currently executing process is preempted. The scheduler selects the process with the shortest remaining burst time.The currently executing process can be preempted by a new arriving process with a shorter remaining burst time. Processes are sorted based on their remaining burst times.

Code:-

```
#include <bits/stdc++.h>

using namespace std;


struct Process {

    int processID; // Process ID

    int burstTime; // Burst Time

    int arrivalTime; // Arrival Time

};


void findWaitingTime(Process proc[], int n, int waitingTime[]) {

    int remainingTime[n];


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

        remainingTime[i] = proc[i].burstTime;


    int complete = 0, currentTime = 0, minBurst = INT_MAX;
```

```
int shortestIndex = 0, finishTime;



bool check = false;

        while (complete != n) {

                for (int j = 0; j < n; j++) {
                        if ((proc[j].arrivalTime <= currentTime) &&
                        (remainingTime[j] < minBurst) && remainingTime[j] > 0) {
                                minBurst = remainingTime[j];
                                shortestIndex = j;
                                check = true;
                        }
                }

                if (check == false) {
                        currentTime++;
                        continue;
                }

                remainingTime[shortestIndex]--;

                minBurst = remainingTime[shortestIndex];
                if (minBurst == 0)
                        minBurst = INT_MAX;
```

```
if (remainingTime[shortestIndex] == 0) {

                        complete++;
                        check = false;


finishTime = currentTime + 1;

                        waitingTime[shortestIndex] = finishTime -
                                                proc[shortestIndex].burstTime -
                                                proc[shortestIndex].arrivalTime;

                        if (waitingTime[shortestIndex] < 0)
                                waitingTime[shortestIndex] = 0;
                }
                currentTime++;
        }
}

void findTurnAroundTime(Process proc[], int n, int waitingTime[], int turnAroundTime[]) {
        for (int i = 0; i < n; i++)
                turnAroundTime[i] = proc[i].burstTime + waitingTime[i];
}

void findAverageTime(Process proc[], int n) {
        int waitingTime[n], turnAroundTime[n], totalWaitingTime = 0, totalTurnaroundTime
= 0;
```

```
findWaitingTime(proc, n, waitingTime);

        findTurnAroundTime(proc, n, waitingTime, turnAroundTime);

        cout << " P\t\t"
                << "BT\t\t"
                << "WT\t\t"
                << "TAT\t\t\n";

        for (int i = 0; i < n; i++) {
                totalWaitingTime = totalWaitingTime + waitingTime[i];
                totalTurnaroundTime = totalTurnaroundTime + turnAroundTime[i];
                cout << " " << proc[i].processID << "\t\t"
                        << proc[i].burstTime << "\t\t " << waitingTime[i]
                        << "\t\t " << turnAroundTime[i] << endl;
        }

        cout << "\nAverage waiting time = "
                << (float)totalWaitingTime / (float)n;
        cout << "\nAverage turn around time = "
                << (float)totalTurnaroundTime / (float)n;
}

int main() {
        Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },
                                        { 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
```

```
int n = sizeof(proc) / sizeof(proc[0]);

        findAverageTime(proc, n);
        return 0;
}
```

Output:-

```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o 2_srtf 2_srtf.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./2_Srtf
 P              BT              WT              TAT
 1              6               7               13
 2              2               0               2
 3              8               14              22
 4              3               0               3
 5              4               2               6

Average waiting time = 4.6
Average turn around time = 9.2%
```

EXPERIMENT-5

Aim:-

Write a program to implement First Come First Serve (FCFS) job scheduling algorithm

Theory:-

The First Come First Serve (FCFS) scheduling algorithm is a non-preemptive scheduling algorithm where processes are executed in the order they arrive. In FCFS, the process that arrives first gets executed first, and other processes wait in a queue. Once a process starts executing, it runs to completion. Processes are executed in the order they arrive, without considering their burst times.

Code:-

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Process {
   int process_id;
   int arrival_time;
   int burst_time;
};

bool compareArrivalTime(const Process &a, const Process &b) {
   return a.arrival_time < b.arrival_time;
}

void fcfs(vector<Process> &processes) {
   // Sort the processes based on arrival time
   sort(processes.begin(), processes.end(), compareArrivalTime);

   // Initialize variables
   int total_waiting_time = 0;
   int total_turnaround_time = 0;
   int current_time = 0;

   cout << "Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";

   for (const auto &p : processes) {
      // Calculate waiting time
      int waiting_time = max(0, current_time - p.arrival_time);
```

```
        // Calculate turnaround time
        int turnaround_time = waiting_time + p.burst_time;

        // Update total waiting and turnaround time
        total_waiting_time += waiting_time;
        total_turnaround_time += turnaround_time;

        // Update current time
        current_time += p.burst_time;

        // Print process details
        cout << p.process_id << "\t" << p.arrival_time << "\t\t" << p.burst_time << "\t\t" <<
waiting_time << "\t\t" << turnaround_time << "\n";
    }

    // Calculate average waiting and turnaround time
    double avg_waiting_time = static_cast<double>(total_waiting_time) / processes.size();
    double avg_turnaround_time = static_cast<double>(total_turnaround_time) /
processes.size();

    cout << "\nAverage Waiting Time: " << avg_waiting_time << endl;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
}

int main() {
    // Example processes: (process_id, arrival_time, burst_time)
    vector<Process> processes = {
        {1, 0, 7},
        {2, 2, 4},
        {3, 4, 1},
        {4, 5, 4},
        {5, 6, 3}
    };

    fcfs(processes);

    return 0;
}
```

OUTPUT:

```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o fcfs fcfs.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./fcfs
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               7               0               7
2       2               4               5               9
3       4               1               7               8
4       5               4               7               11
5       6               3               10              13

Average Waiting Time: 5.8
Average Turnaround Time: 9.6
```

EXPERIMENT-6

Aim:-

Write a program to implement Round Robin Scheduling Algorithm

Theory:-

The Round Robin (RR) scheduling algorithm is a preemptive scheduling algorithm where each process is assigned a fixed time quantum or time slice. Processes are executed in a circular manner, and if a process does not complete within its time quantum, it is moved to the end of the queue.Each process is executed for a fixed time quantum. If a process completes before its time quantum expires, it is removed from the queue. If a process does not complete within its time quantum, it is moved to the end of the queue. Processes are executed in a circular manner until all processes are completed.

Code:-

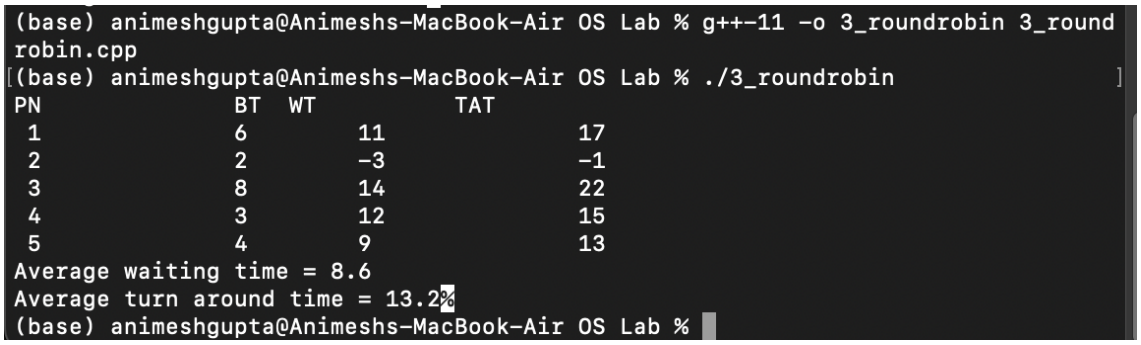
```
#include <iostream>

using namespace std;


struct Process {

        int processID; // Process ID

        int burstTime; // Burst Time

        int arrivalTime; // Arrival Time

};


void calculateWaitingTime(Process proc[], int n, int waitingTime[], int timeQuantum) {

        int remainingBurstTime[n];


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

                remainingBurstTime[i] = proc[i].burstTime;


        int currentTime = 0;
```

```
while (true) {

bool allProcessesDone = true;

        for (int i = 0 ; i < n; i++) {
            if (remainingBurstTime[i] > 0) {
                allProcessesDone = false;

                if (remainingBurstTime[i] > timeQuantum) {
                    currentTime += timeQuantum;
                    remainingBurstTime[i] -= timeQuantum;
                }
                else {
                    currentTime = currentTime + remainingBurstTime[i];
                    waitingTime[i] = currentTime - proc[i].burstTime -
proc[i].arrivalTime;
                    remainingBurstTime[i] = 0;
                }
            }
        }

        if (allProcessesDone)
            break;
    }
}

void calculateTurnAroundTime(Process proc[], int n, int waitingTime[], int
turnAroundTime[]) {
```

```
for (int i = 0; i < n ; i++)
		turnAroundTime[i] = proc[i].burstTime + waitingTime[i];
}

void calculateAverageTime(Process proc[], int n, int timeQuantum) {
	int waitingTime[n], turnAroundTime[n], totalWaitingTime = 0, totalTurnaroundTime
= 0;

	calculateWaitingTime(proc, n, waitingTime, timeQuantum);

	calculateTurnAroundTime(proc, n, waitingTime, turnAroundTime);

	cout << "PN\t " << "\tBT " << " WT " << " \tTAT\n";

	for (int i=0; i<n; i++) {
		totalWaitingTime = totalWaitingTime + waitingTime[i];
		totalTurnaroundTime = totalTurnaroundTime + turnAroundTime[i];
		cout << " " << proc[i].processID << "\t\t" << proc[i].burstTime <<"\t " <<
waitingTime[i] <<"\t\t " << turnAroundTime[i] <<endl;
	}

	cout << "Average waiting time = " << (float)totalWaitingTime / (float)n;
	cout << "\nAverage turn around time = " << (float)totalTurnaroundTime / (float)n;
}

int main() {
	Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 }, { 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
	int n = sizeof(proc) / sizeof(proc[0]);
```

```
    int timeQuantum = 2;
    calculateAverageTime(proc, n, timeQuantum);
    return 0;
}
```

Output:-



```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o 3_roundrobin 3_round
robin.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./3_roundrobin
PN              BT  WT          TAT
 1              6        11              17
 2              2        -3              -1
 3              8        14              22
 4              3        12              15
 5              4        9               13
Average waiting time = 8.6
Average turn around time = 13.2%
(base) animeshgupta@Animeshs-MacBook-Air OS Lab %
```

EXPERIMENT-7

Aim:-

Write a program to implement priority scheduling algorithm.

Theory:

The Priority Scheduling algorithm is a non-preemptive scheduling algorithm where each process is assigned a priority. The process with the highest priority is executed first. If two processes have the same priority, then they are scheduled in a First Come First Serve (FCFS) manner.Once a process starts executing, it runs to completion.Processes are sorted based on their priorities. The process with the highest priority is selected for execution first. Priority scheduling can lead to starvation of lower priority processes if higher priority processes continuously arrive.

Code:-

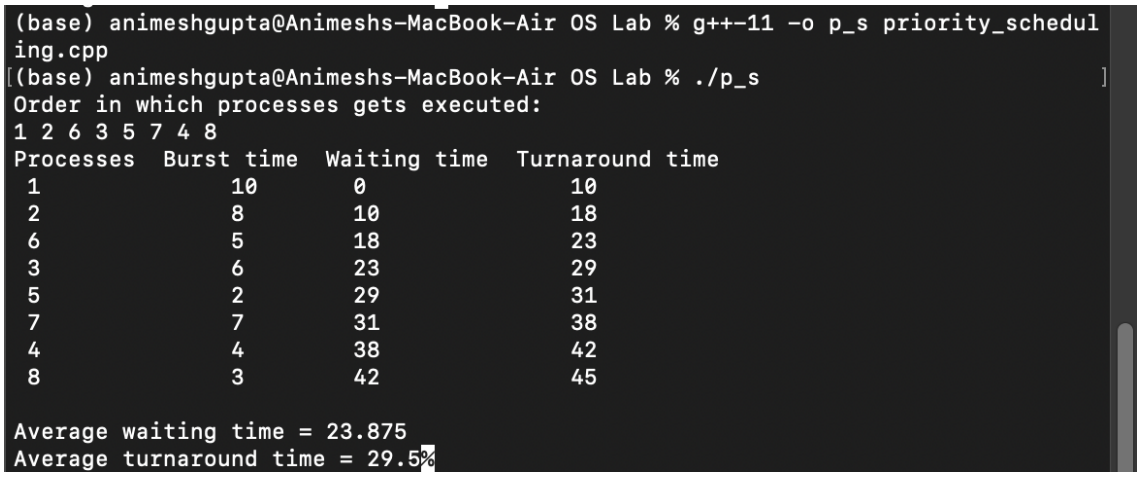
```
#include <iostream>

#include <algorithm>

#include <vector>


using namespace std;


// Structure to represent a process

struct Process {

   int pid;       // Process ID

   int burst_time; // CPU Burst time required

   int priority;   // Priority of this process

};


// Function to sort the Process according to priority

bool comparePriority(Process a, Process b) {

   return (a.priority > b.priority);

}
```

```
// Function to find the waiting time for all processes
void calculateWaitingTime(const vector<Process>& processes, vector<int>& waiting_times)
{
    waiting_times[0] = 0; // Waiting time for the first process is 0

    // Calculate waiting time for subsequent processes
    for (size_t i = 1; i < processes.size(); i++) {
        waiting_times[i] = processes[i - 1].burst_time + waiting_times[i - 1];
    }
}

// Function to calculate turnaround time
void calculateTurnaroundTime(const vector<Process>& processes, const vector<int>& waiting_times, vector<int>& turnaround_times) {
    // Calculate turnaround time by adding burst_time and waiting_time
    for (size_t i = 0; i < processes.size(); i++) {
        turnaround_times[i] = processes[i].burst_time + waiting_times[i];
    }
}

// Function to calculate average time
void calculateAverageTime(const vector<Process>& processes) {
    vector<int> waiting_times(processes.size(), 0);
    vector<int> turnaround_times(processes.size(), 0);
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    // Calculate waiting and turnaround times
    calculateWaitingTime(processes, waiting_times);
    calculateTurnaroundTime(processes, waiting_times, turnaround_times);
```

```
    // Display processes along with all details
    cout << "\nProcesses " << " Burst time " << " Waiting time " << " Turnaround time\n";

    // Calculate total waiting time and total turnaround time
    for (size_t i = 0; i < processes.size(); i++) {
        total_waiting_time += waiting_times[i];
        total_turnaround_time += turnaround_times[i];
        cout << " " << processes[i].pid << "\t\t" << processes[i].burst_time << "\t "
             << waiting_times[i] << "\t\t " << turnaround_times[i] << endl;
    }

    // Calculate and display average waiting time and average turnaround time
    cout << "\nAverage waiting time = " << (float)total_waiting_time / processes.size();
    cout << "\nAverage turnaround time = " << (float)total_turnaround_time / processes.size();
}

// Priority Scheduling Algorithm
void priorityScheduling(vector<Process>& processes) {
    // Sort processes by priority
    sort(processes.begin(), processes.end(), comparePriority);

    cout << "Order in which processes gets executed:\n";
    for (size_t i = 0; i < processes.size(); i++) {
        cout << processes[i].pid << " ";
    }

    calculateAverageTime(processes);
}
```

```
int main() {
    // Example processes
vector<Process> processes = {
        { 1, 10, 3 },
        { 2, 8, 2 },
        { 3, 6, 1 },
        { 4, 4, 0 },
        { 5, 2, 1 },
        { 6, 5, 2 },
        { 7, 7, 1 },
        { 8, 3, 0 }
    };
    // Number of processes
    int n = processes.size();


    // Priority Scheduling Algorithm
    priorityScheduling(processes);


    return 0;
}
```

OUTPUT:



EXPERIMENT-8

Aim:-

Write a program to implement the Peterson’s Solution.

Theory:-

Peterson's Solution is a classic algorithm used to synchronize concurrent processes in computer science, primarily in operating systems. It provides a method for two processes to safely access a shared resource without interference.

Code:-

```
#include <iostream>

#include <thread>

#include <atomic>


using namespace std;


#define NUM_THREADS 2


atomic<int> turn;

atomic<bool> flag[NUM_THREADS];


void critical_section(int id) {

    // Peterson's solution entry section

    int other = 1 - id;

    flag[id] = true;

    turn = other;


    // Busy-wait until it's our turn

    while (flag[other] && turn == other)
```

```
        ;

    // Critical section
    cout << "Thread " << id << " is in the critical section.\n";

    // Peterson's solution exit section
    flag[id] = false;
}

void thread_function(int id) {
    // Simulate some non-critical work
    this_thread::sleep_for(chrono::milliseconds(100 * id));

    // Enter critical section
    critical_section(id);

    // Simulate some more non-critical work
    this_thread::sleep_for(chrono::milliseconds(100 * id));

    // Exit thread
    cout << "Thread " << id << " exited.\n";
}

int main() {
    // Initialize flags
    flag[0] = false;
    flag[1] = false;
```

```
    // Create threads
    thread t[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; ++i) {
        t[i] = thread(thread_function, i);
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; ++i) {
        t[i].join();
    }

    return 0;
}
```

Output:-

```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o ps 4_peterson_soluti
on.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./ps
Thread 0 is in the critical section.
Thread 0 exited.
Thread 1 is in the critical section.
Thread 1 exited.
(base) animeshgupta@Animeshs-MacBook-Air OS Lab %
```

EXPERIMENT-9

Aim:-

Write a program to implement Banker’s algorithm.

Theory:-

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems to ensure that the system can allocate resources to processes in a safe manner without causing deadlocks. The algorithm works by simulating the resource allocation process and checking for safety before granting the resources to a process. Each process declares the maximum number of resources of each type it may need. The system maintains the number of available resources of each type. The system also maintains the number of resources allocated to each process. The algorithm checks if granting the requested resources to a process can lead to a safe state or not. A state is considered safe if there exists a sequence of processes such that each process can obtain its maximum resources and terminate, allowing the next process to complete. The Banker's Algorithm is a practical algorithm used in real-world operating systems to prevent deadlocks and ensure safe resource allocation.

Code:-

```
#include <iostream>

using namespace std;


// Function to check if a process can finish with current available resources

bool canFinishProcess(int process, int need[][4], int available[], int n, int m) {

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

    if (need[process][j] > available[j]) {

      return false; // Process needs more resources than available

    }

  }

  return true;

}


int main() {

  // Number of processes and resources
```

```
int n = 5;  // Processes (P0, P1, P2, P3, P4)
int m = 4;  // Resources (R0, R1, R2, R3)

// Allocation matrix (represents resources currently held by each process)
int allocation[5][4] = {
  {3, 1, 2, 1},  // P0
  {2, 0, 0, 2},  // P1
  {1, 3, 2, 1},  // P2
  {2, 1, 1, 0},  // P3
  {0, 0, 2, 0}   // P4
};

// Maximum matrix (represents maximum resource needs of each process)
int max_need[5][4] = {
  {7, 5, 3, 4},  // P0
  {4, 2, 2, 2},  // P1
  {5, 4, 2, 2},  // P2
  {2, 2, 2, 1},  // P3
  {4, 3, 3, 2}   // P4
};

// Available resources (initially available)
int available[4] = {3, 3, 2, 1};

// Finished processes (stores process IDs in safe sequence)
int finished[n] = {0};
```

```
// Number of finished processes
int num_finished = 0;

// Loop until all processes are finished or a deadlock is detected
while (num_finished < n) {
  int safe = 0; // Flag to check if any process can finish in this iteration

  // Check for each process if it can finish with current available resources
  for (int i = 0; i < n; i++) {
    if (finished[i] == 0 && canFinishProcess(i, max_need, available, n, m)) {
      // Process can finish, update resources and mark as finished
      for (int j = 0; j < m; j++) {
        available[j] += allocation[i][j];
      }
      finished[i] = 1;
      safe = 1;
    }
  }

  // If no process can finish, deadlock detected
  if (safe == 0) {
    cout << "Deadlock detected. System is in unsafe state." << endl;
    break;
  }

  num_finished++;  // Increment finished process count
}
```

```
  // If all processes finished, print the safe sequence
  if (num_finished == n) {
    cout << "Following is the SAFE Sequence" << endl;
    for (int i = 0; i < n - 1; i++) {
      cout << " P" << i << " ->";
    }
    cout << " P" << n - 1 << endl;
  }

  return 0;
}
```

Output:-

```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o ba 6_banker_algo.cpp

(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./ba
Deadlock detected. System is in unsafe state.
```

EXPERIMENT-10

Aim:-

Write a program to implement Dekker’s algorithm using Semaphore

Theory:-

Dekker's Algorithm is one of the earliest known solutions to the mutual exclusion problem in concurrent programming. It allows two processes to share a single resource without conflict. Dekker's Algorithm ensures mutual exclusion by using two flags (one for each process) and a turn variable to control access to the critical section.

Code:-

```
#include <iostream>

using namespace std;

int main()

{

int incomingStream[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0,

3, 2, 1};

int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

int temp[frames];

for(m = 0; m < frames; m++)

{

temp[m] = -1;

}

for(m = 0; m < pages; m++)

{

s = 0;

for(n = 0; n < frames; n++)
```

```
{
if(incomingStream[m] == temp[n])
{
   s++;
pageFaults--;
}
}
pageFaults++;
if((pageFaults <= frames) && (s == 0))
{
temp[m] = incomingStream[m];
}
else if(s == 0)
{
temp[(pageFaults - 1) % frames] =
incomingStream[m];
}
cout << "\n";
cout << incomingStream[m] << "\t\t\t";
for(n = 0; n < frames; n++)
{
if(temp[n] != -1)
cout << temp[n] << "\t\t\t";
else
cout << "- \t\t\t";
}
}
cout << "\n\nTotal Page Faults:\t" << pageFaults;
cout << "\nTotal Hits :\t" << pages - pageFaults;
```

return 0;

}

Output:-

```
PS C:\Users\Zarv!5\Desktop\OS> cd "c:\Users\Zarv!5\Desktop\OS\" ; if ($?) { g++ e
 exp10 } ; if ($?) { .\exp10 }
Incoming        Frame 1         Frame 2         Frame 3
7                       7                       -                       -
0                       7                       0                       -
1                       7                       0                       1
2                       2                       0                       1
0                       2                       0                       1
3                       2                       3                       1
0                       2                       3                       0
4                       4                       3                       0
2                       4                       2                       0
3                       4                       2                       3
0                       0                       2                       3
3                       0                       2                       3
2                       0                       2                       3
1                       0                       1                       3

Total Page Faults:      11
Total Hits :    3
PS C:\Users\Zarv!5\Desktop\OS>
```

EXPERIMENT-11

Aim:-

Write a program to implement Reader and Writer Problem using Semaphore

Theory:

The Reader-Writer Problem is a classical synchronization problem in concurrent programming. The problem involves multiple readers and writers accessing a shared resource. The constraints are as follows:

1. Multiple readers can read the shared resource simultaneously.

2. Only one writer can write to the shared resource at a time.

3. Readers and writers cannot access the shared resource simultaneously.

To implement the Reader-Writer Problem using semaphores, we can use two semaphores:

- `mutex`: A binary semaphore to control access to the `read_count` variable and ensure mutual exclusion.
- `write`: A binary semaphore to control access to the shared resource and ensure that only one writer can write at a time.

- The `read` method simulates the behavior of a reader. It acquires the `mutex` to increment `read_count`, reads from the shared resource, and then releases the `mutex` after decrementing `read_count`.
- The `write` method simulates the behavior of a writer. It acquires the `write` semaphore to block other writers and readers, writes to the shared resource, and then releases the `write` semaphore.
- Multiple reader and writer threads are created and started to simulate concurrent access to the shared resource.

Code:-

```
#include <iostream>

#include <thread>

#include <mutex>

#include <vector>

#include <chrono>


// Constants defining the number of reader and writer threads
```

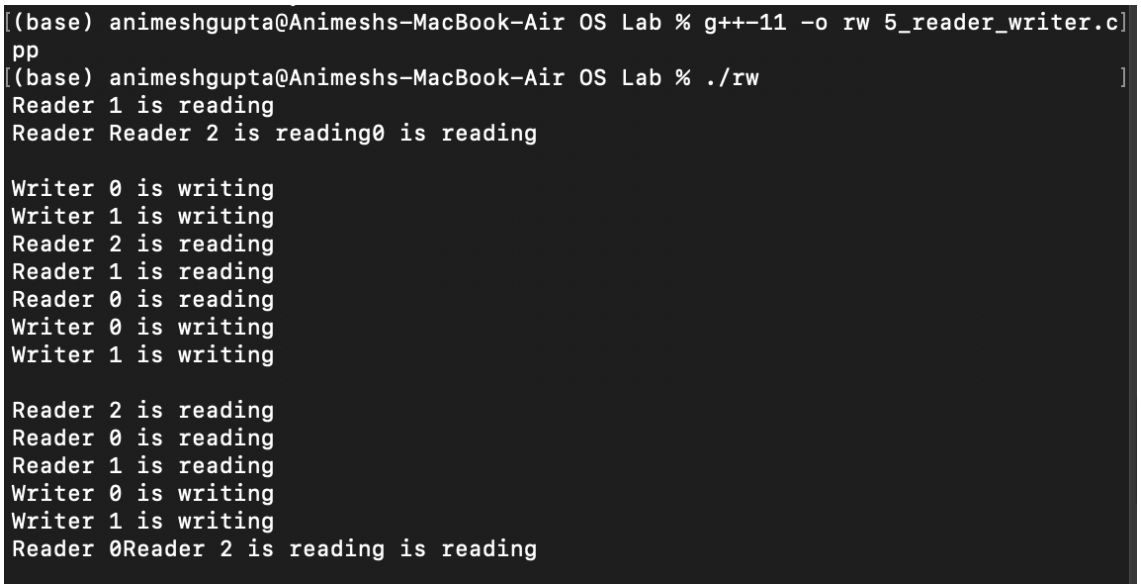
```
const int NUM_READERS = 3;
const int NUM_WRITERS = 2;

// Mutex for protecting the critical section during writes
std::mutex mtx;

// Mutex for reader count and ensuring first reader acquires write lock
std::mutex r_lock;

// Variable to track the number of readers currently reading
int readers_count = 0;

// Function for the reader threads
void reader(int readerID) {
  while (true) {
    // Begin reading section (acquire read lock)
    r_lock.lock();
    readers_count++;
    if (readers_count == 1) {  // First reader acquires write lock
      mtx.lock();
    }
    r_lock.unlock();

    // Simulate reading process
    std::cout << "Reader " << readerID << " is reading" << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
```

```
  // End reading section (release read lock and potentially write lock)
    r_lock.lock();
    readers_count--;
    if (readers_count == 0) {  // Last reader releases write lock
      mtx.unlock();
    }
    r_lock.unlock();

    // Additional actions after reading (sleep for simulation)
    std::this_thread::sleep_for(std::chrono::seconds(1));
  }
}

// Function for the writer threads
void writer(int writerID) {
  while (true) {
    // Begin writing section (acquire write lock)
    mtx.lock();

    // Simulate writing process
    std::cout << "Writer " << writerID << " is writing" << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // End writing section (release write lock)
    mtx.unlock();

    // Additional actions after writing (sleep for simulation)
```

```
 std::this_thread::sleep_for(std::chrono::seconds(1));
  }
}

int main() {
  // Vector to store reader threads
  std::vector<std::thread> reader_threads;

  // Vector to store writer threads
  std::vector<std::thread> writer_threads;

  // Create reader threads
  for (int i = 0; i < NUM_READERS; ++i) {
    reader_threads.push_back(std::thread(reader, i));
  }

  // Create writer threads
  for (int i = 0; i < NUM_WRITERS; ++i) {
    writer_threads.push_back(std::thread(writer, i));
  }

  // Wait for reader threads to finish
  for (auto& thread : reader_threads) {
    thread.join();
  }

  // Wait for writer threads to finish
```

```
  for (auto& thread : writer_threads) {
    thread.join();
  }

  return 0;
}
```

OUTPUT:-



```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o rw 5_reader_writer.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./rw
Reader 1 is reading
Reader Reader 2 is reading0 is reading

Writer 0 is writing
Writer 1 is writing
Reader 2 is reading
Reader 1 is reading
Reader 0 is reading
Writer 0 is writing
Writer 1 is writing

Reader 2 is reading
Reader 0 is reading
Reader 1 is reading
Writer 0 is writing
Writer 1 is writing
Reader 0Reader 2 is reading is reading
```

EXPERIMENT-12

Aim:-

Write a program to implement Optimal page replacement algorithm.

Theory:-

The Optimal Page Replacement Algorithm, also known as the Belady's Algorithm, is an algorithm used in virtual memory management. This algorithm replaces the page that will not be used for the longest period of time in the future. When a page needs to be replaced, the algorithm selects the page that will not be used for the longest period of time in the future. It requires future knowledge of the pages that will be accessed, which is not practical in a real system. Optimal page replacement is used as a theoretical benchmark to evaluate other page replacement algorithms. The Optimal Page Replacement Algorithm is a theoretical algorithm used to evaluate the performance of other page replacement algorithms.

Code:-

```
#include <stdio.h>

#include <stdlib.h>



#define MAX_FRAMES 50

#define MAX_REFERENCE_STR 50



int num_frames, num_references, page_faults = 0;

int reference[MAX_REFERENCE_STR], frames[MAX_FRAMES],
opt_calculations[MAX_FRAMES], recent[10], count = 0;



// Function to find the victim page using the Optimal algorithm

int get_optimal_victim(int index);



int main() {

   printf("\nOPTIMAL PAGE REPLACEMENT ALGORITHM\n");
```

```
// Input number of frames and reference string
    printf("Enter the number of frames: ");
    scanf("%d", &num_frames);
    printf("Enter the size of reference string: ");
    scanf("%d", &num_references);
    printf("Enter the reference string (separated by space): ");
    for (int i = 0; i < num_references; i++)
        scanf("%d", &reference[i]);

    printf("\nOPTIMAL PAGE REPLACEMENT ALGORITHM\n");
    printf("\nReference String\tPage Frames\n");

    // Initialize arrays
    for (int i = 0; i < num_frames; i++) {
        frames[i] = -1;
        opt_calculations[i] = 0;
    }

    for (int i = 0; i < 10; i++)
        recent[i] = 0;

    // Process the reference string
    for (int i = 0; i < num_references; i++) {
        int flag = 0;
        printf("\n%d\t\t\t\t", reference[i]);

        // Check if page already in frames
```

```
    for (int j = 0; j < num_frames; j++) {
            if (frames[j] == reference[i]) {
                flag = 1;
                break;
            }
        }

        // Page fault handling
        if (flag == 0) {
            page_faults++;

            // Find victim page using Optimal algorithm
            int victim = get_optimal_victim(i);
            frames[victim] = reference[i];

            // Display current page frames
            for (int j = 0; j < num_frames; j++)
                printf("%4d", frames[j]);
        }
    }

    // Print total number of page faults
    printf("\n\nNumber of page faults: %d\n", page_faults);
    return 0;
}

// Function to find the victim page using the Optimal algorithm
```

```
int get_optimal_victim(int index) {
    int not_found, temp;
    for (int i = 0; i < num_frames; i++) {
        not_found = 1;
        for (int j = index; j < num_references; j++) {
            if (frames[i] == reference[j]) {
                not_found = 0;
                opt_calculations[i] = j;
                break;
        if (not_found == 1)
            return i;
    temp = opt_calculations[0];
    for (int i = 1; i < num_frames; i++) {
        if (temp < opt_calculations[i])
            temp = opt_calculations[i];
    for (int i = 0; i < num_frames; i++)
        if (frames[temp] == frames[i])
            return i;
    return 0;
```

Output:-

```
OPTIMAL PAGE REPLACEMENT ALGORITHM
Enter the number of frames: 3
Enter the size of reference string: 3
Enter the reference string (separated by space): 1 2 3

OPTIMAL PAGE REPLACEMENT ALGORITHM

Reference String        Page Frames

1                                   1  -1  -1
2                                   2  -1  -1
3                                   3  -1  -1

Number of page faults: 3
(base) animeshgupta@Animeshs-MacBook-Air OS Lab %
```

EXPERIMENT-13

Aim:-

Write a program to implement Least Recently Used (LRU) page replacement algorithm.

Theory:-

The Least Recently Used (LRU) Page Replacement Algorithm is a commonly used algorithm in virtual memory management. It replaces the least recently used page when a new page needs to be brought into memory. The algorithm replaces the page that has not been accessed for the longest period of time. It uses a data structure like a queue or a doubly linked list to keep track of the order in which pages are accessed. When a page is accessed, it is moved to the front of the queue or the head of the linked list. When a page needs to be replaced, the page at the end of the queue or the tail of the linked list (which is the least recently used page) is replaced.

Code:-

```
#include<stdio.h>


int frames_count, references_count, page_faults = 0;

int ref[50], frames[50], recent[10], lru_calculations[50];


// Function to get the victim page using LRU

int get_lru_victim();


int main() {

   printf("\n\t\t\t LRU PAGE REPLACEMENT ALGORITHM");


   // Input number of frames and reference string

   printf("\nEnter the number of frames: ");

   scanf("%d", &frames_count);

   printf("Enter the size of reference string: ");

   scanf("%d", &references_count);

   printf("Enter the reference string (separated by space): ");
```

```
for (int i = 0; i < references_count; i++)
        scanf("%d", &ref[i]);

    // Initialize arrays
    for (int i = 1; i <= frames_count; i++) {
        frames[i] = -1;
        lru_calculations[i] = 0;
    }

    for (int i = 0; i < 10; i++)
        recent[i] = 0;

    // Process the reference string
    printf("\n\n\t\t LRU PAGE REPLACEMENT ALGORITHM\n");
    printf("\nReference String\t\tPage Frames\n");

    for (int i = 0; i < references_count; i++) {
        int flag = 0;
        printf("\n\t %d\t \t \t \t ", ref[i]);

        // Check if page already in frames
        for (int j = 0; j < frames_count; j++) {
            if (frames[j] == ref[i]) {
                flag = 1;
                break;
            }
        }
```

```
        // Page fault handling
        if (flag == 0) {
            page_faults++;

            // Find victim page using LRU
            int victim = get_lru_victim();
            frames[victim] = ref[i];

            // Update recent array
            recent[ref[i]] = i;

            // Display current page frames
            for (int j = 0; j < frames_count; j++)
                printf("%4d", frames[j]);
        }
    }

    // Print total number of page faults
    printf("\n\n\t No. of page faults: %d\n", page_faults);
    return 0;
}

// Function to get the victim page using LRU
int get_lru_victim() {
    int temp1, temp2;

    // Calculate LRU for each frame
```

```
for (int i = 0; i < frames_count; i++) {
        temp1 = frames[i];
        lru_calculations[i] = recent[temp1];
    }

    // Find the page with minimum recent value (LRU)
    temp2 = lru_calculations[0];
    for (int j = 1; j < frames_count; j++) {
        if (temp2 > lru_calculations[j]) {
            temp2 = lru_calculations[j];
        }
    }

    // Find the victim page index
    for (int i = 0; i < frames_count; i++) {
        if (ref[temp2] == frames[i]) {
            return i;
        }
    }
    return 0;
}
```

Output:-

```
                        LRU PAGE REPLACEMENT ALGORITHM
Enter the number of frames: 3
Enter the size of reference string: 3
Enter the reference string (separated by space): 12 345 65


                LRU PAGE REPLACEMENT ALGORITHM

Reference String               Page Frames

        12                                12  -1  -1
        345                              345  -1  -1
        65                                65  -1  -1

        No. of page faults: 3
```

EXPERIMENT-14

Aim:-

Write a program to implement First In First Out (FIFO) page replacement algorithm.

Theory:-

The First In First Out (FIFO) Page Replacement Algorithm is one of the simplest page replacement algorithms. It replaces the oldest page in the memory when a new page needs to be brought in. The algorithm maintains a queue to keep track of the order in which pages are loaded into memory. When a page needs to be replaced, the page at the front of the queue (the oldest page) is replaced. New pages are added to the end of the queue. This algorithm does not consider the frequency of page usage. The FIFO Page Replacement Algorithm is a simple and commonly used algorithm in virtual memory management.

Code:-

```
#include <stdio.h>


int main() {
  const int MAX_FRAMES = 50;
  int frames[MAX_FRAMES];
  int reference[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
  int num_frames = 3;
  int num_references = sizeof(reference) / sizeof(reference[0]);
```

```
int num_page_faults = 0;
 int frame_index = 0;

 printf("FIFO Page Replacement Algorithm\n\n");

 printf("Reference String\tPage Frames\n");

 for (int i = 0; i < num_references; i++) {
  int page_found = 0;
  int page = reference[i];

  for (int j = 0; j < num_frames; j++) {
   if (frames[j] == page) {
    page_found = 1;
    break;
   }
  }

  if (!page_found) {
   frames[frame_index] = page;
   frame_index = (frame_index + 1) % num_frames;
   num_page_faults++;
  }

  printf("%4d\t\t", page);
  for (int j = 0; j < num_frames; j++) {
   printf("%4d", frames[j]);
```

```
  }
    printf("\n");
  }

  printf("\nTotal number of page faults: %d\n", num_page_faults);

  return 0;
}
```

Output:-x

```
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % g++-11 -o ff 7_fifo_page_rep.cpp
(base) animeshgupta@Animeshs-MacBook-Air OS Lab % ./ff
FIFO Page Replacement Algorithm

Reference String        Page Frames
   1              1   0   0
   2              1   2   0
   3              1   2   3
   4              4   2   3
   1              4   1   3
   2              4   1   2
   5              5   1   2
   1              5   1   2
   2              5   1   2
   3              5   3   2
   4              5   3   4
   5              5   3   4

Total number of page faults: 9
```