

**DEPARTMENT OF COMPUTER SCIENCE AND  
ENGINEERING**

**OPERATING SYSTEMS**



**CO-204 LAB FILE**

**SUBMITTED TO:**

Ms. Rajni Jindal

**SUBMITTED BY:**

Ansh Aneja

2K22/CO/68

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 create a child process using fork() system call.		
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"
```

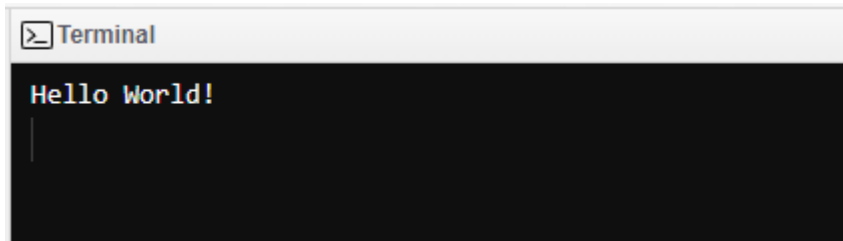
// Save the file and give it execute permission

```
chmod +x hello.sh
```

//Run the script

./hello.sh

### **Output:-**



## **EXPERIMENT-2**

### **Aim:-**

Write a program to implement Prim's 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:-**

```

$ ./"osfile.exe"
Enter no. of vertices 4
Enter no. of edges 4
1 2 5
1 3 10
2 4 20
3 4 40
Edge: 1 - 1 Weight: 40
Total MST Weight: 40

```

## **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>
using namespace std;

int main() {

    // Matrix for storing Process Id, Burst

```

```

// Time, Average Waiting Time & Average
// Turn Around Time.
int A[100][4];
int i, j, n, total = 0, index, temp;
float avg_wt, avg_tat;

cout << "Enter number of process: ";
cin >> n;

cout << "Enter Burst Time:" << endl;

// User Input Burst Time and allotting Process Id.
for (i = 0; i < n; i++) {
    cout << "P" << i + 1 << ": ";
    cin >> A[i][1];
    A[i][0] = i + 1;
}

// Sorting process according to their Burst Time.
for (i = 0; i < n; i++) {
    index = i;
    for (j = i + 1; j < n; j++)
        if (A[j][1] < A[index][1])
            index = j;

    temp = A[i][1];
    A[i][1] = A[index][1];
    A[index][1] = temp;

    temp = A[i][0];
    A[i][0] = A[index][0];
    A[index][0] = temp;
}

A[0][2] = 0;
// Calculation of Waiting Times
for (i = 1; i < n; i++) {
    A[i][2] = 0;
    for (j = 0; j < i; j++)
        A[i][2] += A[j][1];
    total += A[i][2];
}

avg_wt = (float)total / n;
total = 0;
cout << "P    BT    WT    TAT" << endl;

// Calculation of Turn Around Time and printing the
// data.
for (i = 0; i < n; i++) {
    A[i][3] = A[i][1] + A[i][2];

```

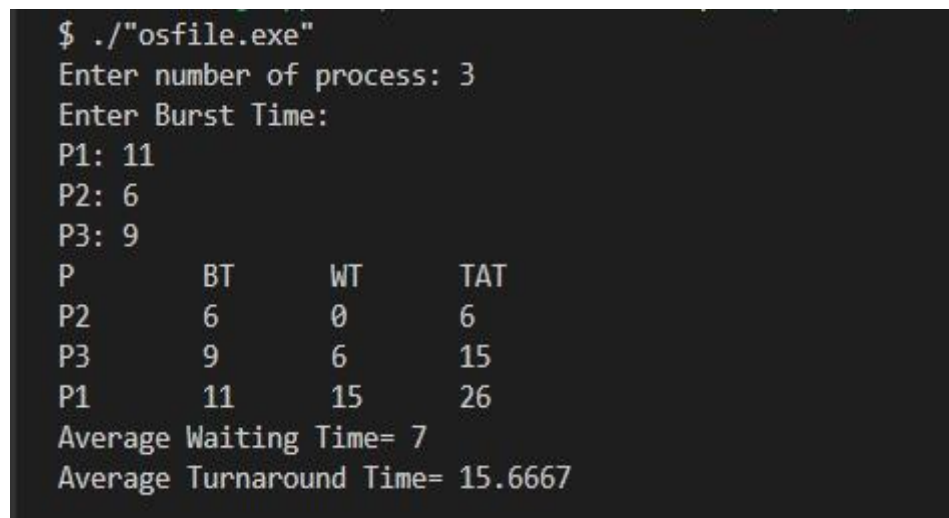
```

        total += A[i][3];
        cout << "P" << A[i][0] << "    " << A[i][1] << "    " << A[i][2] << "    "
<< A[i][3] << endl;
    }

    avg_tat = (float)total / n;
    cout << "Average Waiting Time= " << avg_wt << endl;
    cout << "Average Turnaround Time= " << avg_tat << endl;
}

```

### **OUTPUT:**



```

$ ./"osfile.exe"
Enter number of process: 3
Enter Burst Time:
P1: 11
P2: 6
P3: 9
P      BT      WT      TAT
P2      6       0       6
P3      9       6      15
P1     11     15     26
Average Waiting Time= 7
Average Turnaround Time= 15.6667

```

## **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 pid; // Process ID
    int bt; // Burst Time
    int art; // Arrival Time
};

void findWaitingTime(Process proc[], int n,int wt[])
{
    int rt[n];

    // Copy the burst time into rt[]
    for (int i = 0; i < n; i++)
        rt[i] = proc[i].bt;
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    bool check = false;
    // Process until all processes gets completed
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((proc[j].art <= t) &&
                (rt[j] < minm) && rt[j] > 0) {
                minm = rt[j];
                shortest = j;
                check = true;
            }
        }
        if (check == false) {
            t++;
            continue;
        }
    }
}

```

```

    }

    // Reduce remaining time by one
    rt[shortest]--;

    // Update minimum
    minm = rt[shortest];

    if (minm == 0)
        minm = INT_MAX;

    // If a process gets completely executed
    if (rt[shortest] == 0) {
        // Increment complete
        complete++;

        check = false;

        // Find finish time of current process
        finish_time = t + 1;

        // Calculate waiting time
        wt[shortest] = finish_time - proc[shortest].bt - proc[shortest].art;

        if (wt[shortest] < 0)
            wt[shortest] = 0;
    }

    // Increment time
    t++;
}

}

void findTurnAroundTime(Process proc[], int n,int wt[], int tat[])
{
    // calculating turnaround time by adding bt[i] + wt[i]
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}

```

```

}

// Function to calculate average time
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    findWaitingTime(proc, n, wt);

    findTurnAroundTime(proc, n, wt, tat);

    cout << " P\t\t"

        << "BT\t\t"

        << "WT\t\t"

        << "TAT\t\t\n";

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

        total_wt = total_wt + wt[i];

        total_tat = total_tat + tat[i];

        cout << " " << proc[i].pid << "\t\t"

            << proc[i].bt << "\t\t " << wt[i]

            << "\t\t " << tat[i] << endl;

    }

    cout << "\nAverage waiting time = "

        << (float)total_wt / (float)n;

    cout << "\nAverage turn around time = "

        << (float)total_tat / (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]);

    findavgTime(proc, n);

    return 0;
}

```

```
}
```

### **Output:-**

```
$ ./"osfile.exe"
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>

using namespace std;

struct Process {
    int id, arrival_time, burst_time, completion_time, turnaround_time, waiting_time;
};

void fcfs(vector<Process> processes, int n) {
    int time = 0, count = 0;
    vector<int> completion_time(n), turnaround_time(n), waiting_time(n);

    while (count < n) {
```

```

        if (processes[count].arrival_time == time) {
            time += processes[count].burst_time;
            completion_time[count] = time;
            count++;
        } else {
            time++;
        }
    }

    time = 0;
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = completion_time[i] - processes[i].arrival_time;
        waiting_time[i] = turnaround_time[i] - processes[i].burst_time;
        time += processes[i].burst_time;
    }

    cout << "Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time" << endl;
    for (int i = 0; i < n; i++) {
        cout << i + 1 << "\t\t" << processes[i].arrival_time << "\t\t" << processes[i].burst_time
        << "\t\t" << processes[i].completion_time << "\t\t" << processes[i].turnaround_time << "\t\t"
        << processes[i].waiting_time << endl;
    }

    float avg_waiting_time = 0, avg_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += waiting_time[i];
        avg_turnaround_time += turnaround_time[i];
    }

    cout << "Average Waiting Time: " << avg_waiting_time / n << endl;
    cout << "Average Turnaround Time: " << avg_turnaround_time / n << endl;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for (int i = 0; i < n; i++) {
        cout << "Enter details for process " << i + 1 << endl;
        cout << "Arrival time: ";
        cin >> processes[i].arrival_time;
    }
}

```

```

        cout << "Burst time: ";
        cin >> processes[i].burst_time;
        processes[i].id = i + 1;
        processes[i].completion_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }

    fcfs(processes, n);

    return 0;
}

```

### **OUTPUT:**

```

$ ./"osfile.exe"
Enter the number of processes: 4
Enter details for process 1
Arrival time: 0
Burst time: 10
Enter details for process 2
Arrival time: 1
Burst time: 5
Enter details for process 3
Arrival time: 2
Burst time: 9
Enter details for process 4
Arrival time: 4
Burst time: 6

```

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	0	10	0	0	0
2	1	5	0	0	0
3	2	9	0	0	0
4	4	6	0	0	0

```

Average Waiting Time: 0
Average Turnaround Time: 7.5

```

## **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:-**

```
// C++ program for implementation of RR scheduling

#include<iostream>

using namespace std;

// Function to find the waiting time for all
// processes

void findWaitingTime(int processes[], int n,
                    int bt[], int wt[], int quantum)
{
    // Make a copy of burst times bt[] to store remaining
    // burst times.
    int rem_bt[n];

    for (int i = 0 ; i < n ; i++)
        rem_bt[i] = bt[i];

    int t = 0; // Current time

    // Keep traversing processes in round robin manner
    // until all of them are not done.
    while (1)
    {
        bool done = true;

        // Traverse all processes one by one repeatedly
        for (int i = 0 ; i < n; i++)
```

```

{
    // If burst time of a process is greater than 0
    // then only need to process further
    if (rem_bt[i] > 0)
    {
        done = false; // There is a pending process

        if (rem_bt[i] > quantum)
        {
            // Increase the value of t i.e. shows
            // how much time a process has been processed
            t += quantum;

            // Decrease the burst_time of current process
            // by quantum
            rem_bt[i] -= quantum;
        }

        // If burst time is smaller than or equal to
        // quantum. Last cycle for this process
        else
        {
            // Increase the value of t i.e. shows
            // how much time a process has been processed
            t = t + rem_bt[i];

            // Waiting time is current time minus time
            // used by this process
            wt[i] = t - bt[i];
        }
    }
}

```



```

        // As the process gets fully executed
        // make its remaining burst time = 0
        rem_bt[i] = 0;
    }
}

// If all processes are done
if (done == true)
    break;
}
}

// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n,
                        int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

// Function to calculate average time
void findavgTime(int processes[], int n, int bt[],
                int quantum)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

```

```

// Function to find waiting time of all processes
findWaitingTime(processes, n, bt, wt, quantum);

// Function to find turn around time for all processes
findTurnAroundTime(processes, n, bt, wt, tat);

// Display processes along with all details
cout << "PN\t " << " \tBT "
    << " WT " << " \tTAT\n";

// Calculate total waiting time and total turn
// around time
for (int i=0; i<n; i++)
{
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    cout << " " << i+1 << "\t\t" << bt[i] << "\t "
        << wt[i] << "\t\t " << tat[i] << endl;
}

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

// Driver code
int main()

```

```

{
    // process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    // Burst time of all processes
    int burst_time[] = {10, 5, 8};

    // Time quantum
    int quantum = 2;

    findavgTime(processes, n, burst_time, quantum);

    return 0;
}

```

### **Output:-**

```

$ ./"osfile.exe"
PN      BT  WT      TAT
1       10   13      23
2        5   10      15
3        8   13      21
Average waiting time = 12
Average turn around time = 19.6667

```

## **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 <bits/stdc++.h>

using namespace std;

struct Process {
    int pid; // Process ID
    int bt; // CPU Burst time required
    int priority; // Priority of this process
};

// Function to sort the Process acc. to priority
bool comparison(Process a, Process b)
{
    return (a.priority > b.priority);
}

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n, int wt[])
{
    // waiting time for first process is 0
    wt[0] = 0;

    // calculating waiting time
    for (int i = 1; i < n; i++)
        wt[i] = proc[i - 1].bt + wt[i - 1];
}
```

```

// Function to calculate turn around time
void findTurnAroundTime(Process proc[], int n, int wt[],
                        int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}

// Function to calculate average time
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    // Function to find waiting time of all processes
    findWaitingTime(proc, n, wt);

    // Function to find turn around time for all processes
    findTurnAroundTime(proc, n, wt, tat);

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

    // Calculate total waiting time and total turn

```

```

// around time
for (int i = 0; i < n; i++) {
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    cout << " " << proc[i].pid << "\t\t" << proc[i].bt
        << "\t " << wt[i] << "\t\t " << tat[i]
        << endl;
}

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

void priorityScheduling(Process proc[], int n)
{
    // Sort processes by priority
    sort(proc, proc + n, comparison);

    cout << "Order in which processes gets executed \n";
    for (int i = 0; i < n; i++)
        cout << proc[i].pid << " ";

    findavgTime(proc, n);
}

// Driver code
int main()

```

```

{
    Process proc[]
        = { { 1, 10, 2 }, { 2, 5, 0 }, { 3, 8, 1 } };

    int n = sizeof proc / sizeof proc[0];

    priorityScheduling(proc, n);

    return 0;
}

```

OUTPUT:

```

$ ./"osfile.exe"
Order in which processes gets executed
1 3 2
Processes  Burst time  Waiting time  Turn around time
1           10         0             10
3           8          10             18
2           5          18             23

Average waiting time = 9.33333
Average turn around time = 17

```

## **EXPERIMENT-8**

### **Aim:-**

Write a program to create a child process using fork() system call.

### **Theory:-**

In Unix-like operating systems such as Linux, the `fork()` system call is used to create a new process, which is called a child process. The `fork()` system call creates a new process by duplicating the existing process. After the `fork()` call, both the parent and the child processes continue execution from the next instruction following the `fork()` call. The `fork()` system call is specific to Unix-like operating systems and may not work on Windows. The `fork()` system call is a basic building block for creating new processes in Unix-like operating systems, and it is used extensively in process management and multitasking.

### **Code:-**

```

#include <iostream>

#include <unistd.h>

#include <sys/wait.h>

```

```
int main() {  
    pid_t pid = fork();  
  
    if (pid < 0) {  
        std::cerr << "Error: Fork failed" << std::endl;  
        return 1;  
    } else if (pid == 0) {  
        std::cout << "Child process: My PID is " << getpid() << std::endl;  
        std::cout << "Child process: My parent's PID is " << getppid() << std::endl;  
  
        // Perform child-specific operations here  
  
        return 0;  
    } else {  
        std::cout << "Parent process: I have a child with PID " << pid << std::endl;  
        std::cout << "Parent process: My PID is " << getpid() << std::endl;  
  
        // Perform parent-specific operations here  
  
        int status;  
        waitpid(pid, &status, 0);  
    }  
  
    return 0;  
}
```

**Output:-**



```
/tmp/x9UAqj0GwE.o
Parent process: I have a child with PID 18593
Parent process: My PID is 18592
Child process: My PID is 18593
Child process: My parent's PID is 18592
```

## **EXPERIMENT-8**

### **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;

// Number of processes
#define N 5

// Number of resources
#define M 3

// Function to check if the system is in a safe state
bool isSafe(int allocation[N][M], int max[N][M], int need[N][M], int available[M]) {
```

```

int work[M], finish[N];

// Initialize work and finish arrays
for (int i = 0; i < M; i++)
    work[i] = available[i];
for (int i = 0; i < N; i++)
    finish[i] = 0;

// Find an index i such that both conditions are satisfied
int count = 0;
while (count < N) {
    bool found = false;
    for (int i = 0; i < N; i++) {
        if (finish[i] == 0) {
            int j;
            found = true;
            for (j = 0; j < M; j++)
                if (need[i][j] > work[j])
                    break;
            if (j == M) {
                for (j = 0; j < M; j++)
                    work[j] += allocation[i][j];
                finish[i] = 1;
                count++;
            }
        }
    }
}

// If no process can be executed, the system is in an unsafe state

```

```

        if (!found)
            return false;
    }

    // If all processes have been executed, the system is in a safe state
    return true;
}

int main() {
    int allocation[N][M] = { { 0, 1, 0 }, { 2, 0, 0 }, { 3, 0, 2 }, { 2, 1, 1 }, { 0, 0, 2 } };
    int max[N][M] = { { 0, 2, 2 }, { 3, 0, 2 }, { 4, 0, 3 }, { 3, 2, 2 }, { 2, 0, 2 } };
    int need[N][M];
    int available[M] = { 2, 2, 2 };

    // Calculate the need matrix
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            need[i][j] = max[i][j] - allocation[i][j];

    // Check if the system is in a safe state
    if (isSafe(allocation, max, need, available))
        cout << "System is in safe state.\n";
    else
        cout << "System is in unsafe state.\n";

    return 0;
}

```

**Output:-**

```
$ ./"osfile.exe"  
System is in safe 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:-**

```

$ ./"osfile.exe"
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

```

## **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 <semaphore>
```

```
class ReadersWriters {
```

```
private:
```

```
    std::mutex mutex;
```

```
    sem_t write_mutex;
```

```
    int readers_count;
```

```
public:
```

```
    ReadersWriters() : readers_count(0) {
```

```
        sem_init(&write_mutex, 0, 1);
```

```
    }
```

```
    void start_read() {
```

```

mutex.lock();
readers_count++;
if (readers_count == 1) {
    sem_wait(&write_mutex);
}
mutex.unlock();

// Reading the shared resource
std::cout << "Reader is reading" << std::endl;

mutex.lock();
readers_count--;
if (readers_count == 0) {
    sem_post(&write_mutex);
}
mutex.unlock();
}

void start_write() {
    sem_wait(&write_mutex);

    // Writing to the shared resource
    std::cout << "Writer is writing" << std::endl;

    sem_post(&write_mutex);
}

};

void reader_thread(ReadersWriters& rw, int id) {

```



```
while (true) {  
    // Reading  
    rw.start_read();  
    // Simulating some delay for reading  
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));  
}  
}
```

```
void writer_thread(ReadersWriters& rw, int id) {  
    while (true) {  
        // Writing  
        rw.start_write();  
        // Simulating some delay for writing  
        std::this_thread::sleep_for(std::chrono::milliseconds(2000));  
    }  
}
```

```
int main() {  
    ReadersWriters rw;  
  
    // Creating reader threads  
    std::thread readers[5];  
    for (int i = 0; i < 5; ++i) {  
        readers[i] = std::thread(reader_thread, std::ref(rw), i);  
    }  
  
    // Creating writer threads  
    std::thread writers[2];  
    for (int i = 0; i < 2; ++i) {
```

```

        writers[i] = std::thread(writer_thread, std::ref(rw), i);
    }

    // Joining threads
    for (int i = 0; i < 5; ++i) {
        readers[i].join();
    }
    for (int i = 0; i < 2; ++i) {
        writers[i].join();
    }

    return 0;
}

```

## **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 <iostream>

using namespace std;

int search(int key, int frame_items[], int frame_occupied)
{

```

```

    for (int i = 0; i < frame_occupied; i++)
        if (frame_items[i] == key)
            return 1;
    return 0;
}

void printOuterStructure(int max_frames)
{
    printf("Stream ");
    for (int i = 0; i < max_frames; i++)
        printf("Frame%d ", i + 1);
}

void printCurrFrames(int item, int frame_items[], int frame_occupied, int max_frames)
{
    printf("\n%d \t\t", item);
    for (int i = 0; i < max_frames; i++)
    {
        if (i < frame_occupied)
            printf("%d \t\t", frame_items[i]);
        else
            printf("- \t\t");
    }
}

int predict(int ref_str[], int frame_items[], int refStrLen, int index, int frame_occupied)
{
    int result = -1, farthest = index;
    for (int i = 0; i < frame_occupied; i++)

```

```

{
    int j;
    for (j = index; j < refStrLen; j++)
    {
        if (frame_items[i] == ref_str[j])
        {
            if (j > farthest)
            {
                farthest = j;
                result = i;
            }
            break;
        }
    }
    if (j == refStrLen)
        return i;
}
return (result == -1) ? 0 : result;
}

```

```

void optimalPage(int ref_str[], int refStrLen, int frame_items[], int max_frames)

```

```

{
    int frame_occupied = 0;
    printOuterStructure(max_frames);
    int hits = 0;
    for (int i = 0; i < refStrLen; i++)
    {
        if (search(ref_str[i], frame_items, frame_occupied))
        {

```

```

        hits++;

        printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);

        continue;
    }

    if (frame_occupied < max_frames)
    {
        frame_items[frame_occupied] = ref_str[i];

        frame_occupied++;

        printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
    }
    else
    {
        int pos = predict(ref_str, frame_items, refStrLen, i + 1, frame_occupied);

        frame_items[pos] = ref_str[i];

        printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
    }
}

printf("\n\nHits: %d\n", hits);

printf("Misses: %d", refStrLen - hits);
}

int main()
{
    int ref_str[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

    int refStrLen = sizeof(ref_str) / sizeof(ref_str[0]);

    int max_frames = 3;

    int frame_items[max_frames];

    optimalPage(ref_str, refStrLen, frame_items, max_frames);

    return 0;
}

```

}

### Output:-

```
$ ./"osfile.exe"
Stream Frame1 Frame2 Frame3
7          7          -          -
0          7          0          -
1          7          0          1
2          2          0          1
0          2          0          1
3          2          0          3
0          2          0          3
4          2          4          3
2          2          4          3
3          2          4          3
0          2          0          3
3          2          0          3
2          2          0          3
1          2          0          1
2          2          0          1
0          2          0          1
1          2          0          1
7          7          0          1
0          7          0          1
1          7          0          1

Hits: 11
Misses: 9
```

## **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 <iostream>

#include <unordered_map>

#include <list>

using namespace std;

class LRUCache {
public:
    LRUCache(int capacity) : _capacity(capacity) {}

    int get(int key) {
        auto it = _cache.find(key);
        if (it == _cache.end()) return -1;

        // Move accessed page to the front of the list
        _lru.splice(_lru.begin(), _lru, it->second);

        return it->second->second;
    }

    void put(int key, int value) {
        auto it = _cache.find(key);
        if (it != _cache.end()) {
            // Update the value and move the page to the front of the list
            it->second->second = value;
            _lru.splice(_lru.begin(), _lru, it->second);
        }
        return;
    }
}
```

```

if (_cache.size() >= _capacity) {
    // Remove the least recently used page from the cache
    int lruKey = _lru.back().first;
    _cache.erase(lruKey);
    _lru.pop_back();
}

// Add the new page to the cache and the front of the list
_lru.emplace_front(key, value);
_cache[key] = _lru.begin();
}

```

private:

```

int _capacity;
list<pair<int, int>> _lru; // List to keep track of the access order
unordered_map<int, list<pair<int, int>>::iterator> _cache; // Map to quickly look up a page
};

```

```

int main() {
    LRUCache cache(2);

    cache.put(1, 1);
    cache.put(2, 2);
    cout << cache.get(1) << endl; // Returns 1
    cache.put(3, 3); // Evicts key 2
    cout << cache.get(2) << endl; // Returns -1 (not found)
    cache.put(4, 4); // Evicts key 1
    cout << cache.get(1) << endl; // Returns -1 (not found)
    cout << cache.get(3) << endl; // Returns 3
}

```




```
cout << cache.get(4) << endl; // Returns 4

return 0;

}
```

### **Output:-**



```
$ ./"osfile.exe"
1
-1
-1
3
4
```

## **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:-**

```
// C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
    // To represent set of current pages. We use
    // an unordered_set so that we quickly check
    // if a page is present in set or not
    unordered_set<int> s;
```

```

// To store the pages in FIFO manner
queue<int> indexes;

// Start from initial page
int page_faults = 0;
for (int i=0; i<n; i++)
{
    // Check if the set can hold more pages
    if (s.size() < capacity)
    {
        // Insert it into set if not present
        // already which represents page fault
        if (s.find(pages[i])==s.end())
        {
            // Insert the current page into the set
            s.insert(pages[i]);

            // increment page fault
            page_faults++;

            // Push the current page into the queue
            indexes.push(pages[i]);
        }
    }

    // If the set is full then need to perform FIFO
    // i.e. remove the first page of the queue from
    // set and queue both and insert the current page
    else
    {
        // Check if current page is not already
        // present in the set
        if (s.find(pages[i]) == s.end())
        {
            // Store the first page in the
            // queue to be used to find and
            // erase the page from the set
            int val = indexes.front();

            // Pop the first page from the queue
            indexes.pop();

            // Remove the indexes page from the set
            s.erase(val);

            // insert the current page in the set
            s.insert(pages[i]);

            // push the current page into
            // the queue

```

```

        indexes.push(pages[i]);

        // Increment page faults
        page_faults++;
    }

}

return page_faults;
}

// Driver code
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                  2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}

```

### **Output:-**

```

$ ./"osfile.exe"
7

```