DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM DESIGN

CO-204

PRACTICLE FILE

SUBMITTED TO

Prof. Rajni Jindal

SUBMITTED BY

Anuj Joshi
2K22/CO/74

S. No.	Experiments	Page No.	Sign
1.	Introduction to Linux – Write a program to print Hello World		
2.	Write a program to implement the Prims Algorithm using Disjoint Sets		
3.	Write a program to implement the First Come First Serve (FCFS) job scheduling algorithm.		
4.	Write a program to implement the Shortest Job First (SJF) job scheduling algorithm.		
5.	Write a program to implement the Shortest Remaining Time First (SRTF) job scheduling algorithm.		
6.	Write a program to implement the Round Robin algorithm.		
7.	Write a program to implement a priority scheduling algorithm.		
8.	Write a program to create a child process using a 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
// 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


aj@AnujJoshi: ~
aj@AnujJoshi:~$ touch hello.sh
aj@AnujJoshi:~$ nano hello.sh
aj@AnujJoshi:~$ chmod +x hello.sh
aj@AnujJoshi:~$ ./hello.sh
Hello World
aj@AnujJoshi:~$

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:

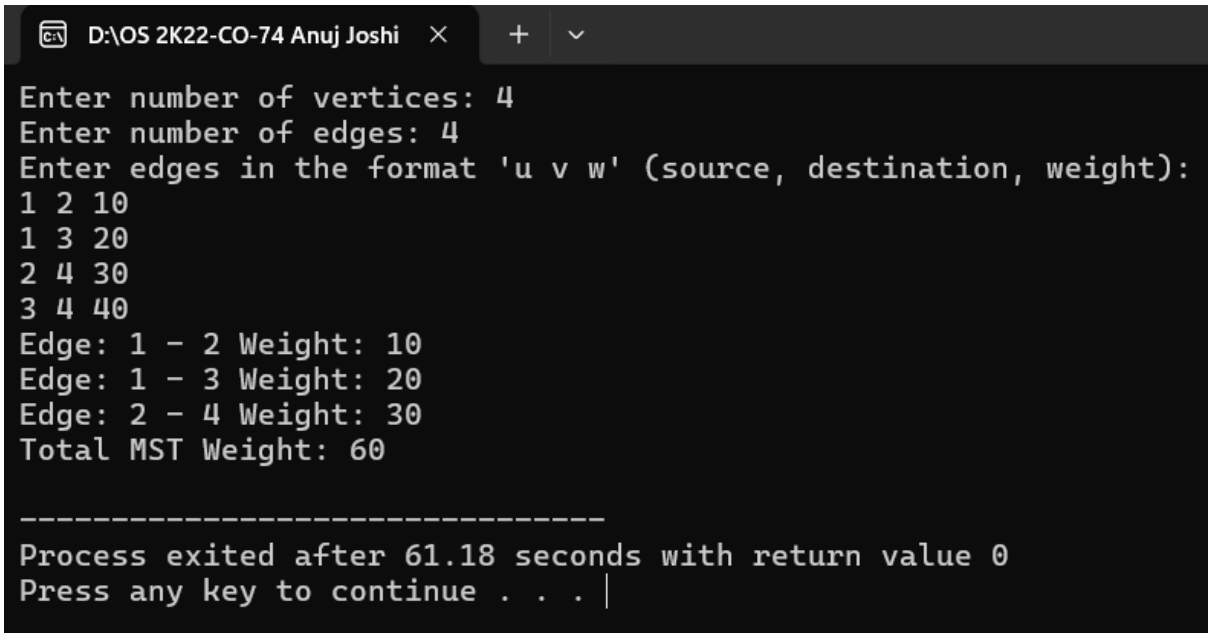
- Union: Merge two sets.
- Find: Determine which set an element belongs to.

Code

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

#define 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]) {
      int temp = x;
      x = y;
      y = temp;
   }
   parent[y] = x;
   if (rank_arr[x] == rank_arr[y]) rank_arr[x]++;
}

typedef struct {
   int u, v, w;
} Edge;
```

```
int compare(const void* a, const void* b) {
    return ((Edge*)a)->w - ((Edge*)b)->w;
}

int main() {
    int n, m;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &m);

    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        rank_arr[i] = 1;
    }

    Edge* edges = (Edge*)malloc(m * sizeof(Edge));

    printf("Enter edges in the format 'u v w' (source, destination, weight):\n");
    for (int i = 0; i < m; i++) {
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].w);
    }

    qsort(edges, m, sizeof(Edge), compare);

    int mst_weight = 0;
    for (int i = 0; i < m; i++) {
        Edge edge = edges[i];
        if (find(edge.u) != find(edge.v)) {
            mst_weight += edge.w;
            union_sets(edge.u, edge.v);
            printf("Edge: %d - %d Weight: %d\n", edge.u, edge.v, edge.w);
        }
    }

    printf("Total MST Weight: %d\n", mst_weight);

    free(edges);
    return 0;
}
```

Output


D:\OS 2K22-CO-74 Anuj Joshi
Enter number of vertices: 4
Enter number of edges: 4
Enter edges in the format 'u v w' (source, destination, weight):
1 2 10
1 3 20
2 4 30
3 4 40
Edge: 1 - 2 Weight: 10
Edge: 1 - 3 Weight: 20
Edge: 2 - 4 Weight: 30
Total MST Weight: 60

Process exited after 61.18 seconds with return value 0
Press any key to continue . . .

EXPERIMENT-3

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 the ready queue. The main idea behind FCFS scheduling is to follow a first-in-first-out approach, where the process that arrives first is executed first. Once a process starts executing, it continues until it completes its execution without any interruption. Processes are sorted based on their arrival times, with the process that arrives earliest being placed at the head of the ready queue.

Code
```
#include <iostream>
using namespace std;

void calculateWaitingTime(int processIDs[], int n, int burstTimes[], int waitingTimes[])
{
   waitingTimes[0] = 0; // Waiting time for the first process is 0
   for (int i = 1; i < n; i++)
      waitingTimes[i] = burstTimes[i - 1] + waitingTimes[i - 1];
}

void calculateTurnAroundTime(int processIDs[], int n, int burstTimes[], int waitingTimes[],
int turnaroundTimes[])
{
   for (int i = 0; i < n; i++)
      turnaroundTimes[i] = burstTimes[i] + waitingTimes[i]; // Turnaround time = Burst time
+ Waiting time
}

void calculateAverageTime(int processIDs[], int n, int burstTimes[])
{
   int waitingTimes[n], turnaroundTimes[n], totalWaitingTime = 0, totalTurnaroundTime =
0;
   calculateWaitingTime(processIDs, n, burstTimes, waitingTimes);
   calculateTurnAroundTime(processIDs, n, burstTimes, waitingTimes, turnaroundTimes);
   cout << "ProcessID BurstTime WaitingTime TurnaroundTime CompletionTime\n";
   for (int i = 0; i < n; i++)
   {
      totalWaitingTime += waitingTimes[i];
      totalTurnaroundTime += turnaroundTimes[i];
      int completionTime = burstTimes[i] + waitingTimes[i];
      cout << processIDs[i] << "\t\t" << burstTimes[i] << "\t\t" << waitingTimes[i] << "\t\t"
<< turnaroundTimes[i] << "\t\t" << completionTime << endl;
   }
   cout << "Average waiting time = " << (float)totalWaitingTime / (float)n;
   cout << "\nAverage turnaround time = " << (float)totalTurnaroundTime / (float)n;
```

```
}

int main()
{
   int processIDs[] = {1, 2, 3, 4};
   int n = sizeof processIDs / sizeof processIDs[0];
   int burstTimes[] = {7, 4, 1, 4};
   calculateAverageTime(processIDs, n, burstTimes);
   return 0;
}
```

OUTPUT

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>fcfs
ProcessID  BurstTime  WaitingTime  TurnaroundTime  CompletionTime
1               7               0               7               7
2               4               7               11              11
3               1               11              12              12
4               4               12              16              16
Average waiting time = 7.5
Average turnaround time = 11.5
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

EXPERIMENT-4

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>
using namespace std;

void calculateWaitingTime(int processIDs[], int n, int burstTimes[], int waitingTimes[])
{
   waitingTimes[0] = 0; // Waiting time for the first process is 0
   for (int i = 1; i < n; i++)
      waitingTimes[i] = burstTimes[i - 1] + waitingTimes[i - 1];
}

void calculateTurnAroundTime(int processIDs[], int n, int burstTimes[], int waitingTimes[], int turnaroundTimes[])
{
   for (int i = 0; i < n; i++)
      turnaroundTimes[i] = burstTimes[i] + waitingTimes[i]; // Turnaround time = Burst time + Waiting time
}

void calculateAverageTime(int processIDs[], int n, int burstTimes[])
{
   int waitingTimes[n], turnaroundTimes[n], totalWaitingTime = 0, totalTurnaroundTime = 0;
   calculateWaitingTime(processIDs, n, burstTimes, waitingTimes);
   calculateTurnAroundTime(processIDs, n, burstTimes, waitingTimes, turnaroundTimes);
   cout << "ProcessID  BurstTime  WaitingTime  TurnaroundTime\n";
   for (int i = 0; i < n; i++)
   {
      totalWaitingTime += waitingTimes[i];
      totalTurnaroundTime += turnaroundTimes[i];
```

```
        cout << processIDs[i] << "\t\t" << burstTimes[i] << "\t\t" << waitingTimes[i] << "\t\t" <<
turnaroundTimes[i] << endl;
    }
    cout << "Average waiting time = " << (float)totalWaitingTime / (float)n;
    cout << "\nAverage turnaround time = " << (float)totalTurnaroundTime / (float)n;
}

int main()
{
    int processIDs[] = {1, 2, 3, 4};
    int n = sizeof processIDs / sizeof processIDs[0];
    int burstTimes[] = {7, 4, 1, 4};

    // Sorting processes according to their Burst Time.
    sort(burstTimes, burstTimes + n);

    calculateAverageTime(processIDs, n, burstTimes);
    return 0;
}
```

Output

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "4. SJF.cpp" -o sjf

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>sjf
ProcessID  BurstTime  WaitingTime  TurnaroundTime
1               1               0               1
2               4               1               5
3               4               5               9
4               7               9               16
Average waiting time = 3.75
Average turnaround time = 7.75
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

EXPERIMENT-5

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 <iostream>
#include <algorithm>
#include <climits>
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 getscompleted
   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

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "5. SRTF.cpp" -o srtf

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>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
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

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>
#include <queue>
#include <vector>
using namespace std;

struct Process
{
   int id;
   int burstTime;
   int remainingTime;
   int arrivalTime;
};

void calculateAverageTimeNonPreemptive(vector<Process> &processes, int quantum)
{
   int n = processes.size();
   queue<Process> readyQueue;
   vector<int> waitingTime(n, 0);
   vector<int> turnaroundTime(n, 0);
   int currentTime = 0;

   for (int i = 0; i < n; ++i)
   {
      readyQueue.push(processes[i]);
   }

   while (!readyQueue.empty())
   {
      Process currentProcess = readyQueue.front();
      readyQueue.pop();

      if (currentProcess.burstTime <= quantum)
```

```
        {
            currentTime += currentProcess.burstTime;
            turnaroundTime[currentProcess.id] = currentTime;
        }
        else
        {
            currentTime += quantum;
            currentProcess.burstTime -= quantum;
            readyQueue.push(currentProcess);
        }
    }

    // Calculating waiting time
    for (int i = 0; i < n; ++i)
    {
        waitingTime[i] = turnaroundTime[i] - processes[i].burstTime - processes[i].arrivalTime;
    }

    // Display results
    cout << "Non-Preemptive Round Robin Schedule:" << endl;
    cout << "ProcessID  BurstTime  WaitingTime  TurnaroundTime" << endl;
    float totalWaitingTime = 0, totalTurnaroundTime = 0;
    for (int i = 0; i < n; ++i)
    {
        totalWaitingTime += waitingTime[i];
        totalTurnaroundTime += turnaroundTime[i];
        cout << processes[i].id << "\t\t" << processes[i].burstTime << "\t\t" << waitingTime[i]
<< "\t\t" << turnaroundTime[i] << endl;
    }
    cout << "Average waiting time = " << totalWaitingTime / n << endl;
    cout << "Average turnaround time = " << totalTurnaroundTime / n << endl;
}

void calculateAverageTimePreemptive(vector<Process> &processes, int quantum)
{
    int n = processes.size();
    queue<Process> readyQueue;
    vector<int> waitingTime(n, 0);
    vector<int> turnaroundTime(n, 0);
    int currentTime = 0;

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

    while (true)
    {
        bool done = true;
        for (int i = 0; i < n; ++i)
```

```
        {
            if (processes[i].remainingTime > 0)
            {
                done = false;
                if (processes[i].remainingTime > quantum)
                {
                    currentTime += quantum;
                    processes[i].remainingTime -= quantum;
                }
                else
                {
                    currentTime += processes[i].remainingTime;
                    processes[i].remainingTime = 0;
                    turnaroundTime[i] = currentTime;
                }
            }
        }
        if (done)
            break;
    }

    // Calculating waiting time
    for (int i = 0; i < n; ++i)
    {
        waitingTime[i] = turnaroundTime[i] - processes[i].burstTime - processes[i].arrivalTime;
    }

    // Display results
    cout << "\nPreemptive Round Robin Schedule:" << endl;
    cout << "ProcessID  BurstTime  WaitingTime  TurnaroundTime" << endl;
    float totalWaitingTime = 0, totalTurnaroundTime = 0;
    for (int i = 0; i < n; ++i)
    {
        totalWaitingTime += waitingTime[i];
        totalTurnaroundTime += turnaroundTime[i];
        cout << processes[i].id << "\t\t" << processes[i].burstTime << "\t\t" << waitingTime[i]
<< "\t\t" << turnaroundTime[i] << endl;
    }
    cout << "Average waiting time = " << totalWaitingTime / n << endl;
    cout << "Average turnaround time = " << totalTurnaroundTime / n << endl;
}

int main()
{
    vector<Process> processes = {{0, 7, 0}, {1, 9, 0}, {2, 6, 0}, {3, 3, 0}}; // Processes with
their burst times and arrival times
    int quantum = 3;                                        // Time quantum

    calculateAverageTimeNonPreemptive(processes, quantum);
    calculateAverageTimePreemptive(processes, quantum);
```

```
    return 0;
}
```

Output

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "6. RoundRobin.cpp" -o rr

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>rr
Non-Preemptive Round Robin Schedule:
ProcessID  BurstTime  WaitingTime  TurnaroundTime
0               7               15              22
1               9               16              25
2               6               15              21
3               3               9               12
Average waiting time = 13.75
Average turnaround time = 20

Preemptive Round Robin Schedule:
ProcessID  BurstTime  WaitingTime  TurnaroundTime
0               7               15              22
1               9               16              25
2               6               15              21
3               3               9               12
Average waiting time = 13.75
Average turnaround time = 20

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

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 <vector>
#include <algorithm>
using namespace std;

struct Process
{
   int id;
   int burstTime;
   int priority;
   int waitingTime;
   int turnaroundTime;
};

bool comparePriority(Process a, Process b)
{
   return a.priority < b.priority;
}

void calculateWaitingTimeTurnaroundTime(vector<Process> &processes)
{
   processes[0].waitingTime = 0;
   processes[0].turnaroundTime = processes[0].burstTime;

   for (int i = 1; i < processes.size(); ++i)
   {
      processes[i].waitingTime = processes[i - 1].burstTime + processes[i - 1].waitingTime;
      processes[i].turnaroundTime = processes[i].burstTime + processes[i].waitingTime;
   }
}
```

```
void calculateAverageTime(vector<Process> &processes)
{
   float totalWaitingTime = 0;
   float totalTurnaroundTime = 0;

   for (int i = 0; i < processes.size(); ++i)
   {
      totalWaitingTime += processes[i].waitingTime;
      totalTurnaroundTime += processes[i].turnaroundTime;
   }

   cout << "ProcessID  BurstTime  Priority  WaitingTime  TurnaroundTime" << endl;
   for (int i = 0; i < processes.size(); ++i)
   {
      cout << processes[i].id << "\t\t" << processes[i].burstTime << "\t" << processes[i].priority
<< "\t"
           << processes[i].waitingTime << "\t\t" << processes[i].turnaroundTime << endl;
   }
   cout << "Average waiting time = " << totalWaitingTime / processes.size() << endl;
   cout << "Average turnaround time = " << totalTurnaroundTime / processes.size() << endl;
}

int main()
{
   vector<Process> processes = {{1, 7, 2}, {2, 4, 1}, {3, 1, 3}, {4, 4, 4}}; // Processes with
their burst times and priorities

   // Sort processes based on priority
   sort(processes.begin(), processes.end(), comparePriority);

   calculateWaitingTimeTurnaroundTime(processes);
   calculateAverageTime(processes);

   return 0;
}
```

OUTPUT

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "7. Priority.cpp" -o priority

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>priority
ProcessID  BurstTime  Priority  WaitingTime  TurnaroundTime
2               4       1       0               4
1               7       2       4               11
3               1       3       11              12
4               4       4       12              16
Average waiting time = 6.75
Average turnaround time = 10.75

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

EXPERIMENT-8

Aim

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

Theory

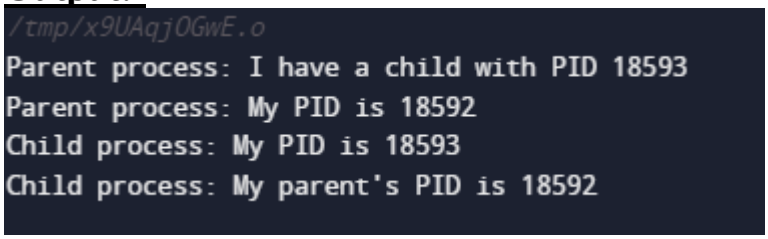
In Unix-like operating systems such as Linux, the `fork()` system call is used to create a new process, which is called a child process. The `fork()` system call creates a new process by duplicating the existing process. After the `fork()` call, both the parent and the child processes continue execution 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>
using namespace std;

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        cerr << "Error: Fork failed" << std::endl;
        return 1;
    } else if (pid == 0) {
        cout << "Child process: My PID is " << getpid() << std::endl;
        cout << "Child process: My parent's PID is " << getppid() << std::endl;

        // Perform child-specific operations here

        return 0;
    } else {
        cout << "Parent process: I have a child with PID " << pid << std::endl;
        cout << "Parent process: My PID is " << getpid() << std::endl;

        // Perform parent-specific operations here

        int status;
        waitpid(pid, &status, 0);
    }

    return 0;
```

}

Output:-


/tmp/x9UAqjOGwE.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;

// Function to check if a process can finish with current available resources
bool canFinish(int process, int maxResources[][4], int available[], int numProcesses, int
numResources) {
  for (int j = 0; j < numResources; j++) {
    if (maxResources[process][j] > available[j]) {
      return false; // Process needs more resources than available
    }
  }
  return true;
}

int main() {
  // Number of processes and resources
  int numProcesses = 5;  // Processes (P0, P1, P2, P3, P4)
  int numResources = 4;  // Resources (R0, R1, R2, R3)

  // Allocation matrix (represents resources currently held by each process)
  int allocationMatrix[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 maxResources[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 availableResources[4] = {3, 3, 2, 1};

  // Finished processes (stores process IDs in safe sequence)
  int finishedProcesses[numProcesses] = {0};

  // Number of finished processes
  int numFinished = 0;

  // Loop until all processes are finished or a deadlock is detected
  while (numFinished < numProcesses) {
    int isSafe = 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 < numProcesses; i++) {
      if (finishedProcesses[i] == 0 && canFinish(i, maxResources, availableResources,
numProcesses, numResources)) {
        // Process can finish, update resources and mark as finished
        for (int j = 0; j < numResources; j++) {
          availableResources[j] += allocationMatrix[i][j];
        }
        finishedProcesses[i] = 1;
        isSafe = 1;
      }
    }

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

    numFinished++;  // Increment finished process count
  }

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

```
    }
    cout << " P" << numProcesses - 1 << endl;
  }

  return 0;
}
```

Output

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "9. banker-algo.cpp" -o banker

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>banker
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>
#include <vector>
using namespace std;

int main() {
   // Incoming page reference stream
   vector<int> incomingStream = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};

   // Number of frames
   int frames = 3;

   // Initialize frame array with -1 (indicating empty frame)
   vector<int> temp(frames, -1);

   // Initialize page faults counter
   int pageFaults = 0;

   // Process incoming page references
   for (int i = 0; i < incomingStream.size(); ++i) {
      int page = incomingStream[i];
      bool pageFound = false;

      // Check if page is already in a frame
      for (int j = 0; j < frames; ++j) {
         if (temp[j] == page) {
            pageFound = true;
            break;
         }
      }

      // If page not found in frames, perform page replacement
      if (!pageFound) {
         // Increment page faults
         pageFaults++;

         // Find the frame to replace using FIFO (First-In-First-Out) algorithm
```

```
            int replaceIndex = i % frames;

            // Replace page in frame
            temp[replaceIndex] = page;
        }

        // Print current frame state
        cout << "Incoming Page: " << page << "\tFrames: ";
        for (int j = 0; j < frames; ++j) {
            if (temp[j] != -1)
                cout << temp[j] << " ";
            else
                cout << "- ";
        }
        cout << endl;
    }

    // Print total page faults
    cout << "\nTotal Page Faults: " << pageFaults << endl;

    return 0;
}
```

Output

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

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "12. optimal-page-replacement.cpp" -o optimal

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>optimal
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
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

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

```
D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>g++ "13. lru-page-replacement.cpp" -o lru

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>lru
1
-1
-1
3
4

D:\OS 2K22-CO-74 Anuj Joshi SEM-IV 2023-24>
```

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