Islamic University

Department of
Information and Communication Technology

Course Code: ICT-3208
Course: Operating System Laboratory

Lab Report

Submitted to:
Dr. Md Sipon Miah
Professor
Department of Information and Communication Technology
Islamic University, Bangladesh

Submitted by:
Name: S. M. Sajjad Hossain Soykot
Roll: 2018040
Reg. No: 1315
Session: 2020-21

INDEX

Experiment 1: Simulate the following CPU scheduling algorithms: First Come First Serve

Theory:

CPU scheduling is one of the fundamental responsibilities of an operating system.
It determines the order in which processes in the **ready queue** are allocated the CPU for execution.
Among different scheduling strategies, the **First Come First Serve (FCFS)** algorithm is the simplest and easiest to implement.

Definition

- The **First Come First Serve (FCFS)** scheduling algorithm is a **non-preemptive** scheduling technique.
- In FCFS, the process that arrives first in the ready queue is executed first, followed by the next process in order of arrival.
- It follows the **FIFO (First In, First Out)** principle, much like servicing customers in a queue.

Characteristics

1. **Non-preemptive**: Once a process starts execution, it cannot be interrupted until it finishes.
2. **Simple & Fair**: Every process is served in the exact order of arrival.
3. **Queue-based Implementation**: Processes are inserted into a queue, and the CPU is allocated based on arrival time.

Important Terms

- **Arrival Time (AT):** The time at which a process enters the ready queue.
- **Burst Time (BT):** The time required by a process to complete execution.
- **Completion Time (CT):** The time at which a process finishes execution.
- **Turnaround Time (TAT):** Total time taken from arrival to completion.

$$TAT = CT - AT$$

- **Waiting Time (WT):** Time spent by a process waiting in the ready queue.

$$WT = TAT - BT$$

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class process{
    public:
    int pid;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnAroundTime;
    int completionTime;
};
bool compare_arrival(process p1, process p2){
    return p1.arrivalTime<p2.arrivalTime;
}
bool compare_pid(process p1, process p2) {
    return p1.pid < p2.pid;   // for restoring PID order
}

int main(){
    int n;
    cout<<"Enter the Number of Process : ";
    cin>>n;
    vector <process> processes(n);

    for(int i=0; i<n; ++i){
        processes[i].pid=i+1;
        cout << "Enter Arrival and Burst Time for P"<<i+1<< ": ";
        cin >> processes[i].arrivalTime>>processes[i].burstTime;

    }

    sort(processes.begin(),processes.end(),compare_arrival);

    int currentTime=0;
    float totalWT=0, totalTAT=0;

    for(int i=0; i<n; ++i){
        if(currentTime<processes[i].arrivalTime){
            currentTime=processes[i].arrivalTime;
        }

        processes[i].completionTime=currentTime+processes[i].burstTime;
        processes[i].turnAroundTime=processes[i].completionTime-
processes[i].arrivalTime;
        processes[i].waitingTime=processes[i].turnAroundTime-
processes[i].burstTime;

        currentTime=processes[i].completionTime;
```

```
        totalWT+=processes[i].waitingTime;
        totalTAT+=processes[i].turnAroundTime;
    }

    // restore PID order for output
    sort(processes.begin(), processes.end(), compare_pid);

    cout<<"\nPID\tAT\tBT\tCT\tTAT\tWT\n";

    for (int i = 0; i < n; i++) {
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].completionTime << "\t"
             << processes[i].turnAroundTime << "\t"
             << processes[i].waitingTime << "\n";
    }

    cout << "\nAverage Turnaround Time = " << totalWT/n<<endl;
    cout << "Average Waiting Time = " << totalTAT/n << endl;

}
```

Output:

```
Enter the Number of Process : 5
Enter Arrival and Burst Time for P1: 3 1
Enter Arrival and Burst Time for P2: 4 5
Enter Arrival and Burst Time for P3: 0 2
Enter Arrival and Burst Time for P4: 3 7
Enter Arrival and Burst Time for P5: 5 5

PID     AT      BT      CT      TAT     WT
P1      3       1       4       1       0
P2      4       5       16      12      7
P3      0       2       2       2       0
P4      3       7       11      8       1
P5      5       5       21      16      11

Average Turnaround Time = 3.8
Average Waiting Time = 7.8
```

Experiment 2: Simulate the following CPU scheduling algorithms: Non-Primitive Priority

Theory:

CPU Scheduling is the process of selecting a process from the ready queue and allocating the CPU to it.

Priority Scheduling is a scheduling algorithm where each process is assigned a priority, and the CPU is allocated to the process with the **highest priority** (smallest priority number if lower number = higher priority).

Non-Preemptive Priority Scheduling

- In **Non-Preemptive Priority Scheduling**, once a process starts execution, it cannot be interrupted until it completes.
- The scheduler selects the process with the highest priority from the ready queue when the CPU becomes idle.
- If two processes have the same priority, they are executed in the **order of their arrival time** (FCFS among them).

Parameters:

- **AT (Arrival Time):** The time when the process arrives in the ready queue.
- **BT (Burst Time):** The time required by a process for execution.
- **PR (Priority):** Defines the importance of the process. Lower value = Higher priority.
- **CT (Completion Time):** Time at which process finishes execution.
- **TAT (Turnaround Time):** `CT - AT`
- **WT (Waiting Time):** `TAT - BT`

Advantages:

- Useful when processes have different importance.
- Ensures high-priority tasks get executed first.

Disadvantages:

- **Starvation:** Low-priority processes may never execute.
- May lead to unfairness if priorities are not managed properly.

Source Code:

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

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int priority;
    int completionTime;
    int turnAroundTime;
```

```
    int waitingTime;
    bool completed = false;
};

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

    Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter Priority, Arrival Time, and Burst Time for P" << i + 1 <<
": ";
        cin >> processes[i].priority >> processes[i].arrivalTime >>
processes[i].burstTime;
    }

    int completedCount = 0, currentTime = 0;
    float totalWT = 0, totalTAT = 0;

    while (completedCount < n) {
        int idx = -1;
        int highestPriority = 1e9; // smaller value = higher priority

        // Find next process to execute
        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrivalTime <=
currentTime) {
                if (processes[i].priority < highestPriority) {
                    idx = i;
                    highestPriority = processes[i].priority;
                } else if (processes[i].priority == highestPriority) {
                    if (processes[i].arrivalTime < processes[idx].arrivalTime) {
                        idx = i;
                    }
                }
            }
        }

        if (idx == -1) {
            currentTime++; // CPU idle
        } else {
            Process &p = processes[idx];
            p.completionTime = currentTime + p.burstTime;
            p.turnAroundTime = p.completionTime - p.arrivalTime;
            p.waitingTime = p.turnAroundTime - p.burstTime;
            p.completed = true;
```

```
            totalWT += p.waitingTime;
            totalTAT += p.turnAroundTime;

            currentTime = p.completionTime;
            completedCount++;
        }
    }
    // Print results
    cout << "\nPID\tAT\tBT\tPR\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].priority << "\t"
             << processes[i].completionTime << "\t"
             << processes[i].turnAroundTime << "\t"
             << processes[i].waitingTime << "\n";
    }

    cout << "\nAverage Waiting Time = " << totalWT / n;
    cout << "\nAverage Turnaround Time = " << totalTAT / n << endl;

    return 0;
}
```

Output:

```
Enter the number of processes: 6
Enter Priority, Arrival Time, and Burst Time for P1: 2 3 5
Enter Priority, Arrival Time, and Burst Time for P2: 1 2 4
Enter Priority, Arrival Time, and Burst Time for P3: 3 5 1
Enter Priority, Arrival Time, and Burst Time for P4: 4 4 7
Enter Priority, Arrival Time, and Burst Time for P5: 3 1 6
Enter Priority, Arrival Time, and Burst Time for P6: 5 0 2

PID     AT      BT      PR      CT      TAT     WT
P1      3       5       2       11      8       3
P2      2       4       1       6       4       0
P3      5       1       3       18      13      12
P4      4       7       4       25      21      14
P5      1       6       3       17      16      10
P6      0       2       5       2       2       0

Average Waiting Time = 6.5
Average Turnaround Time = 10.6667
```

Experiment 3: Simulate the following CPU scheduling algorithms: Preemptive Priority.

Theory:

Definition:
Preemptive Priority Scheduling is a CPU scheduling algorithm where the CPU is assigned to the process with the **highest priority** (smallest priority number = higher priority).

Preemptive Nature:
If a new process arrives with a higher priority than the currently running process, the CPU will **preempt (stop)** the current process and assign the CPU to the new process.

Selection Criteria:

1. Among all processes that have **arrived** and are not completed, the process with the **highest priority** is selected.
2. If two processes have the same priority, the one with the **earlier arrival time** is chosen.

Source Code:

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

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int priority;
    int completionTime;
    int turnAroundTime;
    int waitingTime;
    bool completed = false;
};

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

    Process processes[n];

    // Input processes (Priority → Arrival Time → Burst Time)
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter Priority, Arrival Time and Burst Time for P" << i + 1 <<
": ";
        cin >> processes[i].priority >> processes[i].arrivalTime >>
processes[i].burstTime;
```

```
        processes[i].remainingTime = processes[i].burstTime;
    }

    int currentTime = 0, completedCount = 0;
    float totalWT = 0, totalTAT = 0;

    // Preemptive Priority Scheduling
    while (completedCount < n) {
        int idx = -1;
        int highestPriority = -1; // now higher value = higher priority

        // Select process with highest priority among arrived processes
        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrivalTime <=
currentTime) {
                if (processes[i].priority > highestPriority) {  // flipped here
                    highestPriority = processes[i].priority;
                    idx = i;
                } else if (processes[i].priority == highestPriority) {
                    if (processes[i].arrivalTime < processes[idx].arrivalTime) {
                        idx = i; // tie-breaker: earlier arrival
                    }
                }
            }
        }

        if (idx == -1) {
            currentTime++; // CPU idle
        } else {
            // Run selected process for 1 unit
            processes[idx].remainingTime--;
            currentTime++;

            // If process finishes
            if (processes[idx].remainingTime == 0) {
                processes[idx].completed = true;
                completedCount++;

                processes[idx].completionTime = currentTime;
                processes[idx].turnAroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
                processes[idx].waitingTime = processes[idx].turnAroundTime -
processes[idx].burstTime;

                totalWT += processes[idx].waitingTime;
                totalTAT += processes[idx].turnAroundTime;
            }
        }
    }

    // Output
```

```
    cout << "\nPID\tAT\tBT\tPR\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].priority << "\t"
             << processes[i].completionTime << "\t"
             << processes[i].turnAroundTime << "\t"
             << processes[i].waitingTime << "\n";
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Waiting Time = " << totalWT / n;
    cout << "\nAverage Turnaround Time = " << totalTAT / n << "\n";

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter Priority, Arrival Time and Burst Time for P1: 2 3 5
Enter Priority, Arrival Time and Burst Time for P2: 1 2 4
Enter Priority, Arrival Time and Burst Time for P3: 3 5 6
Enter Priority, Arrival Time and Burst Time for P4: 4 1 7
Enter Priority, Arrival Time and Burst Time for P5: 3 0 5

PID     AT      BT      PR      CT      TAT     WT
P1      3       5       2       23      20      15
P2      2       4       1       27      25      21
P3      5       6       3       18      13      7
P4      1       7       4       8       7       0
P5      0       5       3       12      12      7

Average Waiting Time = 10.00
Average Turnaround Time = 15.40
```

Experiment 4: Simulate the following CPU scheduling algorithms: SJF (Non-preemptive)

Theory:

Definition:

- SJF is a CPU scheduling algorithm where the process with the **shortest burst time** is selected next.
- It is **non-preemptive**, meaning once a process starts execution, it cannot be stopped until it finishes.

Working:

- At any given CPU time, among all the **arrived and incomplete** processes, the one with the **smallest burst time** is chosen.
- If two processes have the same burst time, the one with the **earlier arrival time** gets selected.

Metrics:

- **Completion Time (CT)**: When the process finishes execution.
- **Turnaround Time (TAT)** = CT – Arrival Time
- **Waiting Time (WT)** = TAT – Burst Time

Characteristics:

- **Optimal in terms of Average Waiting Time** (it minimizes the average waiting time).
- May cause **starvation** for long processes if short jobs keep arriving.

Source Code:

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int completionTime;
    int turnAroundTime;
    int waitingTime;
    bool completed = false;
};

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

```
    Process processes[n];

    // Input
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter Arrival Time and Burst Time for P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }

    int currentTime = 0, completedCount = 0;
    float totalWT = 0, totalTAT = 0;

    // Non-preemptive SJF Scheduling
    while (completedCount < n) {
        int idx = -1;
        int minBT = 1e9;

        // Find process with smallest burst time among arrived and incomplete
processes
        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrivalTime <=
currentTime) {
                if (processes[i].burstTime < minBT) {
                    minBT = processes[i].burstTime;
                    idx = i;
                } else if (processes[i].burstTime == minBT) {
                    if (processes[i].arrivalTime < processes[idx].arrivalTime) {
                        idx = i;
                    }
                }
            }
        }

        if (idx == -1) {
            currentTime++; // CPU idle
        } else {
            // Execute process fully (non-preemptive)
            currentTime += processes[idx].burstTime;
            processes[idx].completionTime = currentTime;
            processes[idx].turnAroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
            processes[idx].waitingTime = processes[idx].turnAroundTime -
processes[idx].burstTime;

            processes[idx].completed = true;
            completedCount++;

            totalWT += processes[idx].waitingTime;
            totalTAT += processes[idx].turnAroundTime;
        }
    }
```

```
    // Output
    cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].completionTime << "\t"
             << processes[i].turnAroundTime << "\t"
             << processes[i].waitingTime << "\n";
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Turnaround Time = " << totalTAT / n;
    cout << "\nAverage Waiting Time = " << totalWT / n << "\n";

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter Arrival Time and Burst Time for P1: 4 5
Enter Arrival Time and Burst Time for P2: 0 2
Enter Arrival Time and Burst Time for P3: 1 5
Enter Arrival Time and Burst Time for P4: 6 7
Enter Arrival Time and Burst Time for P5: 2 3

PID     AT      BT      CT      TAT     WT
P1      4       5       15      11      6
P2      0       2       2       2       0
P3      1       5       10      9       4
P4      6       7       22      16      9
P5      2       3       5       3       0

Average Turnaround Time = 8.20
Average Waiting Time = 3.80
```

Experiment 5: Simulate the following CPU scheduling algorithms: SRTF (Preemptive)

Theory:

Definition:

- SRTF is the **preemptive version of Shortest Job First (SJF)**.
- At any moment, the process with the **least remaining burst time** is chosen for execution.

How it works:

- If a new process arrives with a **shorter remaining burst time** than the currently running one, the CPU is **preempted** (switched) to the new process.
- Execution happens in **unit time slices** until a process completes.

Metrics:

- **Completion Time (CT)**: When a process finishes execution.
- **Turnaround Time (TAT)** = CT – Arrival Time.
- **Waiting Time (WT)** = TAT – Burst Time.

Characteristics:

- Minimizes **average waiting time** better than non-preemptive SJF.
- **Starvation problem**: Long processes may be delayed if short processes keep arriving.

Source Code:

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

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int completionTime;
    int turnAroundTime;
    int waitingTime;
    bool completed = false;
};

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

    Process processes[n];
```

```
    // Input
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter Arrival Time & Burst Time for P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

    int currentTime = 0, completedCount = 0;
    float totalWT = 0, totalTAT = 0;

    // SRTF Scheduling Loop (preemptive)
    while (completedCount < n) {
        int idx = -1;
        int minRem = 1e9;

        // Find process with minimum remaining time among arrived ones
        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrivalTime <=
currentTime) {
                if (processes[i].remainingTime < minRem &&
processes[i].remainingTime > 0) {
                    minRem = processes[i].remainingTime;
                    idx = i;
                } else if (processes[i].remainingTime == minRem && idx != -1) {
                    if (processes[i].arrivalTime < processes[idx].arrivalTime) {
                        idx = i;
                    }
                }
            }
        }

        if (idx != -1) {
            // Run the selected process for 1 time unit
            processes[idx].remainingTime--;
            currentTime++;

            // If process finishes
            if (processes[idx].remainingTime == 0) {
                processes[idx].completed = true;
                processes[idx].completionTime = currentTime;
                processes[idx].turnAroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
                processes[idx].waitingTime = processes[idx].turnAroundTime -
processes[idx].burstTime;

                totalWT += processes[idx].waitingTime;
                totalTAT += processes[idx].turnAroundTime;

                completedCount++;
```

```
            }
        } else {
            currentTime++; // CPU idle
        }
    }

    // Output
    cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].completionTime << "\t"
             << processes[i].turnAroundTime << "\t"
             << processes[i].waitingTime << "\n";
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Turnaround Time = " << totalTAT / n;
    cout << "\nAverage Waiting Time = " << totalWT / n << "\n";

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter Arrival Time & Burst Time for P1: 2 6
Enter Arrival Time & Burst Time for P2: 1 3
Enter Arrival Time & Burst Time for P3: 4 2
Enter Arrival Time & Burst Time for P4: 0 5
Enter Arrival Time & Burst Time for P5: 6 4

PID     AT      BT      CT      TAT     WT
P1      2       6       20      18      12
P2      1       3       4       3       0
P3      4       2       6       2       0
P4      0       5       10      10      5
P5      6       4       14      8       4

Average Turnaround Time = 8.20
Average Waiting Time = 4.20
```

Experiment 6: Simulate the following CPU scheduling algorithms: Round Robin

Theory:

Round Robin (RR) is a **preemptive CPU scheduling algorithm** designed to ensure fairness among all processes.

- Each process is assigned a fixed **time quantum** (time slice).
- Processes are maintained in a **ready queue** (circular FIFO queue).
- The CPU executes the first process in the queue for at most one-time quantum:
  - If the process **finishes within the quantum**, it leaves the system.
  - If it **still has remaining burst time**, it is placed at the **end of the queue**, and the CPU picks the next process.
- This continues until all processes are completed.

Characteristics:

1. **Fair scheduling**: Every process gets CPU time in equal intervals.
2. **Preemptive**: A process can be paused and resumed later.
3. **Good response time**: Especially useful in **time-sharing systems**.
4. **Performance** depends heavily on the size of **time quantum (TQ)**:
   - Too small → High overhead due to frequent context switching.
   - Too large → Approaches First-Come, First-Served (FCFS).

Metrics:

- **Completion Time (CT)**: Time when a process finishes.
- **Turnaround Time (TAT)** = CT – AT.
- **Waiting Time (WT)** = TAT – BT.

Source Code:

```
#include <iostream>
#include <iomanip>
#include <queue>
using namespace std;

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int completionTime;
    int turnAroundTime;
    int waitingTime;
    bool completed = false;
};

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

    Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter Arrival Time & Burst Time of P" << processes[i].pid << ":
";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

    cout << "Enter Time Quantum: ";
    cin >> tq;

    queue<int> q;
    int time = 0, completed = 0;
    bool visited[n] = {false};

    // Initially push all processes arriving at time 0
    for (int i = 0; i < n; i++) {
        if (processes[i].arrivalTime == 0) {
            q.push(i);
            visited[i] = true;
        }
    }

    while (completed < n) {
        if (q.empty()) {
            time++; // CPU idle
            for (int i = 0; i < n; i++) {
```

```
                if (!visited[i] && processes[i].arrivalTime <= time) {
                    q.push(i);
                    visited[i] = true;
                }
            }
            continue;
        }

        int idx = q.front(); q.pop();

        if (processes[idx].remainingTime > tq) {
            time += tq;
            processes[idx].remainingTime -= tq;
        } else {
            time += processes[idx].remainingTime;
            processes[idx].remainingTime = 0;
            processes[idx].completed = true;
            completed++;

            processes[idx].completionTime = time;
            processes[idx].turnAroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
            processes[idx].waitingTime = processes[idx].turnAroundTime -
processes[idx].burstTime;
        }

        // Add newly arrived processes
        for (int i = 0; i < n; i++) {
            if (!visited[i] && processes[i].arrivalTime <= time) {
                q.push(i);
                visited[i] = true;
            }
        }

        // If current process is not finished, put it back in queue
        if (!processes[idx].completed) {
            q.push(idx);
        }
    }

    // Output
    float totalTAT = 0, totalWT = 0;
    cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        totalTAT += processes[i].turnAroundTime;
        totalWT += processes[i].waitingTime;
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].completionTime << "\t"
             << processes[i].turnAroundTime << "\t"
```

```
             << processes[i].waitingTime << "\n";
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Turnaround Time = " << totalTAT / n;
    cout << "\nAverage Waiting Time = " << totalWT / n << "\n";

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter Arrival Time & Burst Time of P1: 0 6
Enter Arrival Time & Burst Time of P2: 1 3
Enter Arrival Time & Burst Time of P3: 2 5
Enter Arrival Time & Burst Time of P4: 3 1
Enter Arrival Time & Burst Time of P5: 4 4
Enter Time Quantum: 2

PID     AT      BT      CT      TAT     WT
P1      0       6       16      16      10
P2      1       3       12      11      8
P3      2       5       19      17      12
P4      3       1       9       6       5
P5      4       4       18      14      10

Average Turnaround Time = 12.80
Average Waiting Time = 9.00
```

Experiment 7: Simulate the Bankers' Algorithm for Deadlock Avoidance

Theory:

Banker's Algorithm is a **deadlock avoidance algorithm** developed by Edsger Dijkstra. It ensures that the system never enters an **unsafe state** by carefully allocating resources to processes.

The name comes from the analogy of a **banker** lending money to clients while ensuring he can satisfy all clients' needs in some order, without running out of funds.

Important Terms

1. **Allocation Matrix**: Current allocation of resources to each process.
2. **Max Matrix**: Maximum demand of each process.
3. **Need Matrix** = Max – Allocation.
4. **Available Vector**: Number of available resources of each type.
5. **Safe State**: A state in which there exists at least one sequence of processes (Safe Sequence) such that all processes can finish execution.
6. **Unsafe State**: A state where no such safe sequence exists → can potentially lead to deadlock.

Steps of Banker's Algorithm

1. **Check Initial Safety**:
   - Start with available resources.
   - See if there exists a process whose **Need ≤ Available**.
   - If yes, assume it finishes → release its resources → update Available.
   - Repeat until all processes are finished.
   - If possible → system is in a **Safe State**, otherwise **Unsafe**.
2. **Resource Request Algorithm** (when a process requests resources):
   - Check 1: `Request ≤ Need` → process can only ask up to its max demand.
   - Check 2: `Request ≤ Available` → resources must be available.
   - Temporarily allocate resources (pretend allocation).
   - Run **safety check** again.
   - If still safe → grant the request.
   - Otherwise, roll back allocation → process must wait.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

int n, m; // Global for convenience
int alloc[20][20], maxMat[20][20], need[20][20], avail[20];

// Function to check safe state and print sequence
bool isSafe() {
    int work[20], finish[20], safeSeq[20];
    int count = 0;

    // Initialize work and finish
    for (int i = 0; i < m; i++)
        work[i] = avail[i];
    for (int i = 0; i < n; i++)
        finish[i] = 0;

    while (count < n) {
        bool found = false;
        for (int i = 0; i < n; i++) {
            if (!finish[i]) {
                bool canExecute = true;
                for (int j = 0; j < m; j++) {
                    if (need[i][j] > work[j]) {
                        canExecute = false;
                        break;
                    }
                }
                if (canExecute) {
                    for (int k = 0; k < m; k++)
                        work[k] += alloc[i][k];
                    safeSeq[count++] = i;
                    finish[i] = 1;
                    found = true;
                }
            }
        }
        if (!found) {
            cout << "\nSystem is NOT in a safe state.\n";
            return false; // Not safe
        }
    }

    cout << "\nSystem is in a SAFE state.\nSafe Sequence: ";
    for (int i = 0; i < n; i++) {
        cout << "P" << safeSeq[i];
        if (i != n - 1)
            cout << " -> ";
    }
    cout << "\n";
```

```
    return true; // Safe
}

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

    cout << "\nEnter Allocation Matrix:\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> alloc[i][j];

    cout << "\nEnter Max Matrix:\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> maxMat[i][j];

    cout << "\nEnter Available Resources:\n";
    for (int i = 0; i < m; i++)
        cin >> avail[i];

    // Calculate Need matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = maxMat[i][j] - alloc[i][j];

    cout << "\nNeed Matrix:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << need[i][j] << " ";
        cout << "\n";
    }

    // Check initial safe state
    if (!isSafe()) {
        cout << "\nSystem is NOT in a safe state.\n";
        return 0;
    }

    // Resource request handling
    int reqProcess;
    cout << "\nEnter the process number making a request (0-" << n - 1 << "): ";
    cin >> reqProcess;

    int request[20];
    cout << "Enter request for each resource: ";
    for (int i = 0; i < m; i++)
        cin >> request[i];
```

```
    // Check 1: Request <= Need
    for (int i = 0; i < m; i++) {
        if (request[i] > need[reqProcess][i]) {
            cout << "\nError: Process has exceeded its maximum claim.\n";
            return 0;
        }
    }

    // Check 2: Request <= Available
    for (int i = 0; i < m; i++) {
        if (request[i] > avail[i]) {
            cout << "\nResources not available. Process must wait.\n";
            return 0;
        }
    }

    // Tentatively allocate
    for (int i = 0; i < m; i++) {
        avail[i] -= request[i];
        alloc[reqProcess][i] += request[i];
        need[reqProcess][i] -= request[i];
    }

    // Check if safe after allocation
    if (!isSafe()) {
        cout << "\nRequest cannot be granted (unsafe state).\n";
    } else {
        cout << "\nRequest can be granted safely.\n";
    }

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter number of resources: 3

Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
```

```
Enter Available Resources:
3 3 2

Need Matrix:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

System is in a SAFE state.
Safe Sequence: P1 -> P3 -> P4 -> P0 -> P2

Enter the process number making a request (0-4): 1
Enter request for each resource: 1 0 2

System is in a SAFE state.
Safe Sequence: P1 -> P3 -> P4 -> P0 -> P2

Request can be granted safely.
```

Experiment 8: Simulate Bankers Algorithm for Dead Lock Prevention

Theory:

Introduction:
Deadlock occurs in a multitasking system when a set of processes is **permanently blocked**, each waiting for resources held by the other. To prevent deadlocks, the system must **ensure that at least one of the necessary conditions for deadlock does not hold**.

Necessary Conditions for Deadlock (Coffman Conditions):
A deadlock can occur if all four conditions are simultaneously present:

1. **Mutual Exclusion:** Some resources are non-shareable.
2. **Hold and Wait:** A process holds at least one resource while waiting for others.
3. **No Preemption:** Resources cannot be forcibly removed from a process.
4. **Circular Wait:** A closed chain of processes exists, where each process waits for a resource held by the next process.

Deadlock Prevention:
Deadlock prevention is a set of strategies to ensure that **at least one of the Coffman conditions cannot hold**, thus **avoiding deadlocks entirely**.

Common Deadlock Prevention Techniques:

- **Eliminate Mutual Exclusion:** Make some resources sharable (e.g., read-only files).
- **Eliminate Hold-and-Wait:** Require processes to **request all resources at once**. If all are available, allocation proceeds; otherwise, the process waits without holding any resources.
- **Eliminate No Preemption:** Allow resources to be preempted from processes if necessary.
- **Eliminate Circular Wait:** Impose a **global order** on resources; processes must request resources in ascending order.

Implementation Example (Lab Simulation):
In the lab code:

- Each process requests **all of its maximum required resources at once**.
- If resources are available, they are allocated; the process executes and releases resources after completion.
- If resources are not available, the process waits without holding any resources.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

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

    int maxMat[n][m], avail[m];

    cout << "Enter Max Matrix:\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> maxMat[i][j];

    cout << "Enter Available Resources:\n";
    for (int i = 0; i < m; i++)
        cin >> avail[i];

    cout << "\n--- Deadlock Prevention Simulation ---\n";

    for (int i = 0; i < n; i++) {
        cout << "\nProcess P" << i << " requesting all resources at once...\n";
        bool canAllocate = true;
        for (int j = 0; j < m; j++) {
            if (maxMat[i][j] > avail[j]) {
                canAllocate = false;
                break;
            }
        }
        if (canAllocate) {
            cout << "Resources allocated to P" << i << ".\n";
            for (int j = 0; j < m; j++)
                avail[j] -= maxMat[i][j];

            cout << "Process P" << i << " finished and released resources.\n";
            for (int j = 0; j < m; j++)
                avail[j] += maxMat[i][j];
        } else {
            cout << "Process P" << i << " must wait (cannot hold some and wait
for others).\n";
        }
    }

    return 0;
}
```

Output:

```
Enter number of processes: 3
Enter number of resources: 3
Enter Max Matrix:
7 5 3
3 2 2
9 0 2
Enter Available Resources:
3 3 2

--- Deadlock Prevention Simulation ---

Process P0 requesting all resources at once...
Process P0 must wait (cannot hold some and wait for others).

Process P1 requesting all resources at once...
Resources allocated to P1.
Process P1 finished and released resources.

Process P2 requesting all resources at once...
Process P2 must wait (cannot hold some and wait for others).
```

Experiment 9: Basic Linux Commands Cheat Sheet

Theory:

📂 File & Directory Management

- `ls` → **List (lists directory contents)**
- `cd` → **Change Directory**
- `pwd` → **Print Working Directory (shows current location)**
- `mkdir` → **Make Directory**
- `rmdir` → **Remove Directory**
- `touch` → **Create an empty file (sets file timestamp)**
- `cat` → **Concatenate (display file contents, or join files)**
- `cp` → **Copy**
- `mv` → **Move (or rename files)**
- `rm` → **Remove (delete files)**

⚙️ Permissions & Ownership

- `chmod` → **Change Mode** (change file permissions)
- `chown` → **Change Owner**
- `chgrp` → **Change Group**
- `umask` → **User Mask** (default permissions mask)

🔍 File Viewing & Searching

- `less` → View file content **one screen at a time**
- `more` → Similar to `less` but older
- `head` → Show first lines of a file
- `tail` → Show last lines of a file
- `grep` → **Global Regular Expression Print** (search text in files)
- `find` → Search for files/directories
- `locate` → Find files using database index

📦 Package Management (Debian/Ubuntu)

- `apt` → **Advanced Package Tool**
- `dpkg` → **Debian Package**
- `snap` → Snappy Package Manager

🔒 Privileges & Superuser

- `sudo` → **SuperUser DO** (run command as root/admin)
- `su` → **Substitute User / Switch User** (become another user, often root)

🖥️ System & Process Management

- `ps` → **Process Status**
- `top` → Shows top running processes (real-time)
- `htop` → Improved interactive `top`
- `kill` → Terminate a process
- `df` → **Disk Free** (shows disk usage)
- `du` → **Disk Usage** (per file/folder)
- `uptime` → How long the system has been running
- `uname` → **Unix Name** (system information)
- `whoami` → Shows current logged-in user
- `id` → Show user ID and group ID info

🌐 Networking

- `ping` → **Packet Internet Groper** (test connectivity)
- `ssh` → **Secure Shell** (remote login)
- `scp` → **Secure Copy** (remote file transfer)
- `ftp` → **File Transfer Protocol**
- `curl` → **Client URL** (transfer data to/from server)
- `wget` → **Web Get** (download from web)
- `ifconfig` → **Interface Configurator** (network interfaces)
- `ip` → Replacement for `ifconfig`

📜 Archiving & Compression

- `tar` → **Tape Archive**
- `gzip` → **GNU Zip** (compression)
- `gunzip` → Decompress `.gz` files
- `zip/unzip` → Create/extract zip files

Source Code:

Category	Command	Purpose
System Update & Package Management	`sudo apt update`	Update package index
	`sudo apt upgrade`	Upgrade all installed packages
	`sudo apt install vim`	Install a package
	`sudo apt remove vim`	Remove a package
	`sudo apt remove package_name`	Remove a package (generic)
	`apt list --installed`	List installed packages
File & Directory Management	`ls`	List files in current directory
	`mkdir directory_name` / `mkdir myfolder`	Create a new directory
	`cd ..`	Move to parent directory
	`pwd`	Show current working directory
	`touch filename`	Create a new file
	`cat filename`	Display file contents
	`less filename`	Display file contents page by page
	`cp source_file destination`	Copy file
	`cp source_file destination_directory/`	Copy file to directory
	`mv old_filename new_filename`	Rename a file
	`mv file1.txt file2.txt /path/`	Move a file
	`rm filename`	Delete a file
	`rmdir directory_name`	Remove empty directory
	`rm -r directory_name`	Remove directory recursively
File Permissions	`chmod 644 myfile.txt`	Owner: read/write, Others: read
	`chmod 755 myfile`	Owner: all, Others: read/execute
	`chmod 777 filename`	Full permissions (all users)
	`chmod 000 filename`	Remove all permissions
	`ls -l filename`	Show file permissions
User & Group Management	`sudo useradd username`	Create new user
	`sudo userdel username`	Delete user
	`sudo passwd username`	Change user password
	`sudo passwd sajjad`	Change password for “sajjad”
	`sudo groupadd groupname`	Create group
	`sudo usermod -aG groupname username`	Add user to group
	`sudo gpasswd -d username groupname`	Remove user from group
	`getent group groupname`	List group members
System Control & Info	`sudo shutdown now`	Shutdown system
	`sudo reboot`	Restart system
	`date`	Show date & time
	`whoami`	Show current logged-in user
	`uname`	Show system information
Networking	`ip addr show`	Display IP address