PROGRAM 1

PROBLEM STATEMENT: Write a program to implement CPU scheduling for first come first serve.

THEORY:

First-Come, First-Served (FCFS) is one of the simplest and most straightforward CPU scheduling algorithms used in operating systems. It operates on a queue-based mechanism where processes are executed in the order they arrive in the ready queue. This means that the process that arrives first will be the first to get executed, and it will run to completion before the next process starts.

Basic Operation

- **Queue Management**: In FCFS scheduling, processes are maintained in a queue, typically a FIFO (First In, First Out) queue.
- **Execution Order**: When the CPU becomes available, the process at the head of the queue is selected for execution.
- **Completion**: A process runs to completion before the next process in the queue starts execution.

Waiting Time (WT): The average waiting time in FCFS is calculated as the sum of the waiting times for all processes divided by the number of processes. It tends to be high, especially if there is a large disparity in process burst times.

Turnaround Time (TAT): The turnaround time is the total time taken from the arrival of a process to its completion. In FCFS, turnaround time can also be high due to the sequential nature of execution.

ALGORITHM:

PROBLEM:

Process ID	Arrival Time	Burst Time
P1	0	4
P2	1	3
P3	2	1
P4	3	2
P5	4	5

PROBLEM SOLUTION:

CODE:

```
#include <stdio.h>

struct process
{
    int PID;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turn_around_time;
    int waiting_time;
};

void fcfs_scheduling(struct process processes[], int n)
{
    int current_time = 0;
    for(int i = 0; i < n; i++)
    {
        processes[i].completion_time = current_time +
processes[i].burst_time;
        processes[i].turn_around_time = processes[i].completion_time -
processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turn_around_time -
processes[i].burst_time;
        current_time += processes[i].burst_time;
    }
}

void print_processes_data(struct process processes[], int n)
{
    printf("\nProcess ID \t Arrival Time \t Burst Time \t Completion
Time \t Turn Around Time \t Waiting Time\n");
    for(int i = 0; i < n; i++)
    {
        printf("%d \t\t %d \t\t %d \t\t %d \t\t\t %d \t\t\t %d\n",
            processes[i].PID,
            processes[i].arrival_time,
            processes[i].burst_time,
            processes[i].completion_time,
            processes[i].turn_around_time,
            processes[i].waiting_time);
    }
}

int main()
{
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct process processes[n];

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

```
        printf("Enter Process ID, Process Arrival Time and Burst Time
for Process %d :\n",i+1);
        scanf("%d", &processes[i].PID);
        scanf("%d", &processes[i].arrival_time);
        scanf("%d", &processes[i].burst_time);
    }

    fcfs_scheduling(processes, n);
    print_processes_data(processes, n);

    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 5
Enter Process ID, Process Arrival Time and Burst Time for Process 1 :
1 0 4
Enter Process ID, Process Arrival Time and Burst Time for Process 2 :
2 1 3
Enter Process ID, Process Arrival Time and Burst Time for Process 3 :
3 2 1
Enter Process ID, Process Arrival Time and Burst Time for Process 4 :
4 3 2
Enter Process ID, Process Arrival Time and Burst Time for Process 5 :
5 4 5

Process ID      Arrival Time    Burst Time      Completion Time         Turn Around Time        Waiting Time
1               0               4               4                       4                       0
2               1               3               7                       6                       3
3               2               1               8                       6                       5
4               3               2               10                      7                       5
5               4               5               15                      11                      6
```

LEARNING OUTCOMES:

PROGRAM 2

PROBLEM STATEMENT: Write a program to implement CPU scheduling for shortest job first.

THEORY: Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest execution (burst) time from the ready queue. It is designed to minimize the average waiting time of processes by prioritizing shorter jobs, which typically leads to better overall system performance. SJF can be implemented in either non-preemptive or preemptive (Shortest Remaining Time First, SRTF) forms.

Basic Operation

- **Queue Management**: In SJF scheduling, processes are maintained in a ready queue. The key distinction is that the scheduler selects the process with the shortest burst time, not necessarily the one that arrived first.
- **Execution Order**: When the CPU becomes available, the scheduler scans the ready queue and picks the process with the smallest CPU burst. If the system is using preemptive SJF (SRTF), it will interrupt the current process if a new process arrives with a shorter remaining time.
- **Completion**: In non-preemptive SJF, a selected process runs to completion before the next process with the shortest burst time is chosen. In preemptive SJF (SRTF), the running process may be preempted if a shorter job arrives, and the CPU is assigned to the new process.

ALGORITHM:

PROBLEM:

Processes	Arrival Time	Burst Time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

PROBLEM SOLUTION:

CODE:

```
#include <stdio.h>

typedef struct {
    int id;
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
} process;

int main()
{
    int n = 5;
    process p[n], q[n];

    for (int i = 0; i < n; i++)
    {
        p[i].id = i + 1;
        printf("Enter arrival time and burst time for process P%d: ", i
+ 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (p[j].at > p[j + 1].at)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    int front = 0, rear = 1;
    int time = p[0].at;
    q[front] = p[0];
    int i = 1;

    while (i < n || front != rear)
    {
        time += q[front].bt;
        for (int k = 0; k < n; k++)
        {
            if (p[k].id == q[front].id)
            {
                p[k].ct = time;
                break;
            }
```

```
        }

        while (i < n && p[i].at <= time)
        {
            q[rear] = p[i];
            rear++;
            i++;
        }

        front++;

        for (int j = front + 1; j < rear; j++)
        {
            if (q[front].bt > q[j].bt)
            {
                process temp = q[front];
                q[front] = q[j];
                q[j] = temp;
            }
        }
    }

    for (int i = 0; i < n; i++)
    {
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (p[j].id > p[j + 1].id)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    printf("P\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt,
p[i].ct, p[i].tat, p[i].wt);
    }

    return 0;
}
```

OUTPUT:

```
Enter arrival time and burst time for process P1: 3 1
Enter arrival time and burst time for process P2: 1 4
Enter arrival time and burst time for process P3: 4 2
Enter arrival time and burst time for process P4: 0 6
Enter arrival time and burst time for process P5: 2 3
P       AT      BT      CT      TAT     WT
P1      3       1       7       4       3
P2      1       4       16      15      11
P3      4       2       9       5       3
P4      0       6       6       6       0
P5      2       3       12      10      7
```

LEARNING OUTCOMES:

PROGRAM 3

PROBLEM STATEMENT: Write a program to implement CPU scheduling for Round Robin.

THEORY:

Round Robin (RR) is a CPU scheduling algorithm designed for time-sharing systems. It is one of the simplest and most widely used scheduling techniques. In this algorithm, each process is assigned a fixed **time quantum** or **time slice** during which it can execute. If a process doesn't finish within its assigned time quantum, it is preempted and moved to the back of the ready queue, allowing the next process to execute.

Basic Operation

- **Queue Management**: Processes are maintained in a **FIFO (First In, First Out)** queue. When a process arrives, it is added to the end of the ready queue.
- **Execution Order**: The CPU scheduler picks the process at the front of the queue for execution. The process is allowed to run for a time period equal to the **time quantum**. If the process finishes its execution before the time quantum expires, it is removed from the queue. Otherwise, the process is preempted and placed at the back of the queue, and the next process in line is selected for execution.
- **Completion**: A process continues this cycle of execution and requeueing until it completes. Once a process finishes, it is removed from the system and no longer re-enters the queue.

ALGORITHM:

PROBLEM:

Processes	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

PROBLEM SOLUTION:

CODE:

```
#include <stdio.h>

typedef struct {
    int id;
    int at;  // Arrival Time
    int bt;  // Burst Time
    int ct;  // Completion Time
    int tat; // Turnaround Time
    int wt;  // Waiting Time
} process;

int main()
{
    printf("Enter number of processes: ");
    int n;
    scanf("%d", &n);
    process p[n], rq[64];
    int tq = 2; // Time Quantum
    int front = 0, rear = 1;

    for (int i = 0; i < n; i++)
    {
        p[i].id = i + 1;
        printf("Enter arrival time and burst time for P%d: ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }

    // Sort processes according to arrival time
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (p[j].at > p[j + 1].at)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    rq[front] = p[0];
    int time = p[0].at;
    int i = 1;

    while (i < n || front != rear)
    {
        if (rq[front].bt > tq)
        {
            time += tq;
            rq[front].bt -= tq;
            while (i < n && p[i].at <= time)
            {
```

```
                rq[rear] = p[i];
                rear++;
                i++;
            }
            rq[rear++] = rq[front];  // Place back in the queue
        }
        else
        {
            time += rq[front].bt;
            for (int k = 0; k < n; k++)
            {
                if (p[k].id == rq[front].id)
                {
                    p[k].ct = time;  // Set completion time
                    break;
                }
            }
        }
        front++;
    }

    // Calculate Turnaround Time (TAT) and Waiting Time (WT)
    for (int i = 0; i < n; i++)
    {
        p[i].tat = p[i].ct - p[i].at;  // TAT = CT - AT
        p[i].wt = p[i].tat - p[i].bt;  // WT = TAT - BT
    }

    // Arrange processes according to their IDs
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (p[j].id > p[j + 1].id)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    printf("P\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt,
p[i].ct, p[i].tat, p[i].wt);
    }

    return 0;
}
```

OUTPUT:

```
Enter number of processes: 5
Enter arrival time and burst time for P1: 0 5
Enter arrival time and burst time for P2: 1 3
Enter arrival time and burst time for P3: 2 1
Enter arrival time and burst time for P4: 3 2
Enter arrival time and burst time for P5: 4 3
P       AT      BT      CT      TAT     WT
P1      0       5       13      13      8
P2      1       3       12      11      8
P3      2       1       5       3       2
P4      3       2       9       6       4
P5      4       3       14      10      7
```

LEARNING OUTCOMES:

PROGRAM 4

PROBLEM STATEMENT: Write a program to perform priority scheduling.

THEORY:

Priority Scheduling is a type of scheduling algorithm used in operating systems, where each process is assigned a priority level. The process with the highest priority is selected for execution first. If multiple processes have the same priority, they may be scheduled according to a secondary criterion, such as First-Come, First-Served (FCFS) or Round Robin (RR).

Key Concepts:

1. **Priority Assignment:** Each process is given a priority number, typically an integer. The lower the number, the higher the priority (although in some systems, higher numbers indicate higher priorities). Priorities can be assigned based on various factors such as resource requirements, importance, or time constraints.

2. **Types of Priority Scheduling:**

   - **Preemptive Priority Scheduling:** In this variant, if a new process arrives with a higher priority than the currently running process, the running process is preempted (paused), and the new process is executed.
   - **Non-Preemptive Priority Scheduling:** In this variant, the running process is allowed to complete, even if a new process arrives with a higher priority.
3. **Priority Inversion:** A situation where a lower-priority process holds a resource needed by a higher-priority process, effectively inverting the priority. This can cause the higher-priority process to wait indefinitely. To avoid this, mechanisms like **priority inheritance** can be implemented, where the lower-priority process temporarily inherits the priority of the higher-priority process.

4. **Starvation (Indefinite Blocking):** One drawback of priority scheduling is **starvation**, where lower-priority processes may never get executed if higher-priority processes continuously arrive. To prevent this, some systems use **aging**, where the priority of a process increases the longer it waits.

5. **Advantages:**

   - Efficient for real-time systems where time-sensitive processes must be executed promptly.
   - Helps ensure that critical processes get CPU time quickly.
6. **Disadvantages:**

   - Can lead to starvation of lower-priority processes.
   - Managing priority dynamically may introduce complexity.

ALGORITHM:

PROBLEM:

NUMERICAL:-

Consider the set of 5 processes whose Arrival Time, Burst Time and Priority is given. Calculate the average Turnaround Time and Waiting Time via Priority Scheduling algorithm -

1. **Preemptive mode**
2. **Non-Preemptive mode**

(Higher number represents higher priority)

PROCESS ID	PRIORITY	ARRIVAL TIME	BURST TIME
P1	4	0	4
P2	5	1	5
P3	7	2	1
P4	2	3	2
P5	1	4	3
P6	6	5	6

PROBLEM SOLUTION:

CODE:

a) Preemptive

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

struct Process {
    int id;
    int burstTime;
    int priority;
    int arrivalTime;
    int waitingTime;
    int turnAroundTime;
    int completionTime;
    int remainingTime;
    int isCompleted;
};

int findNextProcess(struct Process proc[], int n, int currentTime)
{
    int highestPriorityIdx = -1;
    for (int i = 0; i < n; i++) {
        if (proc[i].arrivalTime <= currentTime &&
proc[i].isCompleted == 0) {
            if (highestPriorityIdx == -1 || proc[i].priority >
proc[highestPriorityIdx].priority) {
                highestPriorityIdx = i;
            }
        }
    }
    return highestPriorityIdx;
}

void calculateCompletionTime(struct Process proc[], int n) {
    int currentTime = 0;
    int completedProcesses = 0;

    while (completedProcesses < n) {
        int idx = findNextProcess(proc, n, currentTime);

        if (idx == -1) {
            currentTime++;
            continue;
        }

        proc[idx].remainingTime--;
        if (proc[idx].remainingTime == 0) {
            currentTime++;
            proc[idx].completionTime = currentTime;
            proc[idx].isCompleted = 1;
```

```
            completedProcesses++;
        } else {
            currentTime++;
        }
    }
}

void calculateTurnAroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].turnAroundTime = proc[i].completionTime -
proc[i].arrivalTime;
    }
}

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

void displayTable(struct Process proc[], int n) {
    printf("\nProcess ID\tArrival Time\tPriority\tBurst
Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");

printf("--------------------------------------------------------
----------------------------------------------\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
                proc[i].id, proc[i].arrivalTime, proc[i].priority,
proc[i].burstTime,
                proc[i].completionTime, proc[i].turnAroundTime,
proc[i].waitingTime);
    }
}

void displayAvgTime(struct Process proc[], int n) {
    int totalWaitingTime = 0;
    int totalTurnAroundTime = 0;

    for (int i = 0; i < n; i++) {
        totalWaitingTime += proc[i].waitingTime;
        totalTurnAroundTime += proc[i].turnAroundTime;
    }

    printf("\nAverage Waiting Time: %.2f\n",
(float)totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n",
(float)totalTurnAroundTime / n);
}

int main() {
```

```
    int n;

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

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;
        printf("Enter arrival time, burst time and priority for
Process %d: ", proc[i].id);
        scanf("%d", &proc[i].arrivalTime);
        scanf("%d", &proc[i].burstTime);
        scanf("%d", &proc[i].priority);
        proc[i].isCompleted = 0;
        proc[i].remainingTime = proc[i].burstTime;
    }

    calculateCompletionTime(proc, n);
    calculateTurnAroundTime(proc, n);
    calculateWaitingTime(proc, n);

    displayTable(proc, n);
    displayAvgTime(proc, n);

    return 0;
}
```

Output:

```
sriyanshuazad@Sriyanshus-MacBook-Air os % ./priority2
Enter the number of processes: 6
Enter arrival time, burst time and priority for Process 1: 0 4 4
Enter arrival time, burst time and priority for Process 2: 1 5 5
Enter arrival time, burst time and priority for Process 3: 2 1 7
Enter arrival time, burst time and priority for Process 4: 3 2 2
Enter arrival time, burst time and priority for Process 5: 4 3 1
Enter arrival time, burst time and priority for Process 6: 5 6 6

Process ID      Arrival Time    Priority        Burst Time      Completion Time Turnaround Time Waiting Time
------------------------------------------------------------------------------------------------------------
1               0               4               4               16              16              12
2               1               5               5               13              12              7
3               2               7               1               3               1               0
4               3               2               2               18              15              13
5               4               1               3               21              17              14
6               5               6               6               11              6               0

Average Waiting Time: 7.67
Average Turnaround Time: 11.17
```

b) Non Preemptive

```
#include <stdio.h>

struct Process {
    int id;
    int burstTime;
    int priority;
    int arrivalTime;
    int waitingTime;
    int turnAroundTime;
    int completionTime;
    int isCompleted;
};

int findNextProcess(struct Process proc[], int n, int currentTime)
{
    int highestPriorityIdx = -1;
    for (int i = 0; i < n; i++) {
        if (proc[i].arrivalTime <= currentTime &&
proc[i].isCompleted == 0) {
            if (highestPriorityIdx == -1 || proc[i].priority >
proc[highestPriorityIdx].priority) {
                highestPriorityIdx = i;
            }
        }
    }
    return highestPriorityIdx;
}

void calculateCompletionTime(struct Process proc[], int n) {
    int currentTime = 0;
    int completedProcesses = 0;

    while (completedProcesses < n) {
        int idx = findNextProcess(proc, n, currentTime);

        if (idx == -1) {
            currentTime++;
            continue;
        }

        currentTime += proc[idx].burstTime;
        proc[idx].completionTime = currentTime;
        proc[idx].isCompleted = 1;
        completedProcesses++;
    }
}

void calculateTurnAroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
```

```
        proc[i].turnAroundTime = proc[i].completionTime -
proc[i].arrivalTime;
    }
}

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

void displayTable(struct Process proc[], int n) {
    printf("\nProcess ID\tArrival Time\tPriority\tBurst
Time\tCompletionTime\tTurnaroundTime\tWaitingTime\n");

printf("--------------------------------------------------------------------------------------------------------\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
                proc[i].id, proc[i].arrivalTime, proc[i].priority,
proc[i].burstTime,
                proc[i].completionTime, proc[i].turnAroundTime,
proc[i].waitingTime);
    }
}

void displayavgTime(struct Process proc[], int n) {
    int totalWaitingTime = 0;
    int totalTurnAroundTime = 0;

    for (int i = 0; i < n; i++) {
        totalWaitingTime += proc[i].waitingTime;
        totalTurnAroundTime += proc[i].turnAroundTime;
    }

    printf("\nAverage Waiting Time: %.2f\n",
(float)totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n",
(float)totalTurnAroundTime / n);
}

int main() {
    int n;

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

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;
```

```
        printf("Enter arrival time, burst time and priority for Process %d: ", proc[i].id);
        scanf("%d", &proc[i].arrivalTime);
        scanf("%d", &proc[i].burstTime);
        scanf("%d", &proc[i].priority);
        proc[i].isCompleted = 0;
    }

    calculateCompletionTime(proc, n);
    calculateTurnAroundTime(proc, n);
    calculateWaitingTime(proc, n);

    displayTable(proc, n);

    displayavgTime(proc, n);

    return 0;
}
```

Output:

```
Enter the number of processes: 6
Enter arrival time, burst time and priority for Process 1: 0 4 4
Enter arrival time, burst time and priority for Process 2: 1 5 5
Enter arrival time, burst time and priority for Process 3: 2 1 7
Enter arrival time, burst time and priority for Process 4: 3 2 2
Enter arrival time, burst time and priority for Process 5: 4 3 6
Enter arrival time, burst time and priority for Process 6: 5 6 6

Process ID      Arrival Time    Priority        Burst Time      CompletionTime  TurnaroundTime  WaitingTime
-----------------------------------------------------------------------------------------------------------
1               0               4               4               4               4               0
2               1               5               5               19              18              13
3               2               7               1               5               3               2
4               3               2               2               21              18              16
5               4               6               3               8               4               1
6               5               6               6               14              9               3

Average Waiting Time: 5.83
Average Turnaround Time: 9.33
```

LEARNING OUTCOME:

PROGRAM 5

PROBLEM STATEMENT: Write a program for page replacement policy using a) LRU b) FIFO c) Optimal.

THEORY:

A **page replacement policy** is a set of rules used by an operating system to decide which memory pages to swap out, write to disk, or replace when a new page needs to be loaded into main memory but memory is already full. The need for page replacement arises in **paging** systems, where processes may require more memory than what is available in RAM, and so pages of memory are swapped between RAM and secondary storage (like a hard drive or SSD).

The goal of a page replacement policy is to **minimize page faults**. A **page fault** occurs when a program tries to access a page that is not currently in memory, and the system has to retrieve it from disk. Since disk access is much slower than memory access, minimizing page faults improves overall system performance.

Types of Page Replacement Policies:

1. **First-In-First-Out (FIFO)**
   - **Concept**: The oldest page in memory is replaced first.
   - **Implementation**: The pages are stored in the order they arrive. When a new page needs to be loaded, the oldest page (the one that arrived first) is removed.
   - **Pros**: Simple to implement.
   - **Cons**: Does not consider how often or recently a page has been used, leading to **Belady's anomaly** where increasing the number of page frames can lead to more page faults.
2. **Least Recently Used (LRU)**
   - **Concept**: The page that has not been used for the longest time is replaced.
   - **Implementation**: The system keeps track of when each page was last used, either by timestamping or maintaining a linked list of pages in the order of use.
   - **Pros**: Pages that have been used recently are likely to be used again soon, reducing page faults.
   - **Cons**: Requires more overhead to maintain usage history.
3. **Optimal Page Replacement (OPT)**
   - **Concept**: The page that will not be used for the longest period in the future is replaced.
   - **Implementation**: It looks ahead and predicts future requests (which is not feasible in practice but is used for benchmarking other algorithms).
   - **Pros**: Minimizes the number of page faults, as it replaces the page that will not be needed for the longest time.
   - **Cons**: Requires future knowledge of page requests, which is impractical in real systems. It is mainly used as a theoretical comparison standard.

ALGORITHM:

PROBLEM:

Consider a reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. the number of frames in the memory is 3. Find out the number of page faults respective to:

PROBLEM SOLUTION:

LRU

REQUEST	4	7	6	1	7	6	1	2	7	2
FRAME 1										
FRAME 2										
FRAME 3										
HIT/MISS										

Page Faults in LRU-

FIFO

REQUEST	4	7	6	1	7	6	1	2	7	2
FRAME 1										
FRAME 2										
FRAME 3										
HIT/MISS										

Page Faults in FIFO -

Optimal

REQUEST	4	7	6	1	7	6	1	2	7	2
FRAME 1										
FRAME 2										
FRAME 3										
HIT/MISS										

Page Faults in Optimal Page Replacement -

a) **LRU**

CODE:

```
#include <stdio.h>

void getRefString(int *ref_string, int n)
{
    printf("Enter reference string: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ref_string[i]);
    }
}

int findLRUPage(int *last_used, int frame_size)
{
    int min = last_used[0];
    int pos = 0;

    for (int i = 1; i < frame_size; i++)
    {
        if (last_used[i] < min)
        {
            min = last_used[i];
            pos = i;
        }
    }

    return pos;
}

int main()
{
    int n;
    printf("Enter size of reference String: ");
    scanf("%d", &n);
    int ref_string[n];
    getRefString(ref_string, n);
    int frame_size;
    printf("Enter frame size: ");
    scanf("%d", &frame_size);
    int frame[frame_size];
    int last_used[frame_size];
    int hits = 0, page_faults = 0, time = 0;


    for (int i = 0; i < frame_size; i++)
    {
        frame[i] = -1;
```

```
        last_used[i] = -1;
    }

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

        // Check if the page is already in the frame
        for (int j = 0; j < frame_size; j++)
        {
            if (frame[j] == ref_string[i])
            {
                hits++;
                found = 1;
                last_used[j] = time++;
                break;
            }
        }


        if (!found)
        {
            int empty_found = 0;

            for (int j = 0; j < frame_size; j++)
            {
                if (frame[j] == -1)
                {
                    frame[j] = ref_string[i];
                    last_used[j] = time++;
                    page_faults++;
                    empty_found = 1;
                    break;
                }
            }


            if (!empty_found)
            {
                int pos = findLRUPage(last_used, frame_size);
                frame[pos] = ref_string[i];
                last_used[pos] = time++;
                page_faults++;
            }
        }
    }

    double hit_ratio = (double)hits / n;
    double page_fault_ratio = (double)page_faults / n;
    printf("Hits: %d\n", hits);
    printf("Page Faults: %d\n", page_faults);
    printf("Hit Ratio: %f\n", hit_ratio);
```

```
    printf("Page Fault Ratio: %f\n", page_fault_ratio);

    printf("Final frame state: ");
    for (int i = 0; i < frame_size; i++)
    {
        printf("%d ", frame[i]);
    }
    printf("\n");

    return 0;
}
```

OUTPUT:

```
Enter size of reference String: 10
Enter reference string: 4 7 6 1 7 6 1 2 7 2
Enter frame size: 3
Hits: 4
Page Faults: 6
Hit Ratio: 0.400000
Page Fault Ratio: 0.600000
Final frame state: 1 2 7
```

b) **FIFO**

CODE:

```
#include <stdio.h>

void getRefString(int *ref_string, int n)
{
    printf("Enter reference string: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ref_string[i]);
    }
}

void fifoPageReplacement(int *ref_string, int n, int frame_size)
{
    int hits = 0, page_faults = 0, frame_ptr = 0;
    int frame[frame_size];
    double hit_ratio, page_fault_ratio;

    for (int i = 0; i < frame_size; i++)
    {
        frame[i] = -1;
```

```
    }

    for (int i = 0; i < n; i++)
    {
        int found = 0;
        for (int j = 0; j < frame_size; j++)
        {
            // if page is already in frame
            if (frame[j] == ref_string[i])
            {
                hits++;
                found = 1;
                break;
            }
        }

        if (found == 0)
        {

            frame[frame_ptr] = ref_string[i];
            page_faults++;
            frame_ptr = (frame_ptr + 1) % frame_size;
        }
    }
    hit_ratio = (double)hits / n;
    page_fault_ratio = (double)page_faults / n;
    printf("Hits: %d\n", hits);
    printf("Page Faults: %d\n", page_faults);
    printf("Hit Ratio: %f\n", hit_ratio);
    printf("Page Fault Ratio: %f\n", page_fault_ratio);
}

int main()
{
    int n, frame_size;
    printf("Enter size of reference String: ");
    scanf("%d", &n);
    int ref_string[n];
    getRefString(ref_string, n);
    printf("Enter frame size: ");
    scanf("%d", &frame_size);
    int frame[frame_size];

    for (int i = 0; i < frame_size; i++)
    {
        frame[i] = -1;
    }

    fifoPageReplacement(ref_string, n, frame_size);

    return 0;
}
```

OUTPUT:

```
Enter size of reference String: 10
Enter reference string: 4 7 6 1 7 6 1 2 7 2
Enter frame size: 3
Hits: 4
Page Faults: 6
Hit Ratio: 0.400000
Page Fault Ratio: 0.600000
```

c) **Optimal**

CODE:

```
#include <stdio.h>

void getRefString(int *ref_string, int n)
{
    printf("Enter reference string: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &ref_string[i]);
    }
}

int findOptimalPage(int *frame, int frame_size, int *ref_string,
int n, int current_index)
{
    int farthest = current_index;
    int replace_index = -1;

    for (int i = 0; i < frame_size; i++)
    {
        int j;
        for (j = current_index + 1; j < n; j++)
        {
            if (frame[i] == ref_string[j])
            {
                if (j > farthest)
                {
                    farthest = j;
                    replace_index = i;
                }
                break;
            }
        }

              if (j == n)
```

```
        {
            return i;
        }

             if (replace_index == -1)
        {
            replace_index = i;
        }
    }

    return replace_index;
}

int main()
{
    int n;
    printf("Enter size of reference String: ");
    scanf("%d", &n);
    int ref_string[n];
    getRefString(ref_string, n);
    int frame_size;
    printf("Enter frame size: ");
    scanf("%d", &frame_size);
    int frame[frame_size];
    int hits = 0, page_faults = 0;

      for (int i = 0; i < frame_size; i++)
    {
        frame[i] = -1;
    }

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

        // Check if page is already in frame
        for (int j = 0; j < frame_size; j++)
        {
            if (frame[j] == ref_string[i])
            {
                hits++;
                found = 1;
                break;
            }
        }

              if (!found)
        {
            int empty_found = 0;
            // Check if there's an empty frame available
            for (int j = 0; j < frame_size; j++)
            {
```

```
                if (frame[j] == -1)
                {
                    frame[j] = ref_string[i];
                    page_faults++;
                    empty_found = 1;
                    break;
                }
            }

            if (!empty_found)
            {
                // Find the page to replace using the optimal
policy
                int pos = findOptimalPage(frame, frame_size,
ref_string, n, i);
                frame[pos] = ref_string[i];
                page_faults++;
            }}}

    double hit_ratio = (double)hits / n;
    double page_fault_ratio = (double)page_faults / n;
    printf("Hits: %d\n", hits);
    printf("Page Faults: %d\n", page_faults);
    printf("Hit Ratio: %f\n", hit_ratio);
    printf("Page Fault Ratio: %f\n", page_fault_ratio);

    printf("Final frame state: ");
    for (int i = 0; i < frame_size; i++)
    {
        printf("%d ", frame[i]);
    }
    printf("\n");

    return 0;
}
```

OUTPUT:

```
Enter size of reference String: 10
Enter reference string: 4 7 6 1 7 6 1 2 7 2
Enter frame size: 3
Hits: 5
Page Faults: 5
Hit Ratio: 0.500000
Page Fault Ratio: 0.500000
Final frame state: 2 7 6
```

LEARNING OUTCOME:

PROGRAM 6

PROBLEM STATEMENT: Write a program to implement first fit, best fit and worst fit algorithm for memory management.

THEORY:

First Fit, Best Fit, and Worst Fit are memory management algorithms used in operating systems to allocate memory blocks to processes. Each approach has unique strategies to allocate free memory, balancing between minimising fragmentation and maximising resource utilisation.

First Fit Algorithm
The first-fit algorithm searches for the first free partition that is large enough to accommodate the process. The operating system starts searching from the beginning of the memory and allocates the first free partition that is large enough to fit the process.

Best Fit Algorithm
The best-fit algorithm searches for the smallest free partition that is large enough to accommodate the process. The operating system searches the entire memory and selects the free partition that is closest in size to the process.

Worst Fit Algorithm
The worst-fit algorithm searches for the largest free partition and allocates the process to it. This algorithm is designed to leave the largest possible free partition for future use.

ALGORITHM:

PROBLEM:

Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

Memory Block	100 KB	500 KB	200 KB	300 KB	600KB
Processes	212 KB **(P1)**	417KB **(P2)**	112 KB **(P3)**	426 KB **(P4)**	

PROBLEM SOLUTION:

CODE:

```
#include <stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n)
{
    int allocation[n];
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }

    printf("\nFirst Fit Allocation:\nProcess No.\tProcess 
Size\tBlock No.\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

void bestFit(int blockSize[], int m, int processSize[], int n)
{
    int allocation[n];
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++)
    {
        int bestIdx = -1;
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (bestIdx == -1 || blockSize[j] < 
blockSize[bestIdx])
                    bestIdx = j;
            }
```

```
        }

        if (bestIdx != -1)
        {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    printf("\nBest Fit Allocation:\nProcess No.\tProcess
Size\tBlock No.\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

void worstFit(int blockSize[], int m, int processSize[], int n)
{
    int allocation[n];
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++)
    {
        int worstIdx = -1;
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (worstIdx == -1 || blockSize[j] >
blockSize[worstIdx])
                    worstIdx = j;
            }
        }

        if (worstIdx != -1)
        {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }

    printf("\nWorst Fit Allocation:\nProcess No.\tProcess
Size\tBlock No.\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
```

```
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main()
{
    int m, n;

    printf("Enter the number of blocks: ");
    scanf("%d", &m);
    int blockSize[m];
    printf("Enter the sizes of the blocks:\n");
    for (int i = 0; i < m; i++)
        scanf("%d", &blockSize[i]);

    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processSize[n];
    printf("Enter the sizes of the processes:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &processSize[i]);

    int blockSize1[m], blockSize2[m], blockSize3[m];
    for (int i = 0; i < m; i++)
    {
        blockSize1[i] = blockSize[i];
        blockSize2[i] = blockSize[i];
        blockSize3[i] = blockSize[i];
    }

    firstFit(blockSize1, m, processSize, n);
    bestFit(blockSize2, m, processSize, n);
    worstFit(blockSize3, m, processSize, n);

    return 0;
}
```

OUTPUT:

```
● sriyanshuazad@Sriyanshus-MacBook-Air os % code fit.c
● sriyanshuazad@Sriyanshus-MacBook-Air os % cd "/Users/sriy
Enter the number of blocks: 5
Enter the sizes of the blocks:
100 500 200 300 600
Enter the number of processes: 4
Enter the sizes of the processes:
212 417 112 400

First Fit Allocation:
Process No.     Process Size    Block No.
1               212             2
2               417             5
3               112             2
4               400             Not Allocated

Best Fit Allocation:
Process No.     Process Size    Block No.
1               212             4
2               417             2
3               112             3
4               400             5

Worst Fit Allocation:
Process No.     Process Size    Block No.
1               212             5
2               417             2
3               112             5
4               400             Not Allocated
○ sriyanshuazad@Sriyanshus-MacBook-Air os %
```

LEARNING OUTCOME:

PROGRAM 7

PROBLEM STATEMENT: Write a program to implement reader/writer problem using semaphore.

THEORY:

The **Reader-Writer Problem** is a classic synchronization problem in computer science, which deals with allowing multiple processes to access a shared resource (like a database) while preventing data inconsistency. The problem is particularly useful in operating systems and databases where processes (or threads) need to read or write shared data.
In the Reader-Writer problem, there are two types of processes:

1. **Readers**: Processes that only read the shared data. Multiple readers can access the data simultaneously without causing inconsistencies.
2. **Writers**: Processes that modify (write to) the shared data. Only one writer can access the data at a time to avoid data corruption.

Problem Statement
The challenge in the Reader-Writer Problem is to design a system that:
- Allows multiple readers to read simultaneously.
- Restricts writers so that only one writer can access the shared data at a time.
- Prevents readers from accessing the data while a writer is modifying it.

There are two main variations of the problem:
1. **First Reader-Writers Problem (No Starvation for Readers)**: This version gives priority to readers, allowing them to read as long as there is no writer waiting.
2. **Second Reader-Writers Problem (No Starvation for Writers)**: This version gives priority to writers, ensuring that once a writer is waiting, no new readers can start reading until the writer has finished.

Solution Using Semaphores
Semaphores are used to implement synchronisation and avoid race conditions in the Reader-Writer Problem. Here, we commonly use three semaphores:
1. **mutex**: Ensures mutual exclusion when modifying the count of readers.
2. **wrt**: Controls access to the shared resource, ensuring mutual exclusion for writers.
3. **read_count**: Keeps track of the number of readers currently accessing the shared resource.

Semaphores:
- mutex and wrt are binary semaphores, which can take the values 0 or 1.
- read_count is an integer variable, protected by mutex, to keep track of the number of active readers.

PSEUDOCODE:

CODE:

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

pthread_mutex_t x = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t y = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int readercount = 0;
int writercount = 0;

void *reader(void *param) {
    int reader_id = *((int *)param);

   pthread_mutex_lock(&x);
    readercount++;
    if (readercount == 1) {
        pthread_mutex_lock(&y);
   }
    pthread_mutex_unlock(&x);

    printf("Reader %d is reading data\n", reader_id);
    usleep(100000);
    printf("Reader %d is done reading\n", reader_id);

   pthread_mutex_lock(&x);
    readercount--;
    printf("Reader %d is leaving. Remaining readers: %d\n",
reader_id, readercount);
    if (readercount == 0) {
        pthread_mutex_unlock(&y);
   }
    pthread_mutex_unlock(&x);

    return NULL;
}

void *writer(void *param) {
    int writer_id = *((int *)param);

    printf("Writer %d is trying to enter\n", writer_id);
    pthread_mutex_lock(&y);
    printf("Writer %d is writing data\n", writer_id);
    usleep(200000);
    printf("Writer %d is done writing\n", writer_id);
    pthread_mutex_unlock(&y);

    return NULL;
}
```

```c
int main() {
    int n2, i;
    printf("Enter the number of readers and writers: ");
    scanf("%d", &n2);
    printf("\n");

    pthread_t readerthreads[n2], writerthreads[n2];
    int reader_ids[n2], writer_ids[n2];

      for (i = 0; i < n2; i++) {
        reader_ids[i] = i + 1;
        writer_ids[i] = i + 1;
        pthread_create(&readerthreads[i], NULL, reader,
&reader_ids[i]);
        pthread_create(&writerthreads[i], NULL, writer,
&writer_ids[i]);
    }

      for (i = 0; i < n2; i++) {
        pthread_join(readerthreads[i], NULL);
        pthread_join(writerthreads[i], NULL);
    }

    pthread_mutex_destroy(&x);
    pthread_mutex_destroy(&y);

    return 0;
}
```

OUTPUT:

```
sriyanshuazad@Sriyanshus-MacBook-Air os % gcc reader.c -o reader
sriyanshuazad@Sriyanshus-MacBook-Air os % ./reader
Enter the number of readers and writers: 3

Writer 1 is trying to enter
Reader 2 is reading data
Reader 1 is reading data
Writer 2 is trying to enter
Reader 3 is reading data
Writer 3 is trying to enter
Reader 2 is done reading
Reader 2 is leaving. Remaining readers: 2
Reader 1 is done reading
Reader 1 is leaving. Remaining readers: 1
Reader 3 is done reading
Reader 3 is leaving. Remaining readers: 0
Writer 1 is writing data
Writer 1 is done writing
Writer 2 is writing data
Writer 2 is done writing
Writer 3 is writing data
Writer 3 is done writing
sriyanshuazad@Sriyanshus-MacBook-Air os %
```

LEARNING OUTCOME:

PROGRAM 8

PROBLEM STATEMENT:Write a program to implement Producer-Consumer problem using semaphores.

THEORY:

The Producer-Consumer problem is a classic synchronization issue in operating systems. It involves two types of processes: producers, which generate data, and consumers, which process that data. Both share a common buffer.

Problem Statement
The challenge is to ensure that the producer doesn't add data to a full buffer and the consumer doesn't remove data from an empty buffer while avoiding conflicts when accessing the buffer.

Semaphore
A semaphore is a synchronization tool used in computing to manage access to shared resources. It works like a signal that allows multiple processes or threads to coordinate their actions. Semaphores use counters to keep track of how many resources are available, ensuring that no two processes can use the same resource at the same time, thus preventing conflicts and ensuring orderly execution.

In the Producer-Consumer Problem, three semaphores are typically employed:

1. **mutex**: Ensures mutual exclusion, allowing only one process to access the buffer at a time. This avoids simultaneous read/write operations by multiple producers or consumers.
2. **empty**: Counts the number of empty slots available in the buffer, ensuring that the producer only produces when there is space available.
3. **full**: Counts the number of filled slots in the buffer, ensuring that the consumer only consumes when there is data available.

Solution Using Semaphores
Each producer and consumer action is governed by conditions enforced by the semaphores. The basic idea is:

Producer Process:

- Waits on the empty semaphore to check if there's space in the buffer.
- Waits on mutex to get exclusive access to the buffer and produce an item.
- Signals full after producing an item to indicate that there is now one more item for the consumer to consume.

Consumer Process:

- Waits on the full semaphore to check if there's an item to consume.
- Waits on mutex to get exclusive access to the buffer and consume an item.
- Signals empty after consuming an item to indicate there is now one more empty slot for the producer.

PSEUDOCODE:

CODE:

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

int mutex = 1;
int full = 0;
int empty, x = 0;

void producer() {
    --mutex;
    ++full;
    --empty;
    x++;
    printf("Producer produces item %d\n", x);
    ++mutex;
}

void consumer() {
    --mutex;
    --full;
    ++empty;
    printf("Consumer consumes item %d\n", x);
    x--;
    ++mutex;
}

void showBuffer() {
    printf("\nBuffer Status:");
    printf("\nFull slots: %d", full);
    printf("\nEmpty slots: %d", empty);
    printf("\nTotal items in buffer: %d\n", x);
}

int main() {
    int n, bufferSize;

    printf("Enter buffer size: ");
    scanf("%d", &bufferSize);

    empty = bufferSize;

    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit"
           "\n4. Press 4 to Show Buffer Status");

    while (1) {
        printf("\nEnter your choice: ");
        scanf("%d", &n);

        switch (n) {
```

```
        case 1:
            if (mutex == 1 && empty != 0) {
                producer();
            } else {
                printf("Buffer is full!\n");
            }
            break;

        case 2:
            if (mutex == 1 && full != 0) {
                consumer();
            } else {
                printf("Buffer is empty!\n");
            }
            break;

        case 3:
            exit(0);
            break;

        case 4:
            showBuffer();
            break;

        default:
            printf("Invalid choice! Please try again.");
        }
    }
}
```

OUTPUT:

```
sriyanshuazad@Sriyanshus-MacBook-Air os % code producer.c
sriyanshuazad@Sriyanshus-MacBook-Air os % gcc producer.c -o producer
sriyanshuazad@Sriyanshus-MacBook-Air os % ./producer
Enter buffer size: 3

1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
4. Press 4 to Show Buffer Status
Enter your choice: 4

Buffer Status:
Full slots: 0
Empty slots: 3
Total items in buffer: 0

Enter your choice: 1
Producer produces item 1

Enter your choice: 1
Producer produces item 2

Enter your choice: 1
Producer produces item 3

Enter your choice: 1
Buffer is full!

Enter your choice: 2
Consumer consumes item 3

Enter your choice: 4

Buffer Status:
Full slots: 2
Empty slots: 1
Total items in buffer: 2

Enter your choice: 2
Consumer consumes item 2

Enter your choice: 2
Consumer consumes item 1

Enter your choice: 2
Buffer is empty!

Enter your choice: 4

Buffer Status:
Full slots: 0
Empty slots: 3
Total items in buffer: 0

Enter your choice: 3
sriyanshuazad@Sriyanshus-MacBook-Air os %
```

LEARNING OUTCOME:

PROGRAM 9

PROBLEM STATEMENT: Write a program to implement Banker's algorithm for deadlock avoidance.

THEORY:

The Banker's Algorithm is a deadlock avoidance algorithm introduced by Edsger Dijkstra, designed to handle resource allocation in systems where multiple processes compete for limited resources. Named after a banking system analogy, it helps determine if a system can safely allocate resources to processes without running into a deadlock.

Key Concepts in Banker's Algorithm

1. **Safe State**: A system is in a "safe state" if there is a sequence in which processes can complete their execution without running into deadlock. In this state, each process can eventually receive the resources it needs and finish, releasing those resources back to the system.

2. **Unsafe State**: If there's no safe sequence that can ensure the successful completion of all processes, the system is in an "unsafe state." An unsafe state may lead to a deadlock, but it doesn't necessarily mean a deadlock has occurred yet.

3. **Resource Allocation**: Banker's Algorithm works by keeping track of allocated resources, available resources, and maximum resources needed by each process, and dynamically makes decisions about resource allocation based on these values.

Working of Banker's Algorithm

The algorithm operates under the assumption that each process must declare the maximum number of resources it might need. Based on this information, the system decides whether to grant a process's resource request by simulating the allocation and checking if the resulting state is safe.

The algorithm requires three main data structures:

- **Available**: A vector that represents the number of resources of each type currently available in the system.
- **Max**: A matrix where each row represents a process and each column represents the maximum number of resources of a particular type that a process may request.
- **Allocation**: A matrix that represents the number of resources of each type currently allocated to each process.
- **Need**: A matrix calculated by subtracting Allocation from Max for each process, representing the remaining resources each process will need to complete.

PSEUDOCODE:

PROBLEM:

Consider the following example of a system. Check whether the system is safe or not using banker's algorithm. Determine the sequence if it safe.

AVAILABLE RESOURCES		
A	B	C
3	3	2

PROCESS ID	MAXIMUM NEED			CURRENT ALLOCATION			REMAINING NEED		
	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0			
P1	3	2	2	2	0	0			
P2	9	0	2	3	0	2			
P3	2	2	2	2	1	1			
P4	4	3	3	0	0	2			

PROBLEM SOLUTION:

CODE:

```
#include <stdio.h>
#define MAX 5

int available[MAX], max[MAX][MAX], allocation[MAX][MAX], need[MAX]
[MAX], n, m;

void calculateNeed()
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}

int checkSafeState()
{
    int work[MAX], finish[MAX] = {0}, safeSequence[MAX], count =
0;
    for (int i = 0; i < m; i++)
        work[i] = available[i];

    while (count < n)
    {
        int found = 0;
        for (int i = 0; i < n; i++)
        {
            if (finish[i] == 0)
            {
                int j;
                for (j = 0; j < m; j++)
                    if (need[i][j] > work[j])
                        break;

                if (j == m)
                {
                    for (int k = 0; k < m; k++)
                        work[k] += allocation[i][k];
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }

        if (found == 0)
        {
            printf("System is not in a safe state.\n");
            return 0;
        }
    }
```

```
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", safeSequence[i]);
    }
    printf("\n");
    return 1;
}

int main()
{
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resources: ");
    scanf("%d", &m);

    printf("Enter available resources: ");
    for (int i = 0; i < m; i++)
    {
        printf("Resource %d: ", i + 1);
        scanf("%d", &available[i]);
    }

    printf("Enter maximum resources for each process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Enter maximum resources for process %d: ", i + 1);
        for (int j = 0; j < m; j++)
        {

            scanf("%d", &max[i][j]);
        }
    }

    printf("Enter allocated resources for each process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Enter allocated resources for process %d: ", i +
1);
        for (int j = 0; j < m; j++)
        {
            scanf("%d", &allocation[i][j]);
        }
    }
    calculateNeed();
    checkSafeState();

    return 0;
}
```

OUTPUT:

```
sriyanshuazad@Sriyanshus-MacBook-Air os % gcc banker.c -o banker
sriyanshuazad@Sriyanshus-MacBook-Air os % ./banker
Enter the number of processes: 5
Enter the number of resources: 3

Enter available resources:
Resource 1: 3
Resource 2: 3
Resource 3: 2

Enter maximum resources for each process:
Enter maximum resources for process 1: 7 5 3
Enter maximum resources for process 2: 3 2 2
Enter maximum resources for process 3: 9 0 2
Enter maximum resources for process 4: 2 2 2
Enter maximum resources for process 5: 4 3 3

Enter allocated resources for each process:
Enter allocated resources for process 1: 0 1 0
Enter allocated resources for process 2: 2 0 0
Enter allocated resources for process 3: 3 0 2
Enter allocated resources for process 4: 2 1 1
Enter allocated resources for process 5: 0 0 2

System is in a safe state.
Safe sequence is: 1 3 4 0 2
sriyanshuazad@Sriyanshus-MacBook-Air os %
```

LEARNING OUTCOME:

PROGRAM 10

PROBLEM STATEMENT: Write C programs to implement the various File Organization Techniques

THEORY:

THEORY:-
File organization techniques in an operating system (OS) refer to methods used to store, manage, and access files on storage devices efficiently. Different techniques offer varying advantages depending on the requirements for quick access, minimal fragmentation, and easy maintenance. Here are the main file organization techniques:

Sequential Access

Sequential access is a file organization method where records are stored in a specific order, one after another, typically based on a key or a logical sequence. This organization is simple and commonly used for files that need to be processed in sequence, such as logs or transaction files. With sequential access, data is read or written starting from the beginning of the file, making it ideal for batch processing tasks where records are accessed in order.

Direct (or Hashed) Access

Direct, or hashed, access organizes records by computing their storage location using a hashing function, which maps a record's key directly to a specific address in memory or on disk. This method allows for rapid retrieval, as it enables direct access to a record without reading intervening records, making it efficient for applications where records need to be accessed frequently or randomly. When a record is needed, the hashing function is applied to its key, quickly locating the exact storage location.

Indexed File Organization

Indexed file organization improves random access by using an index, similar to a book's table of contents, which maintains pointers to the locations of records within the file. The index is typically built based on one or more key fields, and each entry in the index corresponds to a unique record in the file, containing both the key and the record's address. This organization enables efficient access to records by allowing direct retrieval of the desired record through the index without reading the entire file. Indexed files support both sequential and random access, making them versatile for a variety of applications. They are particularly advantageous when there is a need to frequently search for, insert, or delete records.

ALGORITHM:

CODE:

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

#define MAX_RECORDS 100
#define NAME_LENGTH 20

typedef struct {
    int id; // Unique identifier for the record
    char name[NAME_LENGTH]; // Name associated with the record
} Record;

void writeIndexed();
void readIndexed();
void writeHashed();
void readHashed();
void writeSequential();
void readSequential();

int main() {
    int choice;
    while (1) {
        printf("\n1. Write Indexed File");
        printf("\n2. Read Indexed File");
        printf("\n3. Write Hashed File");
        printf("\n4. Read Hashed File");
        printf("\n5. Write Sequential File");
        printf("\n6. Read Sequential File");
        printf("\n7. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                writeIndexed();
                break;
            case 2:
                readIndexed();
                break;
            case 3:
                writeHashed();
                break;
            case 4:
                readHashed();
                break;
            case 5:
                writeSequential();
                break;
            case 6:
                readSequential();
```

```
                break;
            case 7:
                exit(0);
            default:
                printf("Invalid choice. Please try again.");
        }
    }
    return 0;
}

void writeIndexed() {
    FILE *file = fopen("indexed.txt", "w");
    FILE *indexFile = fopen("index.txt", "w");
    if (!file || !indexFile) {
        printf("Error opening file!\n");
        return;
    }

    Record records[MAX_RECORDS];
    int recordCount = 0;
    char addMore;

    do {
        printf("Enter ID: ");
        scanf("%d", &records[recordCount].id);
        printf("Enter Name: ");
        scanf("%s", records[recordCount].name);
        recordCount++;

        printf("Do you want to add another record? (y/n): ");
        scanf(" %c", &addMore);
    } while (addMore == 'y' && recordCount < MAX_RECORDS);

    for (int i = 0; i < recordCount; i++) {
        fprintf(file, "%d %s\n", records[i].id,
records[i].name); // Write the record
        fprintf(indexFile, "%d %ld\n", records[i].id, ftell(file)
- sizeof(Record)); // Write ID and position to index
    }

    fclose(file);
    fclose(indexFile);
    printf("Records written to indexed.txt and index.txt.\n");
}

void readIndexed() {
    FILE *file = fopen("indexed.txt", "r");
    FILE *indexFile = fopen("index.txt", "r");
    if (!file || !indexFile) {
        printf("Error opening file!\n");
        return;
    }
```

```
    Record record;
    printf("\nIndexed Records:\n");
    while (fscanf(file, "%d %s", &record.id, record.name) != EOF)
{
        printf("ID: %d, Name: %s\n", record.id, record.name);
    }

    fclose(file);
    fclose(indexFile);
}

void writeHashed() {
    FILE *file = fopen("hashed.txt", "a");
    if (!file) {
        printf("Error opening file!\n");
        return;
    }

    Record records[MAX_RECORDS];
    int recordCount = 0;
    char addMore;

    do {
        printf("Enter ID: ");
        scanf("%d", &records[recordCount].id);
        printf("Enter Name: ");
        scanf("%s", records[recordCount].name);
        recordCount++;

        printf("Do you want to add another record? (y/n): ");
        scanf(" %c", &addMore);
    } while (addMore == 'y' && recordCount < MAX_RECORDS);

    for (int i = 0; i < recordCount; i++) {
        fprintf(file, "%d %s\n", records[i].id,
records[i].name); // Write the record
    }

    fclose(file);
    printf("Records written to hashed.txt.\n");
}

void readHashed() {
    FILE *file = fopen("hashed.txt", "r");
    if (!file) {
        printf("Error opening file!\n");
        return;
    }

    Record record;
    printf("\nHashed Records:\n");
```

```
    while (fscanf(file, "%d %s", &record.id, record.name) != EOF)
{
        printf("ID: %d, Name: %s\n", record.id, record.name);
    }

    fclose(file);
}

void writeSequential() {
    FILE *file = fopen("sequential.txt", "a");
    if (!file) {
        printf("Error opening file!\n");
        return;
    }

    Record records[MAX_RECORDS];
    int recordCount = 0;
    char addMore;

    do {
        printf("Enter ID: ");
        scanf("%d", &records[recordCount].id);
        printf("Enter Name: ");
        scanf("%s", records[recordCount].name);
        recordCount++;

        printf("Do you want to add another record? (y/n): ");
        scanf(" %c", &addMore);
    } while (addMore == 'y' && recordCount < MAX_RECORDS);

    for (int i = 0; i < recordCount; i++) {
        fprintf(file, "%d %s\n", records[i].id,
records[i].name); // Write the record
    }

    fclose(file);
    printf("Records written to sequential.txt.\n");
}

void readSequential() {
    FILE *file = fopen("sequential.txt", "r");
    if (!file) {
        printf("Error opening file!\n");
        return;
    }

    Record record;
    printf("\nSequential Records:\n");
    while (fscanf(file, "%d %s", &record.id, record.name) != EOF)
{
        printf("ID: %d, Name: %s\n", record.id, record.name);
    }
```

```
    fclose(file);
}
```

Initial Conditions:

```
∨ fileorganisation
    fileorg
    fileorg.c
    hashed.txt
    sequential.txt
```

```
hashed.txt ×
fileorganisation > has
1   1 Record1
2   2 Record2
3   3 Record3
```

```
sequential.txt ×
fileorganisation >
1   1 Record4
2   2 Record5
3   3 Record6
```

OUTPUT:

```
sriyanshuazad@Sriyanshus-MacBook-Air fileorganisation % ./fileorg

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 2
Error opening file!

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 1
Enter ID: 1
Enter Name: Record7
Do you want to add another record? (y/n): y
Enter ID: 2
Enter Name: Record8
Do you want to add another record? (y/n): n
Records written to indexed.txt and index.txt.

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 2

Indexed Records:
ID: 1, Name: Record7
ID: 2, Name: Record8
```

```
1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 3
Enter ID: 4
Enter Name: Record9
Do you want to add another record? (y/n): n
Records written to hashed.txt.

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 4

Hashed Records:
ID: 1, Name: Record1
ID: 2, Name: Record2
ID: 3, Name: Record3
ID: 4, Name: Record9
```

```
1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 5
Enter ID: 4
Enter Name: Record10
Do you want to add another record? (y/n): n
Records written to sequential.txt.

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 6

Sequential Records:
ID: 1, Name: Record4
ID: 2, Name: Record5
ID: 3, Name: Record6
ID: 4, Name: Record10
```

LEARNING OUTCOME: