PROGRAM - 1

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

BASIC TERMINOLOGY :-

- **Arrival Time (AT)** : Time at which the process arrives in the ready queue.
- **Completion Time (CT)** : Time at which process completes its execution.
- **Burst/Execution Time (BT/ET)** : Time required by a process for CPU execution.
- **Turn Around Time (TAT)** : Time taken by a process from entering ready state till it terminates. **TAT = CT -AT**
- **Waiting Time (WT)** : Time process spends waiting for CPU. **WT = TAT - BT**
- **Response Time :** Time duration between process getting into ready queue and process getting CPU for the first time.
  Response Time = CPU Allocation Time(when the CPU was allocated for the first) - AT

THEORY:- First Come First Serve (FCFS) is one of the simplest scheduling algorithms. Processes are executed in the order they arrive in the ready queue. Once a process starts execution, it runs to completion.
In case of a tie, process with smaller process ID is executed first. It is always non-preemptive in nature. Its implementation is based on FIFO queue.
It suffers from convoy effect.

Convoy effect :-
Convoy Effect is a situation where many processes, who need to use a resource for a short time, are blocked by one process holding that resource for a long time. This cause poor resource management.

ALGORITHM (handwritten):-

1 - Input the processes along with their burst time (BT).
2 - Find waiting time (WT) for all processes.
3 - As first process that comes need not to wait so
waiting time for process 1 will be 0 i.e.WT[0] = 0.
4 - Find waiting time for all other processes i.e. for
process i ->
WT[i] =BT[i-1] +WT[i-1] .
5 - Find turnaround time = waiting_time + burst_time
for all processes.
6 - Find average waiting time =
total_waiting_time / no_of_processes.
7 - Similarly, find average turnaround time =
total_turn_around_time / no_of_processes.

NUMERICAL:-

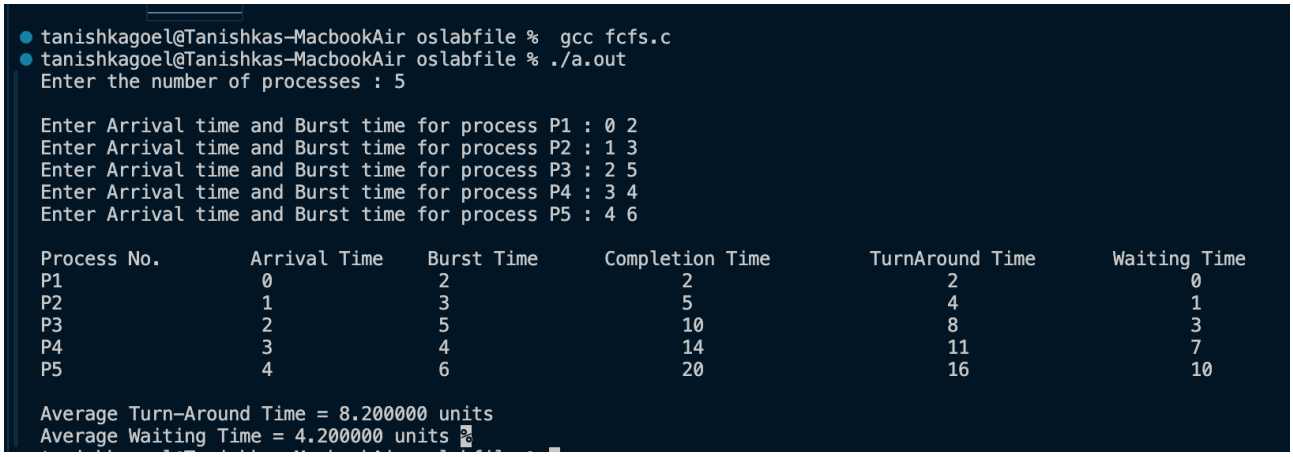
Consider the following processes with given Arrival Time and Burst Time. Calculate the average Waiting Time and Turn-Around Time using FCFS.

PROCESS ID	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P1	0	8			
P2	2	4			
P3	1	9			
P4	3	5			

CODE:-

```
#include <stdio.h>

int main()
{
    int n;

    printf("Enter the number of processes : ");
    scanf("%d", &n);
    printf("\n");
    int arrival_time[n];
    int burst_time[n];
    int completion_time[n];
    int turnaround_time[n];
    int waiting_time[n];
    int current_time = 0;
    float avg_wt = 0.0;
    float avg_tat = 0.0;

    for (int i = 0; i < n; i++)
    {
        printf("Enter Arrival time and Burst time for process P%d : ", i
+ 1);
        scanf("%d %d", &arrival_time[i], &burst_time[i]);
    }

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

        if (current_time < arrival_time[i])
        {
            current_time = arrival_time[i];
        }
        current_time += burst_time[i];
        completion_time[i] = current_time;
        turnaround_time[i] = completion_time[i] - arrival_time[i];
        waiting_time[i] = turnaround_time[i] - burst_time[i];
    }

    for (int i = 0; i < n; i++)
    {
        avg_wt += waiting_time[i];
        avg_tat += turnaround_time[i];
    }

    printf("\n");
    printf("Process No.\t   Arrival Time\t   Burst Time \t   Completion
Time\t   TurnAround Time\t Waiting Time\t");
    for (int i = 0; i < n; i++)
    {
        printf("\nP%d\t\t    %d\t\t    %d\t\t          %d\t\t
%d\t\t        %d\t", i + 1, arrival_time[i], burst_time[i],
completion_time[i], turnaround_time[i], waiting_time[i]);
    }

    printf("\n");
    printf("\nAverage Turn-Around Time = %f units ", avg_tat / n);
    printf("\nAverage Waiting Time = %f units ", avg_wt / n);

    return 0;
}
```

OUTPUT:-



LEARNING OUTCOME:-

PROGRAM - 2

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

THEORY:-

Shortest Job First (SJF) is a **non-preemptive** or **preemptive** CPU scheduling algorithm that selects the process with the smallest burst time to execute next. It aims to minimise the average waiting time and is optimal in that regard.
If two processes have the same burst time, then FCFS is used to break the tie. It is easy to implement in Batch systems where required CPU time is known in advance.

Pre-emptive mode of Shortest Job First is called as Shortest Remaining Time First (SRTF).

PSEUDOCODE (HANDWRITTEN):-

1. Input the number of processes, arrival times, and burst times.

2. Initialize:
current_time = 0
completion = 0
Set remaining_time = burst_time for each process

3. While (completion < total number of processes):
a. Select the process with the shortest burst time among those that have arrived.
b. Execute the process completely.
c. Update its completion time and mark it as completed.
d. Increase current_time based on the burst time of the selected process.

4. Calculate Waiting Time and Turnaround Time:
Waiting Time = Turnaround Time - Burst Time
Turnaround Time = Completion Time - Arrival Time

5. Output the completion time, waiting time, and turnaround time for all processes.

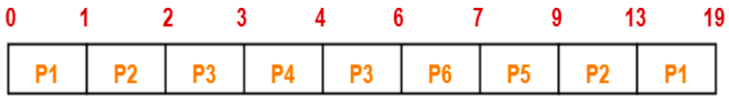
NUMERICAL:-

Consider the following processes with given Arrival Time and Burst Time. Calculate the average Waiting Time and Turn-Around Time using SJF.

PROCESS ID	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P1	0	7			
P2	1	5			
P3	2	3			
P4	3	1			
P5	4	2			
P6	5	1			

Solution-

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	19	19 – 0 = 19	19 – 7 = 12
P2	13	13 – 1 = 12	12 – 5 = 7
P3	6	6 – 2 = 4	4 – 3 = 1
P4	4	4 – 3 = 1	1 – 1 = 0
P5	9	9 – 4 = 5	5 – 2 = 3
P6	7	7 – 5 = 2	2 – 1 = 1

Now,

- Average Turn Around time = (19 + 12 + 4 + 1 + 5 + 2) / 6 = 43 / 6 = 7.17 unit
- Average waiting time = (12 + 7 + 1 + 0 + 3 + 1) / 6 = 24 / 6 = 4 unit

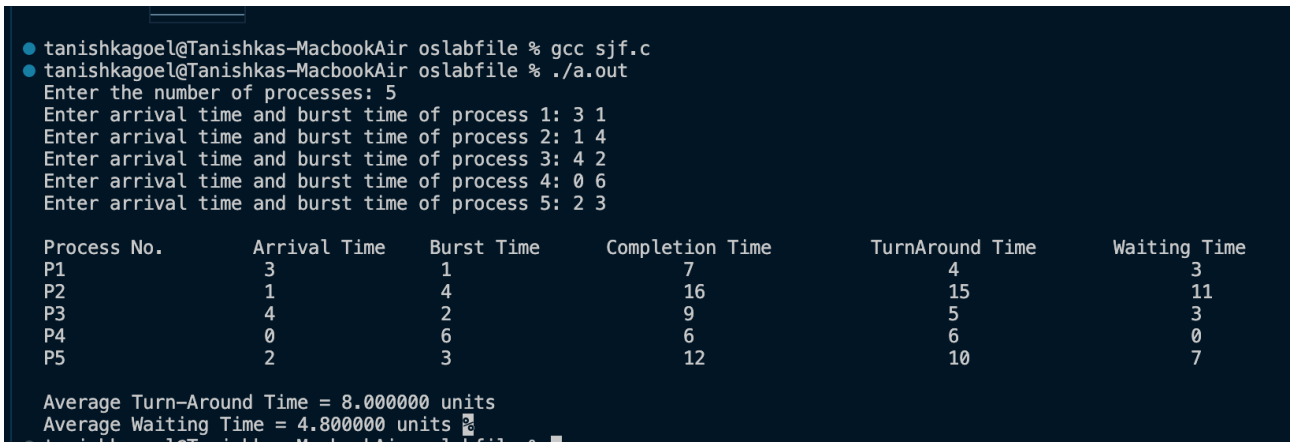
CODE:-

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

typedef struct process
{
    int pid;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turn_around_time;
    int waiting_time;
} process;

void getProcessInfo(process p[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("Enter arrival time and burst time of process %d: ", i +
1);
        scanf("%d %d", &p[i].arrival_time, &p[i].burst_time);
        p[i].pid = i + 1;
    }
}

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

void calculateCompletionTime(process p[], int n)
{
    process q[n];
    int front = 0, rear = 1;
    int time = p[0].arrival_time;
    q[front] = p[0];
    int i = 1;

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

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

        front++;

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

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

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

void printProcessInfo(process p[], int n)
{   float avg_wt = 0.0;
    float avg_tat = 0.0;

    for (int i = 0; i < n; i++)
    {
        avg_wt += p[i].waiting_time;
        avg_tat += p[i].turn_around_time;
    }
    printf("\n");
    printf("Process No.\t   Arrival Time\t   Burst Time \t   Completion
Time\t   TurnAround Time\t Waiting Time\t");
    for (int i = 0; i < n; i++)
    {
```

```
        printf("\nP%d\t\t    %d\t\t    %d\t\t          %d\t\t
%d\t\t        %d\t", i + 1, p[i].arrival_time, p[i].burst_time,
p[i].completion_time, p[i].turn_around_time, p[i].waiting_time);
    }

    printf("\n");
    printf("\nAverage Turn-Around Time = %f units ", avg_tat / n);
    printf("\nAverage Waiting Time = %f units ", avg_wt / n);
}

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

    process p[n];
    getProcessInfo(p, n);
    sortByArrivalTime(p, n);
    calculateCompletionTime(p, n);
    calculateTurnAroundTimeAndWaitingTime(p, n);
    sortByProcessID(p, n);
    printProcessInfo(p, n);

    return 0;
}
```

OUTPUT:-



```
tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc sjf.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out
Enter the number of processes: 5
Enter arrival time and burst time of process 1: 3 1
Enter arrival time and burst time of process 2: 1 4
Enter arrival time and burst time of process 3: 4 2
Enter arrival time and burst time of process 4: 0 6
Enter arrival time and burst time of process 5: 2 3

Process No.        Arrival Time    Burst Time      Completion Time         TurnAround Time       Waiting Time
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

Average Turn-Around Time = 8.000000 units
Average Waiting Time = 4.800000 units %
```

LEARNING OUTCOME:-

PROGRAM - 3

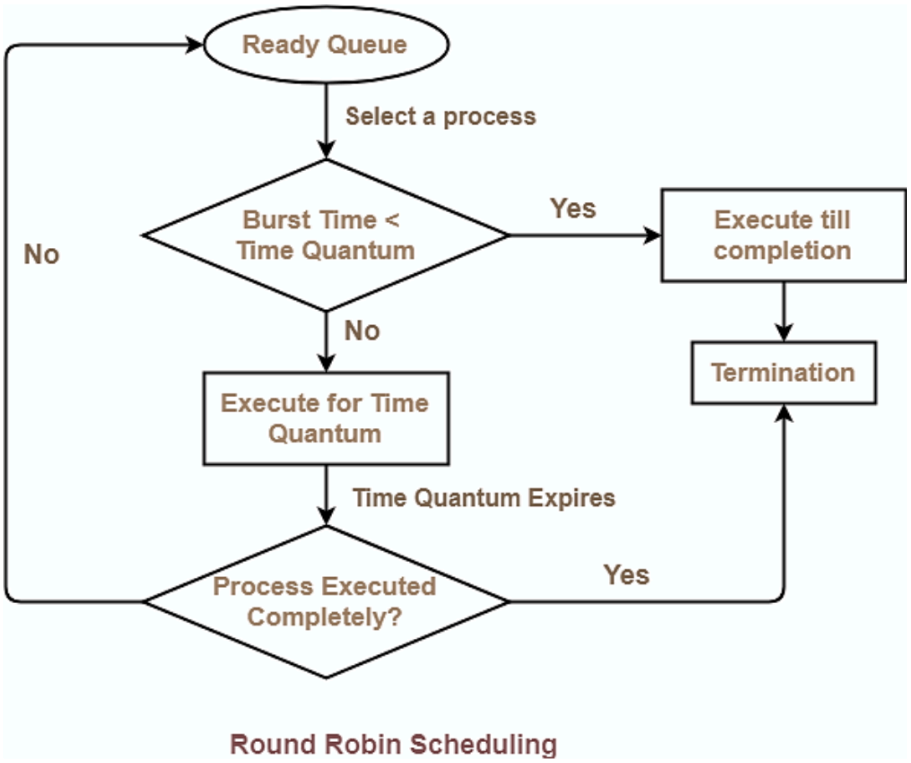
PROBLEM STATEMENT :- Write a program to implement CPU scheduling for Round Robin.

THEORY:-

Round Robin (RR) is a pre-emptive scheduling algorithm used in operating systems to allocate CPU time to processes in a time-sharing system. It is designed to share CPU time fairly among all processes, ensuring that no process monopolises the CPU, which makes it particularly well-suited for interactive systems such as time-sharing systems and real-time operating environments.

- CPU is assigned to the process on the basis of FCFS for a fixed amount of time.
- This fixed amount of time is called as time quantum or time slice.
- After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always preemptive in nature.

FLOWCHART (hand drawn):-

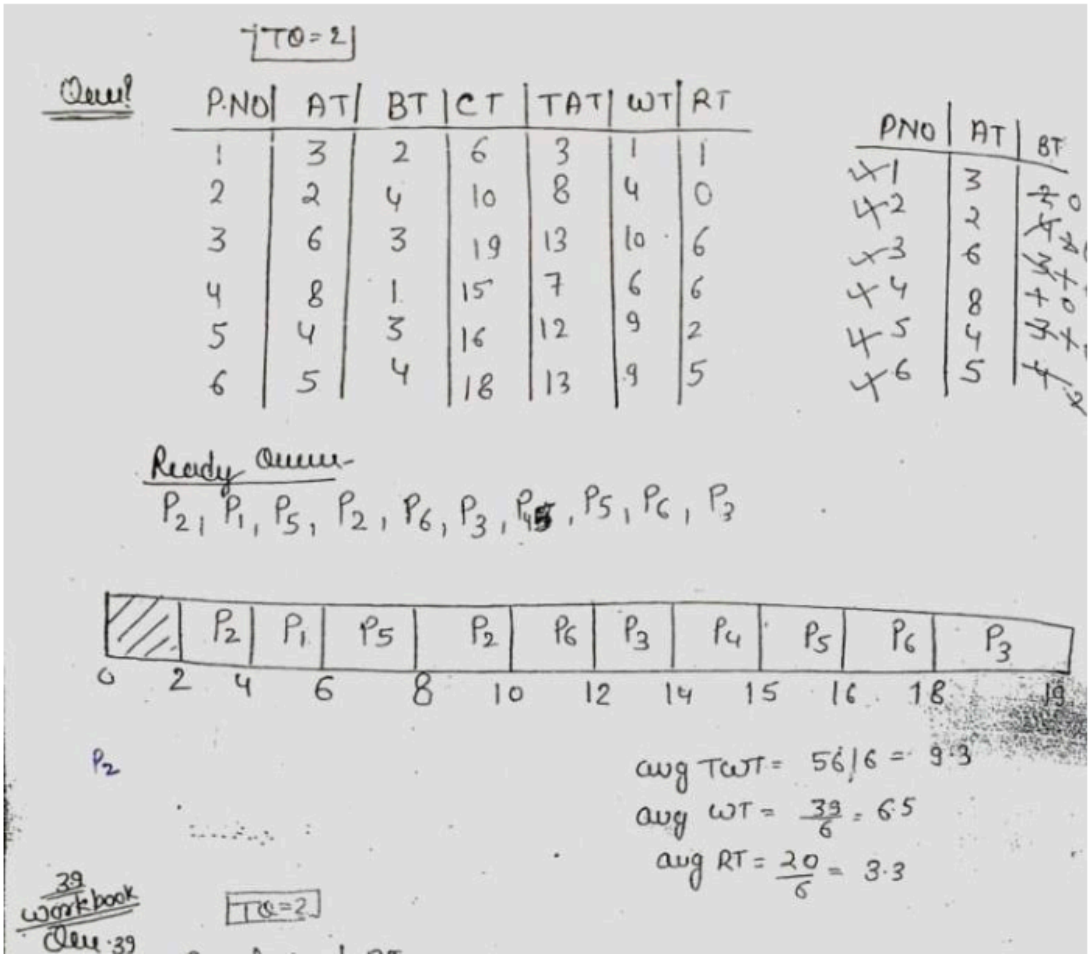


Round Robin Scheduling

NUMERICAL:-

Consider the following processes with given Arrival Time and Burst Time. Calculate the average Waiting Time and Turn-Around Time using Round Robin. Take Time Quantum = 2.

PROCESS ID	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P1	3	2			
P2	2	4			
P3	6	3			
P4	8	1			
P5	4	3			
P6	5	4			



CODE:-

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

typedef struct process
{
    int arrival_time;
    int burst_time;
    int completion_time;
    int remaining_time;
    int turn_around_time;
    int waiting_time;

} process;

void getProcessinfo(process p[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("Enter arrival time and burst time of process %d: ", i +
1);
        scanf("%d %d", &p[i].arrival_time, &p[i].burst_time);
        p[i].remaining_time = p[i].burst_time;
    }
}

void roundRobin(int n, process p[n], int time_quantum)
{
    int current_time = 0;
    int completion = 0;

    bool executed_in_this_cycle;

    while (completion < n)
    {
        executed_in_this_cycle = false;
        for (int i = 0; i < n; i++)
        {
            if (p[i].arrival_time <= current_time && p[i].remaining_time
> 0)
            {
                executed_in_this_cycle = true;
                if (p[i].remaining_time > time_quantum)
                {
                    p[i].remaining_time -= time_quantum;
                    current_time += time_quantum;
                }
                else
                {
                    current_time += p[i].remaining_time;
                    p[i].completion_time = current_time;
                    p[i].remaining_time = 0;
                    completion++;
                }
            }
        }

        if (!executed_in_this_cycle)
        {
```

```
            current_time++;
        }
    }
    for (int i = 0; i < n; i++)
    {
        p[i].turn_around_time = p[i].completion_time - p[i].arrival_time;
        p[i].waiting_time = p[i].turn_around_time - p[i].burst_time;
    }


}


int main()
{

    int n;
    float avg_tat, avg_wt;

    printf("Enter number of processes: ");
    scanf("%d", &n);
    process p[n];
    getProcessinfo(p, n);

    roundRobin(n, p, 2);


    for (int i = 0; i < n; i++)
    {
        avg_wt += p[i].waiting_time;
        avg_tat += p[i].turn_around_time;
    }
    printf("Process No.\t  Arrival Time\t    Burst Time\t    Completion
Time\t   Turn Around Time\t Waiting Time");
    for (int i = 0; i < n; i++)
    {
        printf("\n P%d\t\t      %d\t\t      %d\t\t\t     %d\t\t
%d\t\t\t        %d", i + 1, p[i].arrival_time, p[i].burst_time,
p[i].completion_time, p[i].turn_around_time, p[i].waiting_time);
    }

    printf("\n");
    printf("\nAverage Turn-Around Time = %f units ", avg_tat / n);
    printf("\nAverage Waiting Time = %f units ", avg_wt / n);


    return 0;
}
```

OUTPUT:-

LEARNING OUTCOME:-

```
tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc rr.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out
Enter number of processes: 5
Enter arrival time and burst time of process 1: 0 5
Enter arrival time and burst time of process 2: 1 3
Enter arrival time and burst time of process 3: 2 1
Enter arrival time and burst time of process 4: 3 2
Enter arrival time and burst time of process 5: 4 3
Process No.       Arrival Time      Burst Time      Completion Time        Turn Around Time      Waiting Time
 P1                    0                5                    14                14                    9
 P2                    1                3                    12                11                    8
 P3                    2                1                    5                 3                     2
 P4                    3                2                    7                 4                     2
 P5                    4                3                    13                9                     6

Average Turn-Around Time = 8.200000 units
Average Waiting Time = 5.400000 units %
```

PROGRAM - 4

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

THEORY:-

Page replacement algorithms are used in operating systems to manage how **pages** are swapped in and out of memory when a process requires more memory than is available. When a page that is not currently in memory is needed, the operating system selects a page to replace, i.e., remove from memory, to make room for the new one. The goal of these algorithms is to minimise page faults, which occur when the required page is not in memory.

Common Page Replacement Algorithms:

1. **First-In, First-Out (FIFO):**
    - The oldest loaded page (the first one loaded into memory) is replaced.
    - Pages are loaded into memory in a queue. When a page needs to be replaced, the page at the front of the queue (the one that has been in memory the longest) is removed.
    - It is simple to implement. However, it does not consider how often or how recently a page was used, leading to possible poor performance (e.g., **Belady's Anomaly**).

2. **Optimal Page Replacement (OPT):**
    - It replaces the page that will not be needed for the longest period in the future.
    - The algorithm looks ahead to see which pages will be used and replaces the page that will not be needed for the longest time.
    - It guarantees the lowest number of page faults. However, it is not practical for real systems since it requires future knowledge of memory accesses.

3. **Least Recently Used (LRU):**

- It replaces the page that has not been used for the longest time.
- The algorithm keeps track of the order in which pages are used. When a replacement is needed, it chooses the page that was least recently used.
- It approximates the performance of the optimal algorithm and does not suffer from Belady's Anomaly. However, it requires extra overhead to track the order of usage.

ALGORITHM(HANDWRITTEN):-

EITHER WRITE ALGORITHMS OR PSEUDOCODE

ALGORITHMS

FIFO

Initialize a queue to store the pages currently in memory.
For each page request:

- If the page is **already in memory**, do nothing.
- If the page is **not in memory**:
  - If there is space in memory, load the page.
  - If memory is full, **replace the page at the front** of the queue (oldest page) with the new page.
  - Add the new page to the end of the queue.

Repeat until all page requests are processed.

LRU

Track usage of pages by keeping a record of the time when each page was last used.
For each page request:

- If the page is **already in memory**, update its usage time.
- If the page is **not in memory**:
  - If there is space, load the page and record the time it was used.
  - If memory is full, **find the least recently used page** and replace it with the new page.
  - Record the usage time for the new page.

Repeat until all page requests are processed.

OPTIMAL

1. For each page request:
   - If the page is **already in memory**, do nothing.
   - If the page is **not in memory**:
     - If there is space in memory, load the page.
     - If memory is full, look ahead and **find the page that will not be used for the longest time in the future**.
     - Replace that page with the new page.
2. **Repeat** until all page requests are processed.

PSEUDOCODE————————————————————————————

FIFO

```
Initialize an empty queue
For each page in the sequence:
  If the page is already in memory:
    Do nothing
  Else:
    If there is space in memory:
      Add the page to memory and enqueue it
    Else:
      Dequeue the oldest page and replace it with the new page
      Enqueue the new page
```

LRU

```
For each page in the sequence:
    If the page is already in memory:
        Update its usage time to current time
    Else:
        If there is space in memory:
            Load the page and update its usage time
        Else:
            Find the page with the oldest usage time (LRU)
            Replace it with the new page
            Update the new page's usage time to current time
```

OPTIMAL

```
For each page in the sequence:
    If the page is already in memory:
        Do nothing
    Else:
        If there is space in memory:
            Load the page into memory
        Else:
            Look ahead and find the page that will not be used for the longest time
            Replace that page with the new page
```

NUMERICAL:-

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:

1. Optimal Page Replacement Algorithm

2. FIFO Page Replacement Algorithm

3. LRU Page Replacement Algorithm Optimal Page Replacement Algorithm

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 -

LRU

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

Page Faults in LRU -

2. FIFO Page Replacement Algorithm
3. LRU Page Replacement Algorithm Optimal Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	2	2	2
Frame 2		7	7	7	7	7	7	7	7	7
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Hit	Hit

Number of Page Faults in Optimal Page Replacement Algorithm = 5

LRU Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in LRU = 6

FIFO Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in FIFO = 6

FIRST IN FIRST OUT : -

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;

    // initialize frame with -1
    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];

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

    fifoPageReplacement(ref_string, n, frame_size);

    return 0;
}
```

OUTPUT:-

```
tanishkagoel@Tanishkas-MacbookAir page replacement % gcc fifo.c
tanishkagoel@Tanishkas-MacbookAir page replacement % ./a.out
Enter size of reference String: 9
Enter reference string: 6 5 3 5 6 2 3 7 5
Enter frame size: 3
Hits: 3
Page Faults: 6
Hit Ratio: 0.333333
Page Fault Ratio: 0.666667
```

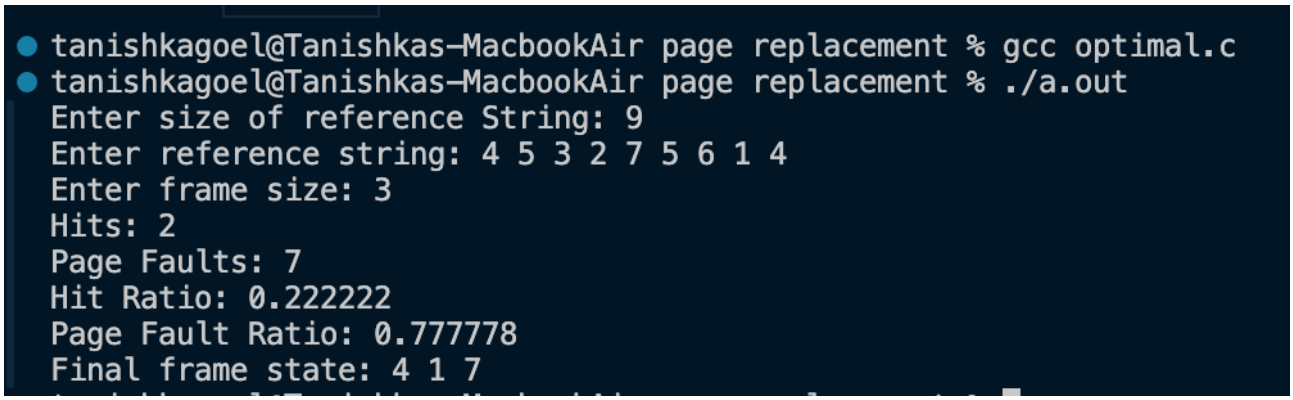
OPTIMAL PAGE REPLACEMENT :-

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:-

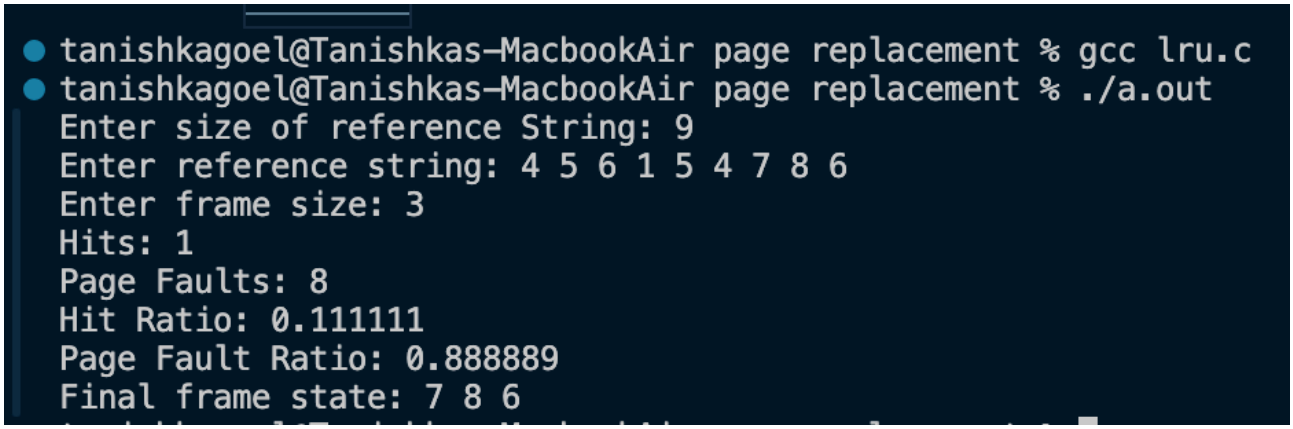

tanishkagoel@Tanishkas-MacbookAir page replacement % gcc optimal.c
tanishkagoel@Tanishkas-MacbookAir page replacement % ./a.out
Enter size of reference String: 9
Enter reference string: 4 5 3 2 7 5 6 1 4
Enter frame size: 3
Hits: 2
Page Faults: 7
Hit Ratio: 0.222222
Page Fault Ratio: 0.777778
Final frame state: 4 1 7

LRU PAGE REPLACEMENT :-
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:-



LEARNING OUTCOME:-

PROGRAM - 5

PROBLEM STATEMENT :- Write a program to perform priority scheduling.

THEORY:-

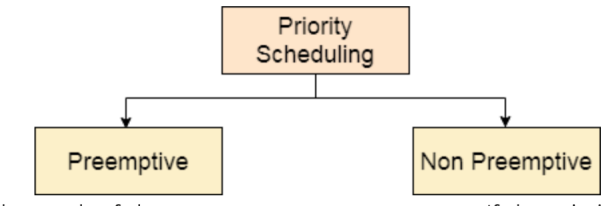
Priority scheduling is a CPU scheduling algorithm used in operating systems where each process is assigned a priority. In this algorithm, the CPU is allocated to the process with the highest priority. If two processes have the same priority, they are scheduled based on other criteria, such as first-come, first-served (FCFS).

Priority of processes depends on some factors such as:

- Time limit
- Memory requirements of the process
- Ratio of average I/O to average CPU burst time

Types of Priority Scheduling Algorithms

There are two types of priority scheduling algorithms in OS:



Non-Preemptive Scheduling

In this type of scheduling:

- If during the execution of a process, another process with a higher priority arrives for execution, even then the currently executing process will not be disturbed.
- The newly arrived high priority process will be put in next for execution since it has higher priority than the processes that are in a waiting state for execution.
- All the other processes will remain in the waiting queue to be processed. Once the execution of the current process is done, the high-priority process will be given the CPU for execution.

Preemptive Scheduling

Preemptive Scheduling as opposed to non-preemptive scheduling will preempt (stop and store the currently executing process) the currently running process if a higher priority process enters the waiting state for execution and will execute the higher priority process first and then resume executing the previous process.

ALGORITHM:-

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	2	0	4
P2	3	1	3
P3	4	2	1
P4	5	3	5
P5	5	4	2

Preemptive mode

Preemptive mode

PROCESS ID	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P1	2	0	4			
P2	3	1	3			
P3	4	2	1			
P4	5	3	5			
P5	5	4	2			

Non-Preemptive mode

Non-Preemptive mode

PROCESS ID	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P1	2	0	4			
P2	3	1	3			
P3	4	2	1			
P4	5	3	5			
P5	5	4	2			

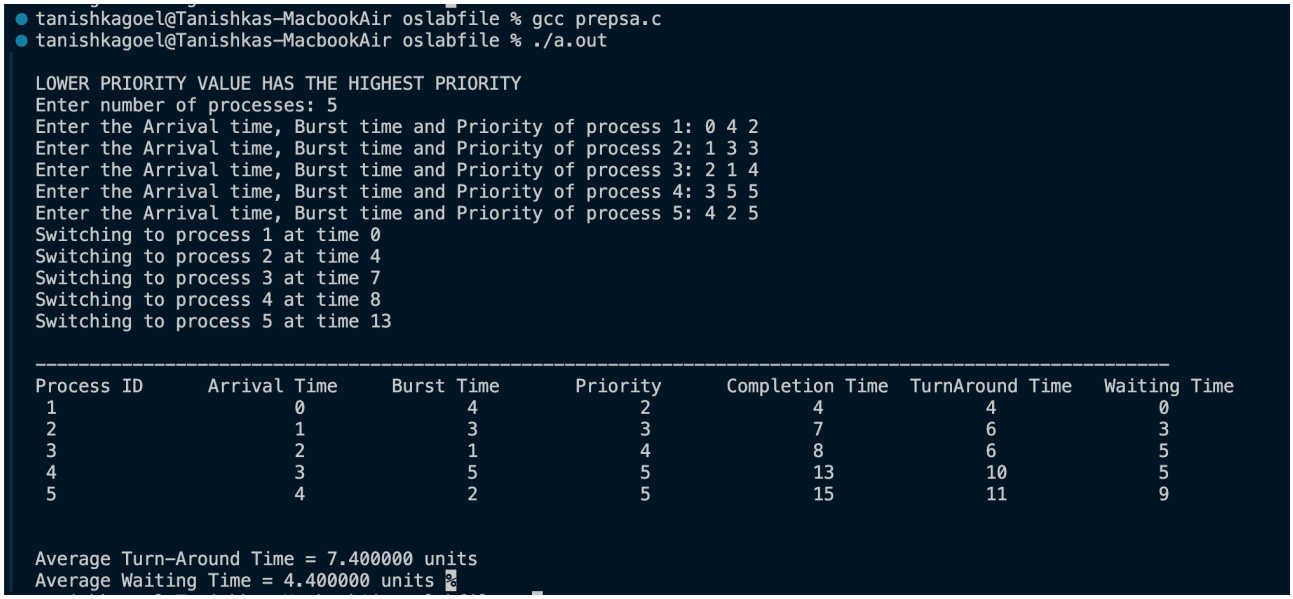
PREEMPTIVE PRIORITY SCHEDULING

CODE:-

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

typedef struct process
{
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int completion_time;
    int turn_around_time;
    int waiting_time;
    int priority;
    bool completed;
} process;

void getProcessInfo(process p[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("Enter the Arrival time, Burst time and Priority of
process %d: ", i + 1);
        scanf("%d %d %d",
&p[i].arrival_time,&p[i].burst_time,&p[i].priority);
        p[i].pid = i + 1;
        p[i].remaining_time = p[i].burst_time;
        p[i].completed = false;
    }
}

int main()
{
    printf("\nLOWER PRIORITY VALUE THE HIGHER PRIORITY\n");
    int n;
    float avg_tat=0.0, avg_wt=0.0;

    printf("Enter number of processes: ");
    scanf("%d", &n);
    process p[n];
    getProcessInfo(p, n);

    int current_time = 0;
    int completed = 0;
    int last_process = -1;

    while (completed < n)
    {
        int highest_priority = INT_MAX;
        int next_process = -1;


        for (int i = 0; i < n; i++)
        {
            if (p[i].arrival_time <= current_time && !p[i].completed &&
p[i].priority < highest_priority && p[i].remaining_time > 0)
            {
```

```
                highest_priority = p[i].priority;
                next_process = i;
            }
        }

        if (next_process != -1)
        {
            if (last_process != next_process) {
                printf("Switching to process %d at time %d\n",
p[next_process].pid, current_time);
                last_process = next_process;
            }

            p[next_process].remaining_time--;
            current_time++;


            if (p[next_process].remaining_time == 0)
            {
                p[next_process].completed = true;
                p[next_process].completion_time = current_time;
                p[next_process].turn_around_time =
p[next_process].completion_time - p[next_process].arrival_time;
                p[next_process].waiting_time =
p[next_process].turn_around_time - p[next_process].burst_time;
                completed++;
            }
        }
        else
        {
            current_time++;
        }
    }
    for (int i = 0; i < n; i++)
    {
        avg_wt += p[i].waiting_time;
        avg_tat += p[i].turn_around_time;
    }


printf("\n----------------------------------------------------------------
-----------------------------------------\n");
    printf("Process ID\tArrival Time\t Burst Time\t  Priority\tCompletion
Time\t TurnAround Time   Waiting Time\n");
    for (int i = 0; i < n; i++)
    {
        printf(" %d\t\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid,
p[i].arrival_time, p[i].burst_time, p[i].priority, p[i].completion_time,
p[i].turn_around_time, p[i].waiting_time);
    }

    printf("\n");
    printf("\nAverage Turn-Around Time = %f units ", avg_tat / n);
    printf("\nAverage Waiting Time = %f units ", avg_wt / n);
    return 0;
}
```

OUTPUT:-


tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc prepsa.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out
LOWER PRIORITY VALUE HAS THE HIGHEST PRIORITY
Enter number of processes: 5
Enter the Arrival time, Burst time and Priority of process 1: 0 4 2
Enter the Arrival time, Burst time and Priority of process 2: 1 3 3
Enter the Arrival time, Burst time and Priority of process 3: 2 1 4
Enter the Arrival time, Burst time and Priority of process 4: 3 5 5
Enter the Arrival time, Burst time and Priority of process 5: 4 2 5
Switching to process 1 at time 0
Switching to process 2 at time 4
Switching to process 3 at time 7
Switching to process 4 at time 8
Switching to process 5 at time 13
Process ID Arrival Time Burst Time Priority Completion Time TurnAround Time Waiting Time
1 0 4 2 4 4 0
2 1 3 3 7 6 3
3 2 1 4 8 6 5
4 3 5 5 13 10 5
5 4 2 5 15 11 9
Average Turn-Around Time = 7.400000 units
Average Waiting Time = 4.400000 units

NON-PREEMPTIVE PRIORITY SCHEDULING

CODE:-

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

typedef struct process
{
    int pid;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turn_around_time;
    int waiting_time;
    int priority;
    bool completed;
} process;

void getProcessInfo(process p[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("Enter the Arrival time, Burst time and Priority of
process %d: ", i + 1);
        scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time,
&p[i].priority);
        p[i].pid = i + 1;
        p[i].completed = false;
    }
}

int main()
{
    printf("\nLOWER PRIORITY VALUE HAS HIGHER PRIORITY\n");
    int n;
    float avg_tat = 0.0, avg_wt = 0.0;

    printf("Enter number of processes: ");
    scanf("%d", &n);
    process p[n];
    getProcessInfo(p, n);

    int current_time = 0, completed = 0;

    while (completed < n)
    {
        int highest_priority_index = -1;
        int highest_priority = INT_MAX;

        for (int i = 0; i < n; i++)
        {
            if (p[i].arrival_time <= current_time && !p[i].completed)
            {
                if (p[i].priority < highest_priority)
                {
                    highest_priority = p[i].priority;
                    highest_priority_index = i;
                }
                else if (p[i].priority == highest_priority &&
p[i].arrival_time < p[highest_priority_index].arrival_time)
                {
```

```
                    highest_priority_index = i;
                }
            }
        }

        if (highest_priority_index != -1)
        {
            int idx = highest_priority_index;
            current_time += p[idx].burst_time;
            p[idx].completion_time = current_time;
            p[idx].turn_around_time = p[idx].completion_time -
p[idx].arrival_time;
            p[idx].waiting_time = p[idx].turn_around_time -
p[idx].burst_time;
            p[idx].completed = true;
            completed++;
        }
        else
        {
            current_time++;
        }
    }

    for (int i = 0; i < n; i++)
    {
        avg_wt += p[i].waiting_time;
        avg_tat += p[i].turn_around_time;
    }


printf("\n----------------------------------------------------------------
----------------------------------------\n");
    printf("Process ID\tArrival Time\tBurst Time\tPriority\tCompletion
Time\tTurnAround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++)
    {
        printf(" %d\t\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid,
p[i].arrival_time, p[i].burst_time, p[i].priority, p[i].completion_time,
p[i].turn_around_time, p[i].waiting_time);
    }

    printf("\n");
    printf("\nAverage Turn-Around Time = %.2f units ", avg_tat / n);
    printf("\nAverage Waiting Time = %.2f units ", avg_wt / n);
    return 0;
}
```

OUTPUT:-

```
tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc nonpsa.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out

LOWER PRIORITY VALUE HAS HIGHER PRIORITY
Enter number of processes: 7
Enter the Arrival time, Burst time and Priority of process 1: 0 3 2
Enter the Arrival time, Burst time and Priority of process 2: 2 6 5
Enter the Arrival time, Burst time and Priority of process 3: 1 4 3
Enter the Arrival time, Burst time and Priority of process 4: 4 2 5
Enter the Arrival time, Burst time and Priority of process 5: 6 9 7
Enter the Arrival time, Burst time and Priority of process 6: 5 4 4
Enter the Arrival time, Burst time and Priority of process 7: 7 10 10

_____________________________________________________________________________________________
Process ID      Arrival Time    Burst Time      Priority        Completion Time TurnAround Time Waiting Time
 1                       0               3               2               3               3               0
 2                       2               6               5               17              15              9
 3                       1               4               3               7               6               2
 4                       4               2               5               19              15              13
 5                       6               9               7               28              22              13
 6                       5               4               4               11              6               2
 7                       7               10              10              38              31              21


Average Turn-Around Time = 14.00 units
Average Waiting Time = 8.57 units %
```

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:-

First Fit Algorithm

1. Traverse the list of memory blocks.
2. For each block, check if it has enough space to accommodate the process.
3. If a block is found that fits, allocate it to the process.
4. If no suitable block is found, the process must wait.

Best Fit Algorithm

1. Traverse the list of memory blocks.
2. Find the smallest block that is large enough to accommodate the process.
3. Allocate the process to this block.
4. If no suitable block is found, the process must wait.

Worst Fit Algorithm

1. Traverse the list of memory blocks.
2. Find the largest block that is large enough to accommodate the process.
3. Allocate the process to this block.
4. If no suitable block is found, the process must wait.

NUMERICAL:-

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)**	

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:-

```
tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc best.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out
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
```

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;
}

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:-

```
tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc reader_writer.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out
Enter the number of readers and writers: 3

Reader 1 is reading data
Reader 3 is reading data
Writer 3 is trying to enter
Reader 2 is reading data
Writer 1 is trying to enter
Writer 2 is trying to enter
Reader 3 is done reading
Reader 3 is leaving. Remaining readers: 2
Reader 2 is done reading
Reader 2 is leaving. Remaining readers: 1
Reader 1 is done reading
Reader 1 is leaving. Remaining readers: 0
Writer 3 is writing data
Writer 3 is done writing
Writer 1 is writing data
Writer 1 is done writing
Writer 2 is writing data
Writer 2 is done writing
```

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.

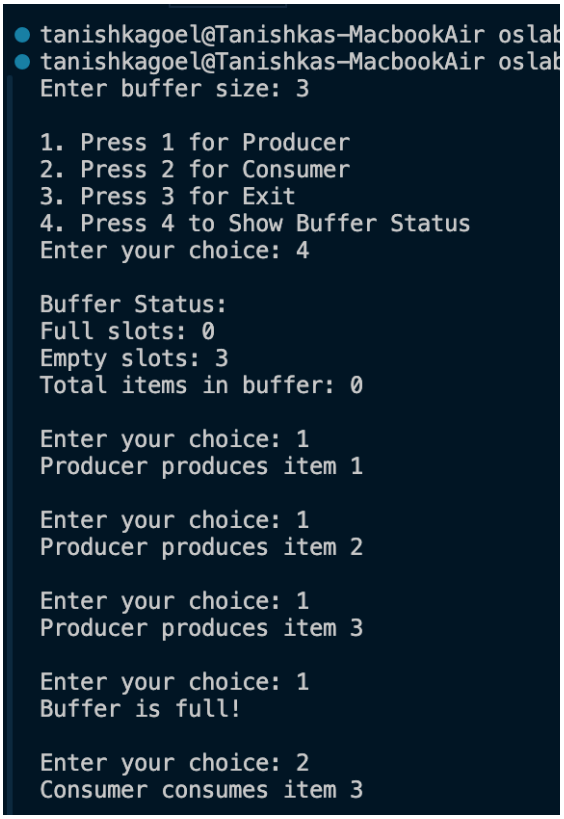
PSEUDOCODE:-

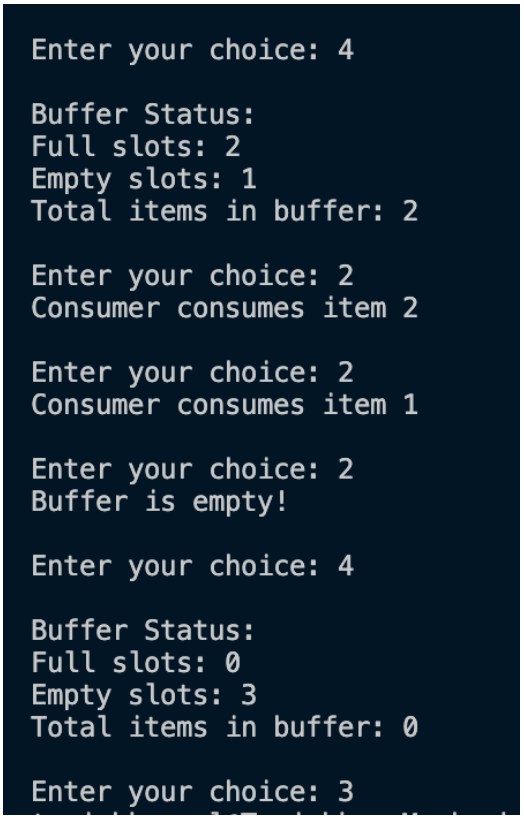
CODE:-

```
#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:-





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:-

NUMERICAL:-

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			

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:-

```
tanishkagoel@Tanishkas-MacbookAir oslabfile % gcc bankers.c
tanishkagoel@Tanishkas-MacbookAir oslabfile % ./a.out
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
```

LEARNING OUTCOME:-

PROGRAM - 10

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

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

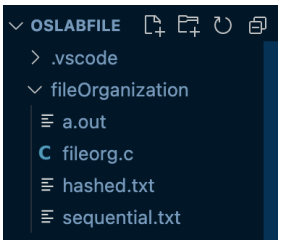
Direct, or hashed, access organizes records by computing their storage location using a hashing function, which maps a record's key directly to a specific address in memory or on disk. This method allows for rapid retrieval, as it enables direct access to a record without reading intervening records, 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.

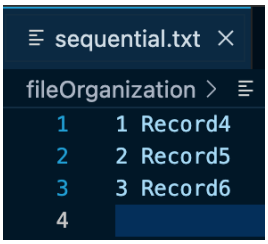
Indexed File Organization

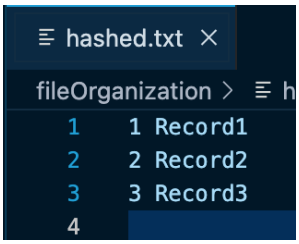
Indexed file organization improves random access by using an index, similar to a book's table of contents, which maintains pointers to the locations of records within the file. The index is typically built based on one or more key fields, and each entry in the index corresponds to a unique record in 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.

PSEUDOCODE:-

INITIAL SET-UP:-







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);
```

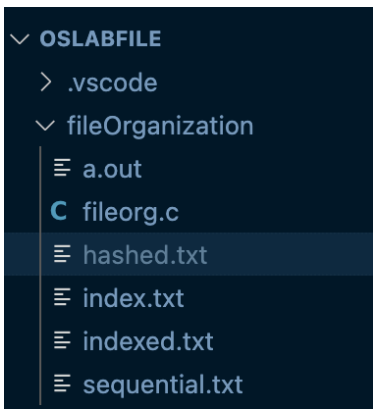
```
    for (int i = 0; i < recordCount; i++) {
        fprintf(file, "%d %s\n", records[i].id, records[i].name); //
Write the record
    }

    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);
}
```

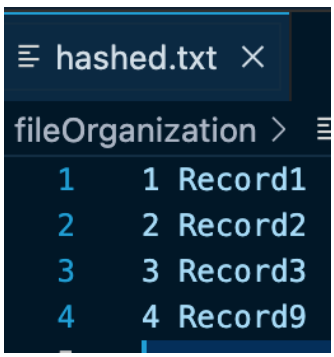
OUTPUT:-

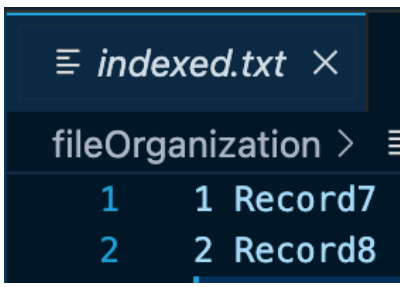
```
tanishkagoel@Tanishkas-MacbookAir fileOrganizat

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:-