VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Operating Systems

(23CS4PCOPS)

Submitted by:

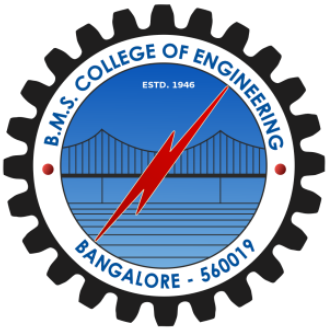
ARUGUNTA HAMSIKA (1BM22CS054)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

June 2024 - August 2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Operating Systems**" carried out by **Arugunta Hamsika (1BM22CS054),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (23CS4PCOPS)** work prescribed for the said degree.

Sowmya T
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table Of Contents

Course Outcomes

CO1: Apply the different concepts and functionalities of Operating System.

CO2: Analyse various Operating system strategies and techniques.

CO3: Demonstrate the different functionalities of Operating System.

CO4: Conduct practical experiments to implement the functionalities of Operating system.

LAB-1

1.Question:

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

(a) FCFS

(b) SJF

Code:

```
#include <stdio.h>
int n, i, j, pos, temp, choice, Burst_time[20], Waiting_time[20], Turn_around_time[20], process[20], total=0;
float avg_Turn_around_time=0, avg_Waiting_time=0;
int FCFS() {
   Waiting_time[0] = 0;
   for(i = 1; i < n; i++) {
      Waiting_time[i] = 0;
      for(j = 0; j < i; j++)
         Waiting_time[i] += Burst_time[j];
   }
   printf("\nProcess\t\tBurst Time\t\tWaiting Time\t\tTurnaround Time");
   for(i = 0; i < n; i++) {
      Turn_around_time[i] = Burst_time[i] + Waiting_time[i];
      avg_Waiting_time += Waiting_time[i];
      avg_Turn_around_time += Turn_around_time[i];
      printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t%d", i+1, Burst_time[i], Waiting_time[i], Turn_around_time[i]);
   }
   avg_Waiting_time = (float)avg_Waiting_time / n;
   avg_Turn_around_time = (float)avg_Turn_around_time / n;
   printf("\nAverage Waiting Time: %.2f", avg_Waiting_time);
   printf("\nAverage Turnaround Time: %.2f\n", avg_Turn_around_time);
   return 0;
}

int SJF() {
   for(i = 0; i < n; i++) {
      pos = i;
      for(j = i + 1; j < n; j++) {
         if(Burst_time[j] < Burst_time[pos])
```

```
            pos = j;
        }
        temp = Burst_time[i];
        Burst_time[i] = Burst_time[pos];
        Burst_time[pos] = temp;

        temp = process[i];
        process[i] = process[pos];
         process[pos] = temp;
    }
    Waiting_time[0] = 0;
    for(i = 1; i < n; i++) {
        Waiting_time[i] = 0;
        for(j = 0; j < i; j++)
            Waiting_time[i] += Burst_time[j];
        total += Waiting_time[i];
    }
    avg_Waiting_time = (float)total / n;
    total = 0;
    printf("\nProcess\t\tBurst Time\t\tWaiting Time\t\tTurnaround Time");
    for(i = 0; i < n; i++) {
        Turn_around_time[i] = Burst_time[i] + Waiting_time[i];
        total += Turn_around_time[i];
        printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d", process[i], Burst_time[i], Waiting_time[i],
Turn_around_time[i]);
    }
    avg_Turn_around_time = (float)total / n;
    printf("\n\nAverage Waiting Time=%.2f", avg_Waiting_time);
    printf("\nAverage Turnaround Time=%.2f\n", avg_Turn_around_time);
    return 0;
}
int main() {
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    printf("\nEnter Burst Time:\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]: ", i+1);
         scanf("%d", &Burst_time[i]);
        process[i] = i+1;
    }
    while(1) {
```

```
        printf("\n-----MAIN MENU-----\n");
        printf("1. FCFS Scheduling\n2. SJF Scheduling\n");
         printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1: FCFS();
                    break;
            case 2: SJF();
                    break;
            default: printf("Invalid Input!!!\n");
        }
    }
    return 0;
}
```

Output:

```
Arugunta Hamsika-1BM22CS054
Enter the total number of processes: 4
Enter Burst Time:
P[1]: 8
P[2]: 4
P[3]: 9
P[4]: 5
-----MAIN MENU-----
1. FCFS Scheduling
2. SJF Scheduling
Enter your choice: 1
Process     Burst Time      Waiting Time        Turnaround Time
P[1]        8           0               8
P[2]        4           8               12
P[3]        9           12              21
P[4]        5           21              26
Average Waiting Time: 10.25
Average Turnaround Time: 16.75

-----MAIN MENU-----
1. FCFS Scheduling
2. SJF Scheduling
Enter your choice: 2
Process     Burst Time      Waiting Time        Turnaround Time
P[2]        4           0               4
P[4]        5           4               9
P[1]        8           9               17
P[3]        9           17              26

Average Waiting Time=7.50
Average Turnaround Time=14.00
```

LAB - 2

2-Question:
Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

(a) Priority (pre-emptive & Non-pre-emptive)

(b) Round Robin (Experiment with different quantum sizes for RR algorithm)

Code:

(a) Priority (Non-pre-emptive)

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

struct process {
    int process_id;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
};

void find_average_time(struct process[], int);
void priority_scheduling(struct process[], int);

int main() {
    int n, i;
    struct process proc[10];
    printf("Arugunta Hamsika-1BM22CS054\n");
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("\nEnter the process ID: ");
        scanf("%d", &proc[i].process_id);
        printf("Enter the burst time: ");
        scanf("%d", &proc[i].burst_time);
        printf("Enter the priority: ");
        scanf("%d", &proc[i].priority);
    }

    priority_scheduling(proc, n);
    return 0;
}

void find_waiting_time(struct process proc[], int n, int wt[]) {
    int i;
    wt[0] = 0;
    for (i = 1; i < n; i++) {
        wt[i] = proc[i - 1].burst_time + wt[i - 1];
```

```
    }
}

void find_turnaround_time(struct process proc[], int n, int wt[], int tat[]) {
    int i;
    for (i = 0; i < n; i++) {
        tat[i] = proc[i].burst_time + wt[i];
    }
}

void find_average_time(struct process proc[], int n) {
    int wt[10], tat[10], total_wt = 0, total_tat = 0, i;
    find_waiting_time(proc, n, wt);
    find_turnaround_time(proc, n, wt, tat);

    printf("\nProcess ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time");
    for (i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].process_id, proc[i].burst_time, proc[i].priority, wt[i],
tat[i]);
    }

    printf("\n\nAverage Waiting Time = %f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %f\n", (float)total_tat / n);
}

void priority_scheduling(struct process proc[], int n) {
    int i, j, pos;
    struct process temp;

    for (i = 0; i < n; i++) {
        pos = i;
        for (j = i + 1; j < n; j++) {
            if (proc[j].priority < proc[pos].priority) {
                pos = j;
            }
        }
        temp = proc[i];
        proc[i] = proc[pos];
        proc[pos] = temp;
    }

    find_average_time(proc, n);
}
```

Priority (Pre-emptive):

CODE:

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

struct process {
```

```
    int process_id;
    int burst_time;
    int priority;
    int arrival_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int is_completed;
};
void find_average_time(struct process[], int);
void priority_scheduling(struct process[], int);

int main() {
    int n, i;
    struct process proc[10];

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

    for (i = 0; i < n; i++) {
        printf("\nEnter the process ID: ");
        scanf("%d", &proc[i].process_id);

        printf("Enter the burst time: ");
        scanf("%d", &proc[i].burst_time);

        printf("Enter the arrival time: ");
        scanf("%d", &proc[i].arrival_time);

        printf("Enter the priority: ");
        scanf("%d", &proc[i].priority);

        proc[i].remaining_time = proc[i].burst_time;
        proc[i].is_completed = 0;
    }

    priority_scheduling(proc, n);
    return 0;
}

void find_waiting_time(struct process proc[], int n) {
```

```
    int time = 0, completed = 0, min_priority, shortest = 0;
    while (completed != n) {
        min_priority = 10000;
 for (int i = 0; i < n; i++) {
            if ((proc[i].arrival_time <= time) && (!proc[i].is_completed) && (proc[i].priority < min_priority)) {
                min_priority = proc[i].priority;
                shortest = i;
            }
        }
        proc[shortest].remaining_time--;
        time++;
        if (proc[shortest].remaining_time == 0) {
            proc[shortest].waiting_time = time - proc[shortest].arrival_time - proc[shortest].burst_time;
            proc[shortest].turnaround_time = time - proc[shortest].arrival_time;
            proc[shortest].is_completed = 1;
            completed++;
        }
    }
}

void find_turnaround_time(struct process proc[], int n) {
}
void find_average_time(struct process proc[], int n) {
    int total_wt = 0, total_tat = 0;
    find_waiting_time(proc, n);
    find_turnaround_time(proc, n);
    printf("\nProcess ID\tBurst Time\tArrival Time\tPriority\tWaiting Time\tTurnaround Time");
for (int i = 0; i < n; i++) {
        total_wt += proc[i].waiting_time;
        total_tat += proc[i].turnaround_time;
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].process_id, proc[i].burst_time,
proc[i].arrival_time, proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("\n\nAverage Waiting Time = %f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %f\n", (float)total_tat / n);
}

void priority_scheduling(struct process proc[], int n) {
    find_average_time(proc, n);
}
```

(b) Round Robin (Non-pre-emptive)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_PROCESSES 10
struct Process {
    int pid;
    int burst_time;
    int arrival_time;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
    int completion_time;
};
void round_robin(struct Process proc[], int n, int quantum) {
    int current_time = 0;
    int completed_processes = 0;
    while (completed_processes < n) {
        bool process_found = false;
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0 && proc[i].arrival_time <= current_time) {
                process_found = true;
                if (proc[i].remaining_time > quantum) {
                    current_time += quantum;
                        proc[i].remaining_time -= quantum;
                } else {
                    current_time += proc[i].remaining_time;
                    proc[i].completion_time = current_time;
                    proc[i].turnaround_time = proc[i].completion_time - proc[i].arrival_time;
                    proc[i].waiting_time = proc[i].turnaround_time - proc[i].burst_time;
                    proc[i].remaining_time = 0;
                        completed_processes++;
```

```
            }
        }
    }
    if (!process_found) {
        current_time++;
    }
}
// Print the results
printf("\nPID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].arrival_time, proc[i].burst_time,
proc[i].completion_time, proc[i].turnaround_time, proc[i].waiting_time);
    total_completion_time += proc[i].completion_time;
    total_turnaround_time += proc[i].turnaround_time;
    total_waiting_time += proc[i].waiting_time;
}
// Calculate and display averages
printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}
int main() {
    int n, quantum;
    printf("Arugunta hamsika-1BM22CS054\n");
    printf("Enter the total number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &n);
    if (n > MAX_PROCESSES) {
        printf("Number of processes exceeds maximum limit.\n");
        return 1;
    }
    struct Process proc[MAX_PROCESSES];
```

```
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &proc[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &proc[i].burst_time);
        proc[i].pid = i + 1;
        proc[i].remaining_time = proc[i].burst_time; // Initialize remaining time
        proc[i].turnaround_time = 0; // Initialize turnaround time
        proc[i].waiting_time = 0; // Initialize waiting time
        proc[i].completion_time = 0; // Initialize completion time
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &quantum);

    round_robin(proc, n, quantum);

    return 0;
}
```

Output:

(a) Priority (Non-pre-emptive)

```
Arugunta Hamsika-1BM22CS054
Enter the number of processes: 3
Enter the process ID: 1
Enter the burst time: 2
Enter the priority: 3
Enter the process ID: 2
Enter the burst time: 5
Enter the priority: 2
Enter the process ID: 3
Enter the burst time: 6
Enter the priority: 1
Process ID  Burst Time  Priority    Waiting Time    Turnaround Time
3       6       1       0       6
2       5       2       6       11
1       2       3       11      13

Average Waiting Time = 5.666667
Average Turnaround Time = 10.000000
```

(b) Round Robin (Non-pre-emptive)

```
Arugunta hamsika-1BM22CS054
Enter the total number of processes (max 10): 4
Enter Arrival Time and Burst Time for each process:
Process 1:
Arrival Time: 5
Burst Time: 5
Process 2:
Arrival Time: 4
Burst Time: 6
Process 3:
Arrival Time: 3
Burst Time: 7
Process 4:
Arrival Time: 1
Burst Time: 9
Enter Time Quantum: 4
PID Arrival Time    Burst Time  Completion Time Turnaround Time Waiting Time
1   5       5       22      17      12
2   4       6       24      20      14
3   3       7       27      24      17
4   1       9       28      27      18

Average Completion Time: 25.25
Average Turnaround Time: 22.00
Average Waiting Time: 15.25
```

LAB- 3

3-Question:

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
};
void FCFS(struct process *queue, int n) {
    int i, j;
    struct process temp;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (queue[i].arrival_time > queue[j].arrival_time) {
                temp = queue[i];
                queue[i] = queue[j];
                queue[j] = temp;
            }
        }
    }
}

int main() {
    int n, i;
    struct process *system_queue, *user_queue;
    int system_n = 0, user_n = 0;
    float avg_waiting_time = 0, avg_turnaround_time = 0;
    printf("Arugunta Hamsika-1BM22CS054\n");
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    system_queue = (struct process *) malloc(n * sizeof(struct process));
    user_queue = (struct process *) malloc(n * sizeof(struct process));
    for (i = 0; i < n; i++) {
        struct process p;
        printf("Enter arrival time, burst time, and priority (0-System/1-User) for process %d: ", i + 1);
        scanf("%d %d %d", &p.arrival_time, &p.burst_time, &p.priority);
        p.pid = i + 1;
        p.waiting_time = 0;
```

```
        p.turnaround_time = 0;
        if (p.priority == 0) {
            system_queue[system_n++] = p;
        } else {
            user_queue[user_n++] = p;
        }
    }

    FCFS(system_queue, system_n);
    FCFS(user_queue, user_n);
    int time = 0;
    int s = 0, u = 0;
    while (s < system_n || u < user_n) {
        if (s < system_n && system_queue[s].arrival_time <= time) {
            if (u < user_n && user_queue[u].arrival_time <= time && user_queue[u].arrival_time <
system_queue[s].arrival_time) {
                user_queue[u].waiting_time = time - user_queue[u].arrival_time;
                time += user_queue[u].burst_time;
                user_queue[u].turnaround_time = user_queue[u].waiting_time + user_queue[u].burst_time;
                avg_waiting_time += user_queue[u].waiting_time;
                avg_turnaround_time += user_queue[u].turnaround_time;
                u++;
            } else {
                system_queue[s].waiting_time = time - system_queue[s].arrival_time;
                time += system_queue[s].burst_time;
                system_queue[s].turnaround_time = system_queue[s].waiting_time + system_queue[s].burst_time;
                avg_waiting_time += system_queue[s].waiting_time;
                avg_turnaround_time += system_queue[s].turnaround_time;
                s++;
            }
        } else if (u < user_n && user_queue[u].arrival_time <= time) {
            user_queue[u].waiting_time = time - user_queue[u].arrival_time;
            time += user_queue[u].burst_time;
            user_queue[u].turnaround_time = user_queue[u].waiting_time + user_queue[u].burst_time;
            avg_waiting_time += user_queue[u].waiting_time;
            avg_turnaround_time += user_queue[u].turnaround_time;
              u++;
        } else {
            if (s < system_n && system_queue[s].arrival_time <= user_queue[u].arrival_time) {
                time = system_queue[s].arrival_time;
            } else {
                time = user_queue[u].arrival_time;
            }
        }
     }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;
    printf("PID\tBurst Time\tPriority\tQueue Type\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < system_n; i++) {
        printf("%d\t%d\t\t%d\t\tSystem\t\t%d\t\t%d\n", system_queue[i].pid, system_queue[i].burst_time,
system_queue[i].priority, system_queue[i].waiting_time, system_queue[i].turnaround_time);
    }
```

```
    for (i = 0; i < user_n; i++) {
        printf("%d\t%d\t\t%d\t\tUser\t\t%d\t\t%d\n", user_queue[i].pid, user_queue[i].burst_time,
user_queue[i].priority, user_queue[i].waiting_time, user_queue[i].turnaround_time);
    }
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    free(system_queue);
    free(user_queue);
    return 0;
}
```

Output:

```
Arugunta Hamsika-1BM22CS054
Enter the number of processes: 4
Enter arrival time, burst time, and priority (0-System/1-User) for process 1:
    0 5 0
Enter arrival time, burst time, and priority (0-System/1-User) for process 2:
    2 3 1
Enter arrival time, burst time, and priority (0-System/1-User) for process 3:
    1 2 0
Enter arrival time, burst time, and priority (0-System/1-User) for process 4:
    4 4 1
PID Burst Time  Priority    Queue Type  Waiting Time    Turnaround Time
1   5       0       System      0       5
3   2       0       System      4       6
2   3       1       User        5       8
4   4       1       User        6       10
Average Waiting Time: 3.75
Average Turnaround Time: 7.25
```

LAB- 3

4-Question:

Write a C program to simulate Real-Time CPU Scheduling algorithms:

(a) Rate- Monotonic

(b) Earliest-deadline First

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#define MAX_PROCESS 10
typedef struct {
    int id;
    int burst_time;
    float priority;
} Task;

int num_of_process;
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
    remain_time[MAX_PROCESS], deadline[MAX_PROCESS],
    remain_deadline[MAX_PROCESS];
void get_process_info(int selected_algo) {
    printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
    scanf("%d", &num_of_process);

    if (num_of_process < 1) {
        exit(0);
    }

    for (int i = 0; i < num_of_process; i++) {
        printf("\nProcess %d:\n", i + 1);
        printf("==> Execution time: ");
        scanf("%d", &execution_time[i]);
        remain_time[i] = execution_time[i];

        if (selected_algo == 2) {
            printf("==> Deadline: ");
            scanf("%d", &deadline[i]);
        } else {
            printf("==> Period: ");
            scanf("%d", &period[i]);
        }
    }
}

int max(int a, int b, int c) {
    int max;
    if (a >= b && a >= c)
        max = a;
```

```
    else if (b >= a && b >= c)
        max = b;
    else if (c >= a && c >= b)
        max = c;
    return max;
}

int get_observation_time(int selected_algo) {
    if (selected_algo == 1) {
        return max(period[0], period[1], period[2]);
    } else if (selected_algo == 2) {
        return max(deadline[0], deadline[1], deadline[2]);
    }
}

void print_schedule(int process_list[], int cycles) {
    printf("\nScheduling:\n\n");
    printf("Time: ");
    for (int i = 0; i < cycles; i++) {
        if (i < 10)
            printf("| 0%d ", i);
        else
            printf("| %d ", i);
    }
    printf("|\n");

    for (int i = 0; i < num_of_process; i++) {
        printf("P[%d]: ", i + 1);
        for (int j = 0; j < cycles; j++) {
            if (process_list[j] == i + 1)
                printf("|####");
            else
                printf("|    ");
        }
        printf("|\n");
    }
}

void rate_monotonic(int time) {
    int process_list[100] = {0}, min = 999, next_process = 0;
    float utilization = 0;

    for (int i = 0; i < num_of_process; i++) {
        utilization += (1.0 * execution_time[i]) / period[i];
    }

    int n = num_of_process;
    int m = (float) (n * (pow(2, 1.0 / n) - 1));

    if (utilization > m) {
        printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");
    }
```

```
    for (int i = 0; i < time; i++) {
        min = 1000;
        for (int j = 0; j < num_of_process; j++) {
            if (remain_time[j] > 0) {
                if (min > period[j]) {
                    min = period[j];
                    next_process = j;
                }
            }
        }
        if (remain_time[next_process] > 0) {
            process_list[i] = next_process + 1;
            remain_time[next_process] -= 1;
        }

        for (int k = 0; k < num_of_process; k++) {
            if ((i + 1) % period[k] == 0) {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }

    print_schedule(process_list, time);
}

void earliest_deadline_first(int time) {
    float utilization = 0;

    for (int i = 0; i < num_of_process; i++) {
        utilization += (1.0 * execution_time[i]) / deadline[i];
    }

    int n = num_of_process;
    int process[num_of_process];
    int max_deadline, current_process = 0, min_deadline, process_list[time];
    bool is_ready[num_of_process];

    for (int i = 0; i < num_of_process; i++) {
        is_ready[i] = true;
        process[i] = i + 1;
    }

    max_deadline = deadline[0];

    for (int i = 1; i < num_of_process; i++) {
        if (deadline[i] > max_deadline)
            max_deadline = deadline[i];
    }

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

```
    for (int j = i + 1; j < num_of_process; j++) {
        if (deadline[j] < deadline[i]) {
            int temp = execution_time[j];
            execution_time[j] = execution_time[i];
            execution_time[i] = temp;
            temp = deadline[j];
            deadline[j] = deadline[i];
            deadline[i] = temp;
            temp = process[j];
            process[j] = process[i];
            process[i] = temp;
        }
    }
}

for (int i = 0; i < num_of_process; i++) {
    remain_time[i] = execution_time[i];
    remain_deadline[i] = deadline[i];
}

for (int t = 0; t < time; t++) {
    if (current_process != -1) {
        --execution_time[current_process];
        process_list[t] = process[current_process];
    } else
        process_list[t] = 0;

    for (int i = 0; i < num_of_process; i++) {
        --deadline[i];

        if ((execution_time[i] == 0) && is_ready[i]) {
            deadline[i] += remain_deadline[i];
            is_ready[i] = false;
        }

        if ((deadline[i] <= remain_deadline[i]) && (is_ready[i] == false)) {
            execution_time[i] = remain_time[i];
            is_ready[i] = true;
        }
    }

    min_deadline = max_deadline;

    current_process = -1;

    for (int i = 0; i < num_of_process; i++) {
        if ((deadline[i] <= min_deadline) && (execution_time[i] > 0)) {
            current_process = i;
            min_deadline = deadline[i];
        }
    }
}
```

```
    print_schedule(process_list, time);
}

int main() {
    int option;
    int observation_time;

    while (1) {
        printf("\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your
choice: ");
        scanf("%d", &option);

        switch (option) {
            case 1:
                get_process_info(option);
                observation_time = get_observation_time(option);
                rate_monotonic(observation_time);
                break;
            case 2:
                get_process_info(option);
                observation_time = get_observation_time(option);
                earliest_deadline_first(observation_time);
                break;
            case 3:
                exit(0);
            default:
                printf("\nInvalid Statement\n");
        }
    }
    return 0;
}
```

Output:

(a) Rate Monotonic:

```
Arugunta Hamsika-1BM22Cs054

1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 1
Enter total number of processes (maximum 10): 2
Process 1:
==> Execution time: 4
==> Period: 6
Process 2:
==> Execution time: 6
==> Period: 2
Given problem is not schedulable under the said scheduling algorithm.

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 |
P[1]: |    |    |    |    |    |    |
P[2]: |####|####|####|####|####|####|
```

(b) Earliest Deadline First:

```
Arugunta Hamsika-1BM22Cs054

1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 2
Enter total number of processes (maximum 10): 2
Process 1:
==> Execution time: 4
==> Deadline: 6
Process 2:
==> Execution time: 3
==> Deadline: 2
Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 |
P[1]: |    |    |    |    |    |    |
P[2]: |####|####|####|####|####|####|
```

LAB- 4

5-Question:
Write a C program to simulate producer-consumer problem using semaphores.

Code:

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

int mutex = 1, full = 0, empty = 3, x = 0;

int wait(int);
int signal(int);
void producer();
void consumer();

int main() {
   int n;
   printf("Arugunta Hamsika-1BM22CS054\n");
   printf("\n1.Producer\n2.Consumer\n3.Exit");
   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;
         default:
            printf("Invalid choice\n");
            break;
      }
   }
   return 0;
}

int wait(int s) {
   return (--s);
}
```

```
int signal(int s) {
   return (++s);
}

void producer() {
   mutex = wait(mutex);
   empty = wait(empty);
   x++;
   printf("\nProducer produces the item %d\n", x);
   full = signal(full);
   mutex = signal(mutex);
}

void consumer() {
   mutex = wait(mutex);
   full = wait(full);
   x--;
   printf("\nConsumer consumes item %d\n", x);
   empty = signal(empty);
   mutex = signal(mutex);
}
```

Output:

```
Arugunta Hamsika-1BM22CS054

1.Producer
2.Consumer
3.Exit
Enter your choice: 1
Producer produces the item 1

Enter your choice: 1
Producer produces the item 2

Enter your choice: 2
Consumer consumes item 1

Enter your choice: 3
```

LAB - 4

6-Question:

Write a C program to simulate the concept of Dining-Philosophers problem.

Code:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h> // for sleep function
#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (i + N - 1) % N
#define RIGHT (i + 1) % N

int state[N];
int phil[N] = {0, 1, 2, 3, 4};
sem_t mutex;
sem_t S[N];

void test(int i)
{
   if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
   {
      state[i] = EATING;
      sleep(2); // Eating for some time
      printf("Philosopher %d takes fork %d and %d\n", i + 1, LEFT + 1, i + 1);
      printf("Philosopher %d is Eating\n", i + 1);
      sem_post(&S[i]);
   }
}

void take_fork(int i)
{
   sem_wait(&mutex);
   state[i] = HUNGRY;
   printf("Philosopher %d is Hungry\n", i + 1);
   test(i);
   sem_post(&mutex);
   sem_wait(&S[i]);
   sleep(1); // Waiting for a moment after picking up forks
}

void put_fork(int i)
{
   sem_wait(&mutex);
   state[i] = THINKING;
   printf("Philosopher %d putting fork %d and %d down\n", i + 1, LEFT + 1, i + 1);
```

```
    printf("Philosopher %d is thinking\n", i + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void *philosopher(void *num)
{
    while (1)
    {
        int *i = num;
        sleep(1); // Thinking for a while
        take_fork(*i);
        sleep(0); // Slight delay
        put_fork(*i);
    }
}

int main()
{
    int i;
    printf("Arugunta Hamsika-1BM22CS054\n");
    pthread_t thread_id[N];

    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++)
    {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)
    {
        pthread_join(thread_id[i], NULL);
    }

    return 0;
}
```

Output:

```
Arugunta Hamsika-1BM22CS054
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
```

LAB - 5

7-Question:

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

Code:

```
#include <stdio.h>
int main()
{
   int n, m, i, j, k;
   printf("Arugunta hamsika-1BM22CS054\n");
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the number of resources: ");
   scanf("%d", &m);

   int allocation[n][m];
   printf("Enter the Allocation Matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < m; j++)
      {
         scanf("%d", &allocation[i][j]);
      }
   }

   int max[n][m];
   printf("Enter the MAX Matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < m; j++)
      {
         scanf("%d", &max[i][j]);
      }
   }

   int available[m];
   printf("Enter the Available Resources:\n");
   for (i = 0; i < m; i++)
   {
      scanf("%d", &available[i]);
   }

   int f[n], ans[n], ind = 0;
   for (k = 0; k < n; k++)
   {
      f[k] = 0;
   }

   int need[n][m];
```

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

int y = 0;
for (k = 0; k < n; k++)
{
   for (i = 0; i < n; i++)
   {
      if (f[i] == 0)
      {
         int flag = 0;
         for (j = 0; j < m; j++)
         {
            if (need[i][j] > available[j])
            {
               flag = 1;
               break;
            }
         }
         if (flag == 0)
         {
            ans[ind++] = i;
            for (y = 0; y < m; y++)
            {
               available[y] += allocation[i][y];
            }
            f[i] = 1;
         }
      }
   }
}

int safe = 1;
for (i = 0; i < n; i++)
{
   if (f[i] == 0)
   {
      safe = 0;
      printf("The system is not in a safe state\n");
      break;
   }
}

if (safe == 1)
{
   printf("Following is the SAFE Sequence: ");
   for (i = 0; i < n - 1; i++)
```

```
        {
            printf("P%d -> ", ans[i]);
        }
        printf("P%d\n", ans[n - 1]);
    }

    return 0;
}
```

Output:

```
Arugunta hamsika-1BM22CS054
Enter the number of processes: 3
Enter the number of resources: 3
Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
Enter the MAX Matrix:
7 5 3
3 2 2
9 0 2
Enter the Available Resources:
3 3 2
The system is not in a safe state
```

```
Arugunta hamsika-1BM22CS054
Enter the number of processes: 3
Enter the number of resources: 3
Enter the Allocation Matrix:
1 0 0
0 2 0
1 1 1
Enter the MAX Matrix:
1 2 1
1 3 1
1 3 2
Enter the Available Resources:
1 1 1
Following is the SAFE Sequence: P1 -> P2 -> P0
```

LAB- 5

8-Question:
Write a C program to simulate deadlock detection.

Code:

```
#include <stdio.h>
static int mark[20];
int i, j, np, nr;

int main()
{
   int alloc[10][10], request[10][10], avail[10], r[10], w[10];
   printf("Arugunta Hamsika-1BM22CS054\n");
   printf("\nEnter the number of processes: ");
   scanf("%d", &np);
   printf("Enter the number of resources: ");
   scanf("%d", &nr);
   for (i = 0; i < nr; i++)
   {
      printf("Total Amount of the Resource R%d: ", i + 1);
      scanf("%d", &r[i]);
   }
   printf("\nEnter the request matrix:\n");
   for (i = 0; i < np; i++)
   {
      for (j = 0; j < nr; j++)
      {
         scanf("%d", &request[i][j]);
      }
   }
   printf("\nEnter the allocation matrix:\n");
   for (i = 0; i < np; i++)
   {
      for (j = 0; j < nr; j++)
      {
         scanf("%d", &alloc[i][j]);
      }
   }
   /* Available Resource calculation */
   for (j = 0; j < nr; j++)
   {
      avail[j] = r[j];
      for (i = 0; i < np; i++)
      {
         avail[j] -= alloc[i][j];
      }
   } // Marking processes with zero allocation
   for (i = 0; i < np; i++)
```

```
{
   int count = 0;
   for (j = 0; j < nr; j++)
   {
      if (alloc[i][j] == 0)
         count++;
      else
         break;
   }
   if (count == nr)
      mark[i] = 1;
}

// Initialize W with avail
for (j = 0; j < nr; j++)
   w[j] = avail[j];

// Mark processes with request less than or equal to W
for (i = 0; i < np; i++)
{
   int canbeprocessed = 0;
   if (mark[i] != 1)
   {
      for (j = 0; j < nr; j++)
      {
         if (request[i][j] <= w[j])
            canbeprocessed = 1;
         else
         {
            canbeprocessed = 0;
            break;
         }
      }
      if (canbeprocessed)
      {
         mark[i] = 1;
         for (j = 0; j < nr; j++)
            w[j] += alloc[i][j];
      }
   }
}

// Checking for unmarked processes
int deadlock = 0;
for (i = 0; i < np; i++)
{
   if (mark[i] != 1)
      deadlock = 1;
}

if (deadlock)
   printf("\nDeadlock detected\n");
```

```
    else
        printf("\nNo Deadlock possible\n");

    return 0;
}
```

Output:

```
Arugunta Hamsika-1BM22CS054

Enter the number of processes: 3
Enter the number of resources: 3
Total Amount of the Resource R1: 7
Total Amount of the Resource R2: 2
Total Amount of the Resource R3: 6
Enter the request matrix:
0 1 0
2 0 0
3 0 2
Enter the allocation matrix:
0 0 0
2 0 0
3 0 2
No Deadlock possible
```

```
Arugunta Hamsika-1BM22CS054

Enter the number of processes: 5
Enter the number of resources: 3
Total Amount of the Resource R1: 0
Total Amount of the Resource R2: 0
Total Amount of the Resource R3: 0
Enter the request matrix:
0 0 0
2 0 2
0 0 0
1 0 0
0 0 2
Enter the allocation matrix:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Deadlock detected
```

LAB 6

9-Question:
Write a C program to simulate the following contiguous memory allocation techniques:
(a) Worst-fit
(b) Best-fit
(c) First-fit

Code:

```
#include <stdio.h>

#define max 25

void firstFit(int b[], int nb, int f[], int nf);
void worstFit(int b[], int nb, int f[], int nf);
void bestFit(int b[], int nb, int f[], int nf);

int main() {
   int b[max], f[max], nb, nf;

   printf("Memory Management Schemes\n");
   printf("\nEnter the number of blocks: ");
   scanf("%d", &nb);
   printf("Enter the number of files: ");
   scanf("%d", &nf);

   printf("\nEnter the size of the blocks:\n");
   for (int i = 1; i <= nb; i++) {
      printf("Block %d: ", i);
      scanf("%d", &b[i]);
   }

   printf("\nEnter the size of the files:\n");
   for (int i = 1; i <= nf; i++) {
      printf("File %d: ", i);
      scanf("%d", &f[i]);
   }

   printf("\nMemory Management Scheme - First Fit");
   firstFit(b, nb, f, nf);

   printf("\n\nMemory Management Scheme - Worst Fit");
   worstFit(b, nb, f, nf);

   printf("\n\nMemory Management Scheme - Best Fit");
   bestFit(b, nb, f, nf);

   return 0;
```

```
}

void firstFit(int b[], int nb, int f[], int nf) {
int bf[max] = {0};
   int ff[max] = {0};
   int frag[max], i, j;

   for (i = 1; i <= nf; i++) {
      for (j = 1; j <= nb; j++) {
         if (bf[j] != 1 && b[j] >= f[i]) {
            ff[i] = j;
            bf[j] = 1;
            frag[i] = b[j] - f[i];
            break;
         }
      }
   }

   printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
   for (i = 1; i <= nf; i++) {
      printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
   }
}

void worstFit(int b[], int nb, int f[], int nf) {
   int bf[max] = {0};
   int ff[max] = {0};
   int frag[max], i, j, temp, highest = 0;

   for (i = 1; i <= nf; i++) {
      for (j = 1; j <= nb; j++) {
         if (bf[j] != 1) {
            temp = b[j] - f[i];
            if (temp >= 0 && highest < temp) {
               ff[i] = j;
               highest = temp;
            }
         }
      }
      frag[i] = highest;
      bf[ff[i]] = 1;
      highest = 0;
   }

   printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
   for (i = 1; i <= nf; i++) {
      printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
   }
}

void bestFit(int b[], int nb, int f[], int nf) {
   int bf[max] = {0};
```

```
    int ff[max] = {0};
    int frag[max], i, j, temp, lowest = 10000;

    for (i = 1; i <= nf; i++) {
        for (j = 1; j <= nb; j++) {
            if (bf[j] != 1) {
                temp = b[j] - f[i];
                if (temp >= 0 && lowest > temp) {
                    ff[i] = j;
                    lowest = temp;
                }
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 10000;
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (i = 1; i <= nf && ff[i] != 0; i++) {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
    }
}
```

Output:

```
Enter the number of blocks: 3
Enter the number of files: 4
Enter the size of the blocks:
Block 1: 10
Block 2: 5
Block 3: 20
Enter the size of the files:
File 1: 12
File 2: 7
File 3: 10
File 4: 15
Memory Management Scheme - First Fit
File_no:    File_size:  Block_no:   Block_size: Fragment
1       12      1       10      8
2       7       Not Allocated   Not Allocated   Not Allocated
3       10      3       20      0
4       15      Not Allocated   Not Allocated   Not Allocated


Memory Management Scheme - Worst Fit
File_no:    File_size:  Block_no:   Block_size: Fragment
1       12      1       10      8
2       7       Not Allocated   Not Allocated   Not Allocated
3       10      3       20      0
4       15      Not Allocated   Not Allocated   Not Allocated


Memory Management Scheme - Best Fit
File_no:    File_size:  Block_no:   Block_size: Fragment
1       12      1       10      8
2       7       Not Allocated   Not Allocated   Not Allocated
3       10      3       20      0
4       15      Not Allocated   Not Allocated   Not Allocated
```

10-Question :
Write a C program to simulate page replacement algorithms:
(a) FIFO
(b) LRU
(c) Optimal

Code:

```
#include <stdio.h>

#define MAX_FRAMES 50
#define MAX_PAGES 100

int n, f, pgfaultcnt = 0;
int in[MAX_PAGES];
int p[MAX_FRAMES];

void getData() {
   printf("\nEnter length of page reference sequence: ");
   scanf("%d", &n);
   printf("Enter the page reference sequence:\n");
   for (int i = 0; i < n; i++)
      scanf("%d", &in[i]);
   printf("Enter number of frames: ");
   scanf("%d", &f);
}

void initialize() {
   pgfaultcnt = 0;
   for (int i = 0; i < f; i++)
      p[i] = -1; // Initialize frame values to -1 (indicating empty frame)
}

int isHit(int data) {
   for (int j = 0; j < f; j++) {
      if (p[j] == data)
         return 1; // Page hit
   }
   return 0; // Page fault
}

int getHitIndex(int data) {
   for (int j = 0; j < f; j++) {
      if (p[j] == data)
         return j;
   }
   return -1;
}

void dispPages() {
   for (int k = 0; k < f; k++) {
```

```
        if (p[k] != -1)
            printf(" %d", p[k]);
    }
}

void dispPgFaultCnt() {
    printf("\nTotal number of page faults: %d\n", pgfaultcnt);
}

void fifo() {
    initialize();
    int frame_index = 0;

    for (int i = 0; i < n; i++) {
        printf("\nFor %d :", in[i]);

        if (!isHit(in[i])) {
            p[frame_index] = in[i];
            frame_index = (frame_index + 1) % f;
            pgfaultcnt++;
            dispPages();
        } else {
            printf(" No page fault");
        }
    }

    dispPgFaultCnt();
}

void optimal() {
    initialize();

    for (int i = 0; i < n; i++) {
        printf("\nFor %d :", in[i]);

        if (!isHit(in[i])) {
            int farthest = i;
            int replace_index = 0;

            for (int j = 0; j < f; j++) {
                int current_page = p[j];
                int found = 0;

                for (int k = i; k < n; k++) {
                    if (current_page == in[k]) {
                        if (k > farthest) {
                            farthest = k;
                            replace_index = j;
                        }
                        found = 1;
                        break;
                    }
```

```
            }

            if (!found) {
                replace_index = j;
                break;
            }
        }

        p[replace_index] = in[i];
        pgfaultcnt++;
        dispPages();
    } else {
        printf(" No page fault");
    }
  }

  dispPgFaultCnt();
}

void lru() {
  initialize();
  int used[MAX_FRAMES] = {0};

  for (int i = 0; i < n; i++) {
    printf("\nFor %d :", in[i]);

    if (!isHit(in[i])) {
        int replace_index = 0;
        int least_used = used[0];

        for (int j = 0; j < f; j++) {
            if (used[j] < least_used) {
                least_used = used[j];
                replace_index = j;
            }
        }

        p[replace_index] = in[i];
        used[replace_index] = i + 1; // Update least recently used time
        pgfaultcnt++;
        dispPages();
    } else {
        printf(" No page fault");
        used[getHitIndex(in[i])] = i + 1; // Update recently used time
    }
  }

  dispPgFaultCnt();
}

int main() {
  int choice;
```

```
    printf("Arugunta hamsika-1BM22CS054\n");
    while (1) {
        printf("\nPage Replacement Algorithms\n");
        printf("1. Enter data\n");
        printf("2. FIFO\n");
        printf("3. Optimal\n");
        printf("4. LRU\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                getData();
                break;
            case 2:
                fifo();
                break;
            case 3:
                optimal();
                break;
            case 4:
                lru();
                break;
            case 5:
                return 0;
            default:
                printf("Invalid choice!\n");
                break;
        }
    }

    return 0;
}
```

Output:

```
Arugunta hamsika-1BM22CS054

Page Replacement Algorithms
1. Enter data
2. FIFO
3. Optimal
4. LRU
5. Exit
Enter your choice: 1
Enter length of page reference sequence: 10
Enter the page reference sequence:
7 0 1 2 0 3 0 4 2 3
Enter number of frames: 3
Page Replacement Algorithms
1. Enter data
2. FIFO
3. Optimal
4. LRU
5. Exit
Enter your choice: 2
For 7 : 7
For 0 : 7 0
For 1 : 7 0 1
For 2 : 2 0 1
For 0 : No page fault
For 3 : 2 3 1
For 0 : 2 3 0
For 4 : 4 3 0
For 2 : 4 2 0
For 3 : 4 2 3
Total number of page faults: 9
```

LAB-7
11-Question:
Write a C program to simulate disk scheduling algorithms:
(a) FCFS

Code:

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

int main() {
   int RQ[100], i, n, TotalHeadMoment = 0, initial;
   printf("Arugunta Hamsika-1BM22Cs054\n");
   printf("Enter the number of requests: ");
   scanf("%d", &n);

   printf("Enter the request sequence: ");
   for (i = 0; i < n; i++)
      scanf("%d", &RQ[i]);

   printf("Enter initial head position: ");
   scanf("%d", &initial);

   // FCFS disk scheduling logic
   for (i = 0; i < n; i++) {
      TotalHeadMoment += abs(RQ[i] - initial);
      initial = RQ[i];
   }

   printf("Total head movement is %d\n", TotalHeadMoment);

   return 0;
}
```

Output:

```
Arugunta Hamsika-1BM22Cs054
Enter the number of requests: 8
Enter the request sequence: 95
180
34
119
11
123
62
64
Enter initial head position: 50
Total head movement is 644
```

(b) SCAN

Code:

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

int main() {
   int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;

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

   printf("Enter the Requests sequence: ");
   for (i = 0; i < n; i++)
      scanf("%d", &RQ[i]);

   printf("Enter initial head position: ");
   scanf("%d", &initial);

   printf("Enter total disk size: ");
   scanf("%d", &size);

   printf("Enter the head movement direction (1 for high, 0 for low): ");
   scanf("%d", &move);

   // Logic to sort the request array (bubble sort)
   for (i = 0; i < n; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (RQ[j] > RQ[j + 1]) {
            int temp = RQ[j];
            RQ[j] = RQ[j + 1];
            RQ[j + 1] = temp;
         }
      }
   }

   // Finding the initial index
   int index;
   for (i = 0; i < n; i++) {
      if (initial < RQ[i]) {
         index = i;
         break;
      }
   }
   if (move == 1) {
      for (i = index; i < n; i++) {
         TotalHeadMoment += abs(RQ[i] - initial);
         initial = RQ[i];
      }
      TotalHeadMoment += abs(size - 1 - RQ[i - 1]);
      initial = size - 1;
      for (i = index - 1; i >= 0; i--) {
         TotalHeadMoment += abs(RQ[i] - initial);
```

```
            initial = RQ[i];
        }
    } else {
        for (i = index - 1; i >= 0; i--) {
            TotalHeadMoment += abs(RQ[i] - initial);
            initial = RQ[i];
        }
        TotalHeadMoment += abs(RQ[i + 1] - 0);
        initial = 0;
        for (i = index; i < n; i++) {
            TotalHeadMoment += abs(RQ[i] - initial);
            initial = RQ[i];
        }
    }

    printf("Total head movement is %d\n", TotalHeadMoment);

    return 0;
}
```

Output:

```
Arugunta hamsika-1BN22Cs054
Enter the number of Requests: 8
Enter the Requests sequence: 98
183
37
122
14
124
65
67
Enter initial head position: 53
Enter total disk size: 199
Enter the head movement direction (1 for high, 0 for low): 0
Total head movement is 236
```

(c) C-SCAN

Code:

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

int main() {
   int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;

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

   printf("Enter the Requests sequence: ");
   for (i = 0; i < n; i++)
      scanf("%d", &RQ[i]);

   printf("Enter initial head position: ");
   scanf("%d", &initial);

   printf("Enter total disk size: ");
   scanf("%d", &size);

   printf("Enter the head movement direction (1 for high to low, 0 for low to high): ");
   scanf("%d", &move);

   // Sorting the request array using bubble sort
   for (i = 0; i < n; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (RQ[j] > RQ[j + 1]) {
            int temp = RQ[j];
            RQ[j] = RQ[j + 1];
            RQ[j + 1] = temp;
         }
      }
   }

   // Finding the initial index
   int index;
   for (i = 0; i < n; i++) {
      if (initial < RQ[i]) {
         index = i;
         break;
      }
   }

   // C-SCAN (Circular SCAN) disk scheduling algorithm
   if (move == 1) { // Movement towards high value
      // Service requests from initial index to the end
      for (i = index; i < n; i++) {
         TotalHeadMoment += abs(RQ[i] - initial);
         initial = RQ[i];
      }
```

```
        // Last movement to the end of the disk
        TotalHeadMoment += abs(size - 1 - RQ[i - 1]);
        // Wrap around to the beginning of the disk and service requests
        TotalHeadMoment += abs(size - 1 - 0);
        initial = 0;
        for (i = 0; i < index; i++) {
            TotalHeadMoment += abs(RQ[i] - initial);
            initial = RQ[i];
        }
    } else { // Movement towards low value
        // Service requests from initial index to the beginning
        for (i = index - 1; i >= 0; i--) {
            TotalHeadMoment += abs(RQ[i] - initial);
            initial = RQ[i];
        }
        // Last movement to the beginning of the disk
        TotalHeadMoment += abs(RQ[i + 1] - 0);
        // Wrap around to the end of the disk and service requests
        TotalHeadMoment += abs(size - 1 - 0);
        initial = size - 1;
        for (i = n - 1; i >= index; i--) {
            TotalHeadMoment += abs(RQ[i] - initial);
            initial = RQ[i];
        }
    }

    printf("Total head movement is %d\n", TotalHeadMoment);

    return 0;
}
```

Output:

```
Arugunta hamsika-1BM22CS054
Enter the number of Requests: 8
Enter the Requests sequence: 98
183
37
122
14
124
65
67
Enter initial head position: 53
Enter total disk size: 199
Enter the head movement direction (1 for high to low, 0 for low to high): 1
Total head movement is 380
```

```
Arugunta hamsika-1BM22CS054
Enter the number of Requests: 8
Enter the Requests sequence: 98
183
37
122
14
124
65
67
Enter initial head position: 53
Enter total disk size: 199
Enter the head movement direction (1 for high to low, 0 for low to high): 0
Total head movement is 384
```