VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

RITESH MOHAN NAYAK (1BM23CS350)

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
Feb-2025 to June-2025

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "OPERATING SYSTEMS – 23CS4PCOPS" carried out by **RITESH MOHAN NAYAK (1BM23CS350)** who is bona fide 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 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Sonika Sharma D
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System
CO2	Analyze 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

PROGRAM-1:

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

a) FCFS
b) SJF

FCFS scheduling:

```
#include <stdio.h>

typedef struct {
    int id, arrival, burst, completion, turnaround, waiting;
} Process;

void sortByArrival(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 > p[j + 1].arrival) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

void fcfs(Process p[], int n, float *avgTAT, float *avgWT) {
    sortByArrival(p, n);
    int time = 0, totalTAT = 0, totalWT = 0;

    for (int i = 0; i < n; i++) {
        if (time < p[i].arrival)
            time = p[i].arrival;

        p[i].completion = time + p[i].burst;
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;

        time = p[i].completion;
        totalTAT += p[i].turnaround;
        totalWT += p[i].waiting;
    }

    *avgTAT = (float)totalTAT / n;
    *avgWT = (float)totalWT / n;
}

void display(Process p[], int n, float avgTAT, float avgWT) {
    printf("\nPID  Arrival  Burst  Completion  Turnaround  Waiting\n");
    for (int i = 0; i < n; i++) {
        printf("%3d  %7d  %5d  %10d  %10d  %7d\n",
               p[i].id, p[i].arrival, p[i].burst,
               p[i].completion,  p[i].turnaround,  p[i].waiting);
    }
    printf("\nAverage Turnaround Time: %.2f", avgTAT);
    printf("\nAverage Waiting Time: %.2f\n", avgWT);
}

int main() {
```

```c
    int n;
    float avgTAT, avgWT;

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

    Process p[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }

    printf("\nFirst Come First Serve (FCFS) Scheduling\n");
    fcfs(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);

    return 0;
}
```

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0
5
P[2]: 1
3
P[3]: 2
8
P[4]: 3
6

First Come First Serve (FCFS) Scheduling

PID  Arrival  Burst  Completion  Turnaround  Waiting
  1        0      5           5           5        0
  2        1      3           8           7        4
  3        2      8          16          14        6
  4        3      6          22          19       13

Average Turnaround Time: 11.25
Average Waiting Time: 5.75

Process returned 0 (0x0)   execution time : 25.615 s
Press any key to continue.
```

SJF scheduling:

```c
#include<stdio.h>

int main() {
    int n, i, j;
    int bt[20], p[20], wt[20], tat[20], temp;
    float avg_wt = 0, avg_tat = 0;

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

    printf("Enter the burst time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("Process %d: ", i+1);
        scanf("%d", &bt[i]);
        p[i] = i+1;
    }
```

```c
    for(i = 0; i < n-1; i++) {
        for(j = i+1; j < n; j++) {
            if(bt[i] > bt[j]) {

                temp = bt[i];
                bt[i]  =  bt[j];
                bt[j] = temp;


                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }

    wt[0] = 0;


    for(i = 1; i < n; i++) {
        wt[i] = wt[i-1] + bt[i-1];
    }


    for(i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
        avg_wt += wt[i];
        avg_tat += tat[i];
    }

    avg_wt /= n;
    avg_tat /= n;

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", avg_wt);
    printf("\nAverage Turnaround Time: %.2f\n", avg_tat);

    return 0;
```

```
Enter the number of processes: 4
Enter the burst time for each process:
Process 1: 6
Process 2: 8
Process 3: 7
Process 4: 3

Process Burst Time      Waiting Time    Turnaround Time
4       3               0               3
1       6               3               9
3       7               9               16
2       8               16              24

Average Waiting Time: 7.00
Average Turnaround Time: 13.00

Process returned 0 (0x0)   execution time : 17.113 s
Press any key to continue.
```

PROGRAM-2:

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.

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

#define MAX_PROCESSES 10
#define MAX_NAME_LENGTH 20


typedef struct {
    int pid;  // Process ID
    char name[MAX_NAME_LENGTH];  // Process name
    int burst_time;  // Burst time for the process
} Process;


void FCFS(Process queue[], int n) {
    int total_wait_time = 0, total_turnaround_time = 0;
    printf("\nScheduling processes with FCFS...\n");
    printf("PID\tName\tBurst Time\tWaiting Time\tTurnaround Time\n");

    int waiting_time = 0;
    for (int i = 0; i < n; i++) {
        int turnaround_time = queue[i].burst_time + waiting_time;
        printf("%d\t%s\t%d\t\t%d\t\t%d\n", queue[i].pid, queue[i].name, queue[i].burst_time, waiting_time, turnaround_time);

        total_wait_time += waiting_time;
        total_turnaround_time += turnaround_time;

        waiting_time += queue[i].burst_time;  // Update waiting time for the next process
    }

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


void input_processes(Process queue[], int *n) {
    printf("Enter the number of processes: ");
    scanf("%d", n);

    for (int i = 0; i < *n; i++) {
        queue[i].pid = i + 1;
        printf("Enter the name of process %d: ", i + 1);
        scanf("%s", queue[i].name);
        printf("Enter the burst time of process %d: ", i + 1);
        scanf("%d", &queue[i].burst_time);
    }
}


int main() {
    Process system_queue[MAX_PROCESSES], user_queue[MAX_PROCESSES];
    int system_count, user_count;


    printf("Enter details for system processes:\n");
    input_processes(system_queue, &system_count);
```

```
    printf("\nEnter details for user processes:\n");
    input_processes(user_queue, &user_count);

    printf("\nScheduling system processes:\n");
    FCFS(system_queue, system_count);

    printf("\nScheduling user processes:\n");
    FCFS(user_queue, user_count);

    return 0;
}
```

```
Enter details for system processes:
Enter the number of processes: 2
Enter the name of process 1: SysA
Enter the burst time of process 1: 5
Enter the name of process 2: SysB
Enter the burst time of process 2: 3

Enter details for user processes:
Enter the number of processes: 3
Enter the name of process 1: UserX
Enter the burst time of process 1: 4
Enter the name of process 2: UserY
Enter the burst time of process 2: 6
Enter the name of process 3: UserZ
Enter the burst time of process 3: 2

Scheduling system processes:

Scheduling processes with FCFS...
PID     Name    Burst Time      Waiting Time    Turnaround Time
1       SysA    5               0               5
2       SysB    3               5               8

Average Waiting Time: 2.50
Average Turnaround Time: 6.50

Scheduling user processes:

Scheduling processes with FCFS...
PID     Name    Burst Time      Waiting Time    Turnaround Time
1       UserX   4               0               4
2       UserY   6               4               10
3       UserZ   2               10              12

Average Waiting Time: 4.67
Average Turnaround Time: 8.67

Process returned 0 (0x0)   execution time : 35.784 s
Press any key to continue.
```

PROGRAM-3:

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

a) Rate- Monotonic

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

#define MAX_TASKS 10

typedef struct {
    int id;
    int period;
    int execution_time;
    int remaining_time;
} Task;

void sort_by_priority(Task tasks[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (tasks[j].period > tasks[j + 1].period) {
                Task temp = tasks[j];
                tasks[j] = tasks[j + 1];
                tasks[j + 1] = temp;
            }
        }
    }
}

void rms_scheduling(Task tasks[], int n, int total_time) {
    sort_by_priority(tasks, n);

    for (int t = 0; t < total_time; t++) {
        int executed = 0;
        for (int i = 0; i < n; i++) {
            if (tasks[i].remaining_time > 0) {
                printf("Time %d: Executing Task %d\n", t, tasks[i].id);
                tasks[i].remaining_time--;

                if (tasks[i].remaining_time == 0)
                    tasks[i].remaining_time = tasks[i].execution_time; // Reset for next period

                executed = 1;
                break;
            }
        }
        if (!executed) {
            printf("Time %d: Idle\n", t);
        }
    }
}

int main() {
    Task tasks[MAX_TASKS] = {
        {1, 3, 1, 1},
        {2, 5, 2, 2},
        {3, 7, 2, 2}
    };
    int num_tasks = 3;
    int total_time = 15; // Simulation time

    rms_scheduling(tasks, num_tasks, total_time);
    return 0;
```

}

```
Time 0: Executing Task 1
Time 1: Executing Task 1
Time 2: Executing Task 1
Time 3: Executing Task 1
Time 4: Executing Task 1
Time 5: Executing Task 1
Time 6: Executing Task 1
Time 7: Executing Task 1
Time 8: Executing Task 1
Time 9: Executing Task 1
Time 10: Executing Task 1
Time 11: Executing Task 1
Time 12: Executing Task 1
Time 13: Executing Task 1
Time 14: Executing Task 1

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

PROGRAM-4:

Write a C program to simulate:

a) Producer-Consumer problem using semaphores.

b) Dining-Philosopher's problem

Producer-Consumer problem using semaphores:

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
int count = 0;

pthread_mutex_t mutex;
pthread_cond_t not_empty, not_full;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = i;

        pthread_mutex_lock(&mutex);

        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&not_full,  &mutex);
        }

        buffer[in] = item;
        printf("Producer produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        count++;

        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);

        sleep(1);
    }
    return NULL;
}

void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex);
        }

        item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        count--;

        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
```

```
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_empty, NULL);
    pthread_cond_init(&not_full, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons,  NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_empty);
    pthread_cond_destroy(&not_full);

    return 0;
}
```

```
Producer produced: 0
Consumer consumed: 0
Producer produced: 1
Consumer consumed: 1
Producer produced: 2
Consumer consumed: 2
Producer produced: 3
Consumer consumed: 3
Producer produced: 4
Consumer consumed: 4
Producer produced: 5
Consumer consumed: 5
Producer produced: 6
Consumer consumed: 6
Producer produced: 7
Consumer consumed: 7
Producer produced: 8
Consumer consumed: 8
Producer produced: 9
Consumer consumed: 9

Process returned 0 (0x0)   execution time : 10.135 s
Press any key to continue.
```

Dining-Philosopher's problem:

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

#define MAX 5

int hungry[MAX] = {0};

void displayStatus(int n) {
    for (int i = 0; i < MAX; i++) {
        if (hungry[i])
            printf("P %d is waiting\n", i + 1);
    }
}

void simulateEating(int id) {
    printf("P %d is granted to eat\n", id + 1);
    sleep(1);
    printf("P %d has finished eating\n", id + 1);
}

int main() {
    int total, hcount;
    int hungry_ids[3];
    int choice;

    printf("Enter the total number of philosophers: ");
    scanf("%d", &total);

    if (total != 5) {
        printf("This simulation only supports 5 philosophers.\n");
        return 1;
    }

    printf("How many are hungry: ");
    scanf("%d", &hcount);

    if (hcount > 3) {
        printf("Only up to 3 hungry philosophers supported in this simulation.\n");
        return 1;
    }

    for (int i = 0; i < hcount; i++) {
        printf("Enter philosopher %d position (1 to 5): ", i + 1);
        scanf("%d", &hungry_ids[i]);
        if (hungry_ids[i] < 1 || hungry_ids[i] > 5) {
            printf("Invalid position. Please enter between 1 and 5.\n");
            return 1;
        }
        hungry[hungry_ids[i] - 1] = 1;
    }

    do {
        printf("\n1. One can eat at a time\n2. Two can eat at a time\n3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Allow one philosopher to eat at any time\n");
                for (int i = 0; i < hcount; i++) {
```

```
                displayStatus(hcount);
                simulateEating(hungry_ids[i] - 1);
                hungry[hungry_ids[i] - 1] = 0;
            }
            break;

        case 2:
            printf("Allow two philosophers to eat at any time\n");
            for (int i = 0; i < hcount; i += 2) {
                displayStatus(hcount);
                simulateEating(hungry_ids[i] - 1);
                hungry[hungry_ids[i] - 1] = 0;
                if (i + 1 < hcount) {
                    simulateEating(hungry_ids[i + 1] - 1);
                    hungry[hungry_ids[i + 1] - 1] = 0;
                }
            }
            break;

        case 3:
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice.\n");
        }
    } while (choice != 3);

    return 0;
}
```

```
Enter the total number of philosophers: 5
How many are hungry: 3
Enter philosopher 1 position (1 to 5): 2
Enter philosopher 2 position (1 to 5): 3
Enter philosopher 3 position (1 to 5): 5

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
Allow one philosopher to eat at any time
P 2 is waiting
P 3 is waiting
P 5 is waiting
P 2 is granted to eat
P 2 has finished eating
P 3 is waiting
P 5 is waiting
P 3 is granted to eat
P 3 has finished eating
P 5 is waiting
P 5 is granted to eat
P 5 has finished eating

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice:
```

PROGRAM-5:

Write a C program to simulate:

a) Banker's algorithm for the purpose of deadlock avoidance.

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

#define MAX 10

int main() {
    int n, m;
    int alloc[MAX][MAX], max[MAX][MAX], need[MAX][MAX];
    int avail[MAX];
    int i, j, k;

    printf("Enter number of processes -- ");
    scanf("%d", &n);
    printf("Enter number of resources -- ");
    scanf("%d", &m);


    for (i = 0; i < n; i++) {
        printf("Enter details for P%d\n", i);
        printf("Enter allocation -- ");
        for (j = 0; j < m; j++) {
            scanf("%d", &alloc[i][j]);
        }
        printf("Enter Max -- ");
        for (j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
        }
    }


    printf("Enter Available Resources -- ");
    for (i = 0; i < m; i++) {
        scanf("%d", &avail[i]);
    }


    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];


    int pid, req[MAX];
    printf("Enter New Request Details -- \n");
    printf("Enter pid -- ");
    scanf("%d", &pid);
    printf("Enter Request for Resources -- ");
    for (i = 0; i < m; i++) {
        scanf("%d", &req[i]);
        if (req[i] > need[pid][i]) {
            printf("Request exceeds maximum claim.\n");
            return 0;
        }
        if (req[i] > avail[i]) {
            printf("Resources not available.\n");
            return 0;
        }
    }
```

```
    for (i = 0; i < m; i++) {
        avail[i] -= req[i];
        alloc[pid][i] += req[i];
        need[pid][i] -= req[i];
    }


    bool finish[MAX] = {false};
    int work[MAX];
    for (i = 0; i < m; i++) work[i] = avail[i];
    int count = 0, safeSeq[MAX];

    while (count < n) {
        bool found = false;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                for (j = 0; j < m; j++)
                    if (need[i][j] > work[j])
                        break;
                if (j == m) {
                    for (k = 0; k < m; k++)
                        work[k] += alloc[i][k];
                    safeSeq[count++] = i;
                    finish[i] = true;
                    found = true;

                    printf("P%d is visited( ", i);
                    for (k = 0; k < m; k++) printf("%d ", work[k]);
                    printf(")\n");
                }
            }
        }
        if (!found) break;
    }

    if (count < n) {
        printf("SYSTEM IS NOT IN SAFE STATE\n");
    } else {
        printf("SYSTEM IS IN SAFE STATE\n");
        printf("The Safe Sequence is -- ( ");
        for (i = 0; i < n; i++) {
            printf("P%d ", safeSeq[i]);
        }
        printf(")\n");
    }


    printf("\n%-10s %-15s %-15s %-15s\n", "Process", "Allocation", "Max", "Need");
    for (i = 0; i < n; i++) {
        printf("P%-9d ", i);
        for (j = 0; j < m; j++) printf("%d ", alloc[i][j]);
        printf("                ");
        for (j = 0; j < m; j++) printf("%d ", max[i][j]);
        printf("                ");
        for (j = 0; j < m; j++) printf("%d ", need[i][j]);
        printf("\n");
    }

    return 0;
}
```

```
Enter number of processes -- 5
Enter number of resources -- 3
Enter details for P0
Enter allocation -- 0
1
0
Enter Max -- 7
5
3
Enter details for P1
Enter allocation -- 2
0
0
Enter Max -- 3
2
2
Enter details for P2
Enter allocation -- 3
0
2
Enter Max -- 9
0
2
Enter details for P3
Enter allocation -- 2
1
1
Enter Max -- 2
2
2
Enter details for P4
Enter allocation -- 0
0
2
Enter Max -- 4
3
3
Enter Available Resources -- 3
3
2
Enter New Request Details --
Enter pid -- 1
Enter Request for Resources -- 1
0
2
P1 is visited( 5 3 2 )
P3 is visited( 7 4 3 )
P4 is visited( 7 4 5 )
P0 is visited( 7 5 5 )
P2 is visited( 10 5 7 )
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- ( P1 P3 P4 P0 P2 )

Process    Allocation      Max             Need
P0         0 1 0                7 5 3                7 4 3
P1         3 0 2                3 2 2                0 2 0
P2         3 0 2                9 0 2                6 0 0
P3         2 1 1                2 2 2                0 1 1
P4         0 0 2                4 3 3                4 3 1

Process returned 0 (0x0)   execution time : 62.313 s
Press any key to continue.
```

PROGRAM-6:

Write a C program to simulate the following contiguous memory allocation techniques.

d) Worst-fit

e) Best-fit

f) First-fit

```
#include <stdio.h>

#define N 10

void allocate(int blocks[], int m, int procs[], int n, char fit) {
    int alloc[N], b[N];
    for (int i = 0; i < m; i++) b[i] = blocks[i];
    for (int i = 0; i < n; i++) {
        int idx = -1;
        for (int j = 0; j < m; j++) {
            if (b[j] >= procs[i]) {
                if (fit == 'F') { idx = j; break; }
                if (fit == 'B' && (idx == -1 || b[j] < b[idx])) idx = j;
                if (fit == 'W' && (idx == -1 || b[j] > b[idx])) idx = j;
            }
        }
        alloc[i] = idx;
        if (idx != -1) b[idx] -= procs[i];
    }

    printf("\n%c-Fit Allocation:\n", fit);
    for (int i = 0; i < n; i++) {
        if (alloc[i] != -1)
            printf("Process %d -> Block %d\n", i + 1, alloc[i] + 1);
        else
            printf("Process %d -> Not Allocated\n", i + 1);
    }
}

int main() {
    int blocks[N], procs[N], m, n;
    printf("Enter number of blocks: "); scanf("%d", &m);
    printf("Enter block sizes: ");
    for (int i = 0; i < m; i++) scanf("%d", &blocks[i]);

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

    allocate(blocks, m, procs, n, 'F'); // First Fit
    allocate(blocks, m, procs, n, 'B'); // Best Fit
    allocate(blocks, m, procs, n, 'W'); // Worst Fit

    return 0;
}
```

```
Enter number of blocks: 5
Enter block sizes: 100
500
200
300
600
Enter number of processes: 4
Enter process sizes: 212
417
112
426

F-Fit Allocation:
Process 1 -> Block 2
Process 2 -> Block 5
Process 3 -> Block 2
Process 4 -> Not Allocated

B-Fit Allocation:
Process 1 -> Block 4
Process 2 -> Block 2
Process 3 -> Block 3
Process 4 -> Block 5

W-Fit Allocation:
Process 1 -> Block 5
Process 2 -> Block 2
Process 3 -> Block 5
Process 4 -> Not Allocated

Process returned 0 (0x0)   execution time : 38.983 s
Press any key to continue.
```

PROGRAM-7:

Write a C program to simulate page replacement algorithms.

a) FIFO

b) LRU

c) Optimal

```
#include <stdio.h>

void FIFO(int frames, int n, int reference[]) {
    int memory[frames];
    int pageFaults = 0, index = 0;

    for(int i = 0; i < frames; i++) memory[i] = -1;

    for(int i = 0; i < n; i++) {
        int found = 0;
        for(int j = 0; j < frames; j++) {
            if(memory[j] == reference[i]) {
                found = 1;
                break;
            }
        }

        if(!found) {
            memory[index] = reference[i];
            index = (index + 1) % frames;
            pageFaults++;
        }

        printf("PF No. %d: ", i + 1);
        for(int j = 0; j < frames; j++) {
            if(memory[j] != -1) printf("%d ", memory[j]);
        }
        printf("\n");
    }
    printf("FIFO Page Faults: %d\n", pageFaults);
}

void LRU(int frames, int n, int reference[]) {
    int memory[frames];
    int pageFaults = 0;
    int lastUsed[frames];

    for(int i = 0; i < frames; i++) memory[i] = -1;

    for(int i = 0; i < n; i++) {
        int found = 0;
        for(int j = 0; j < frames; j++) {
            if(memory[j] == reference[i]) {
                found = 1;
                lastUsed[j] = i;
                break;
            }
        }

        if(!found) {
            int lruIndex = 0;
            for(int j = 1; j < frames; j++) {
```

```
                if(lastUsed[j] < lastUsed[lruIndex]) lruIndex = j;
            }
            memory[lruIndex] = reference[i];
            lastUsed[lruIndex] = i;
            pageFaults++;
        }

        printf("PF No. %d: ", i + 1);
        for(int j = 0; j < frames; j++) {
            if(memory[j] != -1) printf("%d ", memory[j]);
        }
        printf("\n");
    }
    printf("LRU Page Faults: %d\n", pageFaults);
}

int findOptimal(int frames, int n, int reference[], int memory[]) {
    int farthest = -1, idx = -1;
    for(int i = 0; i < frames; i++) {
        int j;
        for(j = 0; j < n; j++) {
            if(memory[i] == reference[j]) break;
        }
        if(j == n) return i;

        if(j > farthest) {
            farthest = j;
            idx = i;
        }
    }
    return idx;
}

void Optimal(int frames, int n, int reference[]) {
    int memory[frames];
    int pageFaults = 0;

    for(int i = 0; i < frames; i++) memory[i] = -1;

    for(int i = 0; i < n; i++) {
        int found = 0;
        for(int j = 0; j < frames; j++) {
            if(memory[j] == reference[i]) {
                found = 1;
                break;
            }
        }

        if(!found) {
            int idx = findOptimal(frames, n, reference, memory);
            memory[idx] = reference[i];
            pageFaults++;
        }

        printf("PF No. %d: ", i + 1);
        for(int j = 0; j < frames; j++) {
            if(memory[j] != -1) printf("%d ", memory[j]);
        }
        printf("\n");
    }
    printf("Optimal Page Faults: %d\n", pageFaults);
}
```

```
int main() {
    int frames, n;

    printf("Enter the number of Frames: ");
    scanf("%d", &frames);
    printf("Enter the length of reference string: ");
    scanf("%d", &n);

    int reference[n];

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

    printf("FIFO Page Replacement Process:\n");
    FIFO(frames, n, reference);

    printf("LRU Page Replacement Process:\n");
    LRU(frames, n, reference);

    printf("Optimal Page Replacement Process:\n");
    Optimal(frames, n, reference);

    return 0;
}
```

```
Enter the number of Frames: 3
Enter the length of reference string: 12
Enter the reference string: 7
0
1
2
0
3
0
4
2
3
0
3
FIFO Page Replacement Process:
PF No. 1: 7
PF No. 2: 7 0
PF No. 3: 7 0 1
PF No. 4: 2 0 1
PF No. 5: 2 0 1
PF No. 6: 2 3 1
PF No. 7: 2 3 0
PF No. 8: 4 3 0
PF No. 9: 4 2 0
PF No. 10: 4 2 3
PF No. 11: 0 2 3
PF No. 12: 0 2 3
FIFO Page Faults: 10
LRU Page Replacement Process:
PF No. 1: 7
PF No. 2: 0 7
PF No. 3: 0 1
PF No. 4: 2 0 1
PF No. 5: 2 0 1
PF No. 6: 2 0 3
PF No. 7: 2 0 3
PF No. 8: 4 0 3
PF No. 9: 4 0 2
PF No. 10: 4 3 2
PF No. 11: 0 3 2
PF No. 12: 0 3 2
LRU Page Faults: 9
Optimal Page Replacement Process:
PF No. 1: 7
PF No. 2: 7 0
PF No. 3: 7 0 1
PF No. 4: 7 0 2
PF No. 5: 7 0 2
PF No. 6: 7 0 3
PF No. 7: 7 0 3
PF No. 8: 7 0 4
PF No. 9: 7 0 2
PF No. 10: 7 0 3
PF No. 11: 7 0 3
PF No. 12: 7 0 3
Optimal Page Faults: 8

Process returned 0 (0x0)   execution time : 19.400 s
Press any key to continue.
```

PROGRAM-8:

Write a C program to simulate the following file allocation strategies.

a) Sequential b) Indexed c) Linked

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

#define MAX_BLOCKS 100

int memory[MAX_BLOCKS];

void sequentialAllocation() {
    int start, length, i, flag = 0;
    scanf("%d %d", &start, &length);

    for (i = start; i < start + length; i++) {
        if (memory[i] == 1) {
            flag = 1;
            break;
        }
    }

    if (flag == 0) {
        for (i = start; i < start + length; i++) {
            memory[i] = 1;
        }
        printf("File allocated from block %d to %d\n", start, start + length - 1);
    } else {
        printf("Blocks already allocated. Cannot allocate file.\n");
    }
}

void linkedAllocation() {
    int blocks, i, index, next;
    int links[MAX_BLOCKS] = {0};
    scanf("%d", &blocks);

    for (i = 0; i < blocks; i++) {
        scanf("%d", &index);
        if (memory[index] == 1) {
            printf("Block %d already allocated. Aborting...\n", index);
            return;
        }
    }

    printf("File blocks linked as follows:\n");
    for (i = 0; i < blocks; i++) {
        scanf("%d", &index);
        memory[index] = 1;
        if (i < blocks - 1) {
            scanf("%d", &next);
            links[index] = next;
            printf("%d -> ", index);
        } else {
            printf("%d -> NULL\n", index);
        }
    }
}
```

```c
void indexedAllocation() {
    int indexBlock, blocks, i, blockNum;
    scanf("%d", &indexBlock);

    if (memory[indexBlock] == 1) {
        printf("Index block already allocated.\n");
        return;
    }

    scanf("%d", &blocks);

    int blockList[blocks];
    for (i = 0; i < blocks; i++) {
        scanf("%d", &blockNum);
        if (memory[blockNum] == 1 || blockNum == indexBlock) {
            printf("Block %d already allocated or is index block. Aborting...\n", blockNum);
            return;
        }
        blockList[i] = blockNum;
    }

    memory[indexBlock] = 1;
    for (i = 0; i < blocks; i++) {
        memory[blockList[i]] = 1;
    }

    printf("File indexed at block %d\n", indexBlock);
    printf("Blocks allocated:\nIndex [%d] -> ", indexBlock);
    for (i = 0; i < blocks; i++) {
        printf("%d ", blockList[i]);
    }
    printf("\n");
}

int main() {
    int choice;

    for (int i = 0; i < MAX_BLOCKS; i++) memory[i] = 0;

    do {
        printf("\nFile Allocation Strategies Menu:\n");
        printf("1. Sequential Allocation\n");
        printf("2. Linked Allocation\n");
        printf("3. Indexed Allocation\n");
        printf("4. Exit\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                sequentialAllocation();
                break;
            case 2:
                linkedAllocation();
                break;
            case 3:
                indexedAllocation();
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
```

```
            printf("Invalid choice! Try again\n");
        }
    } while (choice != 4);

    return 0;
}
```