

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

Operating Systems (23CS4PCOPS)

Submitted by:

ADITHYA RAVIKEERTHI (1BM22CS020)

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 **Adithya Ravikeerthi (1BM22CS020)**, 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 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (23CS4PCOPS)** work prescribed for the said degree.

Basavaraj Jakkalli
Associate Professor
Department of CSE
BMSCE, Bengaluru

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

Table Of Contents

Lab Program No.	Program Details	Page No.
1	FCFS AND SJF	3-6
2	PRIORITY AND ROUND ROBIN	6-12
3		13-16
4	RATE-MONOTONIC AND EARLIEST DEADLINE FIRST	17-23
5	PRODUCER-CONSUMER PROBLEM	24-26
6	DINERS-PHILOSOPHERS PROBLEM	26-29
7	BANKERS ALGORITHM(DEADLOCK AVOIDANCE)	30-32
8	DEADLOCK DETECTION	33-35
9	CONTIGIOUS MEMORY ALLOCATION(FIRST, BEST, WORST FIT)	36-39
10	PAGE REPLACEMENT(FIFO, LRU, OPTIMAL)	40-47
11	DISK SCHEDULING ALGORITHMS(FCFS, SCAN, C-SCAN)	48-53

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.

1. Experiments

1.1 Experiment - 1

1.1.1 Question:

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

(a) FCFS

Input:

```
#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++)
    {
        printf("\nAverage Turnaround Time: %.2f\n", avg_Turn_around_time);

        return 0;
    }
}

int SJF()
{
    //sorting
    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("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("\nEnter your choice:");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1: FCFS();
            break;

        case 2: SJF();
            break;

        default: printf("Invalid Input!!!");
    }
}

```

```

    return 0;
}

```

Output:

```

ArrivalTime.c -o FCFS_ArrivalTime } ; if ($?) { .\FCFS_ArrivalTime }
Enter the number of processes: 4
Enter the process ids:
1 2 3 4
Enter arrival time and burst time for process 1: 0 8
Enter arrival time and burst time for process 2: 1 4
Enter arrival time and burst time for process 3: 2 9
Enter arrival time and burst time for process 4: 3 5

Process Arrival Time    Burst Time    Waiting Time    Turnaround Time
1      0           8           0             8
2      1           4           7            11
3      2           9          10            19
4      3           5          18            23

Average Waiting Time: 8.75
Average Turnaround Time: 15.25
PS C:\Users\Wisarga Gond\OneDrive\Desktop\Wisarga\IV SEM\OS 4th sem\os lab>

```

(b).SJF

Input:

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

```
typedef struct {
```

```
    int pid;
```

```
    int burst_time;
```

```
    int waiting_time;
```

```
    int turnaround_time;
```

```
} Process;
```

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

```
    proc[0].waiting_time = 0; // First process has no waiting time
```

```
    for (int i = 1; i < n; i++) {
```

```
        proc[i].waiting_time = proc[i - 1].waiting_time + proc[i - 1].burst_time;
```

```
    }  
}
```

```
void find_turnaround_time(Process proc[], int n) {  
    for (int i = 0; i < n; i++) {  
        proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;  
    }  
}
```

```
void sort_by_burst_time(Process proc[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (proc[j].burst_time > proc[j + 1].burst_time) {  
                Process temp = proc[j];  
                proc[j] = proc[j + 1];  
                proc[j + 1] = temp;  
            }  
        }  
    }  
}
```

```
void find_avg_time(Process proc[], int n) {  
    int total_wt = 0, total_tat = 0;  
  
    find_waiting_time(proc, n);  
    find_turnaround_time(proc, n);  
  
    printf("Processes Burst Time Waiting Time Turnaround Time\n");  
    for (int i = 0; i < n; i++) {  
        total_wt += proc[i].waiting_time;  
        total_tat += proc[i].turnaround_time;
```



```

        printf("%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time,
proc[i].turnaround_time);
    }
    printf("\nAverage waiting time = %.2f", (float) total_wt / (float) n);
    printf("\nAverage turnaround time = %.2f\n", (float) total_tat / (float) n);
}

```

```

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process proc[n];
    for (int i = 0; i < n; i++) {
        printf("Enter burst time for process %d: ", i + 1);
        proc[i].pid = i + 1;
        scanf("%d", &proc[i].burst_time);
    }

    sort_by_burst_time(proc, n);
    find_avg_time(proc, n);

    return 0;
}

```

Output:

```

P.c -o SJF_NP } ; if ($?) { .\SJF_NP }
Enter the number of processes:
4
Enter the burst time of process 1:
8
Enter the burst time of process 2:
4
Enter the burst time of process 3:
9
Enter the burst time of process 4:
5
BurstTime    WaitingTime    TurnAroundtime
4.00         0.00         4.00
5.00         4.00         9.00
8.00         9.00         17.00
9.00         17.00        26.00
Average waiting time:7.500000
Average turn around time:14.000000

```

Experiment - 2

1.1.2 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)

Input:

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

```

typedef struct {
    int pid;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
    int remaining_time;
} Process;

```

```
void find_waiting_time_preemptive(Process proc[], int n) {
```

```

int complete = 0, t = 0, min_priority = 10000;
int shortest = 0, finish_time;
int check = 0;

while (complete != n) {
    for (int j = 0; j < n; j++) {
        if ((proc[j].remaining_time > 0) && (proc[j].priority < min_priority)) {
            min_priority = proc[j].priority;
            shortest = j;
            check = 1;
        }
    }

    if (check == 0) {
        t++;
        continue;
    }

    proc[shortest].remaining_time--;

    min_priority = proc[shortest].priority;
    if (proc[shortest].remaining_time == 0) {
        min_priority = 10000;
        complete++;
        finish_time = t + 1;

        proc[shortest].waiting_time = finish_time - proc[shortest].burst_time;
        proc[shortest].turnaround_time = finish_time;
    }
    t++;
}

void find_avg_time_preemptive(Process proc[], int n) {
    int total_wt = 0, total_tat = 0;

    find_waiting_time_preemptive(proc, n);

    printf("Processes Burst Time Waiting Time Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += proc[i].waiting_time;
        total_tat += proc[i].turnaround_time;
        printf("%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].burst_time,
        proc[i].waiting_time, proc[i].turnaround_time);
    }
}

```

```

printf("\nAverage waiting time = %.2f", (float) total_wt / (float) n);
printf("\nAverage turnaround time = %.2f\n", (float) total_tat / (float) n);
}

```

```

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process proc[n];
    for (int i = 0; i < n; i++) {
        printf("Enter burst time and priority for process %d: ", i + 1);
        proc[i].pid = i + 1;
        scanf("%d %d", &proc[i].burst_time, &proc[i].priority);
        proc[i].remaining_time = proc[i].burst_time;
    }
}

```

```

    find_avg_time_preemptive(proc, n);

```

```

    return 0;
}

```

```

#include <stdio.h>

```

```

typedef struct {
    int pid;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
    int remaining_time;
} Process;

```

```

void find_waiting_time_preemptive(Process proc[], int n) {
    int complete = 0, t = 0, min_priority = 10000;
    int shortest = 0, finish_time;
    int check = 0;

    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((proc[j].remaining_time > 0) && (proc[j].priority < min_priority)) {
                min_priority = proc[j].priority;
                shortest = j;
                check = 1;
            }
        }
    }
}

```

```

    if (check == 0) {
        t++;
        continue;
    }

    proc[shortest].remaining_time--;

    min_priority = proc[shortest].priority;
    if (proc[shortest].remaining_time == 0) {
        min_priority = 10000;
        complete++;
        finish_time = t + 1;

        proc[shortest].waiting_time = finish_time - proc[shortest].burst_time;
        proc[shortest].turnaround_time = finish_time;
    }
    t++;
}

}

void find_avg_time_preemptive(Process proc[], int n) {
    int total_wt = 0, total_tat = 0;

    find_waiting_time_preemptive(proc, n);

    printf("Processes Burst Time Waiting Time Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += proc[i].waiting_time;
        total_tat += proc[i].turnaround_time;
        printf("%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("\nAverage waiting time = %.2f", (float) total_wt / (float) n);
    printf("\nAverage turnaround time = %.2f\n", (float) total_tat / (float) n);
}

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process proc[n];
    for (int i = 0; i < n; i++) {
        printf("Enter burst time and priority for process %d: ", i + 1);
        proc[i].pid = i + 1;
        scanf("%d %d", &proc[i].burst_time, &proc[i].priority);
    }
}

```

```

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

find_avg_time_preemptive(proc, n);

return 0;
}

```

Output:

```

ity_nonPreemptive.c -o Priority_nonPreemptive } ; if ($?) { .\Priority_nonPreemptive }
Enter the number of processes:
5
Enter the process id:
1 2 3 4 5
Enter the arrival time of the processes:
0 1 2 3 4
Enter the burst time of the processes:
5 3 6 2 4
Enter the priority of processes:
3 2 1 4 5

```

Pid	ArrivalTime	BurstTime	Priority	TAT	WaitingTime
5	4	4	5	5	1
4	3	2	4	8	6
1	0	5	3	5	0
2	1	3	2	13	10
3	2	6	1	18	12

```

Average turn around time:9.8
Average waiting time:5.8
PS C:\Users\Wisarga Gond\OneDrive\Desktop\Wisarga\IV SEM\OS 4th sem\os lab>

```

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

Input:

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

```

typedef struct {
    int pid;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
} Process;

```

```

void find_waiting_time_rr(Process proc[], int n, int quantum) {
    int t = 0; // Current time
    int remain = n; // Number of processes remaining

    while (remain != 0) {
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0) {
                if (proc[i].remaining_time > quantum) {
                    t += quantum;

```

```

        proc[i].remaining_time -= quantum;
    } else {
        t += proc[i].remaining_time;
        proc[i].waiting_time = t - proc[i].burst_time;
        proc[i].remaining_time = 0;
        remain--;
    }
}
}
}
}

void find_turnaround_time_rr(Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;
    }
}

void find_avg_time_rr(Process proc[], int n, int quantum) {
    int total_wt = 0, total_tat = 0;

    find_waiting_time_rr(proc, n, quantum);
    find_turnaround_time_rr(proc, n);

    printf("Processes Burst Time Waiting Time Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += proc[i].waiting_time;
        total_tat += proc[i].turnaround_time;
        printf("%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("\nAverage waiting time = %.2f", (float) total_wt / (float) n);
    printf("\nAverage turnaround time = %.2f\n", (float) total_tat / (float) n);
}

int main() {
    int n, quantum;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process proc[n];
    for (int i = 0; i < n; i++) {
        printf("Enter burst time for process %d: ", i + 1);
        proc[i].pid = i + 1;
        scanf("%d", &proc[i].burst_time);
        proc[i].remaining_time = proc[i].burst_time;
    }
}

```

```
}
```

```
printf("Enter time quantum: ");  
scanf("%d", &quantum);
```

```
find_avg_time_rr(proc, n, quantum);
```

```
return 0;
```

```
}
```

Output:

```
Robin.c -o RoundRobin } ; if ($?) { .\RoundRobin }  
Enter the Number of Processes: 3
```

```
Enter the quantum time: 2
```

```
Enter the process: 1  
Enter the Burst Time:4
```

```
Enter the process: 2  
Enter the Burst Time:3
```

```
Enter the process: 3  
Enter the Burst Time:5
```

Processes	Burst Time	Waiting Time	turnaround time
1	4	4	8
2	3	6	9
3	5	7	12

```
Average waiting time = 5.666667  
Average turnaround time = 9.666667
```


1.2 Experiment - 3

1.2.1 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.

Input:

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

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

```
#define MAX_PROCESSES 100
```

```
typedef struct {
```

```
    int pid;
```

```
    int burst_time;
```

```
    int is_system_process;
```

```
} Process;
```

```
void sort_by_arrival_time(Process *processes, int n) {
```

```
    for (int i = 0; i < n - 1; i++) {
```

```
        for (int j = 0; j < n - 1 - i; j++) {
```

```
            if (processes[j].pid > processes[j + 1].pid) {
```

```
                Process temp = processes[j];
```

```
                processes[j] = processes[j + 1];
```

```
                processes[j + 1] = temp;
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```

void fcfs_scheduling(Process *queue, int n) {
    int time = 0;
    for (int i = 0; i < n; i++) {
        printf("Process %d starts at time %d and finishes at time %d.\n", queue[i].pid,
time, time + queue[i].burst_time);
        time += queue[i].burst_time;
    }
}

```

```

int main() {
    int n;
    Process processes[MAX_PROCESSES];
    Process system_queue[MAX_PROCESSES];
    Process user_queue[MAX_PROCESSES];
    int system_count = 0, user_count = 0;

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

    for (int i = 0; i < n; i++) {
        printf("Enter burst time and process type (1 for system, 0 for user) for process
%d: ", i + 1);
        processes[i].pid = i + 1;
        scanf("%d %d", &processes[i].burst_time, &processes[i].is_system_process);
        if (processes[i].is_system_process) {
            system_queue[system_count++] = processes[i];
        } else {
            user_queue[user_count++] = processes[i];
        }
    }
}

```

```

sort_by_arrival_time(system_queue, system_count);
sort_by_arrival_time(user_queue, user_count);

printf("\nSystem Processes Execution Order:\n");
fcfs_scheduling(system_queue, system_count);

printf("\nUser Processes Execution Order:\n");
fcfs_scheduling(user_queue, user_count);

return 0;
}

```

Output:

```

if ($?) { gcc multilevelqueue.c -o multilevelqueue }; if ($?) { .\multilevelqueue }
Enter the number of processes: 4
Enter arrival time, burst time, and priority (0-System/1-User) for process 1: 0 3 0
Enter arrival time, burst time, and priority (0-System/1-User) for process 2: 1 3 1
Enter arrival time, burst time, and priority (0-System/1-User) for process 3: 8 3 0
Enter arrival time, burst time, and priority (0-System/1-User) for process 4: 8 3 1
PID   Burst Time   Priority   Queue Type   Waiting Time   Turnaround Time
1     3           0         System       0             3
3     3           0         System       0             3
2     3           1         User         2             5
4     3           1         User         3             6
Average Waiting Time: 1.25
Average Turnaround Time: 4.25

```

1.3 Experiment - 4

1.3.1 Question:

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

(a) Rate- Monotonic

Input:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#
    for (int j = 0; j < cycles; j++)
    {
        if (process_list[j] == i + 1)
            printf("|####");
        else
            printf("|  ");
    }
    printf("\n");
}

void rate_monotonic(int time)
{
    int
        if ((i + 1) % period[k] == 0)
        {
            remain_time[k] = execution_time[k];
            next_process = k;
        }
    }
}
print_schedule(process_list, time);
}

void
{

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];
}

print_schedule(process_list, time);
}

int main()
{
    int option;

}
return 0;
}

```

Output:

```

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

Enter your choice: 1
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Period: 20

Process 2:
==> Execution time: 2
==> Period: 5

Process 3:
==> Execution time: 2
==> Period: 10

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: |   |   |   |   |####|   |   |####|####|   |   |   |   |   |   |   |   |   |   |
P[2]: |####|####|   |   |####|####|   |   |####|####|   |   |####|####|   |   |   |   |
P[3]: |   |   |####|####|   |   |   |   |   |   |   |   |####|####|   |   |   |   |

```

(b) Earliest Deadline First:

Input:

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

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

```
typedef struct {  
    int pid;  
    int burst_time;  
    int deadline;  
    int remaining_time;  
} Process;
```

```
void sort_by_deadline(Process *processes, int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - 1 - i; j++) {  
            if (processes[j].deadline > processes[j + 1].deadline) {  
                Process temp = processes[j];  
                processes[j] = processes[j + 1];  
                processes[j + 1] = temp;  
            }  
        }  
    }  
}
```

```
void EDF(Process *processes, int n) {  
    int time = 0, completed = 0;  
  
    while (completed < n) {  
        sort_by_deadline(processes, n);  
        for (int i = 0; i < n; i++) {  
            if (processes[i].remaining_time > 0) {  
                if (time + processes[i].remaining_time > processes[i].deadline) {  
                    printf("Process %d missed its deadline.\n", processes[i].pid);  
                }  
                time += processes[i].remaining_time;  
                processes[i].remaining_time = 0;  
                printf("Process %d completed at time %d.\n", processes[i].pid, time);  
                completed++;  
                break;  
            }  
        }  
    }  
}
```

```
int main() {
```

```

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

Process *processes = (Process *)malloc(n * sizeof(Process));

for (int i = 0; i < n; i++) {
    printf("Enter burst time and deadline for process %d: ", i + 1);
    scanf("%d %d", &processes[i].burst_time, &processes[i].deadline);
    processes[i].pid = i + 1;
    processes[i].remaining_time = processes[i].burst_time;
}

EDF(processes, n);

free(processes);
return 0;
}

```

Output:

```

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

Enter your choice: 2
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Deadline: 7

Process 2:
==> Execution time: 2
==> Deadline: 4

Process 3:
==> Execution time: 2
==> Deadline: 8

Scheduling:

Time:  | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
P[1]:  |   |   | ### | ### | ### |   |   |   |
P[2]:  | ### | ### |   |   |   |   |   | ### |
P[3]:  |   |   |   |   |   | ### | ### |   |

```

Experiment - 5

1.3.2 Question:

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

Input:

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

int *buffer;
int buffer_size;
int in = 0, out = 0;
sem_t empty, full, mutex;

void *producer(void *param) {
    int item, i;
    int num_items = *((int *) param);
    for (i = 0; i < num_items; i++) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        in = (in + 1) % buffer_size;
        printf("Produced: %d\n", item);
        sem_post(&mutex);
        sem_post(&full);
    }
    pthread_exit(0);
}

void *consumer(void *param) {
    int item, i;
    int num_items = *((int *) param);
    for (i = 0; i < num_items; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        out = (out + 1) % buffer_size;
        printf("Consumed: %d\n", item);
        sem_post(&mutex);
        sem_post(&empty);
    }
}
```



```

    pthread_exit(0);
}

int main() {
    int num_produce, num_consume;
    pthread_t prod_thread, cons_thread;

    printf("Enter buffer size: ");
    scanf("%d", &buffer_size);
    printf("Enter number of items to produce: ");
    scanf("%d", &num_produce);
    printf("Enter number of items to consume: ");
    scanf("%d", &num_consume);

    buffer = (int *)malloc(buffer_size * sizeof(int));
    sem_init(&empty, 0, buffer_size);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&prod_thread, NULL, producer, &num_produce);
    pthread_create(&cons_thread, NULL, consumer, &num_consume);

    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    free(buffer);
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
Output:

```

```

rs.c -o Bankers } ; if ($?) { .\Bankers }
Enter number of processes and number of resources required
5 3
Enter the max matrix for all process
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter number of allocated resources 5 for each process
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter number of available resources
3 3 2
Resources can be allocated to Process:2 and available resources are: 3 3 2
Resources can be allocated to Process:4 and available resources are: 5 3 2
Resources can be allocated to Process:5 and available resources are: 7 4 3
Resources can be allocated to Process:1 and available resources are: 7 4 5
Resources can be allocated to Process:3 and available resources are: 7 5 5

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

System is in safe mode
<P2 P4 P5 P1 P3 >

```

2.7 Experiment - 8

1.3.3 Question:

Write a C program to simulate deadlock detection.

(a) FCFS:

Input:

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

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

```
#define NUM_PROCESSES 5
```

```
#define NUM_RESOURCES 3
```

```
bool is_all_done(bool done[]) {  
    for (int i = 0; i < NUM_PROCESSES; i++) {  
        if (!done[i]) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
bool can_allocate(int need[], int available[]) {  
    for (int i = 0; i < NUM_RESOURCES; i++) {  
        if (need[i] > available[i]) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
int main() {  
    int available[NUM_RESOURCES];  
    int max[NUM_PROCESSES][NUM_RESOURCES];  
    int allocated[NUM_PROCESSES][NUM_RESOURCES];  
    int need[NUM_PROCESSES][NUM_RESOURCES];  
    bool done[NUM_PROCESSES] = {false, false, false, false, false};  
    bool deadlock = false;  
  
    printf("Enter the number of available resources for each type:\n");  
    for (int i = 0; i < NUM_RESOURCES; i++) {  
        scanf("%d", &available[i]);  
    }  
  
    printf("Enter the maximum resources needed by each process for each  
type:\n");
```

```

for (int i = 0; i < NUM_PROCESSES; i++) {
    for (int j = 0; j < NUM_RESOURCES; j++) {
        scanf("%d", &max[i][j]);
    }
}

printf("Enter the resources currently allocated to each process for each
type:\n");
for (int i = 0; i < NUM_PROCESSES; i++) {
    for (int j = 0; j < NUM_RESOURCES; j++) {
        scanf("%d", &allocated[i][j]);
    }
}

for (int i = 0; i < NUM_PROCESSES; i++) {
    for (int j = 0; j < NUM_RESOURCES; j++) {
        need[i][j] = max[i][j] - allocated[i][j];
    }
}

while (!is_all_done(done)) {
    bool allocated_in_this_pass = false;

    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (!done[i] && can_allocate(need[i], available)) {
            for (int j = 0; j < NUM_RESOURCES; j++) {
                available[j] += allocated[i][j];
            }
            done[i] = true;
            allocated_in_this_pass = true;
        }
    }

    if (!allocated_in_this_pass) {
        deadlock = true;
        break;
    }
}

if (deadlock) {
    printf("Deadlock detected.\n");
} else {
    printf("No deadlock detected.\n");
}

```

```
    return 0;
}
```

Output:

```
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Total head moment is 644
```

(b) SCAN:

Input:

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

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

```
#define NUM_PROCESSES 5
```

```
#define NUM_RESOURCES 3
```

```
bool is_all_done(bool done[]) {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (!done[i]) {
            return false;
        }
    }
    return true;
}
```

```
bool can_allocate(int need[], int available[]) {
    for (int i = 0; i < NUM_RESOURCES; i++) {
        if (need[i] > available[i]) {
            return false;
        }
    }
    return true;
}
```

```
int calculate_total_need(int need[]) {
    int total = 0;
    for (int i = 0; i < NUM_RESOURCES; i++) {
        total += need[i];
    }
    return total;
}
```

```

}

Voidsort_processes_by_need(int
need[NUM_PROCESSES][NUM_RESOURCES], int order[]) {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        order[i] = i;
    }

    for (int i = 0; i < NUM_PROCESSES - 1; i++) {
        for (int j = i + 1; j < NUM_PROCESSES; j++) {
            if (calculate_total_need(need[order[i]]) >
calculate_total_need(need[order[j]])) {
                int temp = order[i];
                order[i] = order[j];
                order[j] = temp;
            }
        }
    }
}

int main() {
    int available[NUM_RESOURCES];
    int max[NUM_PROCESSES][NUM_RESOURCES];
    int allocated[NUM_PROCESSES][NUM_RESOURCES];
    int need[NUM_PROCESSES][NUM_RESOURCES];
    bool done[NUM_PROCESSES] = {false, false, false, false, false};
    int order[NUM_PROCESSES];
    bool deadlock = false;

    printf("Enter the number of available resources for each type:\n");
    for (int i = 0; i < NUM_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }

    printf("Enter the maximum resources needed by each process for each
type:\n");
    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int j = 0; j < NUM_RESOURCES; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    printf("Enter the resources currently allocated to each process for each
type:\n");
    for (int i = 0; i < NUM_PROCESSES; i++) {

```

```

        for (int j = 0; j < NUM_RESOURCES; j++) {
            scanf("%d", &allocated[i][j]);
        }
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int j = 0; j < NUM_RESOURCES; j++) {
            need[i][j] = max[i][j] - allocated[i][j];
        }
    }

    sort_processes_by_need(need, order);

    while (!is_all_done(done)) {
        bool allocated_in_this_pass = false;

        for (int k = 0; k < NUM_PROCESSES; k++) {
            int i = order[k];
            if (!done[i] && can_allocate(need[i], available)) {
                for (int j = 0; j < NUM_RESOURCES; j++) {
                    available[j] += allocated[i][j];
                }
                done[i] = true;
                allocated_in_this_pass = true;
            }
        }

        if (!allocated_in_this_pass) {
            deadlock = true;
            break;
        }
    }

    if (deadlock) {
        printf("Deadlock detected.\n");
    } else {
        printf("No deadlock detected.\n");
    }

    return 0;
}

```

Output:

```
Enter the number of Requests
6
Enter the Requests sequence
90 120 30 60 50 80
Enter initial head position
70
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 190
```

(c) C-SCAN:

```
Enter the number of Requests
3
Enter the Requests sequence
2 1 0
Enter initial head position
1
Enter total disk size
3
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 4
```


Experiment – 9

Write a C program CONTIGIOUS MEMORY ALLOCATION (FIRST, BEST, WORST FIT)

Input:

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

#define MAX_BLOCKS 100

typedef struct {
    int block_id;
    int size;
    int allocated;
} MemoryBlock;

void display_blocks(MemoryBlock blocks[], int n) {
    printf("Block ID\tSize\tAllocated\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%s\n", blocks[i].block_id, blocks[i].size, blocks[i].allocated ? "Yes" : "No");
    }
    printf("\n");
}

void first_fit(MemoryBlock blocks[], int n, int process_size) {
    int i, j;
    for (i = 0; i < n; i++) {
        if (blocks[i].allocated == 0 && blocks[i].size >= process_size) {
            blocks[i].allocated = 1;
            printf("Process allocated to Block %d using First Fit\n", blocks[i].block_id);
            return;
        }
    }
    printf("Process cannot be allocated using First Fit\n");
}

void best_fit(MemoryBlock blocks[], int n, int process_size) {
    int best_index = -1;
    for (int i = 0; i < n; i++) {
        if (blocks[i].allocated == 0 && blocks[i].size >= process_size) {
            if (best_index == -1 || blocks[i].size < blocks[best_index].size) {
                best_index = i;
            }
        }
    }
    if (best_index != -1) {
        blocks[best_index].allocated = 1;
        printf("Process allocated to Block %d using Best Fit\n", blocks[best_index].block_id);
    } else {
        printf("Process cannot be allocated using Best Fit\n");
    }
}
```

```

void worst_fit(MemoryBlock blocks[], int n, int process_size) {
    int worst_index = -1;
    for (int i = 0; i < n; i++) {
        if (blocks[i].allocated == 0 && blocks[i].size >= process_size) {
            if (worst_index == -1 || blocks[i].size > blocks[worst_index].size) {
                worst_index = i;
            }
        }
    }
    if (worst_index != -1) {
        blocks[worst_index].allocated = 1;
        printf("Process allocated to Block %d using Worst Fit\n", blocks[worst_index].block_id);
    } else {
        printf("Process cannot be allocated using Worst Fit\n");
    }
}

```

```

int main() {
    int n_blocks, choice, process_size;
    MemoryBlock blocks[MAX_BLOCKS];

    printf("Enter number of memory blocks: ");
    scanf("%d", &n_blocks);

    for (int i = 0; i < n_blocks; i++) {
        printf("Enter size of block %d: ", i + 1);
        scanf("%d", &blocks[i].size);
        blocks[i].block_id = i + 1;
        blocks[i].allocated = 0; // 0 for not allocated, 1 for allocated
    }

```

```

    printf("\nEnter the size of the process to allocate: ");
    scanf("%d", &process_size);

```

```

    printf("\nMemory Blocks before allocation:\n");
    display_blocks(blocks, n_blocks);

```

```

    printf("\nChoose Memory Allocation Strategy:\n");
    printf("1. First Fit\n");
    printf("2. Best Fit\n");
    printf("3. Worst Fit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

```

```

    switch (choice) {
        case 1:
            first_fit(blocks, n_blocks, process_size);
            break;
        case 2:
            best_fit(blocks, n_blocks, process_size);
            break;

```

```

        case 3:
            worst_fit(blocks, n_blocks, process_size);
            break;
        default:
            printf("Invalid choice\n");
    }

    printf("\nMemory Blocks after allocation:\n");
    display_blocks(blocks, n_blocks);

    return 0;
}
Output:

```

Enter number of memory blocks: 5
 Enter size of block 1: 100
 Enter size of block 2: 200
 Enter size of block 3: 50
 Enter size of block 4: 300
 Enter size of block 5: 150

Enter the size of the process to allocate: 120

Memory Blocks before allocation:

Block ID	Size	Allocated
1	100	No
2	200	No
3	50	No
4	300	No
5	150	No

Choose Memory Allocation Strategy:

1. First Fit
2. Best Fit
3. Worst Fit

Enter your choice: 1

Process allocated to Block 2 using First Fit

Memory Blocks after allocation:

Block ID	Size	Allocated
1	100	No
2	200	Yes
3	50	No
4	300	No
5	150	No

Process allocated to Block 1 using Best Fit

Memory Blocks after allocation:

Block ID	Size	Allocated
1	100	Yes
2	200	Yes
3	50	No

4	300	No
5	150	No

Process allocated to Block 4 using Worst Fit

Memory Blocks after allocation:

Block ID	Size	Allocated
----------	------	-----------

1	100	No
2	200	No
3	50	No
4	300	Yes
5	150	No

Experiment – 10

Write a C-program PAGE REPLACEMENT (FIFO, LRU, OPTIMAL)

Input:

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

#define MAX_FRAMES 100
#define MAX_PAGES 100

int find_page(int page, int frames[], int n_frames) {
    for (int i = 0; i < n_frames; i++) {
        if (frames[i] == page) {
            return 1;
        }
    }
    return 0;
}

void fifo(int pages[], int n_pages, int n_frames) {
    int frames[MAX_FRAMES] = {0};
    int frame_index = 0;
    int page_faults = 0;
    for (int i = 0; i < n_pages; i++) {
        if (!find_page(pages[i], frames, n_frames)) {
            frames[frame_index] = pages[i];
            frame_index = (frame_index + 1) % n_frames;
            page_faults++;
        }
    }

    printf("FIFO Page Replacement Algorithm:\n");
    printf("Total Page Faults: %d\n\n", page_faults);
}

void lru(int pages[], int n_pages, int n_frames) {
    int frames[MAX_FRAMES] = {0};
    int recent[MAX_FRAMES] = {0};
    int frame_index = 0;

    int page_faults = 0;
    for (int i = 0; i < n_pages; i++) {
        int page = pages[i];
        int found = find_page(page, frames, n_frames);

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

```

        if (recent[j] < recent[replace_index]) {
            replace_index = j;
        }
    }
    frames[replace_index] = page;
    recent[replace_index] = i + 1;
    page_faults++;
} else {
    for (int j = 0; j < n_frames; j++) {
        if (frames[j] == page) {
            recent[j] = i + 1;
            break;
        }
    }
}
}

printf("LRU Page Replacement Algorithm:\n");
printf("Total Page Faults: %d\n\n", page_faults);
}

void optimal(int pages[], int n_pages, int n_frames) {
    int frames[MAX_FRAMES] = {0};
    int next_use[MAX_FRAMES] = {0};
    int page_faults = 0;

    for (int i = 0; i < n_pages; i++) {
        int page = pages[i];
        int found = find_page(page, frames, n_frames);

        if (!found) {
            page_faults++;
            int replace_index = 0;
            int farthest = i + 1;

            for (int j = 0; j < n_frames; j++) {
                int k;
                for (k = i + 1; k < n_pages; k++) {
                    if (pages[k] == frames[j]) {
                        break;
                    }
                }
                if (k == n_pages) {
                    replace_index = j;
                    break;
                }
                if (k > farthest) {
                    farthest = k;
                    replace_index = j;
                }
            }
        }
    }
}

```

```

        frames[replace_index] = page;
    }
}

printf("OPTIMAL Page Replacement Algorithm:\n");
printf("Total Page Faults: %d\n\n", page_faults);
}

int main() {
    int n_pages, n_frames;
    int pages[MAX_PAGES];

    printf("Enter number of pages: ");
    scanf("%d", &n_pages);

    printf("Enter page reference sequence:\n");
    for (int i = 0; i < n_pages; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter number of frames: ");
    scanf("%d", &n_frames);

    fifo(pages, n_pages, n_frames);
    lru(pages, n_pages, n_frames);
    optimal(pages, n_pages, n_frames);

    return 0;
}

```

Output:

```

C:\Users\Adithya\Desktop\adi>adi.exe
Enter number of pages: 12
Enter page reference sequence:
1 2 3 4 5 1 2 3 4 5 1 2
Enter number of frames: 3
FIFO Page Replacement Algorithm:
Total Page Faults: 12

LRU Page Replacement Algorithm:
Total Page Faults: 12

OPTIMAL Page Replacement Algorithm:
Total Page Faults: 8

```

Experiment – 11

Write a C-program DISK SCHEDULING ALGORITHMS (FCFS, SCAN, C-SCAN)

Input:

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

#define MAX_REQUESTS 1000

void sort_requests(int requests[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (requests[j] > requests[j + 1]) {
                int temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }
}

void fcfs(int requests[], int n, int start) {
    printf("FCFS Disk Scheduling Algorithm:\n");
    printf("Sequence of Head Movement:\n");
    printf("%d", start);
    for (int i = 0; i < n; i++) {
        printf(" -> %d", requests[i]);
    }
    printf("\n\n");
}

void scan(int requests[], int n, int start, int disk_size) {
    int total_head_movement = 0;
    int direction = 1;

    sort_requests(requests, n);

    int i, pos;
    for (i = 0; i < n; i++) {
        if (requests[i] >= start) {
            pos = i;
            break;
        }
    }

    printf("SCAN Disk Scheduling Algorithm:\n");
    printf("Sequence of Head Movement:\n");
    printf("%d", start);

    for (i = pos; i < n; i++) {
```



```

        total_head_movement += abs(requests[i] - start);
        start = requests[i];
        printf(" -> %d", start);
    }

    total_head_movement += abs(disk_size - 1 - start);
    start = disk_size - 1;
    printf(" -> %d", start);

    for (i = pos - 1; i >= 0; i--) {
        total_head_movement += abs(requests[i] - start);
        start = requests[i];
        printf(" -> %d", start);
    }

    printf("\nTotal Head Movement: %d\n\n", total_head_movement);
}

void c_scan(int requests[], int n, int start, int disk_size) {
    int total_head_movement = 0;
    sort_requests(requests, n);

    int i, pos;
    for (i = 0; i < n; i++) {
        if (requests[i] >= start) {
            pos = i;
            break;
        }
    }

    printf("C-SCAN Disk Scheduling Algorithm:\n");
    printf("Sequence of Head Movement:\n");
    printf("%d", start);

    // Scan right
    for (i = pos; i < n; i++) {
        total_head_movement += abs(requests[i] - start);
        start = requests[i];
        printf(" -> %d", start);
    }

    total_head_movement += abs(disk_size - 1 - start);
    start = 0;
    printf(" -> %d", start);

    for (i = 0; i < pos; i++) {
        total_head_movement += abs(requests[i] - start);
        start = requests[i];
        printf(" -> %d", start);
    }
}

```

```

    printf("\nTotal Head Movement: %d\n\n", total_head_movement);
}

int main() {
    int requests[MAX_REQUESTS];
    int n, start, disk_size;

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

    printf("Enter starting position of head: ");
    scanf("%d", &start);

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

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

    fcfs(requests, n, start);
    scan(requests, n, start, disk_size);
    c_scan(requests, n, start, disk_size);

    return 0;
}

```

Output:

```
C:\Users\Adithya\Desktop\adi>ad.exe
Enter number of requests: 7
Enter starting position of head: 50
Enter disk size: 200
Enter disk requests:
70
15
100
75
150
20
30
FCFS Disk Scheduling Algorithm:
Sequence of Head Movement:
50 -> 70 -> 15 -> 100 -> 75 -> 150 -> 20 -> 30

SCAN Disk Scheduling Algorithm:
Sequence of Head Movement:
50 -> 70 -> 75 -> 100 -> 150 -> 199 -> 30 -> 20 -> 15
Total Head Movement: 333

C-SCAN Disk Scheduling Algorithm:
Sequence of Head Movement:
50 -> 70 -> 75 -> 100 -> 150 -> 0 -> 15 -> 20 -> 30
Total Head Movement: 179
```