

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT
on
OPERATING SYSTEMS

Submitted by

YATHARTH SINGH (1WA23CS051)

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



CERTIFICATE

This is to certify that the Lab work entitled “OPERATING SYSTEMS – 23CS4PCOPS” carried out by Yatharth Singh (1WA23CS051), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. 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.

Dr Seema Patil
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

Sl. No.	Experiment Title	Page No.
1.	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & Non-preemptive)	1-14
2.	Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. → Priority (pre-emptive & Non-pre-emptive) →Round Robin (Experiment with different quantum sizes for RR algorithm)	14-28
3.	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.	28-34
4.	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling	34-40
5.	Write a C program to simulate producer-consumer problem using semaphores	41-45
6.	Write a C program to simulate the concept of Dining Philosophers problem.	45-50
7.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	50-55
8.	Write a C program to simulate deadlock detection	56-59
9.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	60-67

10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	68-73
-----	-------------------------------------------------------------------------------------------------	-------

Course Outcomes

C01	Apply the different concepts and functionalities of Operating System
C02	Analyse various Operating system strategies and techniques
C03	Demonstrate the different functionalities of Operating System.
C04	Conduct practical experiments to implement the functionalities of Operating system.

Program -1

Question:

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

Code:

```
#include <stdio.h>
void main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int arrival[n], burst[n], waiting[n], turnaround[n],
completion[n], response[n];
    printf("Enter Arrival Time and Burst Time for each
process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d %d", &arrival[i], &burst[i]);
    }
    int currentTime = 0;
    float totalWaiting = 0, totalTurnaround = 0;

printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\tResponse\n");
    for (int i = 0; i < n; i++) {
        if (currentTime < arrival[i])
            currentTime = arrival[i];
        completion[i] = currentTime + burst[i];
        turnaround[i] = completion[i] - arrival[i];
        waiting[i] = turnaround[i] - burst[i];
        response[i] = completion[i] - arrival[i];
        totalWaiting += waiting[i];
        totalTurnaround += turnaround[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, arrival[i],
burst[i], waiting[i], turnaround[i], response[i]);
        currentTime = completion[i];
    }
    printf("\nAverage Waiting Time: %.2f", totalWaiting / n);
```

```
    printf("\nAverage Turnaround Time: %.2f\n", totalTurnaround /  
n);  
}
```

Result:

```
▼ TERMINAL  
PS C:\Users\Admin\Documents\temp> cd "c:\Users\Admin\Documents\temp\" ; if ($?) { gcc fcfs.c -o fcfs } ; if ($?) { ./fcfs }  
Enter number of processes: 4  
Enter Arrival Time and Burst Time for each process:  
Process 1: 0  
7  
Process 2: 0  
3  
Process 3: 0  
4  
Process 4: 0  
6  


| Process | Arrival | Burst | Waiting | Turnaround | Response |
|---------|---------|-------|---------|------------|----------|
| 1       | 0       | 7     | 0       | 7          | 7        |
| 2       | 0       | 3     | 7       | 10         | 10       |
| 3       | 0       | 4     | 10      | 14         | 14       |
| 4       | 0       | 6     | 14      | 20         | 20       |

  
Average Waiting Time: 7.75  
Average Turnaround Time: 12.75
```

Write a C program to implement First Come First Serve algorithm (FCFS).

```
#include <stdio.h>
void main()
{
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int arrival[n], burst[n], waiting[n], turnaround[n], completion[n];
    printf("Enter arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d %d", &arrival[i], &burst[i]);
    }
    int currentTime = 0;
    float totalWaiting = 0, totalTurnaround = 0;
    printf("\nProcess \t Arrival \t Waiting \t Turnaround\n");
    for (int i = 0; i < n; i++) {
        if (currentTime < arrival[i])
            currentTime = arrival[i];
        completion[i] = currentTime + burst[i];
        turnaround[i] = completion[i] - arrival[i];
        waiting[i] = turnaround[i] - burst[i];
        response[i] = completion[i] - arrival[i];
        totalTurnaround += turnaround[i];
        totalWaiting += waiting[i];
    }
    printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1,
           arrival[i], burst[i], waiting[i], turnaround[i],
           completion[i]);
    currentTime = completion[i];
}
```

```

printf("Average Waiting Time: %f ", totalWaiting/4);
printf("Average Turnaround Time: %f ", totalTurnaround);
}

```

Output

Enter number of processes : 4

Enter arrival time and burst time for each process :

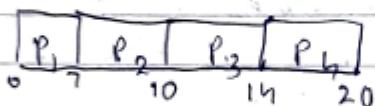
Process 1 : 0 7

Process 2 : 0 3

Process 3 : 0 4

Process 4 : 0 6

Process	AT	BT	WT	TAT	RT	CT
1	0	7	0	7	7	7
2	0	3	7	10	10	10
3	0	4	10	14	14	14
4	0	6	14	20	20	20



Average Waiting Time : 7.75

Average Turnaround Time : 12.75

B
61%
12%

=>SJF(Non-preemptive):

Code:

```
#include <stdio.h>

void nonPreemptiveSJF(int n, int at[], int bt[], int ct[], int
tat[], int wt[], int rt[])
{
    int completed = 0, time = 0, min_bt, shortest, finish_time;
    int remaining_bt[n];
    for (int i = 0; i < n; i++)
    {
        remaining_bt[i] = bt[i];
    }

    while (completed < n)
    {
        min_bt = 9999;
        shortest = -1;
        for (int i = 0; i < n; i++)
        {
            if (at[i] <= time && remaining_bt[i] > 0 && bt[i] <
min_bt)
            {
                min_bt = bt[i];
                shortest = i;
            }
        }
        if (shortest == -1)
        {
            time++;
            continue;
        }
        time += bt[shortest];
        remaining_bt[shortest] = 0;
```

```

        completed++;
        ct[shortest] = time;
        tat[shortest] = ct[shortest] - at[shortest];
        wt[shortest] = tat[shortest] - bt[shortest];
        rt[shortest] = wt[shortest];
    }
}

void displayTable(int n, int at[], int bt[], int ct[], int tat[],
int wt[], int rt[])
{
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i],
ct[i], tat[i], wt[i], rt[i]);
    }
}

int main()
{
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n], ct[n], tat[n], wt[n], rt[n];
    printf("Enter Arrival Time and Burst Time for each
process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Process %d - Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d - Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
    }
}

```

```

    nonPreemptiveSJF(n, at, bt, ct, tat, wt, rt);
    displayTable(n, at, bt, ct, tat, wt, rt);
    return 0;
}

```

Output:

```

Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
Process 1 - Arrival Time: 0
Process 1 - Burst Time: 7
Process 2 - Arrival Time: 8
Process 2 - Burst Time: 3
Process 3 - Arrival Time: 3
Process 3 - Burst Time: 4
Process 4 - Arrival Time: 5
Process 4 - Burst Time: 6

```

Process	AT	BT	CT	TAT	WT	RT
1	0	7	7	7	0	0
2	8	3	14	6	3	3
3	3	4	11	8	4	4
4	5	6	20	15	9	9

B. Write a C program to implement Shortest Job First (SJF) scheduling (Non-preemptive)

```
#include <cs3310.h>
void nonPreemptiveSJF (int n, int at[], int bt[], int ct[],
                      int tat[], int wt[], int rt[])
{
    int completed_id = 0, time = 0, min_bt, shortest, finish_time;
    int remaining_bt[n];
    for (int i = 0; i < n; i++)
    {
        remaining_bt[i] = bt[i];
    }
    while (completed_id < n)
    {
        min_bt = 9999;
        shortest = -1;
        for (int i = 0; i < n; i++)
        {
            if (at[i] <= time && remaining_bt[i] > 0
                && bt[i] < min_bt)
            {
                min_bt = bt[i];
                shortest = i;
            }
        }
        if (shortest == -1)
        {
            time++;
            continue;
        }
        completed_id++;
        ct[shortest] = time;
        bt[shortest] -= ct;
        shortest = i;
    }
}
```

```

time = bt[shortest];
remaining_bt[shortest] = 0;
completed++;
ct[shortest] = time;
bt[shortest] = bt[shortest] - ct[shortest];
wt[shortest] = bt[shortest] - bt[shortest];
rt[shortest] = wt[shortest];
}

```

```

void displayTable(int n, int at[], int bt[], int ct[],
int tat[], int wt[], int rt[])
{

```

```

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

```

Output: No of processes : 4

Process	AT	BT	CT	TAT	WT	RT
1	0	7	7	7	0	0
2	8	3	11	11	3	3
3	3	4	11	8	4	4
4	5	6	20	15	9	9

=> SJF (Preemptive):

Code

```

#include <stdio.h>

void preemptiveSJF(int n, int at[], int bt[], int ct[], int tat[],
int wt[], int rt[])
{
    int remaining_bt[n];
    int completed = 0, time = 0, min_bt, shortest;
    int flag[n];
    for (int i = 0; i < n; i++)
    {

```

```

remaining_bt[i] = bt[i];
flag[i] = 0;
}

while (completed < n)
{
    min_bt = 9999;
    shortest = -1;
    for (int i = 0; i < n; i++)
    {
        if (at[i] <= time && remaining_bt[i] > 0 &&
remaining_bt[i] < min_bt && flag[i] == 0)
        {
            min_bt = remaining_bt[i];
            shortest = i;
        }
    }
    if (shortest == -1)
    {
        time++;
        continue;
    }
    remaining_bt[shortest]--;
    if (remaining_bt[shortest] == 0)
    {
        completed++;
        flag[shortest] = 1;
        ct[shortest] = time + 1;
        tat[shortest] = ct[shortest] - at[shortest];
        wt[shortest] = tat[shortest] - bt[shortest];
        rt[shortest] = wt[shortest];
    }
    time++;
}

```

```

    }

}

void displayTable(int n, int at[], int bt[], int ct[], int tat[],
int wt[], int rt[])
{
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i],
ct[i], tat[i], wt[i], rt[i]);
    }
}

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

    int at[n], bt[n], ct[n], tat[n], wt[n], rt[n];
    printf("Enter Arrival Time and Burst Time for each
process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Process %d - Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d - Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
    }
    preemptiveSJF(n, at, bt, ct, tat, wt, rt);
    displayTable(n, at, bt, ct, tat, wt, rt);

    return 0;
}

```

}

Output:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
Process 1 - Arrival Time: 0
Process 1 - Burst Time: 8
Process 2 - Arrival Time: 1
Process 2 - Burst Time: 4
Process 3 - Arrival Time: 2
Process 3 - Burst Time: 9
Process 4 - Arrival Time: 3
Process 4 - Burst Time: 5
```

Process	AT	BT	CT	TAT	WT	RT
1	0	8	17	17	9	9
2	1	4	5	4	0	0
3	2	9	26	24	15	15
4	3	5	10	7	2	2

a) Preemptive SJF

```

void preemptiveSJF(int n, int at[], int bt[], int ct[], int gnt[],
                    int wt[], int rt[])
{
    int remaining_bt[n];
    int completed = 0, time = 0, min_bt, shortest;
    int flag[1];
    for (int i = 0; i < n; i++)
    {
        remaining_bt[i] = bt[i];
        Flag[i] = 0;
    }
    while (completed < n)
    {
        min_bt = 9999;
        shortest = -1;
        for (int i = 0; i < n; i++)
        {
            if (ct[i] <= time && remaining_bt[i] > 0 &&
                remaining_bt[i] < min_bt && flag[i] == 0)
            {
                min_bt = remaining_bt[i];
                shortest = i;
            }
        }
        if (shortest == -1)
        {
            time++;
            continue;
        }
        remaining_bt[shortest] -=;
    }
}

```

```

if (remaining->bt[shortest] == 0)
{
    completed++;
    flag[shortest] = 1;
    ct[shortest] = 1;
    bt[shortest] = ct[shortest] - at[shortest];
    wt[shortest] = ct[shortest] - bt[shortest];
    rt[shortest] = wt[shortest];
}
time++;
}

```

Output : No. of processes : 4

Process	AT	BT	CT	TAT	WT	RT
1	0	8	17	17	9	9
2	1	4	5	4	0	0
3	2	9	26	24	15	15
4	3	5	10	7	2	2

Program 2

Question

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

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

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

=> Priority Scheduling (Non-preemptive):

Code

```

#include <stdio.h>
//non-preemptive
void priorityScheduling(int n, int at[], int bt[], int pr[], int
ct[], int tat[], int wt[], int rt[]) {
    int completed = 0, time = 0, min_priority, highest_priority;
    int flag[n];
    for (int i = 0; i < n; i++) {
        flag[i] = 0;
    }
    while (completed < n) {
        min_priority = 9999;
        highest_priority = -1;

```

```

        for (int i = 0; i < n; i++) {
            if (at[i] <= time && flag[i] == 0 && pr[i] <
min_priority) {
                min_priority = pr[i];
                highest_priority = i;
            }
        }
        if (highest_priority == -1) {
            time++;
            continue;
        }
        time += bt[highest_priority];
        flag[highest_priority] = 1;
        ct[highest_priority] = time;
        tat[highest_priority] = ct[highest_priority] -
at[highest_priority];
        wt[highest_priority] = tat[highest_priority] -
bt[highest_priority];
        rt[highest_priority] = wt[highest_priority];
        completed++;
    }
}

void displayTable(int n, int at[], int bt[], int pr[], int ct[],
int tat[], int wt[], int rt[]) {
    printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i],
bt[i], pr[i], ct[i], tat[i], wt[i], rt[i]);
    }
}

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n], pr[n], ct[n], tat[n], wt[n], rt[n];
    printf("Enter Arrival Time, Burst Time, and Priority for each
process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d - Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d - Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Process %d - Priority: ", i + 1);
        scanf("%d", &pr[i]);
    }
}

```

```

    }
    priorityScheduling(n, at, bt, pr, ct, tat, wt, rt);
    displayTable(n, at, bt, pr, ct, tat, wt, rt);
    return 0;
}

```

Output:

```

Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1 - Arrival Time: 0
Process 1 - Burst Time: 4
Process 1 - Priority: 2
Process 2 - Arrival Time: 0
Process 2 - Burst Time: 10
Process 2 - Priority: 1
Process 3 - Arrival Time: 0
Process 3 - Burst Time: 3
Process 3 - Priority: 3
Process 4 - Arrival Time: 0
Process 4 - Burst Time: 12
Process 4 - Priority: 4

```

Process	AT	BT	Priority	CT	TAT	WT	RT
1	0	4	2	14	14	10	10
2	0	10	1	10	10	0	0
3	0	3	3	17	17	14	14
4	0	12	4	29	29	17	17

Q. Priority Scheduling algorithm

```
#include <Stdio.h>
```

```
void priorityScheduling (int n, int at[7], int bt[7], int pr[7],
int et[7], int tat[7], int wt[7], int ct[7]) {
    int completed = 0, time = 0, minPriority, highestPriority;
    int flag[7];
    for (int i = 0; i < n; i++) {
        flag[i] = 0;
    }
    while (completed < n) {
        minPriority = 9999;
        highestPriority = -1;
        for (int i = 0; i < n; i++) {
            if (at[i] <= time && flag[i] == 0) {
                if (pr[i] <= minPriority) {
                    minPriority = pr[i];
                    highestPriority = i;
                }
            }
        }
        if (highestPriority == -1) {
            time++;
            continue;
        }
        time += bt[highestPriority];
        flag[highestPriority] = 1;
        ct[highestPriority] = ct[highestPriority] + at[highestPriority];
        wt[highestPriority] = wt[highestPriority] + bt[highestPriority];
        bt[highestPriority] = 0;
        completed++;
    }
}
```

```

void displayTable(int n, int at[], int pr[], int bt[],
    int tat[], int wt[], int rt[]){
    printf("Unprocess NBT BTBT AtPriority Tat
        & TAT & WT & RT \n");
    for (int i = 0; i < n; i++) {
        printf("%d %d %d %d %d %d %d %d %d\n",
            i + 1, at[i], bt[i], pr[i], rt[i], tat[i], wt[i], n);
    }
}

```

```

int main(){
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n], pr[n], rt[n], tat[n], wt[n], n;
    n = 0;
}

```

```

int main(){
    int n;
    printf("Enter Arrival Time, Burst time and Priority for
        each process: \n");
    for (int i = 0; i < n; i++) {
        printf("Process %d - Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d - Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Process %d - priority: ", i + 1);
        scanf("%d", &pr[i]);
    }
    priorityScheduling(n, at, bt, pr, rt, tat, wt, n);
    displayTable(n, at, pr, rt, bt, wt, rt), return 0;
}

```

Output

Enter number of processes : 4

Enter AT, BT, and Priority for each process

Process 1 : AT : 0

BT : 4

P1 Priority : 2

P2 : AT : 0

BT : 10

P : 1

P3 : AT : 0

BT : 3

P : 3

P4 : AT : 0

BT : 12

P : 4

Process	AT	BT	Priority	CT	TAT	WT	RT
1	0	4	2	14	14	10	10
2	0	10	1	10	10	0	0
3	0	3	3	17	17	14	14
4	0	12	4	29	29	17	17

=> Priority Scheduling (Preemptive):

Code

```
#include <stdio.h>

struct Process {
    int id, arrivalTime, burstTime, remainingTime, priority;
    int waitingTime, turnaroundTime, completionTime;
};

int findHighestPriority(struct Process p[], int n, int currentTime)
{
    int highest = -1;
    int highestPriority = 1e9;

    for (int i = 0; i < n; i++) {
        if (p[i].arrivalTime <= currentTime && p[i].remainingTime > 0) {
            if (p[i].priority < highestPriority) {
                highestPriority = p[i].priority;
                highest = i;
            }
        }
    }
}
```

```

    }
    return highest;
}

void priorityScheduling(struct Process p[], int n) {
    int currentTime = 0, completed = 0;
    float totalWaitingTime = 0, totalTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        p[i].remainingTime = p[i].burstTime;
    }
    while (completed < n) {
        int idx = findHighestPriority(p, n, currentTime);

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

        p[idx].remainingTime--;
        currentTime++;
        if (p[idx].remainingTime == 0) {
            completed++;
            p[idx].completionTime = currentTime;
            p[idx].turnaroundTime = p[idx].completionTime -
p[idx].arrivalTime;
            p[idx].waitingTime = p[idx].turnaroundTime -
p[idx].burstTime;

            totalWaitingTime += p[idx].waitingTime;
            totalTurnaroundTime += p[idx].turnaroundTime;
        }
    }

    printf("\nProcess\tArrival\tBurst\tPriority\tCompletion\tTurnaround
\tWaiting\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id,
p[i].arrivalTime, p[i].burstTime,
                p[i].priority, p[i].completionTime,
p[i].turnaroundTime, p[i].waitingTime);
    }

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

```

```

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];

    printf("Enter Arrival Time, Burst Time, and Priority (lower
number = higher priority) for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("Process %d: ", p[i].id);
        scanf("%d %d %d", &p[i].arrivalTime, &p[i].burstTime,
&p[i].priority);
    }
    priorityScheduling(p, n);
    return 0;
}

```

Output:

```

Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority (lower number = higher priority) for each process:
Process 1: 0
5
2
Process 2: 0
3
1
Process 3: 0
8
3
Process 4: 0
2
4

  Process   Arrival   Burst   Priority   Completion   Turnaround   Waiting
  1         0          5        2           8            8            3
  2         0          3        1           3            3            0
  3         0          8        3          16           16            8
  4         0          2        4          18           18           16

Average Waiting Time: 6.75
Average Turnaround Time: 11.25

```

```

        continue;
    }

p[1].remainingTime -=;
currentTime +=;

p[1].completionTime = currentTime;
p[1].turnaroundTime = p[1].completionTime -
    p[1].arrivalTime;
p[1].waitingTime = p[1].completionTime - p[1].arrivalTime;
p[1].waitingTime = p[1].turnaroundTime - p[1].arrivalTime;

totalWaitingTime += p[1].waitingTime;
totalTurnaroundTime += p[1].turnaroundTime;
}

printf("\nProcess \tArrival \tBurst \tPriority \tCompletion
\tTurnaround \tWaiting\n"),
for (int i = 0; i < n; i++) {
    printf("%d \t%d \t%d \t%d \t%d \t%d \n", p[i].id,
        p[i].arrivalTime, p[i].burstTime, p[i].priority,
        p[i].completionTime, p[i].turnaroundTime, p[i].
        waitingTime);
}

printf("\nAverage waiting time : %.2f ", totalWaitingTime /
    n);
printf("\nAverage Turnaround time : %.2f\n");
}

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    printf("Enter Arrival Time, Burst Time and Priority

```

(lower number = higher priority) for each process:

```

for (int i = 0; i < n; i++) {
    pCi3.id = i + 1;
    printf("process %d : ", pCi3.id);
    scanf("%d", &pCi3.arrivalTime, &pCi3.burstTime,
          &pCi3.priority);
}
priorityScheduling(p, n);
return 0;
}

```

Output:

	A1	B1	Priorty	Completion	Turnaround	WT
1	0	5	2	8	8	3
2	0	3	1	3	3	0
3	0	8	3	16	16	8
4	0	2	4	18	18	16

=> Round Robin:

Code

```

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[],
int quantum) {

    int rem_bt[n];
    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        wt[i] = 0;
        wt++;
    }
    int t = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {

```

```

        if (rem_bt[i] > 0) {
            done = 0;
            if (rem_bt[i] > quantum) {
                rem_bt[i] -= quantum;
                //++quantum;
                t += quantum;
            } else {
                t += rem_bt[i];
                wt[i] = t - bt[i];
                rem_bt[i] = 0;
            }
        }
        if (done) break;
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findAvgTime(int processes[], int n, int bt[], int quantum) {
    int wt[n], tat[n];
    findWaitingTime(processes, n, bt, wt, quantum);
    findTurnAroundTime(processes, n, bt, wt, tat);

    int total_wt = 0, total_tat = 0;
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
    }
}

```

```

        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i],
tat[i]);
    }
    printf("\nAverage Waiting Time: %.2f", (float)total_wt / n);
    printf("\nAverage Turnaround Time: %.2f\n", (float)total_tat /
n);
}

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

```

Output:

```
Enter number of processes: 4
Enter burst time for process 1: 10
Enter burst time for process 2: 5
Enter burst time for process 3: 7
Enter burst time for process 4: 3
Enter time quantum: 4

Process Burst Time Waiting Time Turnaround Time
1      10        15          25
2      5         15          20
3      7         16          23
4      3         12          15

Average Waiting Time: 14.50
Average Turnaround Time: 20.75
```

Round Robin:

```
#include <stdio.h>
void findWaitingTime(int processes[], int n, int bt[],
                     int wt[], int quantum) {
    int rem_bt[n];
    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        wt[i] = 0;
    }
    int t = 0;
    while (1) {
        int done = 0;
        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                done = 1;
                if (rem_bt[i] > quantum) {
                    rem_bt[i] -= quantum;
                    t += quantum;
                } else {
                    t += rem_bt[i];
                    rem_bt[i] = 0;
                }
            }
        }
        if (done) break;
    }
}

void findTurnaroundTime(int processes[], int n,
                        int bt[], int wt[], int tt[]) {
    for (int i = 0; i < n; i++) {
        tt[i] = bt[i] + wt[i];
    }
}
```

```

void find Avg Time (int processes[], int n, int b[],
                    int quantum) {
    int wt[4], tat[4];
    find Waiting Time (processes, n, b, quantum);
    find Turn Around Time (processes, n, b, wt, tat);

    int total_WT = 0, total_TAT = 0;
    printf("\n Process \t Burst Time \t Waiting Time \t Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_WT += wt[i];
        total_TAT = tat[i];
        printf("%d\t%d\t%d\t%d\n", processes[i], b[i], wt[i], tat[i]);
    }
    printf("\n Average Waiting Time : %.2f", (float) total_WT / n);
    printf("\n Average Turnaround Time : %.2f", (float) total_TAT / n);
}

```

Output : Some quantum 4

Process	B	WT	TAT
1	10	15	25
2	5	13	20
3	7	16	23
4	3	12	15

Average WT 14.50

Average TAT 20.75

Program 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.

=> Multilevel queue Scheduling

Code

```
#include <stdio.h>
#define TIME_QUANTUM 2
typedef struct {
    int pid, burst_time, arrival_time, queue;
    int waiting_time, turnaround_time, response_time,
remaining_time;
} Process;
void sort_by_arrival(Process p[], int n) {
    Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].arrival_time > p[j].arrival_time) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}
void round_robin(Process p[], int n, int *time) {
    int done, i;
    do {
        done = 1;
        for (i = 0; i < n; i++) {
            if (p[i].remaining_time > 0) {
                done = 0;
                if (p[i].remaining_time > TIME_QUANTUM) {
                    *time += TIME_QUANTUM;
                    p[i].remaining_time -= TIME_QUANTUM;
                } else {
                    *time += p[i].remaining_time;
                    p[i].waiting_time = *time - p[i].arrival_time -
p[i].burst_time;
                    p[i].turnaround_time = p[i].waiting_time +
p[i].burst_time;
                    p[i].response_time = p[i].waiting_time;
                    p[i].remaining_time = 0;
                }
            }
        }
    } while (!done);
}
void fcfs(Process p[], int n, int *time) {
    for (int i = 0; i < n; i++) {
        if (*time < p[i].arrival_time)
```

```

        *time = p[i].arrival_time;
        p[i].waiting_time = *time - p[i].arrival_time;
        p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
        p[i].response_time = p[i].waiting_time;
        *time += p[i].burst_time;
    }
}

int main() {
    int n, i, time = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    Process p[n], system_processes[n], user_processes[n];
    int sys_count = 0, user_count = 0;

    for (i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ",
i + 1);
        p[i].pid = i + 1;
        scanf("%d %d %d", &p[i].burst_time, &p[i].arrival_time,
&p[i].queue);
        p[i].remaining_time = p[i].burst_time;

        if (p[i].queue == 0)
            system_processes[sys_count++] = p[i];
        else
            user_processes[user_count++] = p[i];
    }
    sort_by_arrival(system_processes, sys_count);
    sort_by_arrival(user_processes, user_count);
    printf("\nQueue 1 is System Process\nQueue 2 is User
Process\n");
    round_robin(system_processes, sys_count, &time);
    fcfs(user_processes, user_count, &time);
    Process final_list[n];
    int index = 0;
    for (i = 0; i < sys_count; i++)
        final_list[index++] = system_processes[i];
    for (i = 0; i < user_count; i++)
        final_list[index++] = user_processes[i];

    printf("\nProcess\tWaiting Time\tTurn Around Time\tResponse
Time\n");
    float avg_wt = 0, avg_tat = 0, avg_rt = 0;

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

```

```

        printf("%d\t%d\t\t%d\t\t\t%d\n", final_list[i].pid,
final_list[i].waiting_time, final_list[i].turnaround_time,
final_list[i].response_time);
        avg_wt += final_list[i].waiting_time;
        avg_tat += final_list[i].turnaround_time;
        avg_rt += final_list[i].response_time;
    }
    avg_wt /= n;
    avg_tat /= n;
    avg_rt /= n;
    float throughput = (float)n / time;
    printf("\nAverage Waiting Time: %.2f", avg_wt);
    printf("\nAverage Turn Around Time: %.2f", avg_tat);
    printf("\nAverage Response Time: %.2f", avg_rt);
    printf("\nThroughput: %.2f", throughput);
    return 0;
}

```

Output:

```

Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 2
0
1
Enter Burst Time, Arrival Time and Queue of P2: 1
0
2
Enter Burst Time, Arrival Time and Queue of P3: 5
0

1
Enter Burst Time, Arrival Time and Queue of P4: 3
0

2

Queue 1 is System Process
Queue 2 is User Process

Process Waiting Time      Turn Around Time      Response Time
1      0                  2                      0
2      2                  3                      2
3      3                  8                      3
4      8                  11                     8

Average Waiting Time: 3.25
Average Turn Around Time: 6.00
Average Response Time: 3.25
Throughput: 0.36

```

6. Multilevel Queue Scheduling

```
#include <STDIO.H>
#define TIME_QUANTUM 2
typedef struct {
    int pid, burst_time, arrival_time, queue;
    int waiting_time, turnaround_time, response_time,
        remaining_time;
} Process;

void sort_by_arrival(Process p[6], int n) {
    Process temp;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].arrival_time > p[j].arrival_time) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}

void roundRobin (Process p[6], int n, int *time) {
    int done = 0;
    do {
        done = 1;
        for (int i = 0; i < n; i++) {
            if (p[i].remaining_time > 0) {
                done = 0;
                if (p[i].remaining_time > TIME_QUANTUM) {
                    time += TIME_QUANTUM;
                    p[i].waiting_time = 'time - p[i].arrival_time';
                    p[i].turnaround_time
                    p[i].remaining_time = 0;
                }
            }
        }
    } while (!done);
}
```

```

? while (!done) {
}

void fcfs(Proc* p[], int n, int *time) {
    for (int i = 0; i < n; i++) {
        if (C[i].time < p[i].arrival_time)
            time[i] = p[i].arrival_time;
        else
            time[i] = C[i].waiting_time + p[i].burst_time;
        C[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
        C[i].response_time = C[i].waiting_time;
        time[i] += p[i].burst_time;
    }
}

int main() {
    int n, i, time[10];
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Proc p[10], system_processes[10], user_processes[10];
    int sys_count = 0, user_count = 0;
    for (i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and\n");
        printf("Quue of P%d: ", i + 1);
        p[i].remaining_time = p[i].burst_time;
    }
    return 0;
}

```

Output : Enter number of processes : 4
 Enter BT, AT and Quue of P1 : 2 0 1
 Enter BT, AT and queue of P2 : 1 0 2
 Enter BT, AT and queue of P3 : 5 0 1
 Enter BT, AT and queue of P4 : 3 0 2

Queue 1 is system process

Queue 2 is user Process

Process	Arrival Time	Service Time	Response Time
1	0	2	0
2	2	3	2
3	3	8	3
4	8	11	8

Average WT = 3.25

Average TAT 6.00

Average RT : 3.25

Throughput : 0.36

Program 4

Question

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

- > Rate- Monotonic
- > Earliest-deadline First
- > Proportional scheduling

=> Rate Monotonic Scheduling

Code

```

#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int id;
    int period;
    int execution_time;
    int next_deadline;
    int executed;
} Task;

int compare_tasks(const void *a, const void *b) {

```

```

        return ((Task *)a)->period - ((Task *)b)->period;
    }

void rate_monotonic_scheduling(Task tasks[], int num_tasks, int
total_time) {
    qsort(tasks, num_tasks, sizeof(Task), compare_tasks);

    /*
    for(int i = 0; i < num_tasks; i++)
        printf("Task %d: %d %d\n", tasks[i].id,
tasks[i].execution_time, tasks[i].period);
    */

    for (int i = 0; i < num_tasks; i++)
        tasks[i].next_deadline = tasks[i].period;
    printf("Time\t");
    for (int i = 0; i < num_tasks; i++)
        printf("Task %d\t", tasks[i].id);
    printf("\n");
    for (int current_time = 0; current_time < total_time;
current_time++)
    {
        printf("%d\t", current_time);
        int executed_task = -1;

        for (int i = 0; i < num_tasks; i++)
        {
            if (current_time % tasks[i].period == 0)
            {
                tasks[i].next_deadline = current_time +
tasks[i].period;
                tasks[i].executed = 0;
            }
            if (current_time < tasks[i].next_deadline)
            {
                if(tasks[i].executed < tasks[i].execution_time)
                {
                    executed_task = i;
                    tasks[i].executed++;
                    break;
                }
            }
        }
        if (executed_task != -1)
        {
            for (int i = 0; i < num_tasks; i++)

```

```

        {
            if (i == executed_task) {
                printf("Exec\t");
            } else {
                printf("\t");
            }
        }
    } else {
        for (int i = 0; i < num_tasks; i++) {
            printf("\t");
        }
    }
    printf("\n");
}
}

int main() {
    Task tasks[] = {
        {1, 20, 3},
        {2, 5, 2},
        {3, 10, 2}
    };

    int num_tasks = sizeof(tasks) / sizeof(tasks[0]);
    int total_time = 20;

    rate_monotonic_scheduling(tasks, num_tasks, total_time);
    return 0;
}

```

Output:

Time	Task 2	Task 3	Task 1
0	Exec		
1	Exec		
2		Exec	
3		Exec	
4			Exec
5	Exec		
6	Exec		
7			Exec
8			Exec
9			
10	Exec		
11	Exec		
12			
13			
14			
15	Exec		
16	Exec		
17			
18			
19			

Rate monotonic scheduling

```
#include <stdio.h>
struct Task {
    int id, executionTime, period;
};

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

int main() {
    int n, time = 20;
    printf("Enter number of tasks: ");
    scanf("%d", &n);
    struct Task tasks[n];
    for (int i = 0; i < n; i++) {
        printf("Task %d: execution time and\n"
               "period: ", i + 1);
        scanf("%d %d", &tasks[i].executionTime,
              &tasks[i].period);
        tasks[i].id = i + 1;
    }
    sortTasks(tasks, n);
    printf("RMS: \n");
    for (int t = 0; t < time; t++) {
        int executed = 0;
        for (int i = 0; i < n; i++) {
            if (t % tasks[i].period < tasks[i].executionTime)
                executed++;
        }
        printf("%d ", executed);
    }
}
```

```

2 printf("Time %d : Task %d\n", t, tasks[i].id);
3     executed = 1;
4     break;
5
6 if (!executed)
7     printf("Time %d : Idle\n", t);
8
9 return 0;

```

Output	Time	Task 2	Task 3	Task 1
0		Exe		
1		Free		
2			Free	
3			Free	
4				Free
5		Free		
6		Free		
7			Free	
8			Free	
9				
10		Free		
11		Free		
12				
13				
14				
15		Free		
16		Free		
17				
18				
19				

=> Earliest Deadline First

Code

```

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

typedef struct {
    int id;
    int period;
    int execution_time;
    int deadline;
}

```

```

        int executed;
    } Task;

int compare_tasks(const void *a, const void *b) {
    return ((Task *)a)->deadline - ((Task *)b)->deadline;
}

void earliest_deadline_first_scheduling(Task tasks[], int num_tasks, int total_time) {
    printf("Time\t");
    for (int i = 0; i < num_tasks; i++)
        printf("Task %d\t", tasks[i].id);
    printf("\n");
    for (int current_time = 0; current_time < total_time;
current_time++) {
        printf("%d\t", current_time);
        int executed_task = -1;

        for (int i = 0; i < num_tasks; i++) {
            if (current_time % tasks[i].period == 0) {
                tasks[i].deadline = current_time + tasks[i].period;
                tasks[i].executed = 0;
            }
        }
        qsort(tasks, num_tasks, sizeof(Task), compare_tasks);

        for (int i = 0; i < num_tasks; i++) {
            if (current_time < tasks[i].deadline &&
tasks[i].executed < tasks[i].execution_time) {
                executed_task = i;
                tasks[i].executed++;
                break;
            }
        }
        if (executed_task != -1) {
            for (int i = 0; i < num_tasks; i++) {
                if (i == executed_task) {
                    printf("Exec\t");
                } else {
                    printf("\t");
                }
            }
        } else {
            for (int i = 0; i < num_tasks; i++) {
                printf("\t");
            }
        }
    }
}

```

```

        }
        printf("\n");
    }
}

int main() {
    Task tasks[] = {
        {1, 20, 3, 20, 0},
        {2, 5, 2, 5, 0},
        {3, 10, 2, 10, 0}
//task
    };

    int num_tasks = sizeof(tasks) / sizeof(tasks[0]);
    int total_time = 20;

    earliest_deadline_first_scheduling(tasks, num_tasks,
total_time);
    return 0;
}

```

Output:

Time	Task 1	Task 2	Task 3
0	Exec		
1	Exec		
2		Exec	
3		Exec	
4			Exec
5			Exec
6	Exec		
7			Exec
8			Exec
9			
10	Exec		
11	Exec		
12			Exec
13		Exec	
14			
15			Exec
16			Exec
17			
18			
19			

Program 5

Question

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

=> Producer Consumer

Code

```
#include <stdio.h>

int x = 1, mutex = 1, full = 0, empty = 3;
void wait(int *S)
{
    (*S)--;
}

void signal(int *S)
{
    (*S)++;
}

void producer()
{
    wait(&mutex);
    if (empty > 0)
    {
        wait(&empty);
        signal(&full);
        printf("Item produced: %d\n", x++);
    } else {
        printf("Buffer is Full\n");
    }
    signal(&mutex);
}

void consumer() {
    wait(&mutex);
    if (full > 0) {
        wait(&full);
        signal(&empty);
        printf("Item Consumed: %d\n", --x);
    } else {
        printf("Buffer is Empty\n");
    }
    signal(&mutex);
}

int main() {
```

```
int ch;
printf("1. Produce\n2. Consume\n3. Exit\n");
while (1) {
    printf("Enter Choice: ");
    scanf("%d", &ch);
    switch (ch) {
        case 1: producer(); break;
        case 2: consumer(); break;
        default: return 0;
    }
}
```

Output:

```
1. Produce
2. Consume
3. Exit
Enter Choice: 2
Buffer is Empty
Enter Choice: 2
Item produced: 1
Enter Choice: 1
Item produced: 2
Enter Choice: 1
Item produced: 3
Enter Choice: 1
Buffer is Full
Enter Choice: 1
Buffer is Full
Enter Choice: 2
Item Consumed: 3
Enter Choice: 2
Item Consumed: 2
Enter Choice: 2
Item Consumed: 1
Enter Choice: 2
Buffer is Empty
Enter Choice:
2
Buffer is Empty
Enter Choice: 2
Buffer is Empty
Enter Choice: 3
```

Semaphores or Producer Consumer

Producer Consumer

```
#include <stdio.h>
int Y = 1, mutex = 1, full = 0, empty = 3;
void wait(int *S)
{
    (*S)--;
}
void signal(int *S)
{
    (*S)++;
}
void producer()
{
    wait(&mutex);
    if (empty > 0)
    {
        wait(&empty);
        signal(&full);
        printf("Item produced : %d\n", Y++);
    }
    else
    {
        printf("Buffer is full\n");
    }
    signal(&mutex);
}
void consumer()
{
    wait(&mutex);
    if (full > 0)
    {
        wait(&full);
        signal(&empty);
    }
}
```

Date _____
Page _____

```

printf("Item consumed: %d\n", --x);
}
else {
    printf("Buffer is empty\n");
}
signal(&mutex);
}

int main() {
    int ch;
    printf("1. Produce in 2. Consume in 3. Exit\n");
    while(1) {
        printf("Enter choice:");
        scanf("%d", &ch);
        switch(ch) {
            case 1: producer(); break;
            case 2: consumer(); break;
            default: return 0;
        }
    }
}

Output:
1. Producer
2. Consumer
3. Exit
Enter choice: 1
Item produced: 1
Enter choice: 1
Item produced: 2
Enter choice: 1
Item produced: 3
Enter choice: 1
Buffer is full
Enter choice: 2
Item consumed: 2

```

Program 6

Question

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

=> Dining Philosophers
Code

// PTHRED AND SEMAPHORE LIBRARY ONLY WORK IN CODEBLOCKS, NOT VSC

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

```

#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

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

sem_t mutex;
sem_t S[N];

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

void take_fork(int phnum) {
    sem_wait(&mutex);
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    test(phnum);

    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum) {
    sem_wait(&mutex);

    state[phnum] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n", phnum + 1,
LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);
}

```

```

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philosopher(void* num) {
    while (1) {
        int* i = (int*)num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

int main() {
    int i;
    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,
(void*)&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:

C:\Users\Admin\Documents\t X + ▾

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

Dining Philosophers

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define LEFT (phnum + 4) % N
#define RIGHT ((phnum + 1) % N)

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0

int state[N];
int phID[N] = {0, 1, 2, 3, 4};
sem_t mutex;
sem_t SEM1;
```

void eat (int phnum){

```
    if (state[phnum] == HUNGRY || state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        state[phnum] = EATING;
        sleep(2);
        printf ("Philosopher %d is Eating\n", phnum + 1);
        sem_post(&SEM1);
    }
}
```

}

```
void take_fork (int phnum) {
    sem_wait(&mutex);
    state[phnum] = HUNGRY;
    printf ("Philosopher %d is Hungry\n", phnum + 1);
```

```

    eat (phnum);
    sem - post (Sphumer);
    sem -> eat C & S I phnum];
    sleep (1);
}

void put - fork (int phnum) {
    sem - wait (Sphumer);
    State[phnum] = THINKING;
}

void philosopher (void * num) {
    while (1) {
        int * i = (int *) num;
        sleep (1);
        take_fork (* i);
        sleep (1);
        put_fork (* i);
    }
}

Author: Philosopher 4 is hungry
Philosopher 5 is pretty full & goes down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is eating
Philosopher 1 is hungry

```

Program 7

Question

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

=> Banker's Algorithm / Deadlock Avoidance

Code

```

#include <stdio.h>
#include <stdlib.h>
int condition(int **need, int *work, int i, int m)
{
    for (int j = 0; j < m; j++)

```

```

{
    if (need[i][j] > work[j])
        return 0;
}
return 1;
}
int safety(int m, int n, int **allocated, int **max, int
*available, int *sequence)
{
    // Need Matrix
    int **need = (int**) malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
    {
        need[i] = (int*) malloc(m * sizeof(int));
        for (int j = 0; j < m; j++)
        {
            need[i][j] = max[i][j] - allocated[i][j];
        }
    }
    // Work array
    int *work = (int*) malloc(m * sizeof(int));
    for (int i = 0; i < m; i++)
    {
        work[i] = available[i];
    }
    // Finish array
    int *finish = (int*) malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
    {
        finish[i] = 0;
    }

    int safeIndex = 0;
    int changed;
    do {
        changed = 0;
        for (int i = 0; i < n; i++)
        {
            if (!finish[i] && condition(need, work, i, m))
            {
                for (int j = 0; j < m; j++)
                {
                    work[j] += allocated[i][j];
                }
                finish[i] = 1;
                sequence[safeIndex++] = i;
            }
        }
    } while (changed);
}

int condition(int **need, int *work, int i, int m)
{
    for (int j = 0; j < m; j++)
    {
        if (need[i][j] > work[j])
            return 0;
    }
    return 1;
}

```

```

        changed = 1;
    }
}
} while (changed);

for (int i = 0; i < n; i++)
{
    if (!finish[i])
    {
        return 0;
    }
}
return 1;
}

int main()
{
    int n, m;
    printf("Enter number of processes and resources (n x m order):
");
    scanf("%d", &n);
    scanf("%d", &m);
    // Allocation Matrix
    printf("Enter Allocation Matrix:\n");
    int **allocated = (int **) malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
    {
        allocated[i] = (int *) malloc(m * sizeof(int));
        for (int j = 0; j < m; j++)
        {
            scanf("%d", &allocated[i][j]);
        }
    }

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

    // Available Matrix
    printf("Enter Available matrix:\n");

```

```

int *available = (int *) malloc(m * sizeof(int));
for (int i = 0; i < m; i++)
{
    scanf("%d", &available[i]);
}
// Sequence Matrix
int *sequence = (int *) malloc(n * sizeof(int));

int safe = safety(m, n, allocated, max, available, sequence);
if (safe)
{
    printf("System is in a Safe State.\nSafe Sequence: ");
    for (int i = 0; i < n; i++)
    {
        printf("P%d\t", sequence[i]);
    }
    printf("\n");
}
else
{
    printf("System is not in a Safe State.\n");
}
return 0;
}

```

Output:

```

Enter number of processes and resources (n x m order): 5 3
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Available matrix:
3 3 2
System is in a Safe State.
Safe Sequence: P1      P3      P4      P0      P2

```

8. Randomis Algorithm

```
#include <csdn.h>
#include <stdlib.h>
int condition (int **need, int *work, int i, int m)
{
    for (int j = 0; j < m; j++)
    {
        if (need[i][j] > work[j])
            return 0;
    }
    return 1;
}

int Safe (int m, int n, int **allocated, int **max, int **min,
          int *segmed)
{
    int **need = (int **) malloc (m * sizeof (int *));
    for (int i = 0; i < n; i++)
    {
        need[i] = (int *) malloc (m * sizeof (int));
        for (int j = 0; j < m; j++)
        {
            need[i][j] = (int *) malloc (m * sizeof (int));
            for (int k = 0; k < m; k++)
                need[i][j][k] = max[i][j][k] - allocated[i][j][k];
        }
    }
    int safe_index = 0;
    int changed;
    do
    {
        changed = 0;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                if (condition (need, work, i, m))
                {
                    work[j] += max[i][j][safe_index];
                    changed = 1;
                }
            }
        }
    } while (changed);
}
```

```

if (!Finish[i][j] & condition(need, work, i, m))
{
    for (int j = 0; j < m; j++)
    {
        work[i][j] = allocated[i][j]; }
        Finish[i][j] = 1;
        Sequence[SafeIndex++][j] = i;
        changed = 1; }

    } while (changed);
    for (int i = 0; i < n; i++)
    {
        if (!Finish[i])
        return 0; }

    return 1;
}

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

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

    }

    printf("Enter Max Matrix:\n");
    int **max = (int **) malloc (n * sizeof(int *));
    for (int i = 0; i < n; i++)
    {
        max[i] = (int *) malloc (m * sizeof(int)); }

}

```

Enter Allocation matrix

3 3 2

System is in Safe State

Safe sequence: P1 P3 P4 P0 P2

Program 8

Question

Write a C program to simulate deadlock detection

=> Deadlock Detection

Code

```
#include <stdio.h>
#include <stdbool.h>
#define P 5
#define R 3
int main() {
    int finish[P] = {0};
    int work[R];
    int need[P][R] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };
    int allocation[P][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };
    int available[R] = {3, 3, 2};
    for (int i = 0; i < R; i++) {
        work[i] = available[i];
    }
    bool deadlock = false;
    int count = 0;

    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                bool canFinish = true;
                for (int r = 0; r < R; r++) {
                    if (need[p][r] - allocation[p][r] > work[r]) {
                        canFinish = false;
                        break;
                    }
                }
            }
        }
    }
}
```

```

        if (canFinish) {
            for (int r = 0; r < R; r++) {
                work[r] += allocation[p][r];
            }
            printf("Process %d can finish.\n", p);
            finish[p] = 1;
            found = true;
            count++;
        }
    }
    if (!found) {
        deadlock = true;
        break;
    }
}
if (deadlock) {
    printf("System is in a deadlock state.\n");
} else {
    printf("System is not in a deadlock state.\n");
}
return 0;
}

```

Output:

```

Process 1 can finish.
Process 3 can finish.
Process 4 can finish.
Process 0 can finish.
Process 2 can finish.
System is not in a deadlock state.

```

b. Schedular Selection

#include <stdio.h>

#include <stdlib.h>

#define R 5

#define P 3

int main () {

int finish [P][R] = {{0}};

int work [R];

int need [P][R] = {{

{7,5,3},

{3,2,2},

{4,0,2},

{2,2,2},

{4,3,3},

{3,3,3}}

int allocation [P][R] = {{

{0,0,0},

{2,0,0},

{3,0,2},

{2,1,1},

{0,0,2}},

int available [R] = {3,3,2};

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

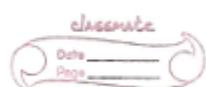
work[i] = available[i];

```

bool deadlock = false;
int count = 0;
while (count < P) {
    bool found = false;
    for (int p = 0; p < R; p++) {
        if (Finish[p] == 0) {
            bool canFinish = true;
            for (int r = 0; r < R; r++) {
                if (need[cp[r]] - allocation[cp[r]] >= 0)
                    canFinish = false;
            }
        }
        if (!canFinish) {
            for (int r = 0; r < R; r++) {
                work[r] += allocation[cp[r]];
            }
        }
    }
    if (!found) {
        deadlock = true;
        break;
    }
}
if (deadlock) {
    printf("System is in deadlock state\n");
} else {
    printf("System is not in deadlock state\n");
    return 0;
}

```

Output : Process 1 can finish
 Process 3 can finish
 Process 4 can finish
 Process 0 can't finish
 Process 2 can finish
 System is not in a deadlock state



Program 9

Question

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

Worst-fit

d) Best-fit

e) First-fit

=> Best fit, worst fit, first fit

Code

```
#include <stdio.h>

struct Block {
    int block_no;
    int block_size;
    int is_free;
};

struct File {
    int file_no;
    int file_size;
};

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("Memory Management Scheme - Best Fit\n");

    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        int best_fit_block = -1;
        int min_fragment = 10000; // Initialize with a large value

        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >=
files[i].file_size) {
                int fragment = blocks[j].block_size -
files[i].file_size;
                if (fragment < min_fragment) {
                    min_fragment = fragment;
                    best_fit_block = j;
                }
            }
        }

        if (best_fit_block != -1) {
```

```

        blocks[best_fit_block].is_free = 0;
        printf("%d\t%d\t%d\t%d\t%d\t%d\n",
files[i].file_no, files[i].file_size,
                blocks[best_fit_block].block_no,
blocks[best_fit_block].block_size, min_fragment);
    }
}

void firstFit(struct Block blocks[], int n_blocks, struct File
files[], int n_files) {
    printf("Memory Management Scheme - First Fit\n");

printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

for (int i = 0; i < n_files; i++) {
    int found = 0;
    for (int j = 0; j < n_blocks; j++) {
        if (blocks[j].is_free && blocks[j].block_size >=
files[i].file_size) {
            blocks[j].is_free = 0;
            int fragment = blocks[j].block_size -
files[i].file_size;
            printf("%d\t%d\t%d\t%d\t%d\t%d\n",
files[i].file_no, files[i].file_size,
                    blocks[j].block_no, blocks[j].block_size,
fragment);
            found = 1;
            break;
        }
    }
    if (!found)
        printf("No suitable block found for File %d\n",
files[i].file_no);
}
}

void worstFit(struct Block blocks[], int n_blocks, struct File
files[], int n_files) {
    printf("Memory Management Scheme - Worst Fit\n");

printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

for (int i = 0; i < n_files; i++) {
    int worst_fit_block = -1;

```

```

        int max_fragment = -1; // Initialize with a small value

        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >=
files[i].file_size) {
                int fragment = blocks[j].block_size -
files[i].file_size;
                if (fragment > max_fragment) {
                    max_fragment = fragment;
                    worst_fit_block = j;
                }
            }
        }

        if (worst_fit_block != -1) {
            blocks[worst_fit_block].is_free = 0;
            printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
files[i].file_no, files[i].file_size,
                    blocks[worst_fit_block].block_no,
blocks[worst_fit_block].block_size, max_fragment);
        }
    }
}

int main() {
    int n_blocks, n_files;
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);
    printf("Enter the number of files: ");
    scanf("%d", &n_files);

    struct Block blocks[n_blocks];
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
    }

    struct File files[n_files];
    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }
    while(1) {

```

```

int choice;
printf("Choose Memory Management Scheme:\n");
printf("1. Best Fit\n");
printf("2. First Fit\n");
printf("3. Worst Fit\n");
printf("[ANY KEY]. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

// Reset blocks for allocation scheme
for (int i = 0; i < n_blocks; i++) {
    blocks[i].is_free = 1;
}

switch (choice) {
    case 1:
        bestFit(blocks, n_blocks, files, n_files);
        break;
    case 2:
        firstFit(blocks, n_blocks, files, n_files);
        break;
    case 3:
        worstFit(blocks, n_blocks, files, n_files);
        break;
    default:
        printf("Closing...");
        return 0;
} }

return 0;
}

```

Output:

```
● Enter the number of blocks: 5
Enter the number of files: 4
Enter the size of block 1: 100
Enter the size of block 2: 500
Enter the size of block 3: 200
Enter the size of block 4: 300
Enter the size of block 5: 600
Enter the size of file 1: 212
Enter the size of file 2: 417
Enter the size of file 3: 112
Enter the size of file 4: 426
Choose Memory Management Scheme:
1. Best Fit
2. First Fit
3. Worst Fit
[ANY KEY]. Exit
Enter your choice: 1
Memory Management Scheme - Best Fit
File_no:      File_size:      Block_no:      Block_size:      Fragment
1            212             4              300            88
2            417             2              500            83
3            112             3              200            88
4            426             5              600           174
Choose Memory Management Scheme:
1. Best Fit
2. First Fit
3. Worst Fit
[ANY KEY]. Exit
Enter your choice: 2
Memory Management Scheme - First Fit
File_no:      File_size:      Block_no:      Block_size:      Fragment
1            212             2              500            288
2            417             5              600            183
3            112             3              200            88
No suitable block found for File 4
Choose Memory Management Scheme:
1. Best Fit
2. First Fit
3. Worst Fit
[ANY KEY]. Exit
Enter your choice: 5
Closing...
```

B. Worst fit, best fit, first fit algorithm

```
#include <stdio.h>
define structure block with block-no, block-size,
define structure file with file-no, file-size.

void bestFit (struct Block blocks, int n-blocks,
    File files)
{
    struct Block {
        int block-no;
        int block-size;
        int is-free;
    };
    struct File {
        int file-no;
        int file-size;
    };
}

void bestFit (struct Block blocks[], int n-blocks, struct File[],
    int n-files) {
    printf ("File-no: %d filesize: %d Block-no: %d Block-size: %d\n",
        for (int i = 0; i < n-blocks; i++) {
            int best-fit-block = -1;
            int min-fragment = 100000;
            for (int j = 0; j < n-blocks; j++) {
                if (blocks[i].is-free && blocks[i].block-size >= files[i].file-size) {
                    if (min-fragment > blocks[i].block-size) {
                        min-fragment = blocks[i].block-size;
                        best-fit-block = i;
                    }
                }
            }
            if (best-fit-block != -1) {
                blocks[best-fit-block].is-free = 0;
                blocks[best-fit-block].block-size -= files[i].file-size;
            }
        }
}
```



```
blocks[i].is-free = 0;
printf("Allocating %d blocks\n", file[i].file-size);
blocks[i].file-no, file[i].file-size,
blocks[i].block-no, blocks[i].block-size
blocks[i].block-size, min-fragment);
}
}
```

```
void firstfit(struct Block blocks[], int n-blocks, struct
File files[], int n-files) {
printf("FirstFit:\n");
printf("File-no: %d File-size: %d block-no: %d block-size:
%d fragment\n");
for (int i = 0; i < n-files; i++) {
int found = 0;
for (int j = 0; j < n-blocks; j++) {
blocks[j].is-free = 0;
if (blocks[j].block-size - files[i].file-size
blocks[j].block-size - files[i].file-size) {
blocks[j].is-free = 1;
blocks[j].file-no = files[i].file-no;
blocks[j].block-no = j;
blocks[j].block-size = files[i].file-size;
found = 1;
break;
}
}
if (found == 1)
printf("%d %d %d %d\n", files[i].file-no, files[i].file-size,
blocks[j].block-no, blocks[j].block-size);
}
}
}
```

```
void worstfit(struct Block blocks[], int
n-blocks) {
printf("Worst fit memory management Scheme - Worst Fit\n");
printf("File-no: %d File-size: %d block-no: %d block-size:
%d fragment\n");
for (int i = 0; i < n-files; i++) {
blocks[i].file-no, file[i].file-size,
blocks[i].block-size, min-fragment);
}
}
```

Output : memory management

Enter the number of blocks : 5.

Enter the number of files : 4

Enter the size of the blocks:

Block 1: 100

Block 2: 500

Block 3: 200

Block 4: 300

Block 5: 600

Enter the size of the files:

File 1: 212

File 2: 417

File 3: 112

File 4: 426

1. First fit

2. Best fit

3. Worst fit

[ANY KEY] Exit

Enter your choice : 1

Best fit

File-no:	File-size:	Block-no:	Block-size:
1	212	4	300
2	417	2	500
3	112	3	200
4	426	5	600

Program 10

Question

Write a C program to simulate page replacement algorithms a) FIFO

d) LRU

e) Optimal

=> LRU & Optimal

Code

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

int search(int key, int frame[], int frameSize) {
    for (int i = 0; i < frameSize; i++) {
        if (frame[i] == key)
            return i;
    }
    return -1;
}

int findOptimal(int pages[], int frame[], int n, int index, int
frameSize) {
    int farthest = index, pos = -1;
    for (int i = 0; i < frameSize; i++) {
        int j;
        for (j = index; j < n; j++) {
            if (frame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    pos = i;
                }
                break;
            }
        }
        if (j == n)
            return i;
    }
    return (pos == -1) ? 0 : pos;
}

void simulateFIFO(int pages[], int n, int frameSize) {
    int frame[frameSize], front = 0, count = 0, hits = 0;

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

```

for (int i = 0; i < n; i++) {
    if (search(pages[i], frame, frameSize) == -1) {
        frame[front] = pages[i];
        front = (front + 1) % frameSize;
        count++;
    } else {
        hits++;
    }
}
printf("FIFO Page Faults: %d, Page Hits: %d\n", count, hits);
}

void simulateLRU(int pages[], int n, int frameSize) {
    int frame[frameSize], time[frameSize], count = 0, hits = 0;

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

    for (int i = 0; i < n; i++) {
        int pos = search(pages[i], frame, frameSize);
        if (pos == -1) {
            int least = 0;
            for (int j = 1; j < frameSize; j++) {
                if (time[j] < time[least])
                    least = j;
            }
            frame[least] = pages[i];
            time[least] = i;
            count++;
        } else {
            hits++;
            time[pos] = i;
        }
    }
    printf("LRU Page Faults: %d, Page Hits: %d\n", count, hits);
}

void simulateOptimal(int pages[], int n, int frameSize) {
    int frame[frameSize], count = 0, hits = 0;

    for (int i = 0; i < frameSize; i++)
        frame[i] = -1;

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

```

```

        if (search(pages[i], frame, frameSize) == -1) {
            int index = -1;
            for (int j = 0; j < frameSize; j++) {
                if (frame[j] == -1) {
                    index = j;
                    break;
                }
            }
            if (index != -1) {
                frame[index] = pages[i];
            } else {
                int replaceIndex = findOptimal(pages, frame, n, i + 1, frameSize);
                frame[replaceIndex] = pages[i];
            }
            count++;
        } else {
            hits++;
        }
    }
    printf("Optimal Page Faults: %d, Page Hits: %d\n", count, hits);
}

int main() {
    int n, frameSize;
    printf("Enter the size of the pages: ");
    scanf("%d", &n);

    int pages[n];
    printf("Enter the page strings: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);

    printf("Enter the no of page frames: ");
    scanf("%d", &frameSize);

    simulateFIFO(pages, n, frameSize);
    simulateOptimal(pages, n, frameSize);
    simulateLRU(pages, n, frameSize);

    return 0;
}

```

Output:

```

Enter the size of the pages: 7
Enter the page strings: 1 3 0 3 5 6 3
Enter the no of page frames: 3
FIFO Page Faults: 6, Page Hits: 1
Optimal Page Faults: 5, Page Hits: 2
LRU Page Faults: 5, Page Hits: 2

```

Q. LRU, FIFO, Optimal C program

```

#include <stdio.h>
#include <stdlib.h>
int search (int page, int frame[], int frameSize) {
    for (int i = 0; i < frameSize; i++) {
        if (frame[i] == page)
            return i;
    }
    return -1;
}
int findOptimal (int pages[], int frame[], int n,
                 int index, int frameSize) {
    int farthest = index, pos = -1;
    for (int i = 0; i < frameSize; i++) {
        int j;
        for (j = index; j < n; j++) {
            if (frame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    pos = i;
                }
            }
        }
        break;
    }
    if (j == n)
        return -1;
    return (pos == -1) ? 0 : pos;
}

```

```

void simulate (int pages[], int n, int frameSize,
               int frame[], int frameSize, int bnt = 0, count = 0, hits = 0)
{
    for (int i = 0; i < n; i++) {
        frame[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        if (!search (pages[i], frame, frameSize))
            frame[findOptimal (pages, frame, n, i, frameSize)] = pages[i];
    }
}

```



```
front = (front + 1) % frameSize;
count++;
else {
    hits++;
}
printf ("FIFO Page Faults: %d, Page Hits: %d\n",
       count, hits); }

void simulateLRU (int pages[], int n, int frameSize) {
    int frame[frameSize], timeFrames[n], count=0, hits=0;
    for (int i=0; i<frameSize; i++) {
        frame[i] = -1;
        timeFrames[i] = 0;
    }
    for (int i=0; i<n; i++) {
        int pos = search(pages[i], frame, frameSize);
        if (pos == -1) {
            int least = 0;
            for (int j=1; j < frameSize; j++) {
                if (timeFrames[j] < timeFrames[least])
                    least = j;
            }
            frame[least] = pages[i];
            timeFrames[least] = i;
            count++;
        } else {
            hits++;
            timeFrames[pos] = i;
        }
    }
    printf ("LRU page Faults: %d, Page Hits: %d\n",
           count, hits); }

void simulateOptimal (int pages[], int n, int frameSize) {
    int frame[frameSize], count = 0, hits = 0;
    for (int i=0; i<frameSize; i++)
        frame[i] = -1;
```

Simulate LRU (page s, n; framesize);
return 0;

Output:

Enter the size of the pages:
7

Enter the ~~no.~~ page ~~names~~ strings:

1 3 0 3 5 6 3

Enter the no. of page frames:
3

FIFO: Page Faults: 6, Page Hits: 1

Optimal: Page Faults: 5, Page Hits: 2

LRU: Page Faults: 5, Page Hits: 2

45