VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

MAKADIA RISHIT DILIPBHAI (1BM23CS177)

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 Jun 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 **MAKADIA RISHIT DILIPBHAI (1BM23CS177),** who is a 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 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. Selva kumar S
Associate Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System
CO2	Analyze various Operating system strategies and techniques
CO3	Demonstrate the different functionalities of Operating System
CO4	Conduct practical experiments to implement the functionalities of Operating system

Program - 1

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

a) FCFS b) SJF c) Priority d) Round Robin

Code:

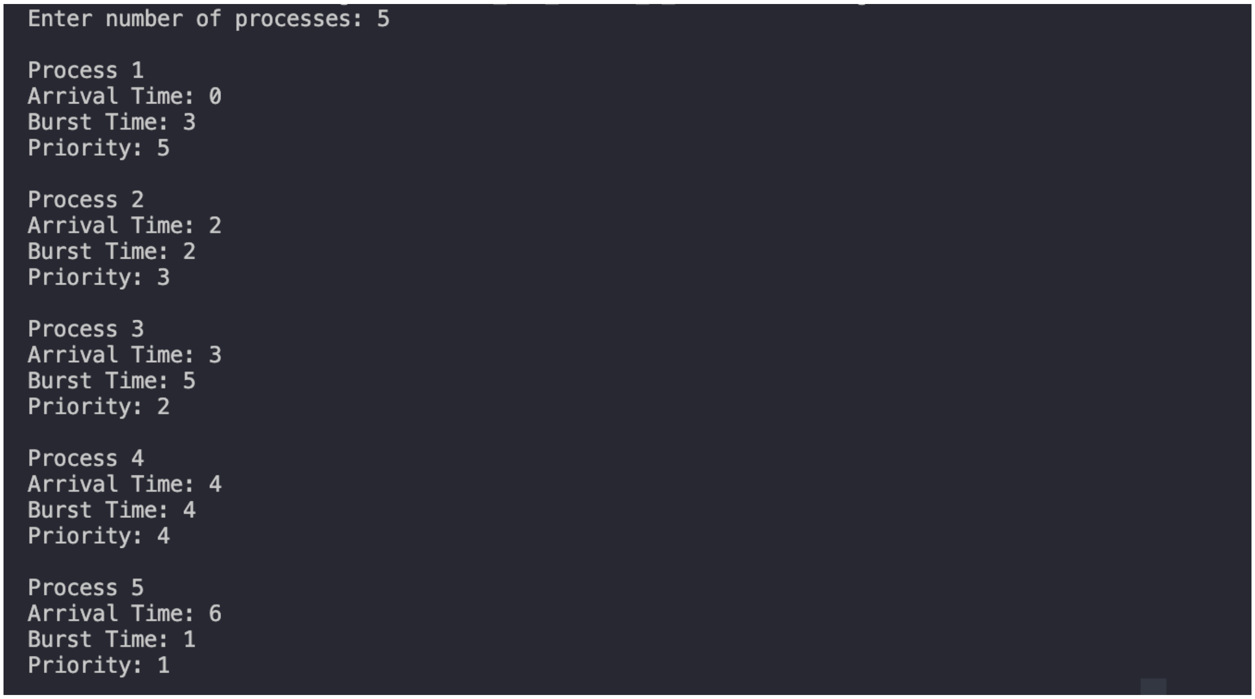
```
#include <stdio.h>

#include <stdlib.h>

#define QUANTUM 4

typedef struct process {

  int pID, aT, bT, cT, tT, wT, priority, remainingB;

} proc;

void sortByArrival(proc p[], int n) {

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

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

      if (p[i].aT > p[j].aT) {

        proc temp = p[i];

        p[i] = p[j];

        p[j] = temp;

      }

    }

  }

}

void sortByPriority(proc p[], int n) {

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

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

      if (p[i].aT > p[j].aT || (p[i].aT == p[j].aT && p[i].priority > p[j].priority)) {

        proc temp = p[i];
```

```
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}
void calculateTAT(proc p[], int n) {
    for (int i = 0; i < n; i++) {
        p[i].tT = p[i].cT - p[i].aT;
    }
}
void calculateWT(proc p[], int n) {
    for (int i = 0; i < n; i++) {
        p[i].wT = p[i].tT - p[i].bT;
    }
}
void printResults(proc p[], int n, const char* algorithm) {
    double avgTAT = 0, avgWT = 0;

    printf("\n%s Scheduling Results:\n", algorithm);
    printf("Process ID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
              p[i].pID, p[i].aT, p[i].bT, p[i].cT, p[i].tT, p[i].wT);
        avgTAT += p[i].tT;
        avgWT += p[i].wT;
    }

    printf("\nAverage Turnaround Time: %.2f\n", avgTAT / n);
    printf("Average Waiting Time: %.2f\n\n", avgWT / n);
}
void FCFS(proc p[], int n) {
    proc temp[n];
    for (int i = 0; i < n; i++) temp[i] = p[i];
    sortByArrival(temp, n);
    temp[0].cT = temp[0].aT + temp[0].bT;
    for (int i = 1; i < n; i++) {
        if (temp[i].aT > temp[i-1].cT) {
            temp[i].cT = temp[i].aT + temp[i].bT;
        } else {
            temp[i].cT = temp[i-1].cT + temp[i].bT;
        }
    }
    calculateTAT(temp, n);
    calculateWT(temp, n);
    printResults(temp, n, "FCFS");
}
void SJF(proc p[], int n) {
    proc temp[n];
    for (int i = 0; i < n; i++) temp[i] = p[i];
    sortByArrival(temp, n);
    int completed = 0, currentTime = 0;
    int isCompleted[n];
    for (int i = 0; i < n; i++) isCompleted[i] = 0;

    while (completed < n) {
        int idx = -1;
        int minBT = 9999;
        for (int i = 0; i < n; i++) {
            if (temp[i].aT <= currentTime && !isCompleted[i]) {
                if (temp[i].bT < minBT) {
                    minBT = temp[i].bT;
                    idx = i;
                }
            }
        }
        if (idx == -1) {
            currentTime++;
        } else {
            temp[idx].cT = currentTime + temp[idx].bT;
            currentTime = temp[idx].cT;
            isCompleted[idx] = 1;
            completed++;
        }
    }
    calculateTAT(temp, n);
    calculateWT(temp, n);
    printResults(temp, n, "SJF");
}
void PriorityScheduling(proc p[], int n) {
```

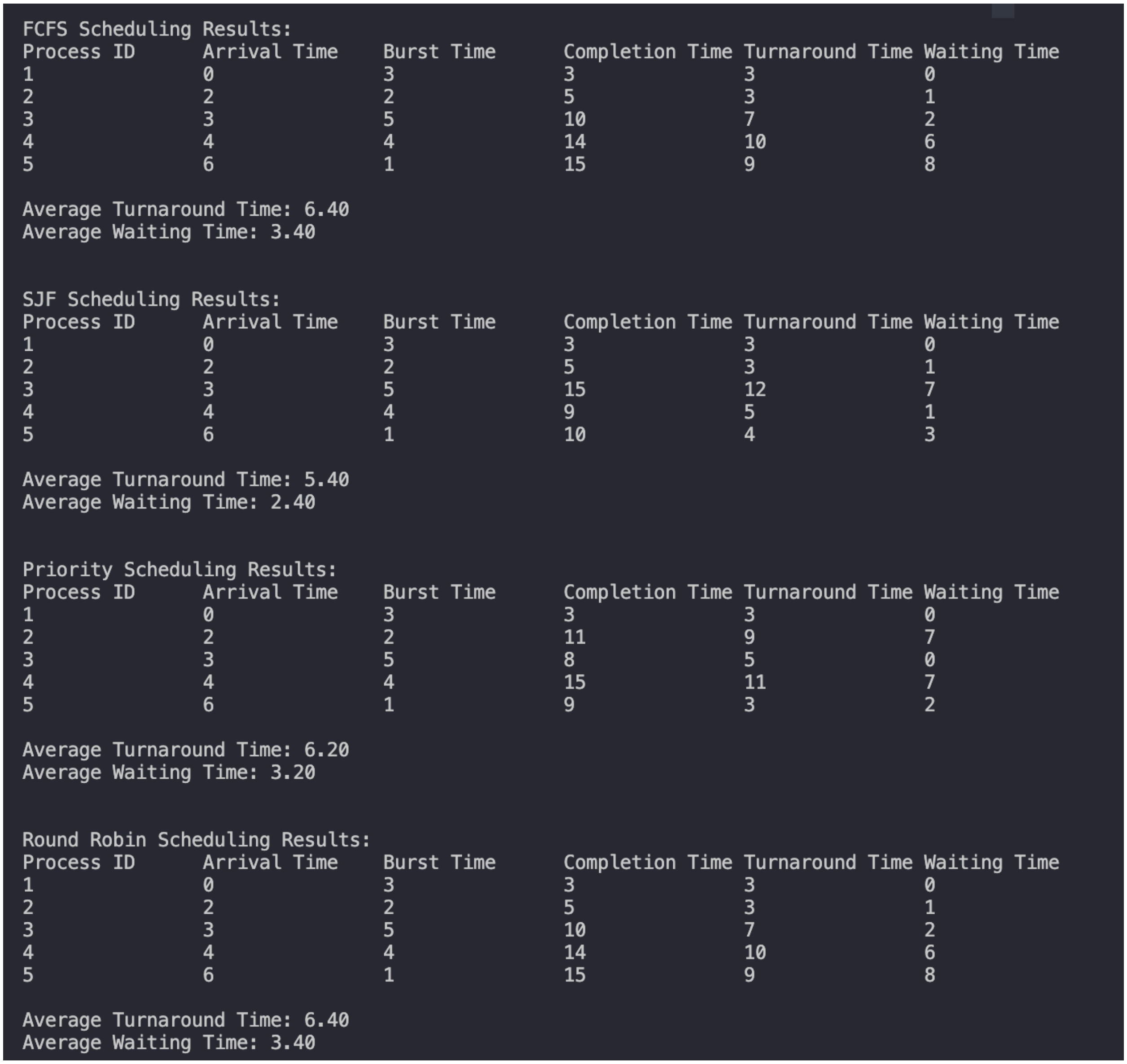
```
proc temp[n];
for (int i = 0; i < n; i++) temp[i] = p[i];
sortByPriority(temp, n);
int completed = 0, currentTime = 0;
int isCompleted[n];
for (int i = 0; i < n; i++) isCompleted[i] = 0;
while (completed < n) {
   int idx = -1;
   int minPri = 9999;
   for (int i = 0; i < n; i++) {
      if (temp[i].aT <= currentTime && !isCompleted[i]) {
         if (temp[i].priority < minPri) {
            minPri = temp[i].priority;
            idx = i;
         }
      }
   }
   if (idx == -1) {
      currentTime++;
   } else {
      temp[idx].cT = currentTime + temp[idx].bT;
      currentTime = temp[idx].cT;
      isCompleted[idx] = 1;
      completed++;
   }
}
calculateTAT(temp, n);
```

```
    calculateWT(temp, n);
    printResults(temp, n, "Priority");
}
void RoundRobin(proc p[], int n) {
    proc temp[n];
    for (int i = 0; i < n; i++) {
        temp[i] = p[i];
        temp[i].remainingB = temp[i].bT;
    }
    sortByArrival(temp, n);
    int completed = 0, currentTime = 0;
    int isCompleted[n];
    for (int i = 0; i < n; i++) isCompleted[i] = 0;
    while (completed < n) {
        int idx = -1;
        for (int i = 0; i < n; i++) {
            if (temp[i].aT <= currentTime && !isCompleted[i] && temp[i].remainingB > 0) {
                idx = i;
                break;
            }
        }
        if (idx == -1) {
            currentTime++;
        } else {
            int timeSlice = (temp[idx].remainingB > QUANTUM) ? QUANTUM : temp[idx].remainingB;
            temp[idx].remainingB -= timeSlice;
            currentTime += timeSlice;
            if (temp[idx].remainingB == 0) {
                temp[idx].cT = currentTime;
                isCompleted[idx] = 1;
                completed++;
            }
        }
    }
    calculateTAT(temp, n);
    calculateWT(temp, n);
    printResults(temp, n, "Round Robin");
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    proc processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pID = i + 1;
        printf("\nProcess %d\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].aT);
        printf("Burst Time: ");
        scanf("%d", &processes[i].bT);
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
    }
    // Call all scheduling algorithms directly

    FCFS(processes, n);

    SJF(processes, n);

    PriorityScheduling(processes, n);

    RoundRobin(processes, n);

    return 0;

}
```

Output:



```
Enter number of processes: 5

Process 1
Arrival Time: 0
Burst Time: 3
Priority: 5

Process 2
Arrival Time: 2
Burst Time: 2
Priority: 3

Process 3
Arrival Time: 3
Burst Time: 5
Priority: 2

Process 4
Arrival Time: 4
Burst Time: 4
Priority: 4

Process 5
Arrival Time: 6
Burst Time: 1
Priority: 1
```



```
FCFS Scheduling Results:
Process ID      Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1               0               3               3               3               0
2               2               2               5               3               1
3               3               5               10              7               2
4               4               4               14              10              6
5               6               1               15              9               8

Average Turnaround Time: 6.40
Average Waiting Time: 3.40


SJF Scheduling Results:
Process ID      Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1               0               3               3               3               0
2               2               2               5               3               1
3               3               5               15              12              7
4               4               4               9               5               1
5               6               1               10              4               3

Average Turnaround Time: 5.40
Average Waiting Time: 2.40


Priority Scheduling Results:
Process ID      Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1               0               3               3               3               0
2               2               2               11              9               7
3               3               5               8               5               0
4               4               4               15              11              7
5               6               1               9               3               2

Average Turnaround Time: 6.20
Average Waiting Time: 3.20


Round Robin Scheduling Results:
Process ID      Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1               0               3               3               3               0
2               2               2               5               3               1
3               3               5               10              7               2
4               4               4               14              10              6
5               6               1               15              9               8

Average Turnaround Time: 6.40
Average Waiting Time: 3.40
```

Program 2

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

Code:

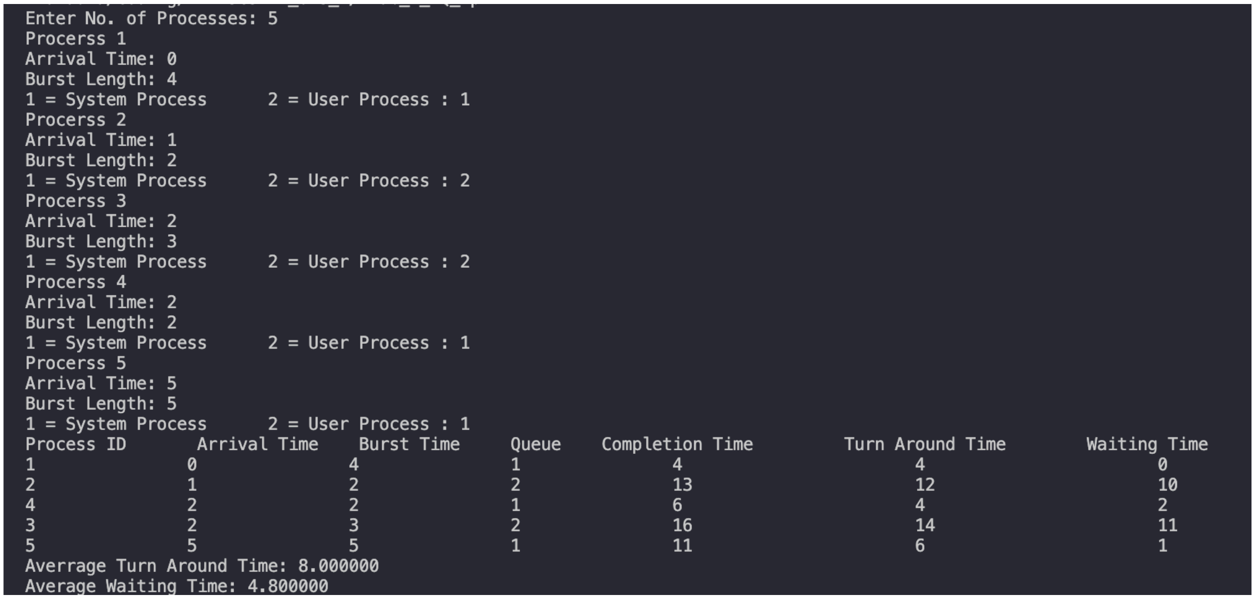
```
#include <stdio.h>

#include <stdlib.h>

typedef struct process{

  int aT,bT, pID, cT, tT, wT, priority;

}proc;

void completionT(proc p[], int n) {

  int completed = 0, currentTime1 = 0, currentTime2=0;

  int isCompleted[n];

  for (int i = 0; i < n; i++) isCompleted[i] = 0;

  while (completed < n) {

    int idx = -1;

    int minPri = 9999;

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

      if (p[i].aT <= currentTime1 && !isCompleted[i]) {

        if (p[i].priority < minPri) {

          minPri = p[i].priority;

          idx = i;

        }

      }

      else if (p[i].priority == minPri && p[i].aT < p[idx].aT) {

        idx = i;

      }
```

```
        }
        if (idx == -1) {
            currentTime1++;
            currentTime2++;
        }
        else if(idx!=(-1) && minPri == 1){
            p[idx].cT = currentTime1 + p[idx].bT;
            currentTime1 = p[idx].cT;
            currentTime2 = p[idx].cT;
            isCompleted[idx] = 1;
            completed++;
        }
        else if(idx!=(-1) && minPri == 2){
            p[idx].cT = currentTime2 + p[idx].bT;
            currentTime2 = p[idx].cT;
            currentTime1 = p[idx].cT;
            isCompleted[idx] = 1;
            completed++;
        }
    }
}
void tatT(proc p[], int n){
    for (int k=0; k<n; k++){
        p[k].tT=p[k].cT-p[k].aT;
    }
}
```

```
void waitingT(proc p[], int n){
    for (int l=0; l<n; l++){
        p[l].wT=p[l].tT-p[l].bT;
    }
}
void MultiQueue(proc arr[], int n){
    double TATavg=0;
    double WTavg=0;
    sort(arr, n);
    completionT(arr, n);
    tatT(arr, n);
    waitingT(arr, n);
    for(int i=0;i<n;i++){
        TATavg+=arr[i].tT;
        WTavg+=arr[i].wT;
    }
    printf("");
    printf("Process ID\t Arrival Time\t Burst Time\tQueue\t Completion Time\t Turn Around Time\t Waiting Time\n");
    for(int i=0;i<n;i++){
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\t\t\t%d\n",arr[i].pID,arr[i].aT,arr[i].bT,arr[i].priority,arr[i].cT,arr[i].tT,arr[i].wT);
    }
    printf("Averrage Turn Around Time: %f\n",(TATavg/n));
    printf("Average Waiting Time: %f\n",(WTavg/n));
}
```

```
void sort(proc p[], int n){
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].aT > p[j].aT || (p[i].aT == p[j].aT && p[i].priority > p[j].priority)) {
                proc temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}
int main(){
    int no;
    printf("Enter No. of Processes: ");
    scanf("%d", &no);
    proc process[no];
    for (int i=0; i<no ; i++){
        process[i].pID = i + 1;
        printf("Procerss %d \n", (i+1));
        printf("Arrival Time: ");
        scanf("%d", &process[i].aT);
        printf("Burst Length: ");
        scanf("%d", &process[i].bT);
        printf("1 = System Process\t2 = User Process : ");
        scanf("%d", &process[i].priority);
    }
    MultiQueue(process, no);
}
```

Output:



```
Enter No. of Processes: 5
Procerss 1
Arrival Time: 0
Burst Length: 4
1 = System Process      2 = User Process : 1
Procerss 2
Arrival Time: 1
Burst Length: 2
1 = System Process      2 = User Process : 2
Procerss 3
Arrival Time: 2
Burst Length: 3
1 = System Process      2 = User Process : 2
Procerss 4
Arrival Time: 2
Burst Length: 2
1 = System Process      2 = User Process : 1
Procerss 5
Arrival Time: 5
Burst Length: 5
1 = System Process      2 = User Process : 1
Process ID       Arrival Time    Burst Time     Queue    Completion Time         Turn Around Time        Waiting Time
1               0               4               1               4                       4                       0
2               1               2               2               13                      12                      10
4               2               2               1               6                       4                       2
3               2               3               2               16                      14                      11
5               5               5               1               11                      6                       1
Averrage Turn Around Time: 8.000000
Average Waiting Time: 4.800000
```

Program 3

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

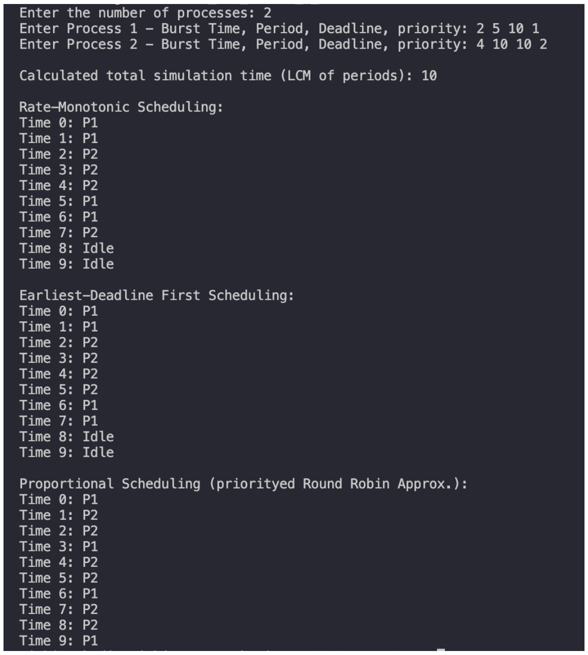
a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling

Code:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 10

typedef struct {

   int id;

   int bT;

   int period;

   int deadline;

   int remT;

   int priority;

   int nextA;

   int nextD;

} Task;

// GCD and LCM functions

int gcd(int a, int b) {

   return b == 0 ? a : gcd(b, a % b);

}

int lcm(int a, int b) {

   return (a * b) / gcd(a, b);

}

int calculate_lcm(Task *tasks, int n) {

   int result = tasks[0].period;
```

```
    for (int i = 1; i < n; i++) {
        result = lcm(result, tasks[i].period);
    }
    return result;
}


void insert(Task *tasks, int n) {
    for (int i = 0; i < n; i++) {
        printf("Enter Process %d - Burst Time, Period, Deadline, priority: ", i + 1);
        scanf("%d %d %d %d", &tasks[i].bT, &tasks[i].period, &tasks[i].deadline, &tasks[i].priority);
        tasks[i].id = i + 1;
        tasks[i].remT = 0;
        tasks[i].nextA = 0;
        tasks[i].nextD = 0;
    }
}
void rate_monotonic(Task *tasks, int n, int hyper_period) {
    printf("\nRate-Monotonic Scheduling:\n");
    for (int t = 0; t < hyper_period; t++) {
        int current = -1;
        for (int i = 0; i < n; i++) {
            if (t == tasks[i].nextA) {
                tasks[i].remT = tasks[i].bT;
                tasks[i].nextA += tasks[i].period;
                tasks[i].nextD = t + tasks[i].period;
            }
            if (tasks[i].remT > 0 && (current == -1 || tasks[i].period < tasks[current].period)) {
                current = i;
            }
        }
        if (current != -1) {
            printf("Time %d: P%d\n", t, tasks[current].id);
            tasks[current].remT--;
        } else {
            printf("Time %d: Idle\n", t);
        }
    }
}
void earliest_deadline_first(Task *tasks, int n, int hyper_period) {
    printf("\nEarliest-Deadline First Scheduling:\n");
    for (int t = 0; t < hyper_period; t++) {
        int current = -1;
        for (int i = 0; i < n; i++) {
            if (t == tasks[i].nextA) {
                tasks[i].remT = tasks[i].bT;
                tasks[i].nextA += tasks[i].period;
                tasks[i].nextD = t + tasks[i].deadline;
            }
            if (tasks[i].remT > 0 && (current == -1 || tasks[i].nextD < tasks[current].nextD)) {
                current = i;
            }
        }
        if (current != -1) {
            printf("Time %d: P%d\n", t, tasks[current].id);
            tasks[current].remT--;
        } else {
            printf("Time %d: Idle\n", t);
        }
    }
}
void proportional_scheduling(Task *tasks, int n, int total_time) {
    printf("\nProportional Scheduling (priorityed Round Robin Approx.):\n");
    int time = 0;
    while (time < total_time) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < tasks[i].priority && time < total_time; j++) {
                printf("Time %d: P%d\n", time, tasks[i].id);
                time++;
            }
        }
    }
}
int main() {
    int n;
    Task *tasks;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    tasks = (Task *)malloc(n * sizeof(Task));
    if (!tasks) {
        printf("Memory allocation failed.\n");
        return 1;
    }


    insert(tasks, n);
    int hyper_period = calculate_lcm(tasks, n);
    printf("\nCalculated total simulation time (LCM of periods): %d\n", hyper_period);
    rate_monotonic(tasks, n, hyper_period);
    for (int i = 0; i < n; i++) {
        tasks[i].remT = 0;
        tasks[i].nextA = 0;
        tasks[i].nextD = 0;
    }
    earliest_deadline_first(tasks, n, hyper_period);
    proportional_scheduling(tasks, n, hyper_period);
    free(tasks);
}
```

Output:



```
Enter the number of processes: 2
Enter Process 1 - Burst Time, Period, Deadline, priority: 2 5 10 1
Enter Process 2 - Burst Time, Period, Deadline, priority: 4 10 10 2

Calculated total simulation time (LCM of periods): 10

Rate-Monotonic Scheduling:
Time 0: P1
Time 1: P1
Time 2: P2
Time 3: P2
Time 4: P2
Time 5: P1
Time 6: P1
Time 7: P2
Time 8: Idle
Time 9: Idle

Earliest-Deadline First Scheduling:
Time 0: P1
Time 1: P1
Time 2: P2
Time 3: P2
Time 4: P2
Time 5: P2
Time 6: P1
Time 7: P1
Time 8: Idle
Time 9: Idle

Proportional Scheduling (priorityed Round Robin Approx.):
Time 0: P1
Time 1: P2
Time 2: P2
Time 3: P1
Time 4: P2
Time 5: P2
Time 6: P1
Time 7: P2
Time 8: P2
Time 9: P1
```

Program 4

Question: Write a C program to simulate: a) Producer-Consumer problem using semaphores. b) Dining-Philosopher's problem

Code:

a) Producer - Consumer

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define max 5
int mutex = 1;
int full = 0;
int empty = max;
int buf[max];
int in = 0, out = 0;
void Wait(int *s) {
   while (*s <= 0)
      ;
   (*s)--;
}
void Signal(int *s) {
   (*s)++;
}
void* Producer(void* arg) {
   while (1) {
      int item = rand() % 100;
      Wait(&empty);
      Wait(&mutex);
```

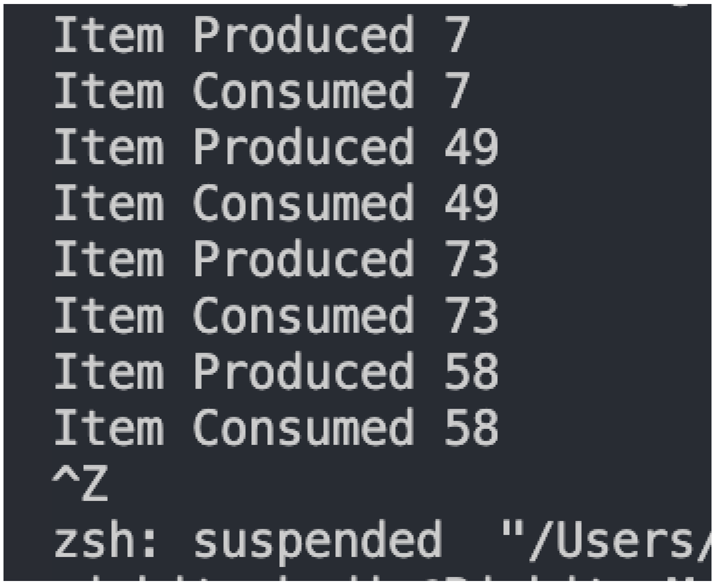
```
        full++;
        printf("Item Produced %d\n", item);
        buf[in] = item;
        in = (in + 1) % max;
        Signal(&mutex);
        Signal(&full);
        sleep(1);
    }
    return NULL;
}
void* Consumer(void* arg) {
    while (1) {
        Wait(&full);
        Wait(&mutex);
        int item = buf[out];
        printf("Item Consumed %d\n", item);
        full--;
        out = (out + 1) % max;
        Signal(&mutex);
        Signal(&empty);
        sleep(1);
    }
    return NULL;
}
int main() {
    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, Producer, NULL);
    pthread_create(&consumer_thread, NULL, Consumer, NULL);
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
    return 0;
}
```

b) Dining- Philosopher

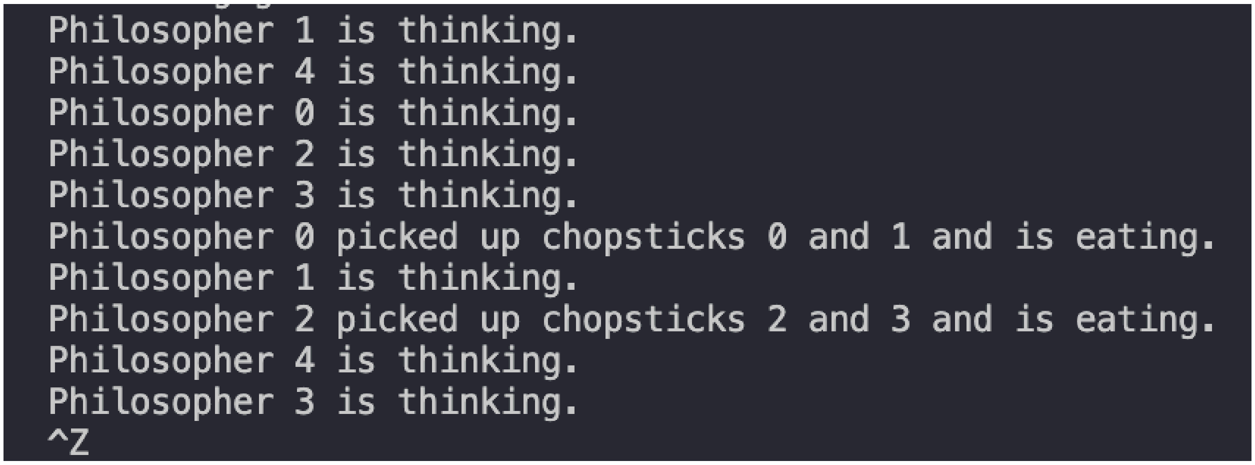
```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define MAX 5
int chopstick[MAX] = {1, 1, 1, 1, 1};  // 1 = free, 0 = taken
int mutex = 1;
int philosopher_id = 0;
void Wait(int *s) {
    while (*s <= 0);
    (*s)--;
}
void Signal(int *s) {
    (*s)++;
}
void* philosopher(void* arg) {
    int id;
    Wait(&mutex);
    id = philosopher_id++;
    Signal(&mutex);
    int left = id;
    int right = (id + 1) % MAX;
    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(1);
        Wait(&mutex);
        if (chopstick[left] && chopstick[right]) {
            chopstick[left] = chopstick[right] = 0;
            printf("Philosopher %d picked up chopsticks %d and %d and is eating.\n", id, left, right);
            Signal(&mutex);
            sleep(2); // Eating
            Wait(&mutex);
            chopstick[left] = chopstick[right] = 1;
            printf("Philosopher %d put down chopsticks %d and %d.\n", id, left, right);
            Signal(&mutex);
        } else {
            Signal(&mutex);
        }
    }
}
void main() {
    pthread_t philosophers[5];
    for (int i = 0; i < 5; i++) {
        pthread_create(&philosophers[i], NULL, philosopher, NULL);
    }
```

```
  for (int i = 0; i < 5; i++) {
    pthread_join(philosophers[i], NULL);
  }
}
```

Output:



```
Item Produced 7
Item Consumed 7
Item Produced 49
Item Consumed 49
Item Produced 73
Item Consumed 73
Item Produced 58
Item Consumed 58
^Z
zsh: suspended  "/Users/
```



```
Philosopher 1 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 0 picked up chopsticks 0 and 1 and is eating.
Philosopher 1 is thinking.
Philosopher 2 picked up chopsticks 2 and 3 and is eating.
Philosopher 4 is thinking.
Philosopher 3 is thinking.
^Z
```

Program 5

Question: Write a C program to simulate a) Bankers' algorithm for the purpose of deadlock avoidance. b) Deadlock Detection

Code:

a) Bankers Algorithm

```
#include <stdio.h>
#define MAX 10
void in(int arr[][MAX], int m, int n)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
}
void bankers(int processes, int res, int alloc[][MAX], int max[][MAX], int instances[])
{
    int finish[MAX] = {0}, safeSeq[MAX], need[MAX][MAX], work[MAX];
    for (int i = 0; i < res; i++)
        work[i] = instances[i];
    for (int i = 0; i < processes; i++)
    {
        for (int j = 0; j < res; j++)
        {
```

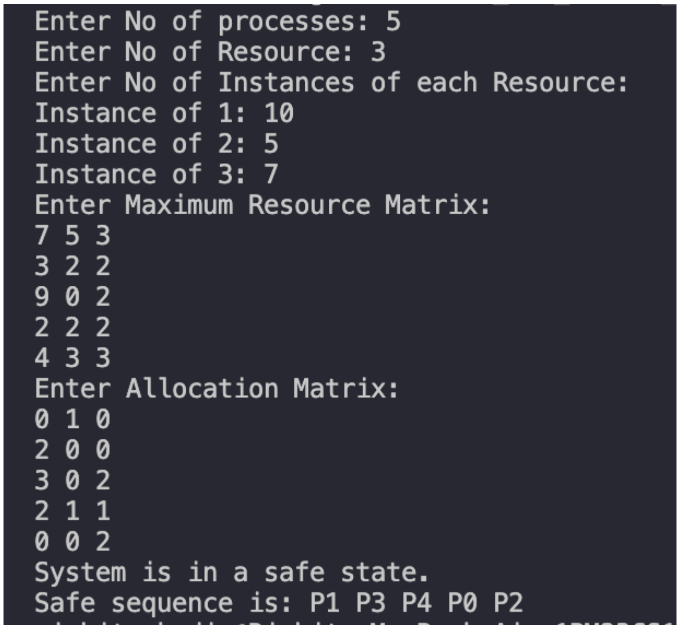
```
        need[i][j] = max[i][j] - alloc[i][j];  // Calculate Need
    }
}
for (int i = 0; i < processes; i++)
{
    for (int j = 0; j < res; j++)
    {
        work[j] -= alloc[i][j];
    }
}
int count = 0;
while (count < processes)
{
    int found = 0;
    for (int p = 0; p < processes; p++)
    {
        if (finish[p] == 0)
        {
            int j;
            for (j = 0; j < res; j++)
                if (need[p][j] > work[j])
                    break;
            if (j == res)
            {
                for (int k = 0; k < res; k++)
                    work[k] += alloc[p][k];
                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }
        if (found == 0)
        {
            printf("System is not in a safe state.\n");
            return;
        }
    }
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < processes; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n");
}
int main()
{
    int res, proce;
    printf("Enter No of processes: ");
    scanf("%d", &proce);
    printf("Enter No of Resource: ");
    scanf("%d", &res);
    int instances[MAX], allocated[MAX][MAX], max[MAX][MAX];
    printf("Enter No of Instances of each Resource: \n");
    for (int i = 0; i < res; i++)
    {
        printf("Instance of %d: ", (i + 1));
        scanf("%d", &instances[i]);
    }
    printf("Enter Maximum Resource Matrix:\n");
    in(max, proce, res);
    printf("Enter Allocation Matrix:\n");
    in(allocated, proce, res);
    bankers(proce, res, allocated, max, instances);
    return 0;
}
```

b) Deadlock Detection

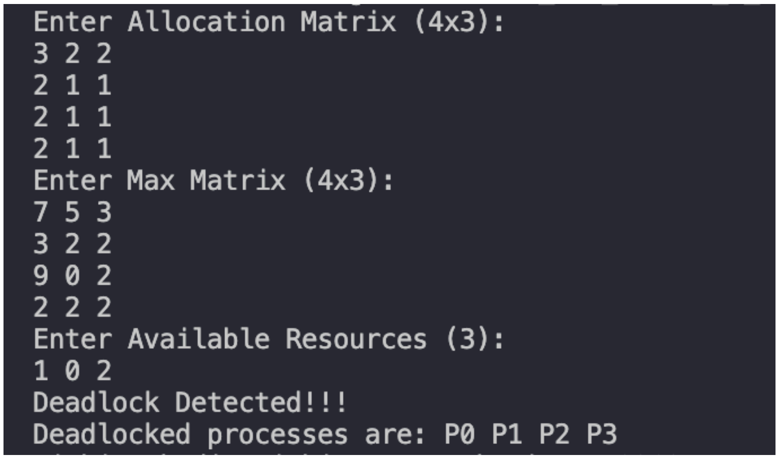
```
#include <stdio.h>
#include <stdlib.h>
#define N 4    // Number of processes
#define M 3    // Number of resources
int deadlock_detection(int alloc[N][M], int max[N][M], int avail[M]) {
    int finish[N] = {0};
    int work[M];
    int need[N][M];
    int deadlocked = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
```

```
for (int i = 0; i < M; i++) {
    work[i] = avail[i];
}
int count = 0;
while (count < N) {
    int found = 0;
    for (int i = 0; i < N; i++) {
        if (finish[i] == 0) { // Process not finished
            int j;
            for (j = 0; j < M; j++) {
                if (need[i][j] > work[j])
                    break;
            }
            if (j == M) { // If all needs can be satisfied
                for (int k = 0; k < M; k++) {
                    work[k] += alloc[i][k];
                }
                finish[i] = 1;
                found = 1;
                count++;
            }
        }
    }
    if (found == 0) {
        break; // No process could be finished
    }
    }

    for (int i = 0; i < N; i++) {
        if (finish[i] == 0) {
            deadlocked = 1;
            break;
        }
    }
    if (deadlocked) {
        printf("Deadlock Detected!!!\nDeadlocked processes are: ");
        for (int i = 0; i < N; i++)
            if (finish[i] == 0)
                printf("P%d ", i);
        printf("\n");
        return 1; // Return 1 to indicate deadlock
    } else {
        printf("Deadlock Not Detected!!!!\n");
        return 0; // Return 0 to indicate no deadlock
    }
}
int main() {
    int alloc[N][M], max[N][M], avail[M];
    printf("Enter Allocation Matrix (%dx%d):\n", N, M);
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter Max Matrix (%dx%d):\n", N, M);
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            scanf("%d", &max[i][j]);
    printf("Enter Available Resources (%d):\n", M);
    for (int i = 0; i < M; i++)
        scanf("%d", &avail[i]);
    deadlock_detection(alloc, max, avail);
}
```

Output:



```
Enter No of processes: 5
Enter No of Resource: 3
Enter No of Instances of each Resource:
Instance of 1: 10
Instance of 2: 5
Instance of 3: 7
Enter Maximum Resource Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
System is in a safe state.
Safe sequence is: P1 P3 P4 P0 P2
```



```
Enter Allocation Matrix (4x3):
3 2 2
2 1 1
2 1 1
2 1 1
Enter Max Matrix (4x3):
7 5 3
3 2 2
9 0 2
2 2 2
Enter Available Resources (3):
1 0 2
Deadlock Detected!!!
Deadlocked processes are: P0 P1 P2 P3
```

Program 6

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

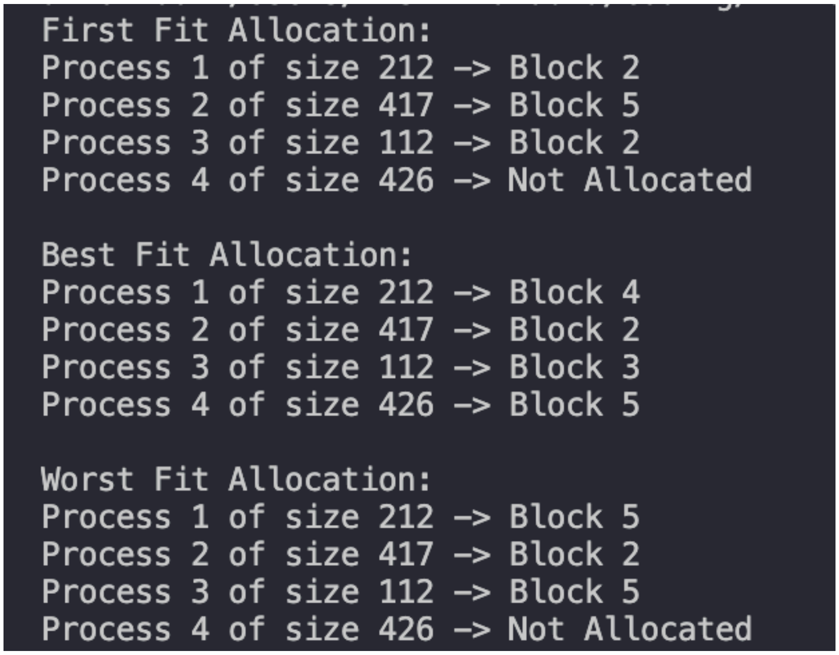
Code:

```
#include <stdio.h>

void worst_fit(int blockSize[], int blocks, int processSize[], int processes) {

  int allocation[processes];

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

    int worstIdx = -1;

    for (int j = 0; j < blocks; j++) {

      if (blockSize[j] >= processSize[i]) {

        if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx]) {

          worstIdx = j;

        }

      }

    }

    if (worstIdx != -1) {

      allocation[i] = worstIdx;

      blockSize[worstIdx] -= processSize[i];

    } else {

      allocation[i] = -1;

    }

  }

  printf("Worst Fit Allocation:\n");

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

    printf("Process %d of size %d -> ", i + 1, processSize[i]);
```

```
        if (allocation[i] != -1)
            printf("Block %d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
void best_fit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[processes];
    for (int i = 0; i < processes; i++) {
        int bestIdx = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx]) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        } else {
            allocation[i] = -1;
        }
    }
    printf("Best Fit Allocation:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d of size %d -> ", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("Block %d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
void first_fit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[processes];
    for (int i = 0; i < processes; i++) {
        allocation[i] = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }
    printf("First Fit Allocation:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d of size %d -> ", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("Block %d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
```

```
int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int n = sizeof(blockSize) / sizeof(blockSize[0]);
    int m = sizeof(processSize) / sizeof(processSize[0]);
    int blockSize1[5] = {100, 500, 200, 300, 600};
    int blockSize2[5] = {100, 500, 200, 300, 600};
    int blockSize3[5] = {100, 500, 200, 300, 600};
    first_fit(blockSize1, n, processSize, m);
    printf("\n");
    best_fit(blockSize2, n, processSize, m);
    printf("\n");
    worst_fit(blockSize3, n, processSize, m);
}
```

Output:



```
First Fit Allocation:
Process 1 of size 212 -> Block 2
Process 2 of size 417 -> Block 5
Process 3 of size 112 -> Block 2
Process 4 of size 426 -> Not Allocated

Best Fit Allocation:
Process 1 of size 212 -> Block 4
Process 2 of size 417 -> Block 2
Process 3 of size 112 -> Block 3
Process 4 of size 426 -> Block 5

Worst Fit Allocation:
Process 1 of size 212 -> Block 5
Process 2 of size 417 -> Block 2
Process 3 of size 112 -> Block 5
Process 4 of size 426 -> Not Allocated
```

Program 7

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

Code:

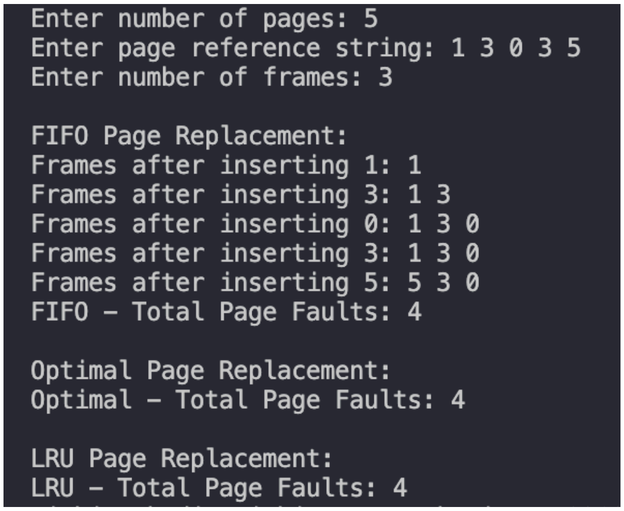
```
#include <stdio.h>

#define MAX 100

void printFrames(int frame[], int capacity) {

   for (int j = 0; j < capacity; j++) {

      if (frame[j] != -1)

         printf("%d ", frame[j]);

   }

   printf("\n");

}

void fifo(int pages[], int n, int f) {

   int frames[f], i, j, k = 0, faults = 0, hit;

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

   printf("\nFIFO Page Replacement:\n");

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

      hit = 0;

      for (j = 0; j < f; j++) {

         if (frames[j] == pages[i]) {

            hit = 1; break;

         }

      }

      if (!hit) {

         frames[k % f] = pages[i];

         k++;    faults++;
```

```
        }
        printf("Frames after inserting %d: ", pages[i]);
        printFrames(frames, f);
    }
    printf("FIFO - Total Page Faults: %d\n", faults);
}
void lru(int pages[], int n, int f) {
    int frames[f], time[f], i, j, t = 0, faults = 0, hit, pos;
    for (i = 0; i < f; i++) frames[i] = -1;
    printf("\nLRU Page Replacement:\n");
    for (i = 0; i < n; i++) {
        hit = 0;
        for (j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                hit = 1;      time[j] = t++;        break;
            }
        }
        if (!hit) {
            pos = -1;
            for (j = 0; j < f; j++) {
                if (frames[j] == -1) {
                    pos = j;
                    break;
                }
            }
            if (pos == -1) {
                pos = 0;
                for (j = 1; j < f; j++) {
                    if (time[j] < time[pos])
                        pos = j;
                }
            }
            frames[pos] = pages[i];
            time[pos] = t++;
            faults++;
        }
        printf("Frames after inserting %d: ", pages[i]);
        printFrames(frames, f);
    }
    printf("LRU - Total Page Faults: %d\n", faults);
}
void optimal(int pages[], int n, int f) {
    int frames[f], i, j, k, faults = 0, hit, farthest, index;
    for (i = 0; i < f; i++) frames[i] = -1;
    printf("\nOptimal Page Replacement:\n");
    for (i = 0; i < n; i++) {
        hit = 0;
        for (j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                hit = 1;
                break;
            }
        }
        if (!hit) {
```

```
        int found = 0;
        for (j = 0; j < f; j++) {
            if (frames[j] == -1) {
                frames[j] = pages[i];
                found = 1;
                break;
            }
        }
        if (!found) {
            farthest = -1;
            index = -1;
            for (j = 0; j < f; j++) {
                int pos;
                for (pos = i + 1; pos < n; pos++) {
                    if (frames[j] == pages[pos])     break;
                }
                if (pos > farthest) {
                    farthest = pos;        index = j;
                }
            }
            frames[index] = pages[i];
        }
        faults++;
    }
    printf("Frames after inserting %d: ", pages[i]);
    printFrames(frames, f);
}
```

```
    printf("Optimal - Total Page Faults: %d\n", faults);
}
int main() {
    int choice, n, f, i;
    int pages[MAX];
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page reference string: ");
    for (i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &f);
    fifo(pages, n, f);
    optimal(pages, n, f);
    lru(pages, n, f);
}
```

Output:



```
Enter number of pages: 5
Enter page reference string: 1 3 0 3 5
Enter number of frames: 3

FIFO Page Replacement:
Frames after inserting 1: 1
Frames after inserting 3: 1 3
Frames after inserting 0: 1 3 0
Frames after inserting 3: 1 3 0
Frames after inserting 5: 5 3 0
FIFO - Total Page Faults: 4

Optimal Page Replacement:
Optimal - Total Page Faults: 4

LRU Page Replacement:
LRU - Total Page Faults: 4
```