VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

DIPTANSHU SHEKHAR (1BM23CS361)

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 **DIPTANSHU SHEKHAR(1BM23CS361),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Radhika A D Assistant Professor Department of CSE BMSCE, Bengaluru

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

Index

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. (Any one)

a) FCFS

b) SJF

Code:

a) FCFS

```
#include <stdio.h>

int main() {
    int bt[20], wt[20], tat[20], at[20];
    float wtavg = 0, tatavg = 0;
    int n, i;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter the arrival time and burst time for each process:\n");
    for (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]);
    }
    wt[0] = 0;
    tat[0] = bt[0];
    for (i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1] - at[i];
        if (wt[i] < 0) wt[i] = 0;
        tat[i] = wt[i] + bt[i];
        wtavg += wt[i];
        tatavg += tat[i];
    }
```



```
    printf("\nFCFS Scheduling\n");
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, at[i], bt[i], wt[i],
    tat[i]); }

    printf("\nAverage Waiting Time: %.2f\n", wtavg / n);
    printf("Average Turnaround Time: %.2f\n", tatavg / n);

    return 0;
}
```

b) SJF

```
#include <stdio.h>

void main() {
    int n, i, j, temp;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int bt[n], wt[n], tat[n], at[n];
```

```
float wtavg = 0, tatavg = 0;

printf("Enter the arrival time and burst time for each process:\n");
for (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]);
}
```



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

wt[0] = 0;
for (i = 1; i < n; i++) {
    wt[i] = wt[i - 1] + bt[i - 1] - at[i];
    if (wt[i] < 0) wt[i] = 0;
}

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

```
        wtavg += wt[i];
        tatavg += tat[i];
    }

    printf("\nSJF (Non-Preemptive) Scheduling\n");
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, at[i], bt[i], wt[i],
    tat[i]); }
```



```
    printf("\nAverage Waiting Time: %.2f\n", wtavg / n);
    printf("Average Turnaround Time: %.2f\n", tatavg / n);
}
```

Result:

```
Enter the number of processes: 4
Enter the arrival time and burst time for each process:
Process 1 - Arrival Time: 1
Process 1 - Burst Time: 5
Process 2 - Arrival Time: 2
Process 2 - Burst Time: 3
Process 3 - Arrival Time: 3
Process 3 - Burst Time: 1
Process 4 - Arrival Time: 4
Process 4 - Burst Time: 7

FCFS Scheduling
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       1               5               0               5
2       2               3               3               6
3       3               1               3               4
4       4               7               0               7

Average Waiting Time: 1.50
Average Turnaround Time: 4.25
```

```
Enter the number of processes: 4
Enter the arrival time and burst time for each process:
Process 1 - Arrival Time: 1
Process 1 - Burst Time: 5
Process 2 - Arrival Time: 2
Process 2 - Burst Time: 3
Process 3 - Arrival Time: 3
Process 3 - Burst Time: 1
Process 4 - Arrival Time: 4
Process 4 - Burst Time: 7

SJF (Non-Preemptive) Scheduling
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       3               1               0               1
2       2               3               0               3
3       1               5               2               7
4       4               7               3               10

Average Waiting Time: 1.25
Average Turnaround Time: 5.25
```

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:

a)using FCFS

```
#include <stdio.h>


int main() {
    int bt[20], wt[20], tat[20], ct[20], queue[20];
    int n;
    float wtavg = 0, tatavg = 0;
    int sys_bt[20], user_bt[20];
    int sys_count = 0, user_count = 0;

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

    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Enter Queue Number (1 = System, 2 = User) for Process %d: ", i +
        1);  scanf("%d", &queue[i]);
```

```
    if (queue[i] == 1) {
        sys_bt[sys_count++] = bt[i];
    } else if (queue[i] == 2) {
        user_bt[user_count++] = bt[i];
    }
}
```


5

```
int total_count = 0;
int current_time = 0;

for (int i = 0; i < sys_count; i++) {
    if (total_count == 0) {
        wt[total_count] = 0;
    } else {
        wt[total_count] = current_time;
    }

    tat[total_count] = wt[total_count] +
    sys_bt[i];  ct[total_count] = current_time +
    sys_bt[i];

    current_time = ct[total_count];

    wtavg += wt[total_count];
    tatavg += tat[total_count];
    total_count++;
}

for (int i = 0; i < user_count; i++) {
    if (total_count == 0) {
        wt[total_count] = 0;
    } else {
        wt[total_count] = current_time;
```

```
        }

        tat[total_count] = wt[total_count] +
        user_bt[i];  ct[total_count] = current_time +
        user_bt[i];

        current_time = ct[total_count];
        wtavg += wt[total_count];
```



```
        tatavg += tat[total_count];
        total_count++;
    }

    printf("\nPROCESS\tBURST TIME\tQUEUE\tCOMPLETION TIME\tWAITING
TIME\tTURNAROUND TIME\n");
    int sys_index = 0, user_index = 0;
    for (int i = 0; i < total_count; i++) {
        if (sys_index < sys_count) {
            printf("%d\t%d\t\tSystem\t%d\t\t%d\t\t%d\n", i + 1, sys_bt[sys_index], ct[i], wt[i], tat[i]);
            sys_index++;
        } else if (user_index < user_count) {
            printf("%d\t%d\t\tUser\t%d\t\t%d\t\t%d\n", i + 1, user_bt[user_index], ct[i], wt[i],
            tat[i]);  user_index++;
        }
    }

    printf("\nAverage Waiting Time: %.2f", wtavg / total_count);
    printf("\nAverage Turnaround Time: %.2f\n", tatavg / total_count);

    return 0;
}
```

b)using Round robin

```
#include <stdio.h>
void roundRobin(int bt[], int n, int quantum, int queue[], int sys_count, int user_count)
```

```
{  int remaining_bt[20];
int wt[20] = {0}, tat[20] = {0}, ct[20] = {0};
int total_count = sys_count + user_count;
int queue_index = 0;
int current_time = 0, total_bt = 0;
float wtavg = 0, tatavg = 0;

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



```
   remaining_bt[i] = bt[i];
   total_bt += bt[i];
}

while (total_bt > 0) {
   for (int i = 0; i < total_count; i++) {
      if (remaining_bt[i] > 0) {
         int time_slice = (remaining_bt[i] <= quantum) ? remaining_bt[i] :
         quantum;  current_time += time_slice;
         remaining_bt[i] -= time_slice;
         total_bt -= time_slice;

         if (remaining_bt[i] == 0) {
            ct[i] = current_time;
            tat[i] = ct[i] - (total_bt - bt[i]);
            wt[i] = tat[i] - bt[i];
         }
      }
   }
}

for (int i = 0; i < total_count; i++) {
   wtavg += wt[i];
   tatavg += tat[i];
}
```

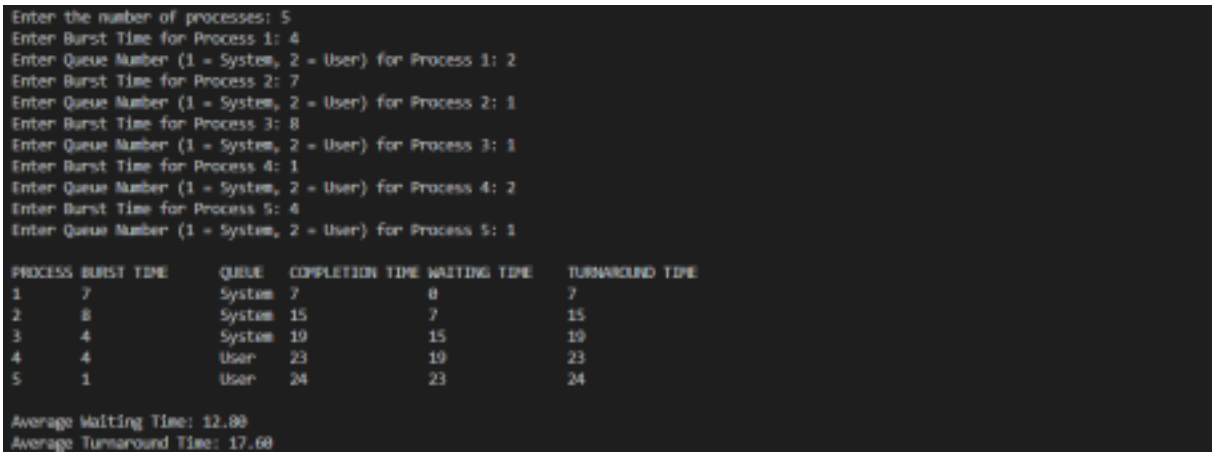
```
    printf("\nPROCESS\tBURST TIME\tQUEUE\tCOMPLETION
TIME\tWAITING TIME\tTURNAROUND TIME\n");
    for (int i = 0; i < total_count; i++) {
        if (queue[i] == 1) {
            printf("%d\t%d\t\tSystem\t%d\t\t%d\t\t%d\n", i + 1, bt[i], ct[i], wt[i],
        tat[i]); } else if (queue[i] == 2) {
            printf("%d\t%d\t\tUser\t%d\t\t%d\t\t%d\n", i + 1, bt[i], ct[i], wt[i], tat[i]);
```

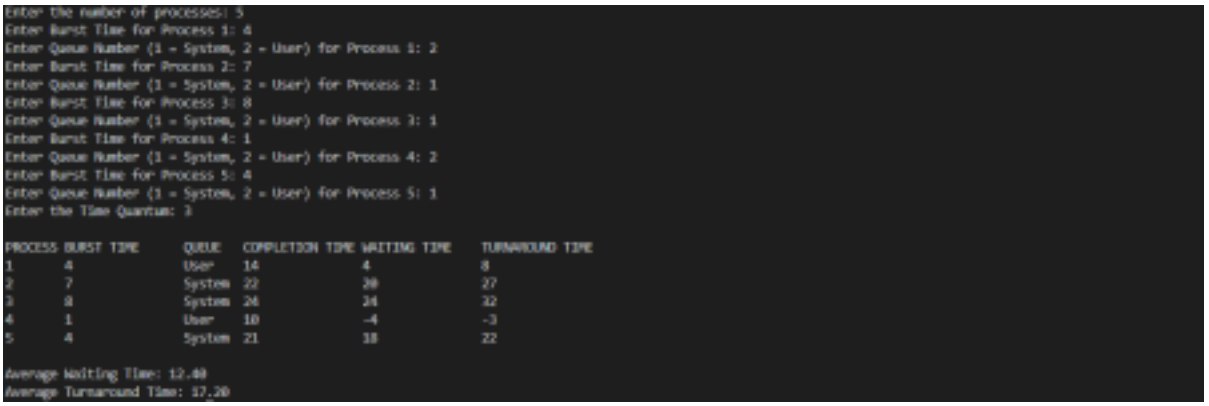


```
        }
    }

    printf("\nAverage Waiting Time: %.2f", wtavg / total_count);
    printf("\nAverage Turnaround Time: %.2f\n", tatavg /
total_count); }

int main() {
    int bt[20], queue[20];
    int n, quantum;
    int sys_count = 0, user_count = 0;

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

    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Enter Queue Number (1 = System, 2 = User) for Process %d: ", i +
        1);  scanf("%d", &queue[i]);

        if (queue[i] == 1) {
            sys_count++;
        } else if (queue[i] == 2) {
            user_count++;
        }
```

```
    }

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

    roundRobin(bt, n, quantum, queue, sys_count, user_count);

    return 0;
}
```


Result:





Program -3

Question:

Write a C program to simulate Real-Time CPU Scheduling

algorithms a) Rate- Monotonic

Code:

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

#define MAX_TASKS 10

typedefstruct {
    int id;
    int execution_time;
    int period;
    int time_remaining;
    int next_start_time;
} Task;

// Function to calculate GCD
int gcd(int a, int b) {
    return (b == 0) ? a : gcd(b, a % b);
}

// Function to calculate LCM of all task periods
int find_lcm(int periods[], int n) {
```

```
    int lcm = periods[0];
    for (int i = 1; i < n; i++) {
        lcm = (lcm * periods[i]) / gcd(lcm, periods[i]);
    }
    return lcm;
```



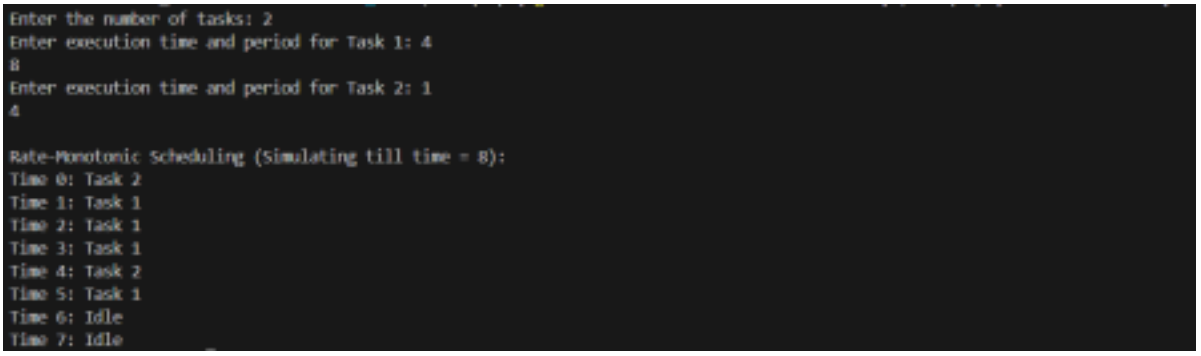
```
}

void rate_monotonic(Task tasks[], int n) {
    int periods[MAX_TASKS];
    for (int i = 0; i < n; i++) {
        periods[i] = tasks[i].period;
    }

    int simulation_time = find_lcm(periods, n); // Set simulation time to LCM of periods
    printf("\nRate-Monotonic Scheduling (Simulating till time = %d):\n",
    simulation_time);

    for (int time = 0; time < simulation_time; time++) {
        int chosen_task = -1;

        // Check if any task arrives at this time
        for (int i = 0; i < n; i++) {
            if (time == tasks[i].next_start_time) {
                tasks[i].time_remaining = tasks[i].execution_time;
                tasks[i].next_start_time += tasks[i].period;
            }
        }

        // Pick the highest-priority (shortest period) ready task
        for (int i = 0; i < n; i++) {
            if (tasks[i].time_remaining > 0) {
                if (chosen_task == -1 || tasks[i].period < tasks[chosen_task].period)
```

```
                {  chosen_task = i;
            }
        }
    }

    // Execute the chosen task or idle
```



```
    if (chosen_task != -1) {
        printf("Time %d: Task %d\n", time, tasks[chosen_task].id);
        tasks[chosen_task].time_remaining--;
    } else {
        printf("Time %d: Idle\n", time);
    }
  }
}

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

  Task tasks[MAX_TASKS];
  for (int i = 0; i < n; i++) {
    printf("Enter execution time and period for Task %d: ", i + 1);
    scanf("%d %d", &tasks[i].execution_time, &tasks[i].period);
    tasks[i].id = i + 1;
    tasks[i].time_remaining = 0;
    tasks[i].next_start_time = 0;
  }

  rate_monotonic(tasks, n);
  return 0;
}
```

Result:



Program -4

Question:

Write a C program to simulate:

a) Producer-Consumer problem using

semaphores. b) Dining-Philosopher's problem

Code:

a) Producer-Consumer problem using

semaphores. #include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

```
#define MAX_ITEMS 5
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t mutex;
sem_t full;
sem_t empty;
int produced_count = 0, consumed_count = 0;

void *producer(void *arg) {
    sem_wait(&empty);
    sem_wait(&mutex);

    buffer[in] = produced_count + 1;
    printf("Producer has produced: Item %d\n", buffer[in]);
    in = (in + 1) % BUFFER_SIZE;
    produced_count++;
    sem_post(&mutex);
```



```
    sem_post(&full);
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    sem_wait(&full);
    sem_wait(&mutex);

    int last_item_index = (in - 1 + BUFFER_SIZE) % BUFFER_SIZE;
    printf("Consumer has consumed: Item %d\n",
    buffer[last_item_index]);  buffer[last_item_index] = 0;
    consumed_count++;
```

```
    in = (in - 1 + BUFFER_SIZE) % BUFFER_SIZE;

    sem_post(&mutex);
    sem_post(&empty);
    pthread_exit(NULL);
}

int main() {
    pthread_t prod_thread, cons_thread;
    int choice;

    sem_init(&mutex, 0, 1);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, MAX_ITEMS);
    while (1) {
        printf("Enter 1.Producer 2.Consumer 3.exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
```



```
        switch (choice) {
            case 1:
                if (produced_count < MAX_ITEMS) {
                    pthread_create(&prod_thread, NULL, producer, NULL);
                    pthread_join(prod_thread, NULL);
                } else {
                    printf("Buffer is full. Cannot produce more
                items.\n"); }
                break;
            case 2:
                if (consumed_count < produced_count) {
                    pthread_create(&cons_thread, NULL, consumer, NULL);
```

```
                pthread_join(cons_thread, NULL);
            } else {
                printf("Buffer is empty. Cannot consume more
            items.\n"); }
            break;
        case 3:
            sem_destroy(&mutex);
            sem_destroy(&full);
            sem_destroy(&empty);
            return 0;
        default:
            printf("Invalid choice.\n");
    }
  }

  return 0;
}
```

b) Dining-Philosopher's problem

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



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

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

int state[NUM_PHILOSOPHERS];
int phil_ids[NUM_PHILOSOPHERS];

sem_t mutex;
```

```
sem_t S[NUM_PHILOSOPHERS];

void test(int i) {
    if (state[i] == HUNGRY &&
        state[(i + 4) % NUM_PHILOSOPHERS] != EATING &&
        state[(i + 1) % NUM_PHILOSOPHERS] != EATING) {

        state[i] = EATING;
        sleep(1);
        printf("Philosopher %d takes forks %d and %d and starts eating\n", i + 1, (i + 4)
%  NUM_PHILOSOPHERS + 1, i + 1);
        sem_post(&S[i]);
    }
}

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



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

void put_fork(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    printf("Philosopher %d puts down forks %d and %d and starts thinking\n", i + 1, (i + 4)
%  NUM_PHILOSOPHERS + 1, i + 1);
    test((i + 4) % NUM_PHILOSOPHERS);
    test((i + 1) % NUM_PHILOSOPHERS);
    sem_post(&mutex);
}
```

```
void* philosopher(void* num) {
    int i = *(int*)num;

    while (1) {
        printf("Philosopher %d is thinking\n", i + 1);
        sleep(1);
        take_fork(i);
        sleep(2);
        put_fork(i);
    }

    return NULL;
}

int main() {
    int i;
    pthread_t thread_id[NUM_PHILOSOPHERS];
    sem_init(&mutex, 0, 1);
```



```
for (i = 0; i < NUM_PHILOSOPHERS; i++) {
sem_init(&S[i], 0, 0);
phil_ids[i] = i;
}

for (i = 0; i < NUM_PHILOSOPHERS; i++) {
pthread_create(&thread_id[i], NULL, philosopher, &phil_ids[i]);
printf("Philosopher %d is seated at the table\n", i + 1);
}

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

return 0;

}

Result:

```
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 1
Producer has produced: Item 1
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 1
Producer has produced: Item 2
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 1
Producer has produced: Item 3
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 1
Producer has produced: Item 4
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 2
Consumer has consumed: Item 4
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 2
Consumer has consumed: Item 3
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 2
Consumer has consumed: Item 2
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 2
Consumer has consumed: Item 1
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 2
Buffer is empty. Cannot consume more items.
Enter 1.Producer 2.Consumer 3.exit
Enter choice: 3
```



```
Philosopher 1 is seated at the table
Philosopher 2 is seated at the table
Philosopher 3 is seated at the table
Philosopher 4 is seated at the table
Philosopher 5 is seated at the table
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is hungry
Philosopher 3 takes forks 2 and 3 and starts eating
Philosopher 1 is hungry
Philosopher 1 takes forks 5 and 1 and starts eating
Philosopher 2 is hungry
Philosopher 5 is hungry
Philosopher 4 is hungry
Philosopher 3 puts down forks 2 and 3 and starts thinking
Philosopher 4 takes forks 3 and 4 and starts eating
Philosopher 3 is thinking
Philosopher 1 puts down forks 5 and 1 and starts thinking
Philosopher 2 takes forks 1 and 2 and starts eating
Philosopher 3 is hungry
Philosopher 1 is thinking
Philosopher 1 is hungry
Philosopher 4 puts down forks 3 and 4 and starts thinking
Philosopher 5 takes forks 4 and 5 and starts eating
Philosopher 4 is thinking
Philosopher 2 puts down forks 1 and 2 and starts thinking
Philosopher 3 takes forks 2 and 3 and starts eating
Philosopher 2 is thinking
Philosopher 4 is hungry
Philosopher 2 is hungry
Philosopher 5 puts down forks 4 and 5 and starts thinking
Philosopher 1 takes forks 5 and 1 and starts eating
Philosopher 5 is thinking
Philosopher 3 puts down forks 2 and 3 and starts thinking
Philosopher 4 takes forks 3 and 4 and starts eating
```


Program -5

Question:

Write a C program to simulate:

a) Bankers' algorithm for the purpose of deadlock avoidance. **Code:**

```
#include<stdio.h>
int main()
{
int n, m, i, j, k;
printf("Enter the number of processes: ");
scanf("%d", &n);
printf("Enter the number of resources: ");
scanf("%d", &m);
int allocation[n][m];
printf("Enter the Allocation Matrix:\n");
for (i = 0; i < n; i++){
   for (j = 0; j < m; j++){
      scanf("%d", &allocation[i][j]);
   }
}
int max[n][m];
printf("Enter the MAX Matrix:\n");
for (i = 0; i < n; i++){
   for (j = 0; j < m; j++){
      scanf("%d", &max[i][j]);
   }
}
int available[m];
printf("Enter the Available Resources:\n");
for (i = 0; i < m; i++){
   scanf("%d", &available[i]);
```



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

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



```
int flag = 1;
for (i = 0; i < n; i++){
if (f[i] == 0){
flag = 0;
```

```
printf("The following system is not safe\n");
break;
}
}
if (flag == 1){
printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++){
printf(" P%d ->", ans[i]);
}
printf(" P%d\n", ans[n - 1]);
}
return 0;
}
```

Result:

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>
#define MAX 10
```

```
void firstFit(int blockSize[], int blocks, int processSize[], int processes)
   {  int allocation[MAX];
   for (int i = 0; i < processes; i++) allocation[i] = -1;

   for (int i = 0; i < processes; i++) {
      for (int j = 0; j < blocks; j++) {
         if (blockSize[j] >= processSize[i]) {
            allocation[i] = j;
            blockSize[j] -= processSize[i];
            break;
         }
      }
   }

   printf("\nFirst-Fit Allocation:\n");
   for (int i = 0; i < processes; i++) {
      if (allocation[i] != -1)
         printf("Process %d of size %d -> Block %d\n", i + 1, processSize[i], allocation[i] + 1);
      else
         printf("Process %d of size %d -> Not Allocated\n", i + 1,
   processSize[i]); }
```



```
}

void bestFit(int blockSize[], int blocks, int processSize[], int processes)
   {  int allocation[MAX];
   for (int i = 0; i < processes; i++) allocation[i] = -1;

   for (int i = 0; i < processes; i++) {
      int best = -1;
      for (int j = 0; j < blocks; j++) {
         if (blockSize[j] >= processSize[i]) {
```

```
            if (best == -1 || blockSize[j] < blockSize[best]) best = j;
        }
    }
    if (best != -1) {
        allocation[i] = best;
        blockSize[best] -= processSize[i];
    }
  }

  printf("\nBest-Fit Allocation:\n");
  for (int i = 0; i < processes; i++) {
    if (allocation[i] != -1)
        printf("Process %d of size %d -> Block %d\n", i + 1, processSize[i], allocation[i] + 1);
    else
        printf("Process %d of size %d -> Not Allocated\n", i + 1,
  processSize[i]); }
}

void worstFit(int blockSize[], int blocks, int processSize[], int processes)
  {  int allocation[MAX];
  for (int i = 0; i < processes; i++) allocation[i] = -1;
```



```
  for (int i = 0; i < processes; i++) {
    int worst = -1;
    for (int j = 0; j < blocks; j++) {
        if (blockSize[j] >= processSize[i]) {
            if (worst == -1 || blockSize[j] > blockSize[worst]) worst = j;
        }
    }
    if (worst != -1) {
        allocation[i] = worst;
        blockSize[worst] -= processSize[i];
    }
```

```
    }

    printf("\nWorst-Fit Allocation:\n");
    for (int i = 0; i < processes; i++) {
        if (allocation[i] != -1)
            printf("Process %d of size %d -> Block %d\n", i + 1, processSize[i], allocation[i] + 1);
        else
            printf("Process %d of size %d -> Not Allocated\n", i + 1,
    processSize[i]); }
}

int main() {
    int blockSize[MAX], processSize[MAX], blocks, processes, choice;

    printf("Enter number of memory blocks: ");
    scanf("%d", &blocks);
    printf("Enter size of each block:\n");
    for (int i = 0; i < blocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }
```



```
    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter size of each process:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    printf("\nMemory Allocation Techniques:\n");
    printf("1. First Fit\n2. Best Fit\n3. Worst Fit\nEnter choice: ");
    scanf("%d", &choice);
```

```
    int originalBlockSize[MAX];
    for (int i = 0; i < blocks; i++) originalBlockSize[i] = blockSize[i];

    switch (choice) {
        case 1:
            firstFit(originalBlockSize, blocks, processSize,
            processes);  break;
        case 2:
            for (int i = 0; i < blocks; i++) blockSize[i] =
            originalBlockSize[i];  bestFit(blockSize, blocks, processSize,
            processes);
            break;
        case 3:
            for (int i = 0; i < blocks; i++) blockSize[i] =
            originalBlockSize[i];  worstFit(blockSize, blocks, processSize,
            processes);
            break;
        default:
            printf("Invalid choice.\n");
    }

    return 0;
```



```
}
```

Result:

```
Enter number of memory blocks: 5
Enter size of each block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter number of processes: 4
Enter size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Memory Allocation Techniques:
1. First Fit
2. Best Fit
1. Worst Fit
Enter choice: 1

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
```

```
Enter number of memory blocks: 5
Enter size of each block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter number of processes: 4
Enter size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Memory Allocation Techniques:
1. First Fit
2. Best Fit
1. Worst Fit
Enter choice: 2

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
```

```
Enter number of memory blocks: 5
Enter size of each block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter number of processes: 4
Enter size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Memory Allocation Techniques:
1. First Fit
2. Best Fit
1. Worst Fit
Enter choice: 3

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>
#include <limits.h>

void fifo(int pages[], int n, int capacity) {
    int frames[capacity], index = 0, faults = 0;
    for (int i = 0; i < capacity; i++) frames[i] = -1;

    printf("\nFIFO PageReplacement\n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            frames[index] = pages[i];
            index = (index + 1) % capacity;
            faults++;
        }

        printf("Frames: ");
                                                    for (int j = 0; j < capacity; j++) {
```



```
            if (frames[j] == -1)
                printf(" - ");
            else
                printf(" %d ", frames[j]);
        }
        printf("\n");
    }
```

```
    printf("Total Page Faults: %d\n", faults);
}

void lru(int pages[], int n, int capacity) {
    int frames[capacity], recent[capacity], faults =

    0; for (int i = 0; i < capacity; i++) frames[i] =

    -1;

    printf("\nLRU PageReplacement\n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frames[j] == pages[i]) {
                recent[j] = i;
                found = 1;
                break;
            }
        }

        if (!found) {
            int lru_index = 0;
            for (int j = 1; j < capacity; j++) {
                if (frames[j] == -1 || recent[j] < recent[lru_index])
                    lru_index = j;
            }
```



```
            frames[lru_index] = pages[i];
            recent[lru_index] = i;
            faults++;
        }
```

```
        printf("Frames: ");
        for (int j = 0; j < capacity; j++) {
            if (frames[j] == -1)
                printf(" - ");
            else
                printf(" %d ", frames[j]);
        }
        printf("\n");
    }
    printf("Total Page Faults: %d\n",
faults); }

void optimal(int pages[], int n, int capacity) {
    int frames[capacity], faults = 0;

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

    printf("\nOptimal PageReplacement\n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }
```



```
        if (!found){
            int opt_index = -1, farthest = i;
            for (intj= 0;j< capacity; j++){ if
            (frames[j] ==-1) {
                    opt_index = j;
```

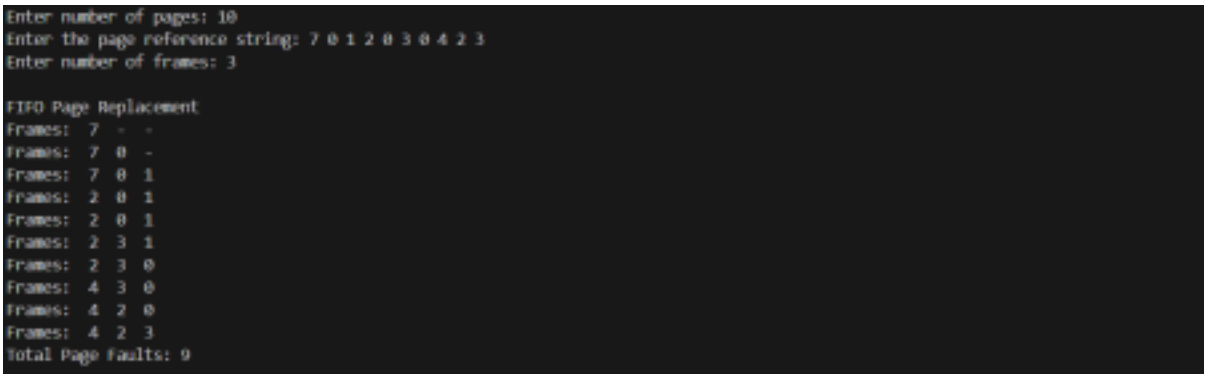
```
            break;
        }
        int next_use= INT_MAX;
        for (intk=i+ 1;k< n; k++) {  if
            (frames[j] == pages[k]) {
            next_use= k;
            break;
        }
        }
        if (next_use> farthest) {
            farthest = next_use;
            opt_index = j;
        }
    }
    frames[opt_index]= pages[i];
    faults++;
  }

  printf("Frames: ");
  for (intj= 0;j< capacity; j++) {
    if (frames[j] == -1)
      printf("- ");
    else
      printf(" %d ", frames[j]);
  }
  printf("\n");
}
```

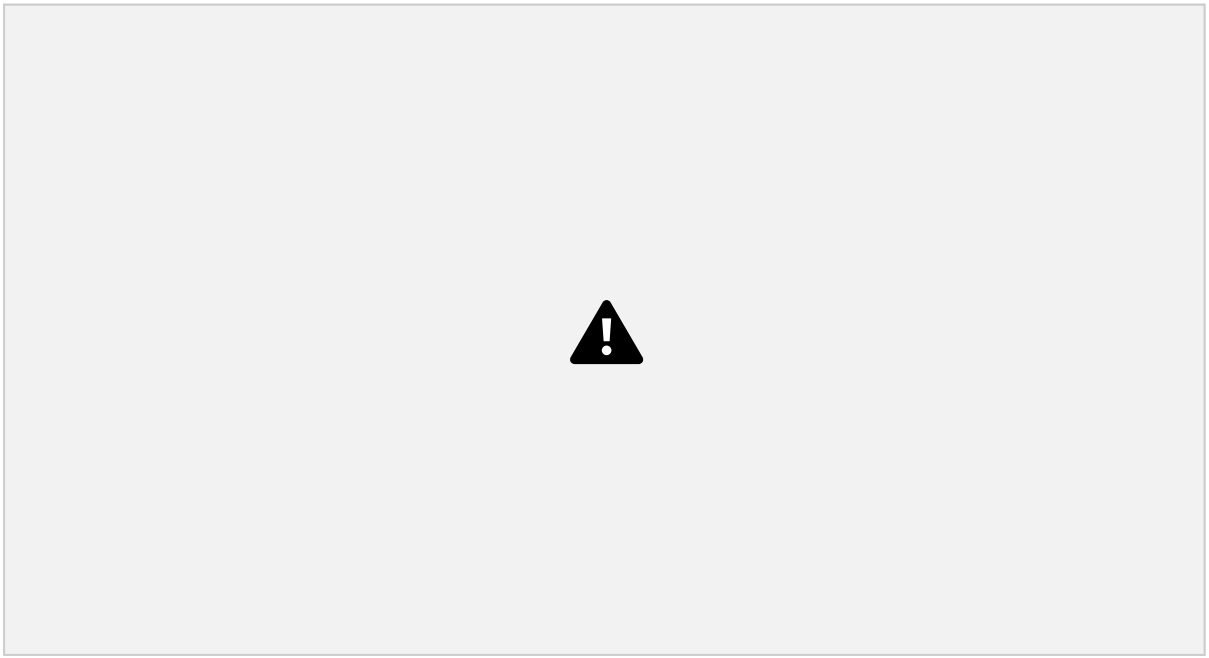


```
printf("Total Page Faults: %d\n", faults);
}

int main() {
int n, capacity;
printf("Enter number of pages: ");
```

```
scanf("%d", &n);
int pages[n];
printf("Enter the page reference string: ");
for (int i = 0; i < n; i++) scanf("%d", &pages[i]);
printf("Enter number of frames: ");
scanf("%d", &capacity);

fifo(pages, n, capacity);
lru(pages, n, capacity);
optimal(pages, n, capacity);

return 0;
}
```

Result:





Program -8

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

Code:

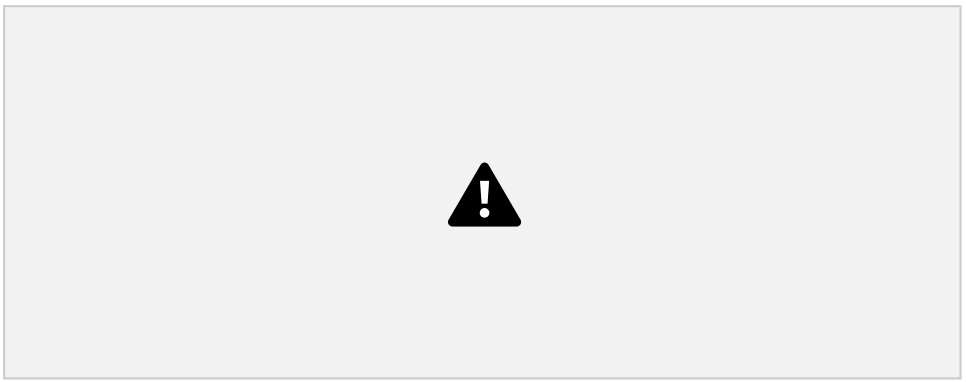
```
#include <stdio.h>

int main() {
    int memory[100], i, start, length, j, n;

    for (i = 0; i < 100; i++)
        memory[i] = 0;

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

    for (i = 0; i < n; i++) {
        printf("Enter starting block and length of file %d: ", i +
        1);  scanf("%d %d", &start, &length);

        int flag = 0;
        for (j = start; j < start + length; j++) {
```

```
            if (memory[j] != 0) {
                flag = 1;
                break;
            }
        }

        if (flag == 0) {
```



```
for (j = start; j < start + length; j++)
memory[j] = i + 1;
printf("File %d allocated successfully.\n", i + 1);
} else {
printf("File %d cannot be allocated.\n", i + 1);
}
}

printf("\nMemory Allocation:\n");
for (i = 0; i < 100; i++) {
printf("%d ", memory[i]);
if ((i + 1) % 10 == 0)
printf("\n");
}

return 0;
}
```

Result:


37