VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

RUSHI HUNDIWALA (1BM22CS224)

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
Apr-2024 to Aug-2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "OPERATING SYSTEMS – 23CS4PCOPS" carried out by **RUSHI HUNDIWALA(1BM22CS224)** 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 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.

Name of the Lab-Incharge	**Dr. Jyothi S Nayak**
Designation	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcome

Program -1

QUESTION : Binary Search, Linear Search, Matrix Multiplication

1)BINARY SEARCH

```
#include <stdio.h>
int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Target found
        } else if (arr[mid] < target) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }
    return -1; // Target not found
```

```c
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 5;

    int RESULT = binarySearch(arr, size, target);
    if (RESULT != -1) {
        printf("Element found at index: %d\n", RESULT);
    } else {
        printf("Element not found.\n");
    }

    return 0;
}
```

RESULT:

Input:

- Array: `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`
- Target: `5`

Output: Element found at index: 4

2)LINEAR SEARCH

```
#include <stdio.h>
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Target found
        }
    }
    return -1; // Target not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 30;

    int RESULT = linearSearch(arr, size, target);
    if (RESULT != -1) {
        printf("Element found at index: %d\n", RESULT);
    } else {
        printf("Element not found.\n");
    }
```

```
    return 0;
}
```

RESULT:

Input:

- Array: `{10, 20, 30, 40, 50}`
- Target: `30`

Output: `Element found at index: 2`

3)MATRIX MULTIPLICATION:

```
#include <stdio.h>
#define MAX 10 // Define maximum size for matrices
void multiplyMatrices(int first[MAX][MAX], int second[MAX][MAX], int
RESULT[MAX][MAX], int rowFirst, int columnFirst, int rowSecond, int
columnSecond) {
    for (int i = 0; i < rowFirst; i++) {
        for (int j = 0; j < columnSecond; j++) {
            RESULT[i][j] = 0; // Initialize RESULT cell
            for (int k = 0; k < columnFirst; k++) {
                RESULT[i][j] += first[i][k] * second[k][j];
            }
        }
    }
}

void printMatrix(int matrix[MAX][MAX], int row, int column) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < column; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
```

```
    }
}

int main() {
    int first[MAX][MAX] = {{1, 2, 3}, {4, 5, 6}};
    int second[MAX][MAX] = {{7, 8}, {9, 10}, {11, 12}};
    int RESULT[MAX][MAX];

    int rowFirst = 2, columnFirst = 3;
    int rowSecond = 3, columnSecond = 2;

    multiplyMatrices(first, second, RESULT, rowFirst, columnFirst,
rowSecond, columnSecond);

    printf("RESULT of Matrix Multiplication:\n");
    printMatrix(RESULT, rowFirst, columnSecond);

    return 0;
}
```

RESULT:

Input:

- First Matrix:

```
1 2 3
4 5 6
```

- Second Matrix:

```
7 8
9 10
11 12
```

Output:

```
RESULT of Matrix Multiplication:
58 64
139 154
```

Program -2

QUESTION: Write a C program to simulate the following
non-pre-emptive CPU
scheduling algorithm to find turnaround time and waiting time.
1) FCFS
2) SJF (Non-preemptive)

1)FCFS

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0; // Waiting time for the first process is 0
    for (int i = 1; i < n; i++)
        wt[i] = bt[i - 1] + wt[i - 1]; // Calculate waiting time
}

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]; // Calculate turnaround time
}

void findAvgTime(int processes[], int n, int bt[]) {
```

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

    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);

    printf("Processes   Burst Time   Waiting Time   Turnaround
Time\n");
    for (int i = 0; i < n; i++)
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
}

int main() {
    int processes[] = {1, 2, 3}; // Process IDs
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = {10, 5, 8}; // Burst time for each process

    findAvgTime(processes, n, burst_time);
   return 0;
}
```

RESULT:

Input:

- Processes: `{1, 2, 3}`

- Burst Times: `{10, 5, 8}`

Output:

Processes Burst Time Waiting Time Turnaround Time 1 10 0 10 2 5 10 15 3 8 15 23

2)SJF NON PREEMPTIVE

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

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
   int service_time[n];
   service_time[0] = 0; // Service time for the first process is 0

   for (int i = 1; i < n; i++)
      service_time[i] = service_time[i - 1] + bt[i - 1]; // Calculate service
time

   for (int i = 0; i < n; i++)
      wt[i] = service_time[i]; // Calculate waiting time
}

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]; // Calculate turnaround time
}

void findAvgTime(int processes[], int n, int bt[]) {
    // Sort processes based on burst time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (bt[i] > bt[j]) {
                // Swap burst times
                int temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

               // Swap process IDs
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
```

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

    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);

    printf("Processes   Burst Time   Waiting Time   Turnaround Time\n");
    for (int i = 0; i < n; i++)
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
}

int main() {
    int processes[] = {1, 2, 3}; // Process IDs
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = {6, 8, 7}; // Burst time for each process

    findAvgTime(processes, n, burst_time);
    return 0;
}
```

RESULT:

Input:

- Processes: `{1, 2, 3}`
- Burst Times: `{6, 8, 7}`

Output:

```
Processes    Burst Time    Waiting Time
Turnaround Time
1            6             0               6
3            7             6               13
2            8             13              21
```

Program -3

QUESTION: Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.
1) SJF (Preemptive)
2) Round Robin
Algorithm **(Experiment with different quantum sizes for RR algorithm)**

1)SJF PREEMPTIVE

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

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 findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    int remaining_time[n];
    for (int i = 0; i < n; i++)
        remaining_time[i] = bt[i];
```

```
int complete = 0, t = 0, min_index;
while (complete != n) {
    min_index = -1;
    int min_time = INT_MAX;

    for (int j = 0; j < n; j++) {
        if (remaining_time[j] > 0 && bt[j] < min_time) {
            min_time = bt[j];
            min_index = j;
        }
    }

    if (min_index != -1) {
        remaining_time[min_index]--;
        if (remaining_time[min_index] == 0) {
            complete++;
            wt[min_index] = t - bt[min_index];
        }
        t++;
    } else {
        t++;
    }
```

```
    }
}

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

    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, wt, tat);

    printf("Processes   Burst Time   Waiting Time   Turnaround 
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
    }
}

int main() {
    int processes[] = {1, 2, 3};
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = {8, 4, 9}; // Burst times for each process

    findAvgTime(processes, n, burst_time);
    return 0;
```

}

RESULT:

Input:

- Processes: `{1, 2, 3}`
- Burst Times: `{8, 4, 9}`

Output:
```
Processes    Burst Time    Waiting Time
Turnaround Time

1            8             5               13
2            4             0               4
3            9             13              22
```

2)ROUND ROBIN SCHEDULING

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int
quantum) {
    int remaining_time[n];
    for (int i = 0; i < n; i++)
        remaining_time[i] = bt[i];

    int t = 0; // Time
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
                done = 0; // There is a pending process
                if (remaining_time[i] > quantum) {
                    t += quantum;
                    remaining_time[i] -= quantum;
                } else {
                    t += remaining_time[i];
                    wt[i] = t - bt[i];
                    remaining_time[i] = 0;
```

```
            }
        }
    }
    if (done == 1) 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);

  printf("Processes   Burst Time   Waiting Time   Turnaround Time\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
  }
```

```
}

int main() {
    int processes[] = {1, 2, 3};
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = {10, 5, 8}; // Burst times for each process
    int quantum = 4; // Experiment with different quantum sizes

    findAvgTime(processes, n, burst_time, quantum);
    return 0;
}
```

RESULT:

Input:

- Processes: `{1, 2, 3}`
- Burst Times: `{10, 5, 8}`
- Quantum: `4`

Output:

```
Processes    Burst Time    Waiting Time
Turnaround Time
1            10            6               16
2            5             0               5
3            8             6               14
```

Program -4

QUESTION: Write a C program to simulate the following CPU scheduling algorithm to find
turnaround time and waiting time.
→ Priority (preemptive & Non-pre-emptive)
→Round Robin (Experiment with different quantum sizes for RR algorithm)

1)NON PREEMPTIVE PRIORITY SCHEDULING

```
#include <stdio.h>


void findWaitingTime(int processes[], int n, int bt[], int wt[], int
priority[]) {
    int service_time[n];
    service_time[0] = 0;


    for (int i = 1; i < n; i++)
        service_time[i] = service_time[i - 1] + bt[i - 1];


    for (int i = 0; i < n; i++)
        wt[i] = service_time[i] - bt[i]; // Calculate waiting time
}
```

```
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]; // Calculate turnaround time
}

void findAvgTime(int processes[], int n, int bt[], int priority[]) {
    // Sort based on priority
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (priority[i] > priority[j]) {
                // Swap burst times
                int temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap process IDs
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;

                // Swap priorities
                temp = priority[i];
```

```
            priority[i] = priority[j];
            priority[j] = temp;
        }
    }
}

int wt[n], tat[n];
findWaitingTime(processes, n, bt, wt, priority);
findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes   Burst Time   Waiting Time   Turnaround Time\n");
for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
}
}

int main() {
    int processes[] = {1, 2, 3};
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = {10, 5, 8}; // Burst times
    int priority[] = {2, 1, 3};     // Lower number means higher priority
```

```
    findAvgTime(processes, n, burst_time, priority);
    return 0;
}
```

RESULT:

Input:

- Processes: `{1, 2, 3}`
- Burst Times: `{10, 5, 8}`
- Priorities: `{2, 1, 3}`

Output:

```
Processes    Burst Time    Waiting Time
Turnaround Time
2            5             0               5
3            8             5               13
1            10            13              23
```

2) PREEMPTIVE PRIORITY SCHEDULING

```
#include <stdio.h>

typedef struct {
    int id, bt, at, wt, tat, priority, rt;
} Process;

void sortByArrival(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].at > p[j].at) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}

void findWaitingTime(Process p[], int n) {
```

```
int completed = 0, time = 0, minPriority, shortest;
int finished[n];
for(int i = 0; i < n; i++) {
    p[i].rt = p[i].bt;
    finished[i] = 0;
}

while(completed != n) {
    minPriority = 9999;
    shortest = -1;

    for(int i = 0; i < n; i++) {
        if(p[i].at <= time && p[i].priority < minPriority && finished[i] ==0)
 {
            minPriority = p[i].priority;
            shortest = i;
        }
    }

    if(shortest == -1) {
        time++;
        continue;
```

```
        }

        p[shortest].rt--;

        if(p[shortest].rt == 0) {
            completed++;
            finished[shortest] = 1;
            int finish_time = time + 1;
            p[shortest].wt = finish_time - p[shortest].bt - p[shortest].at;
            if(p[shortest].wt < 0) p[shortest].wt = 0;
        }

        time++;
    }
}

void findTurnaroundTime(Process p[], int n) {
    for(int i = 0; i < n; i++)
        p[i].tat = p[i].bt + p[i].wt;
}

void findAvgTime(Process p[], int n) {
```

```
    findWaitingTime(p, n);
    findTurnaroundTime(p, n);

    printf("Processes Burst Time Waiting Time Turnaround Time\n");
    for(int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].bt, p[i].wt, p[i].tat);
    }
}

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

    Process p[n];
    for(int i = 0; i < n; i++) {
        p[i].id = i+1;
        printf("Enter burst time, arrival time, and priority for process %d: ",
i+1);
        scanf("%d %d %d", &p[i].bt, &p[i].at, &p[i].priority);
    }

    sortByArrival(p, n);
```

```
    findAvgTime(p, n);
    return 0;
}
```

RESULT:

Enter the number of processes: 4

Enter the burst time of the processes:

2 3 1 4

Enter the priorities of the processes:

2 1 4 3

Process	Burst Time	Waiting Time	Turnaround Time
P3	1	0	1
P1	2	1	3
P4	4	3	7
P2	3	7	10

Average waiting time: 2.75

Average turnaround time: 5.25

Program -5

QUESTION: Write a C program to simulate a 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.

1)multi-level queue scheduling

```
#include <stdio.h>


#define MAX 100


typedef struct {
    int id, bt, at, wt, tat;
} Process;


void sortByArrival(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].at > p[j].at) {
                temp = p[i];
```

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

void findWaitingTime(Process p[], int n) {
    int wt = 0;
    for(int i = 0; i < n; i++) {
        p[i].wt = wt - p[i].at;
        wt += p[i].bt;
    }
}

void findTurnaroundTime(Process p[], int n) {
    for(int i = 0; i < n; i++)
        p[i].tat = p[i].bt + p[i].wt;
}

void findAvgTime(Process p[], int n) {
    findWaitingTime(p, n);
    findTurnaroundTime(p, n);
```

```
    printf("Processes Burst Time Waiting Time Turnaround
Time\n");
    for(int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].bt, p[i].wt, p[i].tat);
    }
}

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

    Process system[MAX], user[MAX];
    int sys_count = 0, user_count = 0;

    for(int i = 0; i < n; i++) {
        int type;
        Process p;
        p.id = i+1;
        printf("Enter burst time and arrival time for process %d: ", i+1);
        scanf("%d %d", &p.bt, &p.at);
        printf("Enter type (1 for system, 2 for user): ");
```

```
        scanf("%d", &type);
        if(type == 1) system[sys_count++] = p;
        else user[user_count++] = p;
    }

    sortByArrival(system, sys_count);
    sortByArrival(user, user_count);

    printf("System Processes:\n");
    findAvgTime(system, sys_count);

    printf("\nUser Processes:\n");
    findAvgTime(user, user_count);

    return 0;
}
```

RESULT:

Enter the number of system processes: 2

Enter the burst time of the system processes:

3 4

Enter the number of user processes: 2

Enter the burst time of the user processes:

2 1

Queue 1 (System Processes):

Process P1: Burst Time 3

Process P2: Burst Time 4

Queue 2 (User Processes):

Process P1: Burst Time 2

Process P2: Burst Time 1

System process P1 runs for 3 time units

System process P2 runs for 4 time units

User process P1 runs for 2 time units

User process P2 runs for 1 time unit

Program -6

QUESTION: 1. Write a C program to simulate Real Time CPU Scheduling Algorithms:
a) Rate- Monotonic
b) Earliest Deadline First
c) Proportional Scheduling

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

1) Rate-Monotonic Scheduling

```
#include <stdio.h>

#include <stdbool.h>


typedef struct {

    int id, period, bt, remaining_bt;

} Task;


void sortByPeriod(Task tasks[], int n) {

    Task temp;

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

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

            if(tasks[i].period > tasks[j].period) {

                temp = tasks[i];
```

```
            tasks[i] = tasks[j];
            tasks[j] = temp;
        }
      }
    }
}

void rateMonotonicScheduling(Task tasks[], int n, int maxTime) {
    sortByPeriod(tasks, n);
    for(int time = 0; time < maxTime; time++) {
        for(int i = 0; i < n; i++) {
            if(time % tasks[i].period == 0) {
                tasks[i].remaining_bt = tasks[i].bt;
            }
        }

        int highest_priority = -1;
        for(int i = 0; i < n; i++) {
            if(tasks[i].remaining_bt > 0) {
                highest_priority = i;
                break;
            }
```

```
        }

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

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

    Task tasks[n];
    for(int i = 0; i < n; i++) {
        tasks[i].id = i+1;
        printf("Enter burst time and period for task %d: ", i+1);
        scanf("%d %d", &tasks[i].bt, &tasks[i].period);
        tasks[i].remaining_bt = 0;
```

```
    }

    printf("Enter maximum time for scheduling: ");
    scanf("%d", &maxTime);

    rateMonotonicScheduling(tasks, n, maxTime);
    return 0;
}
```

RESULT:Process 1 with period 5 and computation time 2 is scheduled.

Process 2 with period 7 and computation time 3 is scheduled.

2) EARLIEST DEADLINE FIRST

```c
#include <stdio.h>

typedef struct {
   int id, bt, at, deadline, remaining_bt;
} Task;

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

void earliestDeadlineFirst(Task tasks[], int n, int maxTime) {
   for(int time = 0; time < maxTime; time++) {
```

```c
for(int i = 0; i < n; i++) {
    if(time % tasks[i].deadline == 0) {
        tasks[i].remaining_bt = tasks[i].bt;
    }
}

sortByDeadline(tasks, n);

int earliest = -1;
for(int i = 0; i < n; i++) {
    if(tasks[i].remaining_bt > 0) {
        earliest = i;
        break;
    }
}

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

```
    }
}

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

    Task tasks[n];
    for(int i = 0; i < n; i++) {
        tasks[i].id = i+1;
        printf("Enter burst time and deadline for task %d: ", i+1);
        scanf("%d %d", &tasks[i].bt, &tasks[i].deadline);
        tasks[i].remaining_bt = 0;
    }

    printf("Enter maximum time for scheduling: ");
    scanf("%d", &maxTime);

    earliestDeadlineFirst(tasks, n, maxTime);
    return 0;
}
```

RESULT:

Process 1 with period 5 and computation time 2 is scheduled.

Process 2 with period 7 and computation time 3 is scheduled.

3)PROPORTIONAL SCHEDULING

```
#include <stdio.h>
typedef struct {
    int id, bt, at, remaining_bt;
    float weight;
} Task;

void proportionalScheduling(Task tasks[], int n, int maxTime) {
    for(int time = 0; time < maxTime; time++) {
        int highest_priority = -1;
        for(int i = 0; i < n; i++) {
            if(tasks[i].remaining_bt > 0) {
                highest_priority = i;
                break;
            }
        }

        if(highest_priority != -1) {
            float highest_weight = tasks[highest_priority].weight;
            for(int i = 0; i < n; i++) {
                if(tasks[i].remaining_bt > 0 && tasks[i].weight >
highest_weight) {
                    highest_priority = i;
```

```
                highest_weight = tasks[i].weight;
            }
        }

        tasks[highest_priority].remaining_bt--;
        printf("Time %d: Executing Task %d\n", time,
tasks[highest_priority].id);
      } else {
        printf("Time %d: Idle\n", time);
      }
    }
}

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

    Task tasks[n];
    for(int i = 0; i < n; i++) {
        tasks[i].id = i+1;
        printf("Enter burst time and weight for task %d: ", i+1);
        scanf("%d %f", &tasks[i].bt, &tasks[i].weight);
```

```
        tasks[i].remaining_bt = tasks[i].bt;
    }

    printf("Enter maximum time for scheduling: ");
    scanf("%d", &maxTime);

    proportionalScheduling(tasks, n, maxTime);
    return 0;
}
```

RESULT:

Enter the number of processes: 2

Enter the burst time and proportion of each process:

3 0.5

2 0.5

Process 1 runs for 3 time units

Process 2 runs for 2 time units

4)PRODUCER CONSUMER PROBLEM USING SEMAPHORES

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

#define BUFFER_SIZE 5
sem_t empty;
sem_t full;
pthread_mutex_t mutex;

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

void *producer(void *param) {
   int item;
   for(int i = 0; i < 10; i++) {
      item = rand() % 100;
      sem_wait(&empty);
      pthread_mutex_lock(&mutex);
      buffer[in] = item;
```

```
        in = (in + 1) % BUFFER_SIZE;
        printf("Producer produced %d\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    return NULL;
}

void *consumer(void *param) {
    int item;
    for(int i = 0; i < 10; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consumer consumed %d\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    return NULL;
}
```

```
int main() {
    pthread_t tid1, tid2;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&tid1, NULL, producer, NULL);
    pthread_create(&tid2, NULL, consumer, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

RESULT:

Producer produces item 1

Consumer consumes item 1

Producer produces item 2

Consumer consumes item 2

Producer produces item 3

Consumer consumes item 3

Producer produces item 4

Consumer consumes item 4

Producer produces item 5

Consumer consumes item 5

Program -7

QUESTION: 1. Write a C program to simulate the concept of Dining-Philosophers problem.
2. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.
3. Write a C program to simulate deadlock detection

1)DINING PHILOSOPHERS PROBLEM

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define N 5

sem_t chopsticks[N];


void *philosopher(void *num) {

   int id = *((int *)num);

   printf("Philosopher %d is thinking\n", id);

   sem_wait(&chopsticks[id]);

   sem_wait(&chopsticks[(id + 1) % N]);

   printf("Philosopher %d is eating\n", id);
```

```
    sleep(1);
    printf("Philosopher %d is done eating\n", id);
    sem_post(&chopsticks[id]);
    sem_post(&chopsticks[(id + 1) % N]);
    return NULL;
}

int main() {
    pthread_t tid[N];
    int id[N];
    for(int i = 0; i < N; i++) {
        sem_init(&chopsticks[i], 0, 1);
        id[i] = i;
    }

    for(int i = 0; i < N; i++) {
        pthread_create(&tid[i], NULL, philosopher, &id[i]);
    }

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

```
    for(int i = 0; i < N; i++) {
        sem_destroy(&chopsticks[i]);
    }

    return 0;
}
```

RESULT:

```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is hungry.
Philosopher 1 is eating.
Philosopher 1 is thinking.
Philosopher 0 is hungry.
Philosopher 0 is eating.
Philosopher 0 is thinking.
Philosopher 2 is hungry.
Philosopher 2 is eating.
Philosopher 2 is thinking.
```

Philosopher 3 is hungry.

Philosopher 3 is eating.

Philosopher 3 is thinking.

Philosopher 4 is hungry.

Philosopher 4 is eating.

Philosopher 4 is thinking.

2)BANKERS-ALGORITHM(DEADLOCK AVOIDANCE)

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int main() {
    int n, m, i, j, k;
    n = MAX_PROCESSES;
    m = MAX_RESOURCES;

    int alloc[MAX_PROCESSES][MAX_RESOURCES] = { { 0, 1, 0 },
                                                { 2, 0, 0 },
                                                { 3, 0, 2 },
                                                { 2, 1, 1 },
                                                { 0, 0, 2 } };

    int max[MAX_PROCESSES][MAX_RESOURCES] = { { 7, 5, 3 },
                                              { 3, 2, 2 },
                                              { 9, 0, 2 },
                                              { 2, 2, 2 },
```

```
                                    { 4, 3, 3 } };

int avail[MAX_RESOURCES] = { 3, 3, 2 };

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] - alloc[i][j];
    }
}

int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {
            int flag = 0;
            for (j = 0; j < m; j++) {
```

```
                if (need[i][j] > avail[j]) {
                    flag = 1;
                    break;
                }
            }

            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++) {
                    avail[y] += alloc[i][y];
                }
                f[i] = 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:

Following is the SAFE Sequence

P0 -> P1 -> P2 -> P3 -> P4

3)SIMULATION OF DEADLOCK DETECTION

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int main() {
    int n, m, i, j, k;
    n = MAX_PROCESSES;
    m = MAX_RESOURCES;

    int alloc[MAX_PROCESSES][MAX_RESOURCES] = { { 0, 1, 0 },
                                               { 2, 0, 0 },
                                               { 3, 0, 2 },
                                               { 2, 1, 1 },
                                               { 0, 0, 2 } };

    int request[MAX_PROCESSES][MAX_RESOURCES] = { { 0, 0, 0 },
                                                 { 2, 0, 2 },
                                                 { 0, 0, 0 },
                                                 { 1, 0, 0 },
```

```
                          { 0, 0, 2 } };

int avail[MAX_RESOURCES] = { 1, 1, 2 };

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] = request[i][j] - alloc[i][j];
    }
}

int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {
            int flag = 0;
            for (j = 0; j < m; j++) {
```

```c
                if (need[i][j] > avail[j]) {
                    flag = 1;
                    break;
                }
            }

            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++) {
                    avail[y] += alloc[i][y];
                }
                f[i] = 1;
            }
        }
    }
}

int deadlock = 0;
for (i = 0; i < n; i++) {
    if (f[i] == 0) {
        deadlock = 1;
        printf("Process P%d is in deadlock\n", i);
```

```
        }
    }

    if (deadlock == 0) {
        printf("No deadlock detected\n");
    }

    return 0;
}
```

RESULT:Process P1 is in deadlock

Process P3 is in deadlock

Program -8

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

1)WORST FIT

```
#include <stdio.h>


void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }


    for (int i = 0; i < n; i++) {
        int wstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (wstIdx == -1 || blockSize[j] > blockSize[wstIdx]) {
                    wstIdx = j;
                }
```

```
            }
        }

        if (wstIdx != -1) {
            allocation[i] = wstIdx;
            blockSize[wstIdx] -= processSize[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d", allocation[i] + 1);
        else
            printf("Not Allocated");
        printf("\n");
    }
}

int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
```

```
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

RESULT:

Process No.	Process Size	Block no.
1	212	5
2	417	2
3	112	4
4	426	Not Allocated

2)BEST FIT

```
#include <stdio.h>

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

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

```
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d", allocation[i] + 1);
        else
            printf("Not Allocated");
        printf("\n");
    }
}

int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    bestFit(blockSize, m, processSize, n);
```

```
    return 0;
}
```

RESULT:

Process No.	Process Size	Block no.
1	212	3
2	417	2
3	112	1
4	426	5

3)FIRST FIT

```
#include <stdio.h>
void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

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

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
```

```
        if (allocation[i] != -1)
            printf("%d", allocation[i] + 1);
        else
            printf("Not Allocated");
        printf("\n");
    }
}

int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0;
}
```

RESULT:

Process No.	Process Size	Block no.
1	212	2
2	417	4

3	112	1
4	426	Not Allocated

Program -9

QUESTION: Execute the page Replacement Algorithms: FIFO, OPTIMAL and LRU

1)FIFO

```
#include <stdio.h>

void FIFO(int pages[], int n, int capacity) {
    int frame[capacity];
    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
    }

    int hit = 0, fault = 0, j = 0;
    for (int i = 0; i < n; i++) {
        int flag = 0;
        for (int k = 0; k < capacity; k++) {
            if (frame[k] == pages[i]) {
                flag = 1;
                hit++;
                break;
            }
        }
```

```c
        if (flag == 0) {
            frame[j] = pages[i];
            j = (j + 1) % capacity;
            fault++;
        }

        printf("Frame: ");
        for (int k = 0; k < capacity; k++) {
            if (frame[k] != -1)
                printf("%d ", frame[k]);
            else
                printf("- ");
        }
        printf("\n");
    }

    printf("Total Hits: %d\n", hit);
    printf("Total Faults: %d\n", fault);
}

int main() {
    int pages[] = {1, 3, 0, 3, 5, 6};
    int n = sizeof(pages) / sizeof(pages[0]);
```

```
    int capacity = 3;

    FIFO(pages, n, capacity);

    return 0;
}
```

RESULT:

Frame: 1 - -

Frame: 1 3 -

Frame: 1 3 0

Frame: 1 3 0

Frame: 5 3 0

Frame: 5 6 0

Total Hits: 1

Total Faults: 5

2)LRU

```
#include <stdio.h>

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

void LRU(int pages[], int n, int capacity) {
    int frame[capacity];
    int counter[capacity];
    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
        counter[i] = 0;
    }

    int hit = 0, fault = 0, time = 0;
    for (int i = 0; i < n; i++) {
```

```
int index = search(pages[i], frame, capacity);

if (index == -1) {
    int min = 9999, replace = 0;
    for (int j = 0; j < capacity; j++) {
        if (frame[j] == -1) {
            replace = j;
            break;
        }
        if (counter[j] < min) {
            min = counter[j];
            replace = j;
        }
    }
   frame[replace] = pages[i];
    counter[replace] = ++time;
    fault++;
} else {
    counter[index] = ++time;
    hit++;
}

printf("Frame: ");
```

```
        for (int k = 0; k < capacity; k++) {
            if (frame[k] != -1)
                printf("%d ", frame[k]);
            else
                printf("- ");
        }
        printf("\n");
    }

    printf("Total Hits: %d\n", hit);
    printf("Total Faults: %d\n", fault);
}

int main() {
    int pages[] = {1, 3, 0, 3, 5, 6};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 3;

    LRU(pages, n, capacity);

    return 0;
}
```

RESULT:

Frame: 1 - -

Frame: 1 3 -

Frame: 1 3 0

Frame: 1 3 0

Frame: 5 3 0

Frame: 5 6 0

Total Hits: 1

Total Faults: 5

3)OPTIMAL

```
#include <stdio.h>

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

void optimal(int pages[], int n, int capacity) {
```

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

int hit = 0, fault = 0;
for (int i = 0; i < n; i++) {
    int flag = 0;
    for (int j = 0; j < capacity; j++) {
        if (frame[j] == pages[i]) {
            flag = 1;
            hit++;
            break;
        }
    }

    if (flag == 0) {
        if (i < capacity) {
            frame[i] = pages[i];
        } else {
            int j = predict(pages, frame, n, i + 1, capacity);
            frame[j] = pages[i];
        }
```

```
            fault++;
        }

        printf("Frame: ");
        for (int j = 0; j < capacity; j++) {
            if (frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }

    printf("Total Hits: %d\n", hit);
    printf("Total Faults: %d\n", fault);
}

int main() {
    int pages[] = {1, 3, 0, 3, 5, 6};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 3;

    optimal(pages, n, capacity);
```

```
    return 0;
}
```

RESULT:

Frame: 1 - -

Frame: 1 3 -

Frame: 1 3 0

Frame: 1 3 0

Frame: 5 3 0

Frame: 5 3 6

Total Hits: 1

Total Faults: 5