VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

NANDINI A T(1BM23CS411)

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 **NANDINI A T (1BM23CS411),** 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.

Name of the Lab-Incharge **Dr. Jyothi S Nayak**

Designation
Department of CSE
BMSCE, Bengaluru

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

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

Code:

a.FCFS

```
#include <stdio.h>
#define MAX 10
void fcfs(int n, int at[], int bt[]) {
   int ct[MAX];
   int tat[MAX];
   int wt[MAX];
   int total_wt = 0;
   int total_tat = 0;
   int current_time = 0;
   for (int i = 0; i < n; i++) {
      ct[i] = -1;
   }
   for (int i = 0; i < n; i++) {
      if (current_time < at[i]) {
         current_time = at[i];
      }
      ct[i] = current_time + bt[i];
      current_time = ct[i];
   }
   for (int i = 0; i < n; i++) {
      tat[i] = ct[i] - at[i];
      total_tat += tat[i];
   }
   for (int i = 0; i < n; i++) {
```

```
        wt[i] = tat[i] - bt[i];
        total_wt += wt[i];
    }
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage waiting time: %.2f", (float)total_wt / n);
    printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}
int main() {
    int n, i;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n];
    printf("Enter the arrival time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }
    printf("Enter the burst time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
    }
    fcfs(n, at, bt);
    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter the arrival time:
0 1 5 6
Enter the burst time:
2 2 3 4

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       0               2               2               2               0
2       1               2               4               3               1
3       5               3               8               3               0
4       6               4               12              6               2

Average waiting time: 0.75
Average turnaround time: 3.50
Process returned 0 (0x0)   execution time : 16.999 s
Press any key to continue.
```

b. SJF (pre-emptive)

Code:

```
#include <stdio.h>

#define MAX 10

void sjf_preemptive(int n, int at[], int bt[]) {
   int ct[MAX];
   int tat[MAX];
   int wt[MAX];
   int rt[MAX];
   int total_wt = 0;
   int total_tat = 0;
   int completed = 0;
   int current_time = 0;
   int is_completed[MAX] = {0};
   for (int i = 0; i < n; i++) {
      rt[i] = bt[i];
   }
  while (completed < n) {
```

```
        int shortest_job = -1;
        int min_bt = 9999;
  for (int i = 0; i < n; i++) {
            if (at[i] <= current_time && rt[i] < min_bt && rt[i] > 0) {
                shortest_job = i;
                min_bt = rt[i];
            }
        }
 if (shortest_job == -1) {
            current_time++;
            continue;
        }
 rt[shortest_job]--;
 if (rt[shortest_job] == 0) {
            completed++;
ct[shortest_job] = current_time + 1;
 tat[shortest_job] = ct[shortest_job] - at[shortest_job];
            total_tat += tat[shortest_job];


            wt[shortest_job] = tat[shortest_job] - bt[shortest_job];
            if (wt[shortest_job] < 0) wt[shortest_job] = 0;
            total_wt += wt[shortest_job];
     is_completed[shortest_job] = 1;
        }
    current_time++;
    }
  printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
  for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage waiting time: %.2f", (float)total_wt / n);
```

```
    printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}
int main() {
    int n, i;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n];
   printf("Enter the arrival time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }
    printf("Enter the burst time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
    }
    sjf_preemptive(n, at, bt);
     return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter the arrival time:
2 1 4 0 2
Enter the burst time:
1 5 1 6 3

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       2               1               3               1               0
2       1               5               11              10              5
3       4               1               5               1               0
4       0               6               16              16              10
5       2               3               7               5               2

Average waiting time: 3.40
Average turnaround time: 6.60
Process returned 0 (0x0)   execution time : 21.791 s
Press any key to continue.
```

c.SJF (Non-preemptive)

Code:

```
#include <stdio.h>
#define MAX 10
void sjf_non_preemptive(int n, int at[], int bt[]) {
    int ct[MAX];
    int tat[MAX];
    int wt[MAX];
    int rt[MAX];
    int total_wt = 0;
    int total_tat = 0;
    int completed = 0;
    int current_time = 0;
    int shortest_job = 0;
    int min_bt = 9999;
    int is_completed[MAX] = {0};
    for (int i = 0; i < n; i++) {
        rt[i] = bt[i];
    }
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (at[i] <= current_time && rt[i] < min_bt && !is_completed[i]) {
                shortest_job = i;
                min_bt = rt[i];
            }
        }
        rt[shortest_job]--;
        if (rt[shortest_job] == 0) {
            completed++;
            min_bt = 9999;
            is_completed[shortest_job] = 1;
            ct[shortest_job] = current_time + 1;
            tat[shortest_job] = ct[shortest_job] - at[shortest_job];
```

```
            total_tat += tat[shortest_job];
            wt[shortest_job] = tat[shortest_job] - bt[shortest_job];
            if (wt[shortest_job] < 0) wt[shortest_job] = 0;
            total_wt += wt[shortest_job];
        }
        current_time++;
    }
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("\nAverage waiting time: %.2f", (float)total_wt / n);
    printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}
int main() {
    int n, i;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n];
    printf("Enter the arrival time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }
    printf("Enter the burst time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
    }
sjf_non_preemptive(n, at, bt);
    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter the arrival time:
0 0 0 0
Enter the burst time:
6 8 7 3

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       0               6               9               9               3
2       0               8               24              24              16
3       0               7               16              16              9
4       0               3               3               3               0

Average waiting time: 7.00
Average turnaround time: 13.00
Process returned 0 (0x0)   execution time : 18.046 s
Press any key to continue.
```

Program 2

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

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

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

Program:

a.Priority (pre-emptive)

Code:

```
#include<stdio.h>
void sort (int proc_id[], int p[], int at[], int bt[], int b[], int n){
  int min = p[0], temp = 0;
  for (int i = 0; i < n; i++){
            min = p[i];
            for (int j = i; j < n; j++){
                        if (p[j] < min){
                                    temp = at[i];
                                    at[i] = at[j];
                                    at[j] = temp;
                                    temp = bt[j];
                                    bt[j] = bt[i];
                                    bt[i] = temp;
                                    temp = b[j];
                                    b[j] = b[i];
                                    b[i] = temp;
                                    temp = p[j];
                                    p[j] = p[i];
                                    p[i] = temp;
                                    temp = proc_id[i];
                                    proc_id[i] = proc_id[j];
                                    proc_id[j] = temp;
                                   }
```

```
                    }
            }
}
void main (){
  int n, c = 0;
  printf ("Enter number of processes: ");
  scanf ("%d", &n);
  int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], m[n], b[n], rt[n], p[n];
  double avg_tat = 0.0, ttat = 0.0, avg_wt = 0.0, twt = 0.0;
  for (int i = 0; i < n; i++){
              proc_id[i] = i + 1;
              m[i] = 0;
            }
  printf ("Enter priorities:\n");
  for (int i = 0; i < n; i++)
            scanf ("%d", &p[i]);
  printf ("Enter arrival times:\n");
  for (int i = 0; i < n; i++)
            scanf ("%d", &at[i]);
  printf ("Enter burst times:\n");
  for (int i = 0; i < n; i++){
              scanf ("%d", &bt[i]);
              b[i] = bt[i];
              m[i] = -1;
              rt[i] = -1;
            }
sort(proc_id, p, at, bt, b, n);
  int count = 0, pro = 0, priority = p[0];
  int x = 0;
  c = 0;
  while (count < n){
```

```
        for (int i = 0; i < n; i++){
                if (at[i] <= c && p[i] >= priority && b[i] > 0 && m[i] != 1){
                        x = i;
                        priority = p[i];
                       }
               }
        if (b[x] > 0){
                if (rt[x] == -1)
                        rt[x] = c - at[x];
                b[x]--;
                c++;
               }
        if (b[x] == 0){
                count++;
                ct[x] = c;
                m[x] = 1;
                while (x >= 1 && b[x] == 0)
                        priority = p[--x];
               }
         if (count == n)
                break;
       }
for (int i = 0; i < n; i++)
        tat[i] = ct[i] - at[i];
for (int i = 0; i < n; i++)
        wt[i] = tat[i] - bt[i];
printf ("Priority scheduling(Pre-Emptive):\n");
printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++)
        printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], p[i], at[i],
                        bt[i], ct[i], tat[i], wt[i], rt[i]);
```

```
 for (int i = 0; i < n; i++){
            ttat += tat[i];
            twt += wt[i];
          }
 avg_tat = ttat / (double) n;
 avg_wt = twt / (double) n;
 printf ("\nAverage turnaround time:%lfms\n", avg_tat);
 printf ("\nAverage waiting time:%lfms\n", avg_wt);
}
```

Output:

```
Enter number of processes: 4
Enter priorities:
10 20 30 40
Enter arrival times:
0 1 2 4
Enter burst times:
5 4 2 1
Priority scheduling(Pre-Emptive):
PID     Prior   AT      BT      CT      TAT     WT      RT
P1       10             0       5       12      12      7       0
P2       20             1       4       8       7       3       0
P3       30             2       2       4       2       0       0
P4       40             4       1       5       1       0       0

Average turnaround time:5.500000ms

Average waiting time:2.500000ms

Process returned 33 (0x21)   execution time : 22.246 s
Press any key to continue.
```

b. Priority (Non-pre-emptive)

Code:

```
#include<stdio.h>
void sort (int proc_id[], int p[], int at[], int bt[], int n){
 int min = p[0], temp = 0;
 for (int i = 0; i < n; i++)
        {
          min = p[i];
```

```
            for (int j = i; j < n; j++)
                    {
                     if (p[j] < min)
                            {
                             temp = at[i];
                             at[i] = at[j];
                             at[j] = temp;
                             temp = bt[j];
                             bt[j] = bt[i];
                             bt[i] = temp;
                             temp = p[j];
                             p[j] = p[i];
                             p[i] = temp;
                             temp = proc_id[i];
                             proc_id[i] = proc_id[j];
                             proc_id[j] = temp;
                            }
                    }
           }
}
void main (){
  int n, c = 0;
  printf ("Enter number of processes: ");
  scanf ("%d", &n);
  int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], m[n], rt[n], p[n];
  double avg_tat = 0.0, ttat = 0.0, avg_wt = 0.0, twt = 0.0;
  for (int i = 0; i < n; i++){
            proc_id[i] = i + 1;
            m[i] = 0;
  }
  printf ("Enter priorities:\n");
```

```
for (int i = 0; i < n; i++)
        scanf ("%d", &p[i]);
printf ("Enter arrival times:\n");
for (int i = 0; i < n; i++)
        scanf ("%d", &at[i]);
printf ("Enter burst times:\n");
for (int i = 0; i < n; i++)
        {
         scanf ("%d", &bt[i]);
         m[i] = -1;
         rt[i] = -1;
        }
sort (proc_id, p, at, bt, n);
int count = 0, pro = 0, priority = p[0];
int x = 0;
c = 0;
while (count < n){
        for (int i = 0; i < n; i++){
                if (at[i] <= c && p[i] >= priority && m[i] != 1){
                        x = i;
                        priority = p[i];
                }
        }
        if (rt[x] == -1)
          rt[x] = c - at[x];
        if (at[x] <= c)
          c += bt[x];
        else
      c += at[x] - c + bt[x];
         count++;
         ct[x] = c;
```

```
            m[x] = 1;
            while (x >= 1 && m[--x] != 1)    {
                        priority = p[x];
                        break;
            }
            x++;
            if (count == n)
                        break;
           }
  for (int i = 0; i < n; i++)
           tat[i] = ct[i] - at[i];
  for (int i = 0; i < n; i++)
           wt[i] = tat[i] - bt[i];
  printf ("\nPriority scheduling:\n");
  printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");
  for (int i = 0; i < n; i++)
           printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], p[i], at[i],
                                bt[i], ct[i], tat[i], wt[i], rt[i]);
  for (int i = 0; i < n; i++){
            ttat += tat[i];
            twt += wt[i];
           }
  avg_tat = ttat / (double) n;
  avg_wt = twt / (double) n;
  printf ("\nAverage turnaround time:%lfms\n", avg_tat);
  printf ("\nAverage waiting time:%lfms\n", avg_wt);
}
```

Output:

```
Enter number of processes: 4
Enter priorities:
10 20 30 40
Enter arrival times:
0 1 2 4
Enter burst times:
5 4 2 1

Priority scheduling:
PID     Prior   AT      BT      CT      TAT     WT      RT
P1       10             0       5       5       5       0       0
P2       20             1       4       12      11      7       7
P3       30             2       2       8       6       4       4
P4       40             4       1       6       2       1       1

Average turnaround time:6.000000ms

Average waiting time:3.000000ms

Process returned 33 (0x21)   execution time : 22.748 s
Press any key to continue.
```

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

```
#include <stdio.h>
#define MAX 10
void round_robin(int n, int bt[], int quantum) {
   int wt[MAX] = {0};
   int tat[MAX] = {0};
   int remaining_bt[MAX];
   int total_wt = 0, total_tat = 0;
   int time = 0;
   for (int i = 0; i < n; i++) {
      remaining_bt[i] = bt[i];
   }
   while (1) {
      int done = 1;
      for (int i = 0; i < n; i++) {
         if (remaining_bt[i] > 0) {
```

```
            done = 0;
            if (remaining_bt[i] > quantum) {
                time += quantum;
                remaining_bt[i] -= quantum;
            } else {
                time += remaining_bt[i];
                wt[i] = time - bt[i];
                remaining_bt[i] = 0;
            }
        }
    }
    if (done == 1) break;
  }
  for (int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
    total_wt += wt[i];
    total_tat += tat[i];
  }
  printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
  }
  printf("\nAverage waiting time: %.2f", (float)total_wt / n);
  printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}
int main() {
  int n, quantum;
  printf("Enter the number of processes: ");
  scanf("%d", &n);
  int bt[MAX];
  printf("Enter Burst Time for each process:\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &bt[i]);
    }
    printf("Enter the size of time slice (quantum): ");
    scanf("%d", &quantum);
    round_robin(n, bt, quantum);
    return 0;
}
```

Output:

```
Enter the number of processes: 3
Enter Burst Time for each process:
Process 1: 24
Process 2: 3
Process 3: 3
Enter the size of time slice (quantum): 3

Process Burst Time      Waiting Time    Turnaround Time
1       24              6               30
2       3               3               6
3       3               6               9

Average waiting time: 5.00
Average turnaround time: 15.00
Process returned 0 (0x0)   execution time : 19.434 s
Press any key to continue.
```

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

Code:

```
#include <stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[]) {
   wt[0] = 0;
   for (int i = 1; i < n; i++) {
      wt[i] = bt[i-1] + wt[i-1] - at[i-1];
      if (wt[i] < 0)
         wt[i] = 0;
   }
}
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 roundRobin(int processes[], int n, int bt[], int at[], int quantum) {
   int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
   int remaining_bt[n];
   int completed = 0;
   int time = 0;
   for (int i = 0; i < n; i++) {
      remaining_bt[i] = bt[i];
   }
   while (completed < n) {
      for (int i = 0; i < n; i++) {
```

```
            if (remaining_bt[i] > 0 && at[i] <= time) {
                if (remaining_bt[i] <= quantum) {
                    time += remaining_bt[i];
                    remaining_bt[i] = 0;
                    ct[i] = time;
                    completed++;
                } else {
                    time += quantum;
                    remaining_bt[i] -= quantum;
                }
            }
        }
    }
    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);
    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("Average Waiting Time (Round Robin) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (Round Robin) = %f\n", (float)total_tat / n);
}
void fcfs(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);
    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        ct[i] = at[i] + bt[i];
```

```
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("Average Waiting Time (FCFS) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (FCFS) = %f\n", (float)total_tat / n);
}
int main() {
    int processes[] = {1, 2, 3, 4, 5};
    int n = sizeof(processes) / sizeof(processes[0]);
    int bt[] = {10, 5, 8, 12, 15};
    int at[] = {0, 1, 2, 3, 4};
    int quantum = 2;
    roundRobin(processes, n, bt, at, quantum);
    fcfs(processes, n, bt, at);
    return 0;
}
```

Output:

```
Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time
P1              10             0              0               10              39
P2              5              1              10              15              23
P3              8              2              14              22              33
P4              12             3              20              32              45
P5              15             4              29              44              50
Average Waiting Time (Round Robin) = 14.600000
Average Turnaround Time (Round Robin) = 24.600000
Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time
P1              10             0              0               10              10
P2              5              1              10              15              6
P3              8              2              14              22              10
P4              12             3              20              32              15
P5              15             4              29              44              19
Average Waiting Time (FCFS) = 14.600000
Average Turnaround Time (FCFS) = 24.600000

Process returned 0 (0x0)   execution time : 0.141 s
Press any key to continue.
```

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

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

Code:

a) Rate- Monotonic

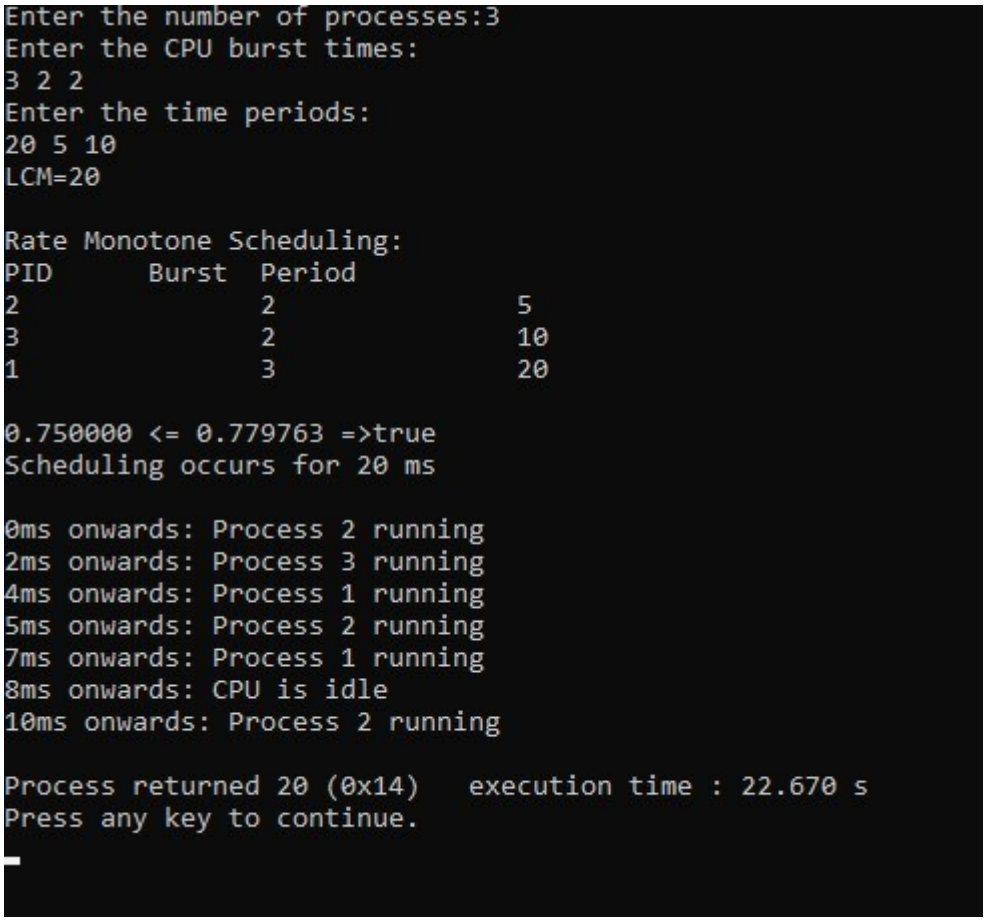
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
voidsort (int proc[], int b[], int pt[], int n){
  int temp = 0;
  for (int i = 0; i < n; i++){
            for (int j = i; j < n; j++){
                      if (pt[j] < pt[i]){
                                temp = pt[i];
                                pt[i] = pt[j];
                                pt[j] = temp;
                                temp = b[j];
                                b[j] = b[i];
                                b[i] = temp;
                                temp = proc[i];
                                proc[i] = proc[j];
                                proc[j] = temp;
                               }
                     }
           }
}
int gcd (int a, int b){
  int r;
  while (b > 0){
             r = a % b;
```

```
            a = b;
            b = r;
           }
  return a;
}
int lcmul (int p[], int n){
  int lcm = p[0];
  for (int i = 1; i < n; i++){
            lcm = (lcm * p[i]) / gcd (lcm, p[i]);
           }
  return lcm;
}
void main (){
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], rem[n];
  printf ("Enter the CPU burst times:\n");
  for (int i = 0; i < n; i++){
            scanf ("%d", &b[i]);
            rem[i] = b[i];
           }
  printf ("Enter the time periods:\n");
  for (int i = 0; i < n; i++)
           scanf ("%d", &pt[i]);
  for (int i = 0; i < n; i++)
           proc[i] = i + 1;
  sort (proc, b, pt, n);
  int l = lcmul (pt, n);
  printf ("LCM=%d\n", l);
  printf ("\nRate Monotone Scheduling:\n");
```

```
printf ("PID\t Burst\tPeriod\n");
for (int i = 0; i < n; i++)
        printf ("%d\t\t%d\t\t%d\n", proc[i], b[i], pt[i]);
double sum = 0.0;
for (int i = 0; i < n; i++){
         sum += (double) b[i] / pt[i];
        }
double rhs = n * (pow (2.0, (1.0 / n)) - 1.0);
printf ("\n%lf <= %lf =>%s\n", sum, rhs, (sum <= rhs) ? "true" : "false");
if (sum > rhs)
        exit (0);
printf ("Scheduling occurs for %d ms\n\n", l);
int time = 0, prev = 0, x = 0;
while (time < l){
         int f = 0;
         for (int i = 0; i < n; i++){
                 if (time % pt[i] == 0)
                         rem[i] = b[i];
                 if (rem[i] > 0){
                          if (prev != proc[i]){
                                  printf ("%dms onwards: Process %d running\n", time,
                                                          proc[i]);
                                  prev = proc[i];
                                 }
                          rem[i]--;
                          f = 1;
                          break;
                          x = 0;
                         }
                }
         if (!f){
```

```
                    if (x != 1){
                            printf ("%dms onwards: CPU is idle\n", time);
                            x = 1;
                           }
                  }
          time++;
         }
}
```

Output:



b) Earliest-deadline First

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void sort (int proc[], int d[], int b[], int pt[], int n){
```

```
  int temp = 0;
  for (int i = 0; i < n; i++){
            for (int j = i; j < n; j++){
                    if (d[j] < d[i]){
                            temp = d[j];
                            d[j] = d[i];
                            d[i] = temp;
                            temp = pt[i];
                            pt[i] = pt[j];
                            pt[j] = temp;
                            temp = b[j];
                            b[j] = b[i];
                            b[i] = temp;
                            temp = proc[i];
                            proc[i] = proc[j];
                            proc[j] = temp;
                           }
                    }
           }
}
int gcd (int a, int b){
  int r;
  while (b > 0){
            r = a % b;
            a = b;
            b = r;
           }
  return a;
}
int lcmul (int p[], int n){
  int lcm = p[0];
```

```
  for (int i = 1; i < n; i++){
              lcm = (lcm * p[i]) / gcd (lcm, p[i]);
             }
  return lcm;
}
void main (){
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], d[n], rem[n];
  printf ("Enter the CPU burst times:\n");
  for (int i = 0; i < n; i++){
               scanf ("%d", &b[i]);
               rem[i] = b[i];
             }
  printf ("Enter the deadlines:\n");
  for (int i = 0; i < n; i++)
             scanf ("%d", &d[i]);
  printf ("Enter the time periods:\n");
  for (int i = 0; i < n; i++)
             scanf ("%d", &pt[i]);
  for (int i = 0; i < n; i++)
             proc[i] = i + 1;
  sort (proc, d, b, pt, n);
  int l = lcmul (pt, n);
  printf ("\nEarliest Deadline Scheduling:\n");
  printf ("PID\t Burst\tDeadline\tPeriod\n");
  for (int i = 0; i < n; i++)
             printf ("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);
  printf ("Scheduling occurs for %d ms\n\n", l);
  int time = 0, prev = 0, x = 0;
```

```
 int nextDeadlines[n];
 for (int i = 0; i < n; i++){
            nextDeadlines[i] = d[i];
            rem[i] = b[i];
          }
 while (time < l){
            for (int i = 0; i < n; i++){
                  if (time % pt[i] == 0 && time != 0){
                        nextDeadlines[i] = time + d[i];
                        rem[i] = b[i];
                       }
                 }
            int minDeadline = l + 1;
            int taskToExecute = -1;
            for (int i = 0; i < n; i++){
                  if (rem[i] > 0 && nextDeadlines[i] < minDeadline){
                        minDeadline = nextDeadlines[i];
                        taskToExecute = i;
                       }
                 }
            if (taskToExecute != -1){
                  printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);
                  rem[taskToExecute]--;
                 }
            else{
                  printf ("%dms: CPU is idle.\n", time);
                 }
            time++;
          }
}
```

Output:

```
Enter the CPU burst times:
0 1 2
Enter the deadlines:
8 5 4
Enter the time periods:
3 4 6

Earliest Deadline Scheduling:
PID      Burst  Deadline        Period
3               2               4               6
2               1               5               4
1               0               8               3
Scheduling occurs for 12 ms

0ms : Task 3 is running.
1ms : Task 3 is running.
2ms : Task 2 is running.
3ms: CPU is idle.
4ms : Task 2 is running.
5ms: CPU is idle.
6ms : Task 3 is running.
7ms : Task 3 is running.
8ms: CPU is idle.
9ms: CPU is idle.
10ms: CPU is idle.
11ms: CPU is idle.

Process returned 12 (0xC)   execution time : 18.265 s
Press any key to continue.
```

c) Proportional scheduling

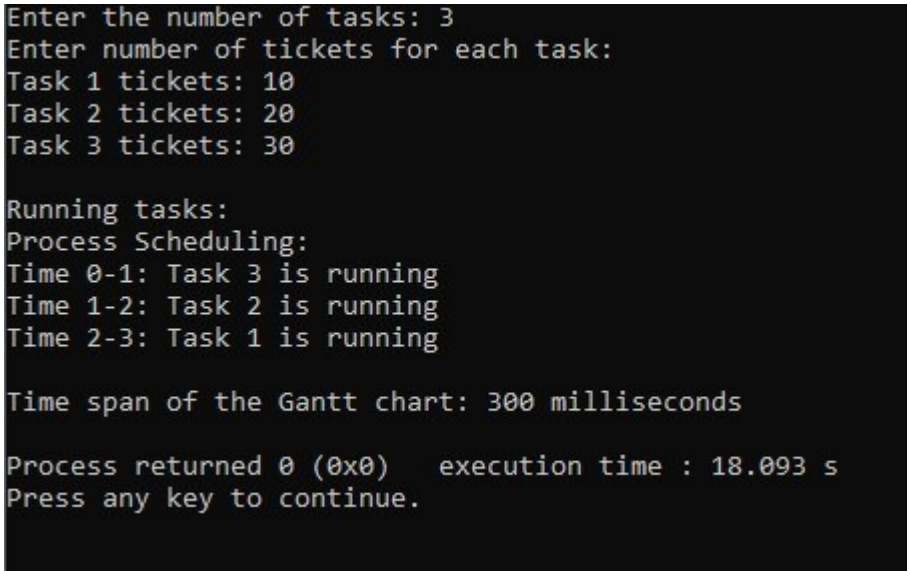
Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX_TASKS 10

#define MAX_TICKETS 100

#define TIME_UNIT_DURATION_MS 100

struct Task {

   int tid;

   int tickets;

};

void schedule(struct Task tasks[], int num_tasks, int *time_span_ms) {

   int total_tickets = 0;

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

```
        total_tickets += tasks[i].tickets;
    }
    srand(time(NULL));
    int current_time = 0;
    int completed_tasks = 0;
    printf("Process Scheduling:\n");
    while (completed_tasks < num_tasks) {
        int winning_ticket = rand() % total_tickets;
        int cumulative_tickets = 0;
        for (int i = 0; i < num_tasks; i++) {
            cumulative_tickets += tasks[i].tickets;
            if (winning_ticket < cumulative_tickets) {
                printf("Time %d-%d: Task %d is running\n", current_time, current_time + 1, tasks[i].tid);
                current_time++;
                break;
            }
        }
        completed_tasks++;
    }
    *time_span_ms = current_time * TIME_UNIT_DURATION_MS;
}
int main() {
    struct Task tasks[MAX_TASKS];
    int num_tasks;
    int time_span_ms;
    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);
    if (num_tasks <= 0 || num_tasks > MAX_TASKS) {
        printf("Invalid number of tasks. Please enter a number between 1 and %d.\n", MAX_TASKS);
        return 1;
    }
```

```
    printf("Enter number of tickets for each task:\n");
    for (int i = 0; i < num_tasks; i++) {
        tasks[i].tid = i + 1;
        printf("Task %d tickets: ", tasks[i].tid);
        scanf("%d", &tasks[i].tickets);
    }
    printf("\nRunning tasks:\n");
    schedule(tasks, num_tasks, &time_span_ms);
    printf("\nTime span of the Gantt chart: %d milliseconds\n", time_span_ms);
    return 0;
}
```

Output:



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

Code:

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty = 5, x = 0;
int main(){
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.Producer\n2.Consumer\n3.Exit");
while (1){
    printf("\nEnter your choice:");
    scanf("%d", &n);
    switch (n){
        case 1:
            if ((mutex == 1) && (empty != 0))
            producer();
            else
            printf("Buffer is full!!");
            break;
        case 2:
            if ((mutex == 1) && (full != 0))
                consumer();
            else
                printf("Buffer is empty!!");
                break;
        case 3:
            printf("Program execution completed.\n");
            exit(0);
```

```
            break;
        }
}
return 0;
}
int wait(int s){
        return (--s);
}
int signal(int s){
        return (++s);
}
void producer(){
      mutex = wait(mutex);
      full = signal(full);
      empty = wait(empty);
      x++;
      printf("\nProducer produces the item %d", x);
      mutex = signal(mutex);
}
void consumer() {
     mutex = wait(mutex);
     full = wait(full);
     empty = signal(empty);
     printf("\nConsumer consumes item %d", x);
     x--;
     mutex = signal(mutex);
 }
```

Output:

```
1.Producer
2.Consumer
3.Exit
Enter your choice:2
Buffer is empty!!
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:1

Producer produces the item 4
Enter your choice:1

Producer produces the item 5
Enter your choice:1
Buffer is full!!
Enter your choice:2

Consumer consumes item 5
Enter your choice:2

Consumer consumes item 4
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:
2
Buffer is empty!!
Enter your choice:3
Program execution completed.
```

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

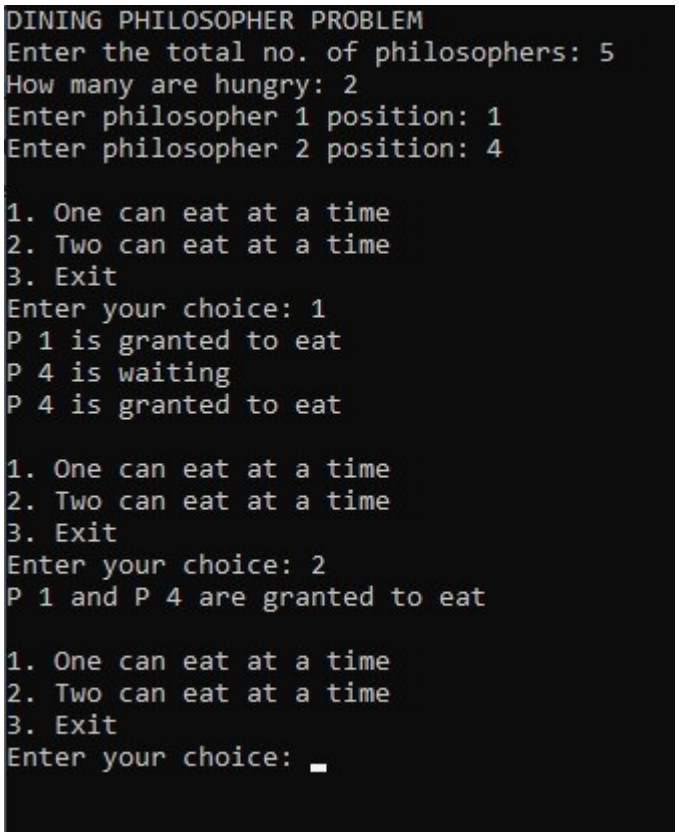
Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PHILOSOPHERS 5
void allow_one_to_eat(int hungry[], int n) {
   int isWaiting[MAX_PHILOSOPHERS];
   for (int i = 0; i < n; i++) {
      isWaiting[i] = 1;
   }
   for (int i = 0; i < n; i++) {
      printf("P %d is granted to eat\n", hungry[i]);
      isWaiting[hungry[i]] = 0;
      for (int j = 0; j < n; j++) {
         if (isWaiting[hungry[j]]) {
            printf("P %d is waiting\n", hungry[j]);
         }
      }
      for (int k = 0; k < n; k++) {
         isWaiting[k] = 1;
      }
      isWaiting[hungry[i]] = 0;
   }
}
void allow_two_to_eat(int hungry[], int n) {
   if (n < 2 || n > MAX_PHILOSOPHERS) {
      printf("Invalid number of philosophers.\n");
      return;
   }
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
```

```
            printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);
            for (int k = 0; k < n; k++) {
                if (k != i && k != j) {
                    printf("P %d is waiting\n", hungry[k]);
                }
            }
        }
    }
}
int main() {
    int total_philosophers, hungry_count;
    int hungry_positions[MAX_PHILOSOPHERS];
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: ");
    scanf("%d", &total_philosophers);
    if (total_philosophers > MAX_PHILOSOPHERS || total_philosophers < 2) {
        printf("Invalid number of philosophers.\n");
        return 1;
    }
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);
    if (hungry_count < 1 || hungry_count > total_philosophers) {
        printf("Invalid number of hungry philosophers.\n");
        return 1;
    }
    for (int i = 0; i < hungry_count; i++) {
        printf("Enter philosopher %d position: ", i + 1);
        scanf("%d", &hungry_positions[i]);
        if (hungry_positions[i] < 0 || hungry_positions[i] >= total_philosophers) {
            printf("Invalid philosopher position.\n");
            return 1;
```

```
        }
    }
    int choice;
    while (1) {
        printf("\n1. One can eat at a time\n");
        printf("2. Two can eat at a time\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                allow_one_to_eat(hungry_positions, hungry_count);
                break;
            case 2:
                allow_two_to_eat(hungry_positions, hungry_count);
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Output:


DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 2
Enter philosopher 1 position: 1
Enter philosopher 2 position: 4
1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
P 1 is granted to eat
P 4 is waiting
P 4 is granted to eat
1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 2
P 1 and P 4 are granted to eat
1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice:

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

Code:

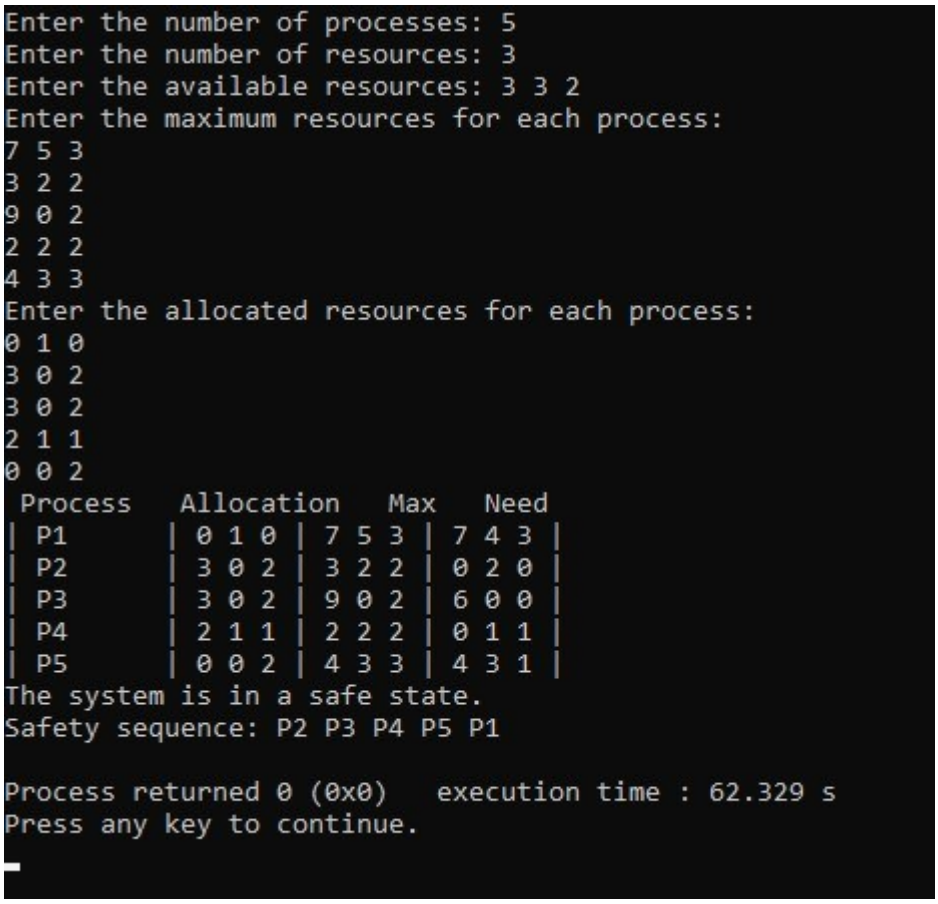
```
#include <stdio.h>
int main() {
   int n, m;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the number of resources: ");
   scanf("%d", &m);
   int available[m];
   printf("Enter the available resources: ");
   for (int i = 0; i < m; i++) {
      scanf("%d", &available[i]);
   }
   int maximum[n][m];
   printf("Enter the maximum resources for each process:\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         scanf("%d", &maximum[i][j]);
      }
   }
   int allocation[n][m];
   printf("Enter the allocated resources for each process:\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         scanf("%d", &allocation[i][j]);
      }
   }
   int need[n][m];
   for (int i = 0; i < n; i++) {
```

```
    for (int j = 0; j < m; j++) {
        need[i][j] = maximum[i][j] - allocation[i][j];
    }
}
printf(" Process   Allocation   Max   Need        \n");
for (int i = 0; i < n; i++) {
    printf("| P%d      | ", i + 1);
    for (int j = 0; j < m; j++) {
        printf("%d ", allocation[i][j]);
    }
    printf("| ");
    for (int j = 0; j < m; j++) {
        printf("%d ", maximum[i][j]);
    }
    printf("| ");
    for (int j = 0; j < m; j++) {
        printf("%d ", need[i][j]);
    }
    printf("|\n");
}
int work[m];
for (int i = 0; i < m; i++) {
    work[i] = available[i];
}
int finish[n];
for (int i = 0; i < n; i++) {
    finish[i] = 0;
}
int safeSequence[n];
int count = 0;
int safe = 1;
```

```
while (count < n) {
    int found = 0;
    for (int i = 0; i < n; i++) {
        if (finish[i] == 0) {
            int j;
            for (j = 0; j < m; j++) {
                if (need[i][j] > work[j]) {
                    break;
                }
            }
            if (j == m) {
                for (j = 0; j < m; j++) {
                    work[j] += allocation[i][j];
                }
                finish[i] = 1;
                safeSequence[count++] = i;
                found = 1;
            }
        }
    }
    if (!found) {
        safe = 0;
        break;
    }
}
if (safe) {
    printf("The system is in a safe state.\n");
    printf("Safety sequence: ");
    for (int i = 0; i < n; i++) {
        printf("P%d ", safeSequence[i] + 1);
    }
```

```
        printf("\n");
    } else {
        printf("The system is in an unsafe state and might lead to deadlock.\n");
    }
    return 0;
}
```

Output:



8. Write a C program to simulate deadlock detection

Code:

```
#include<stdio.h>
void main(){
   int n,m,i,j;
   printf("Enter the number of processes and number of types of resources:\n");
   scanf("%d %d",&n,&m);
   int max[n][m],need[n][m],all[n][m],ava[m],flag=1,finish[n],dead[n],c=0;
   printf("Enter the maximum number of each type of resource needed by each process:\n");
   for(i=0;i<n;i++){
      for(j=0;j<m;j++){
         scanf("%d",&max[i][j]);
      }
   }
   printf("Enter the allocated number of each type of resource needed by each process:\n");
   for(i=0;i<n;i++){
      for(j=0;j<m;j++){
         scanf("%d",&all[i][j]);
      }
   }
   printf("Enter the available number of each type of resource:\n");
   for(j=0;j<m;j++){
      scanf("%d",&ava[j]);
   }
   for(i=0;i<n;i++){
      for(j=0;j<m;j++)
      {
         need[i][j]=max[i][j]-all[i][j];
      }
   }
```

```
for(i=0;i<n;i++){
   finish[i]=0;
}
while(flag){
   flag=0;
   for(i=0;i<n;i++){
      c=0;
      for(j=0;j<m;j++){
         if(finish[i]==0 && need[i][j]<=ava[j]){
            c++;
            if(c==m){
               for(j=0;j<m;j++){
                  ava[j]+=all[i][j];
                  finish[i]=1;
                  flag=1;
               }
               if(finish[i]==1){
                  i=n;
               }
            }
         }
      }
   }
}
j=0;
flag=0;
for(i=0;i<n;i++){
   if(finish[i]==0){
      dead[j]=i;
      j++;
      flag=1;
```

```
        }
    }
    if(flag==1){
        printf("Deadlock has occured:\n");
        printf("The deadlock processes are:\n");
        for(i=0;i<n;i++){
            printf("P%d ",dead[i]);
        }
    }
    else
    printf("No deadlock has occured!\n");
}
```

Output:

```
Enter the number of processes and number of types of resources:
4 3
Enter the allocated number of each type of resource needed by each process:
1 0 2
2 1 1
1 0 3
2 2 2
Enter the available number of each type of resource:
0 0 0
Enter the request number of each type of resource needed by each process:
0 0 1
1 0 2
0 0 0
3 3 0
Deadlock has occured:
The deadlock processes are:
P3
Process returned 1 (0x1)   execution time : 47.994 s
Press any key to continue.
```

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

a) Worst-fit

b) Best-fit

c) First-fit

Code:

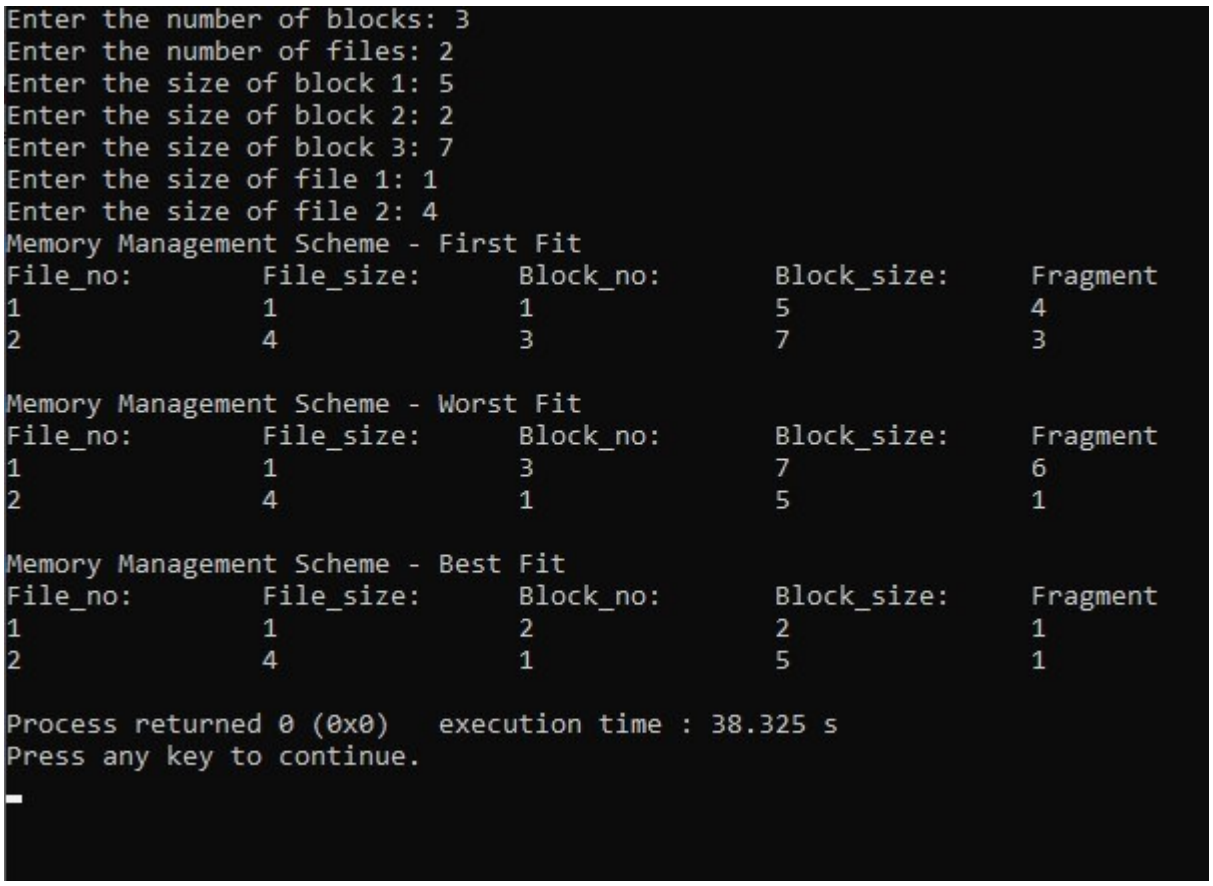
```
#include <stdio.h>
struct Block {
   int block_no;
   int block_size;
   int is_free;
};
struct File {
   int file_no;
   int file_size;
};
void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - First Fit\n");
   printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
   for (int i = 0; i < n_files; i++) {
      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            blocks[j].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size, blocks[j].block_no,
blocks[j].block_size, blocks[j].block_size - files[i].file_size);
            break;
         }
      }
   }
}
void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - Worst Fit\n");
```

```
    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
    for (int i = 0; i < n_files; i++) {
        int worst_fit_block = -1;
        int max_fragment = -1;
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment > max_fragment) {
                    max_fragment = fragment;
                    worst_fit_block = j;
                }
            }
        }
        if (worst_fit_block!= -1) {
            blocks[worst_fit_block].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[worst_fit_block].block_no, blocks[worst_fit_block].block_size, max_fragment);
        }
    }
}
void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("Memory Management Scheme - Best Fit\n");
    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
    for (int i = 0; i < n_files; i++) {
        int best_fit_block = -1;
        int min_fragment = 10000;
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment < min_fragment) {
                    min_fragment = fragment;
                    best_fit_block = j;
```

```
            }
          }
        }
        if (best_fit_block!= -1) {
            blocks[best_fit_block].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[best_fit_block].block_no, blocks[best_fit_block].block_size, min_fragment);
        }
    }
}
int main() {
    int n_blocks, n_files;
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);
    printf("Enter the number of files: ");
    scanf("%d", &n_files);
    struct Block blocks[n_blocks];
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
    }
    struct File files[n_files];
    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }
    firstFit(blocks, n_blocks, files, n_files);
    printf("\n");
    for (int i = 0; i < n_blocks; i++) {
```

```
        blocks[i].is_free = 1;
    }
    worstFit(blocks, n_blocks, files, n_files);
    printf("\n");
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }
    bestFit(blocks, n_blocks, files, n_files);
    return 0;
}
```

Output:



Program 10

10. Write a C program to simulate page replacement algorithms

a) FIFO

b) LRU

c) Optimal

Code:

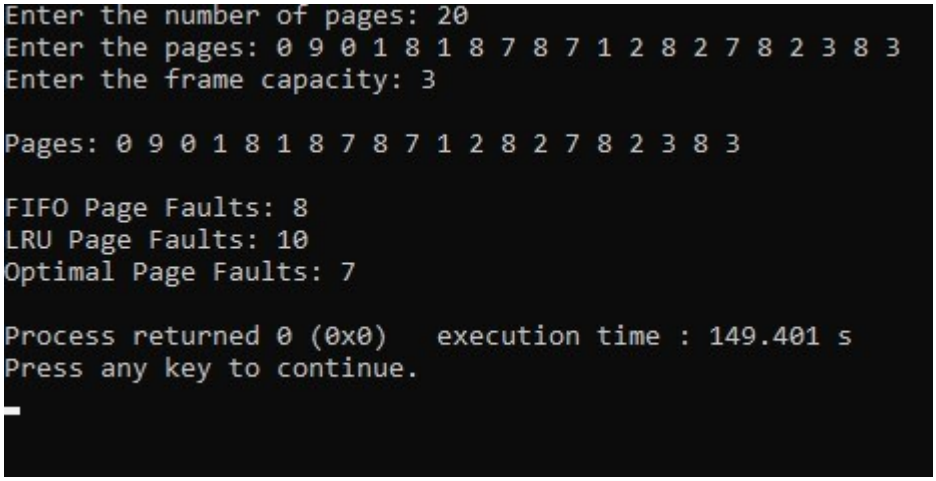
```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
void fifo(int pages[], int n, int capacity) {
   int frame[capacity], index = 0, page_faults = 0;
   for (int i = 0; i < capacity; i++)
      frame[i] = -1;
   for (int i = 0; i < n; i++) {
      int found = 0;
      for (int j = 0; j < capacity; j++) {
         if (frame[j] == pages[i]) {
            found = 1;
            break;
         }
      }
      if (!found) {
         frame[index] = pages[i];
         index = (index + 1) % capacity;
         page_faults++;
      }
   }
   printf("FIFO Page Faults: %d\n", page_faults);
}
void lru(int pages[], int n, int capacity) {
   int frame[capacity], counter[capacity], time = 0, page_faults = 0;
```

```
    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
        counter[i] = 0;
    }
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                counter[j] = time++;
                break;
            }
        }
        if (!found) {
            int min = INT_MAX, min_index = -1;
            for (int j = 0; j < capacity; j++) {
                if (counter[j] < min) {
                    min = counter[j];
                    min_index = j;
                }
            }
            frame[min_index] = pages[i];
            counter[min_index] = time++;
            page_faults++;
        }
    }
    printf("LRU Page Faults: %d\n", page_faults);
}
void optimal(int pages[], int n, int capacity) {
    int frame[capacity], page_faults = 0;
    for (int i = 0; i < capacity; i++)
```

```
    frame[i] = -1;
for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < capacity; j++) {
        if (frame[j] == pages[i]) {
            found = 1;
            break;
        }
    }
    if (!found) {
        int farthest = i + 1, index = -1;
        for (int j = 0; j < capacity; j++) {
            int k;
            for (k = i + 1; k < n; k++) {
                if (frame[j] == pages[k])
                    break;
            }
            if (k > farthest) {
                farthest = k;
                index = j;
            }
        }
        if (index == -1) {
            for (int j = 0; j < capacity; j++) {
                if (frame[j] == -1) {
                    index = j;
                    break;
                }
            }
        }
        frame[index] = pages[i];
```

```
            page_faults++;
        }
    }
    printf("Optimal Page Faults: %d\n", page_faults);
}
int main() {
    int n, capacity;
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int *pages = (int*)malloc(n * sizeof(int));
    printf("Enter the pages: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter the frame capacity: ");
    scanf("%d", &capacity);
    printf("\nPages: ");
    for (int i = 0; i < n; i++)
        printf("%d ", pages[i]);
    printf("\n\n");
    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    optimal(pages, n, capacity);
    free(pages);
    return 0;
}
```

Output:


Enter the number of pages: 20
Enter the pages: 0 9 0 1 8 1 8 7 8 7 1 2 8 2 7 8 2 3 8 3
Enter the frame capacity: 3
Pages: 0 9 0 1 8 1 8 7 8 7 1 2 8 2 7 8 2 3 8 3
FIFO Page Faults: 8
LRU Page Faults: 10
Optimal Page Faults: 7
Process returned 0 (0x0) execution time : 149.401 s
Press any key to continue.