VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

Nallabothula Sai Sruthi(1BM22CS170)

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 **Nallabothula Sai Sruthi(1BM22CS170),** 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.

Dr. Radhika A D
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1.	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & Non-preemptive)	4
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)	13
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.	19
4.	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate-Monotonic b) Earliest-deadline First c) Proportional scheduling	22
5.	Write a C program to simulate producer-consumer problem using semaphores.	31
6.	Write a C program to simulate the concept of Dining-Philosophers problem.	33
7.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	37
8.	Write a C program to simulate deadlock detection	41
9.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	44
10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	47

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

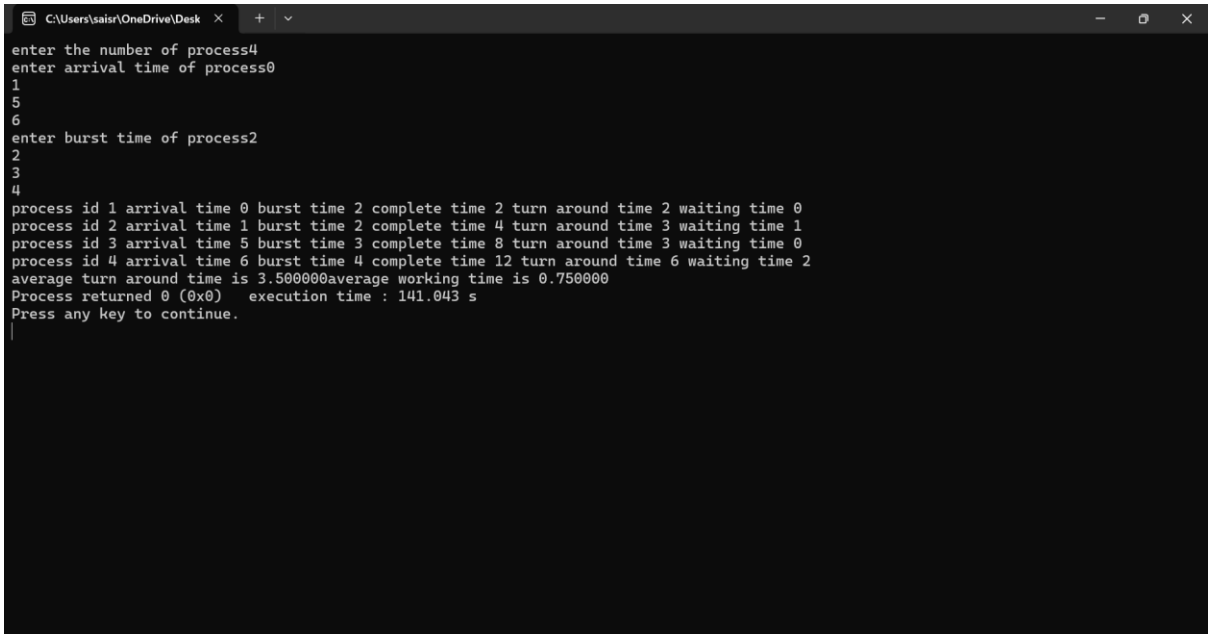
1. Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & Non-preemptive)

a.FCFS scheduling using array

```
#include<stdio.h>
int main(){
   int n,i;
   float atat=0,awt=0;
   printf("enter the number of process");
   scanf("%d",&n);
   int atime1[n],btime2[n],ctime3[n],tattime4[n],wtime5[n];
   printf("enter arrival time of process");
   for(i=0;i<n;i++){
      scanf("%d",&atime1[i]);
   }
   printf("enter burst time of process");
   for(i=0;i<n;i++){
      scanf("%d",&btime2[i]);
   }
   for(i=0;i<n;i++){
      if(i==0){
         ctime3[i]=atime1[i]+btime2[i];
      }
      else{
         if(ctime3[i-1]<atime1[i]){
            ctime3[i]=(atime1[i]-ctime3[i-1])+ctime3[i-1]+btime2[i];
         }
         else{
            ctime3[i]=ctime3[i-1]+btime2[i];
         }
      }
   }
```

```
    for(i=0;i<n;i++){
        tattime4[i]=ctime3[i]-atime1[i];
    }
    for(i=0;i<n;i++){
        wtime5[i]=tattime4[i]-btime2[i];
    }
    for(i=0;i<n;i++){
        atat=atat+tattime4[i];
    }
    atat=(atat/n);
    for(i=0;i<n;i++){
        awt=awt+wtime5[i];
    }
    awt=(awt/n);
    for(i=0;i<n;i++){
        printf("process id %d arrival time %d burst time %d complete time %d turn around time %d
waiting time %d\n",i+1,atime1[i],btime2[i],ctime3[i],tattime4[i],wtime5[i]);
    }
    printf("average turn around time is %f",atat);
    printf("average working time is %f",awt);
}
```

output:



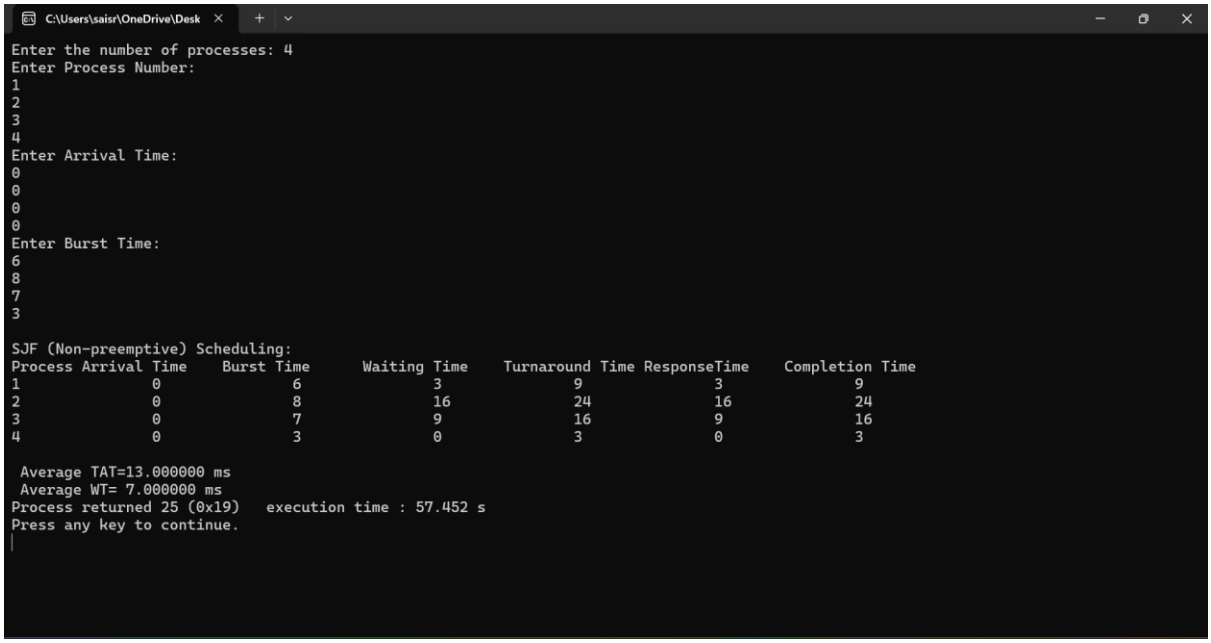
b.SJF(non-preemptive)scheduling using array

```
#include<stdio.h>
 void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int rt[], int ct[])
 {
 int completion[n];
 int remaining[n];
 for (int i = 0; i < n; i++)
 remaining[i] = bt[i];
 int currentTime = 0;
 for (int i = 0; i < n; i++)
 {
 int shortest =-1;
 for (int j = 0; j < n; j++)
 {
 if (at[j] <= currentTime && remaining[j] > 0)
 {
 if (shortest ==-1 || remaining[j] < remaining[shortest])
 shortest = j;
 }
 }
```

```
 if (shortest ==-1)
 {
 currentTime++;
 continue;
 }
 completion[shortest] = currentTime + remaining[shortest];
 currentTime = completion[shortest];
 wt[shortest] = currentTime- bt[shortest]- at[shortest];
 tat[shortest] = currentTime- at[shortest];
 rt[shortest] = wt[shortest];
 remaining[shortest] = 0;
 }
 printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\tResponseTime\tCompletion Time\n");
 for (int i = 0; i < n; i++)
 {
 ct[i] = completion[i];
 printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], at[i], bt[i], wt[i], tat[i], rt[i], ct[i]);
 }
 float avg_tat=tat[0];
 for(int i=1;i<n;i++)
 {
avg_tat+=tat[i];
 }
 printf("\n Average TAT=%f ms",avg_tat/n);
 float avg_wt=wt[0];
 for(int i=1;i<n;i++)
 {
 avg_wt+=wt[i];
 }
 printf("\n Average WT= %f ms",avg_wt/n);
```

```
}
void main()
{
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
int processes[n];
int burst_time[n];
int arrival_time[n];
printf("Enter Process Number:\n");
for (int i = 0; i < n; i++)
{
scanf("%d", & processes[i]);
}
printf("Enter Arrival Time:\n");
for (int i = 0; i < n; i++)
{
scanf("%d", &arrival_time[i]);
}
printf("Enter Burst Time:\n");
for (int i = 0; i < n; i++)
{
scanf("%d", &burst_time[i]);
}
int wt[n], tat[n], rt[n], ct[n];
for (int i = 0; i < n; i++)
rt[i] =-1;
printf("\nSJF (Non-preemptive) Scheduling:\n");
findCompletionTime(processes, n, burst_time, arrival_time, wt, tat, rt, ct);
}
```

output:



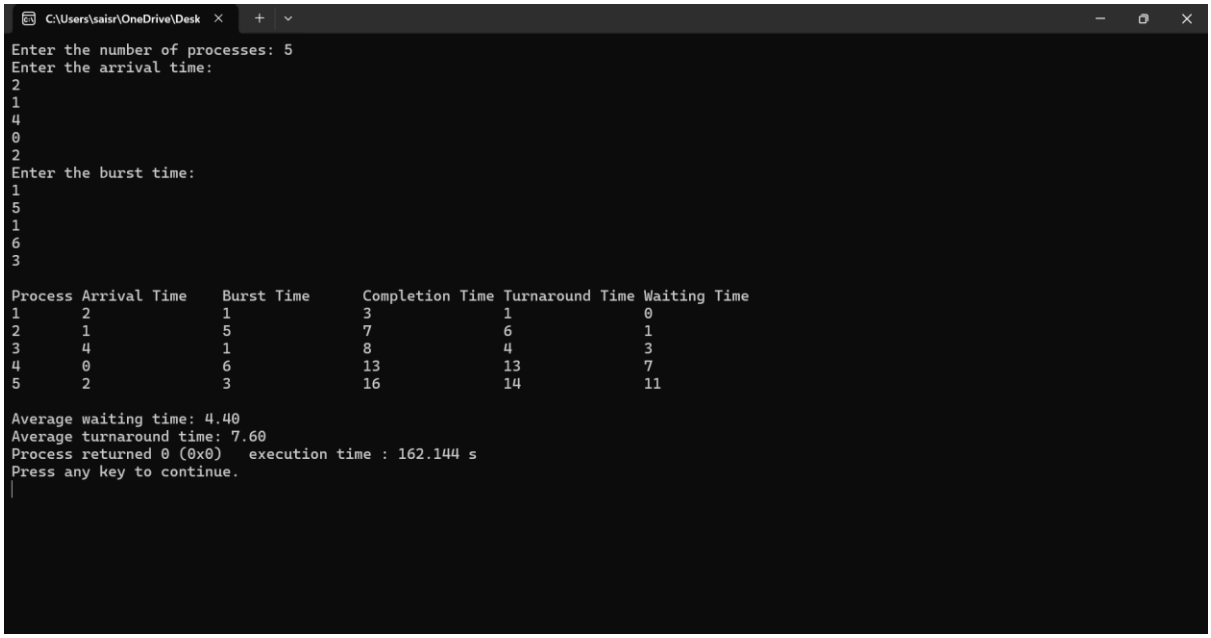
c.SJF(preemptive)scheduling using array

```
#include <stdio.h>

#define MAX 10

int find_min(int arr[], int n) {
    int min = arr[0];
    int index = 0;
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) {
            min = arr[i];
            index = i;
        }
    }
    return index;
}

void sjf_preemptive(int n, int at[], int bt[]) {
    int ct[MAX] = {0};
    int tat[MAX] = {0};
    int wt[MAX] = {0};
    int rt[MAX];
    int total_wt = 0;
```

```
int total_tat = 0;
for (int i = 0; i < n; i++) {
    rt[i] = bt[i];
}
int current_time = 0;
int completed_processes = 0;
while (completed_processes < n) {
    int available_processes[MAX];
    int available_count = 0;
    for (int i = 0; i < n; i++) {
        if (at[i] <= current_time && rt[i] > 0) {
            available_processes[available_count] = i;
            available_count++;
        }
    }
    if (available_count == 0) {
        current_time++;
        continue;
    }
    int shortest_job_index = available_processes[find_min(rt, available_count)];
    rt[shortest_job_index]--;
    current_time++;
    if (rt[shortest_job_index] == 0) {
        completed_processes++;
        ct[shortest_job_index] = current_time;
        tat[shortest_job_index] = ct[shortest_job_index] - at[shortest_job_index];
        wt[shortest_job_index] = tat[shortest_job_index] - bt[shortest_job_index];
        total_wt += wt[shortest_job_index];
        total_tat += tat[shortest_job_index];
    }
}
```

```
    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;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int at[MAX], bt[MAX];
    printf("Enter the arrival time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }
    printf("Enter the burst time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
    }
    sjf_preemptive(n, at, bt);
    return 0;
}
```

output:


C:\Users\saisr\OneDrive\Desk
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 7 6 1
3 4 1 8 4 3
4 0 6 13 13 7
5 2 3 16 14 11
Average waiting time: 4.40
Average turnaround time: 7.60
Process returned 0 (0x0) execution time : 162.144 s
Press any key to continue.

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)

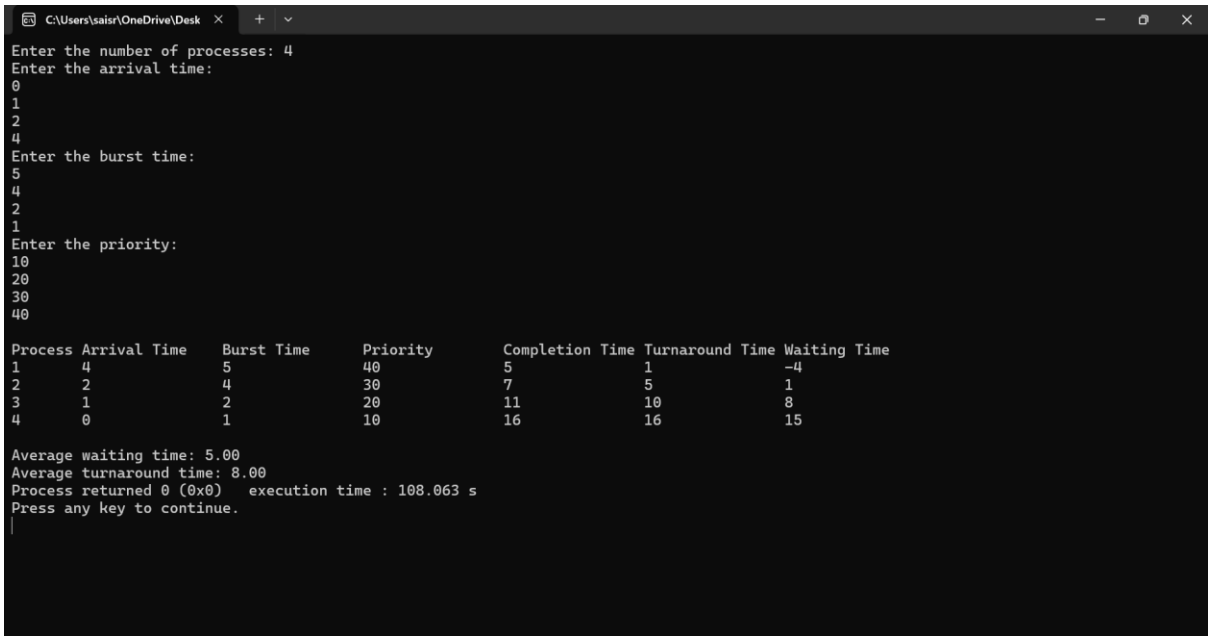
a.priority(non preemptive)scheduling

```
#include<stdio.h>

#define MAX 10

void priority_non_preemptive(int n, int at[], int bt[], int p[]) {
   int ct[MAX] = {0};
   int tat[MAX] = {0};
   int wt[MAX] = {0};
   int total_wt = 0;
   int total_tat = 0;
   int bt_copy[MAX];
   for (int i = 0; i < n; i++) {
      bt_copy[i] = bt[i];
   }
   for (int i = 0; i < n; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i] < p[j]) {
            int temp = at[i];
            at[i] = at[j];
            at[j] = temp;
            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
         }
      }
   }
   ct[0] = at[0] + bt[0];
```

```
    tat[0] = ct[0] - at[0];
    wt[0] = tat[0] - bt_copy[0];
    total_wt += wt[0];
    total_tat += tat[0];
    for (int i = 1; i < n; i++) {
        ct[i] = ct[i - 1] + bt[i];
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt_copy[i];
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("\nProcess\tArrival Time\tBurst Time\tPriority\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\t\t%d\n", i+1, at[i], bt_copy[i], p[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;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int at[MAX], bt[MAX], p[MAX];
    printf("Enter the arrival time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }
    printf("Enter the burst time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
```

```
    }
    printf("Enter the priority:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &p[i]);
    }
    priority_non_preemptive(n, at, bt, p);
    return 0;
}
```

output:



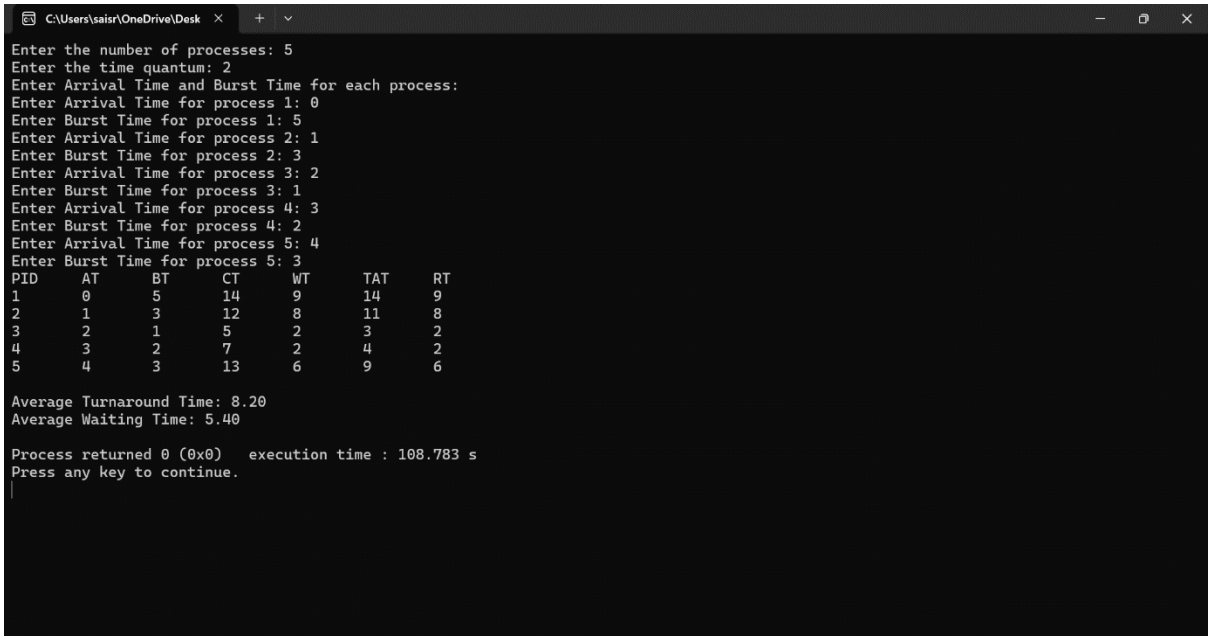
b.Round robin scheduling

```
#include <stdio.h>
 struct Process {
 int pid;
 int burst_time;
 int arrival_time;
 int remaining_time;
 };
 void roundRobin(struct Process processes[], int n, int time_quantum) {
 int remaining_processes = n;
 int current_time = 0;
```

```
int completed[n];
int ct[n], wt[n], tat[n], rt[n];
for (int i = 0; i < n; i++) {
completed[i] = 0;
}
while (remaining_processes > 0) {
for (int i = 0; i < n; i++) {
if (completed[i] == 0 && processes[i].arrival_time <= current_time) {
if (processes[i].remaining_time > 0) {
if (processes[i].remaining_time <= time_quantum) {
current_time += processes[i].remaining_time;
processes[i].remaining_time = 0;
completed[i] = 1;
remaining_processes--;
ct[i] = current_time;
tat[i] = ct[i]- processes[i].arrival_time;
} else {
current_time += time_quantum;
processes[i].remaining_time-= time_quantum;
}
wt[i] = ct[i]- processes[i].arrival_time- processes[i].burst_time;
rt[i] = wt[i];
}
}
}
}
printf("PID\tAT\tBT\tCT\tWT\tTAT\tRT\n");
float avg_tat = 0, avg_wt = 0;
for (int i = 0; i < n; i++) {
printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].arrival_time,
processes[i].burst_time, ct[i], wt[i], tat[i], rt[i]);
```

```
 avg_tat += tat[i];
 avg_wt += wt[i];
}
 avg_tat /= n;
 avg_wt /= n;
 printf("\nAverage Turnaround Time: %.2f\n", avg_tat);
 printf("Average Waiting Time: %.2f\n", avg_wt);
 }
 int main() {
 int n, time_quantum;
 printf("Enter the number of processes: ");
 scanf("%d", &n);
 printf("Enter the time quantum: ");
 scanf("%d", &time_quantum);
 struct Process processes[n];
 printf("Enter Arrival Time and Burst Time for each process:\n");
 for (int i = 0; i < n; i++) {
 printf("Enter Arrival Time for process %d: ", i+1);
 scanf("%d", & processes[i].arrival_time);
 printf("Enter Burst Time for process %d: ", i+1);
 scanf("%d", & processes[i].burst_time);
 processes[i].pid = i+1;
 processes[i].remaining_time = processes[i].burst_time;
 }
 roundRobin(processes, n, time_quantum);
 }
```

output:


C:\Users\saisr\OneDrive\Desk
Enter the number of processes: 5
Enter the time quantum: 2
Enter Arrival Time and Burst Time for each process:
Enter Arrival Time for process 1: 0
Enter Burst Time for process 1: 5
Enter Arrival Time for process 2: 1
Enter Burst Time for process 2: 3
Enter Arrival Time for process 3: 2
Enter Burst Time for process 3: 1
Enter Arrival Time for process 4: 3
Enter Burst Time for process 4: 2
Enter Arrival Time for process 5: 4
Enter Burst Time for process 5: 3
PID AT BT CT WT TAT RT
1 0 5 14 9 14 9
2 1 3 12 8 11 8
3 2 1 5 2 3 2
4 3 2 7 2 4 2
5 4 3 13 6 9 6
Average Turnaround Time: 8.20
Average Waiting Time: 5.40
Process returned 0 (0x0) execution time : 108.783 s
Press any key to continue.

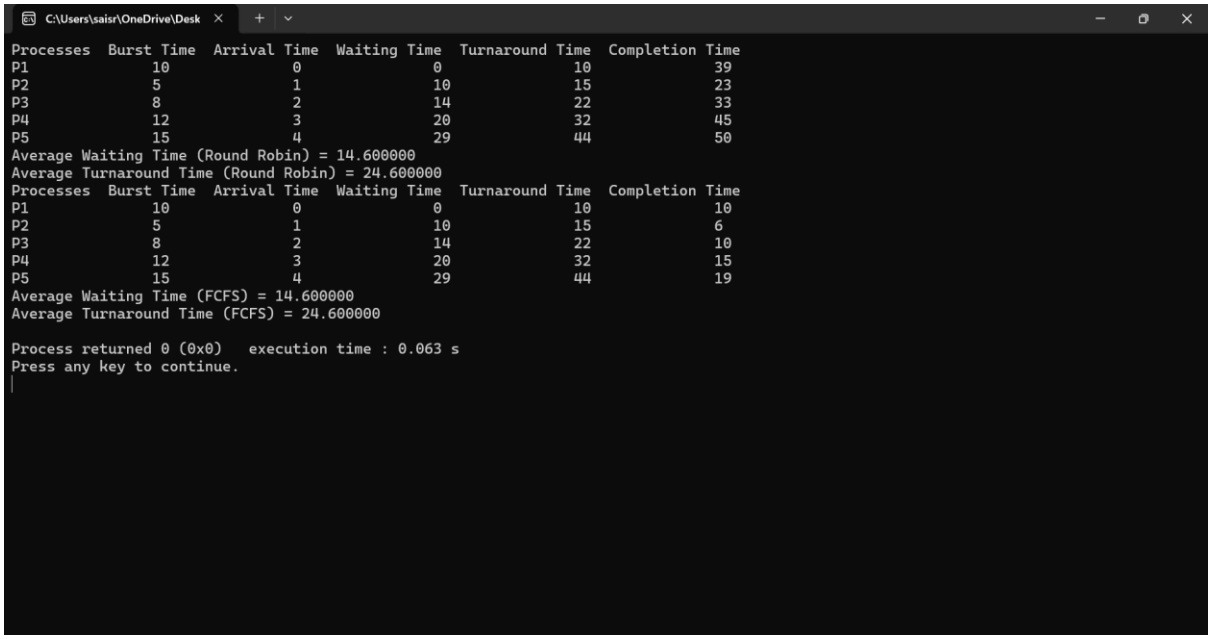
3. Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling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

```
#include <stdio.h>

// Function to find waiting time for FCFS

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;

   }

}

// Function to find turnaround 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];

   }

}

// Function to implement Round Robin scheduling

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);
}
// Function to implement FCFS scheduling
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:



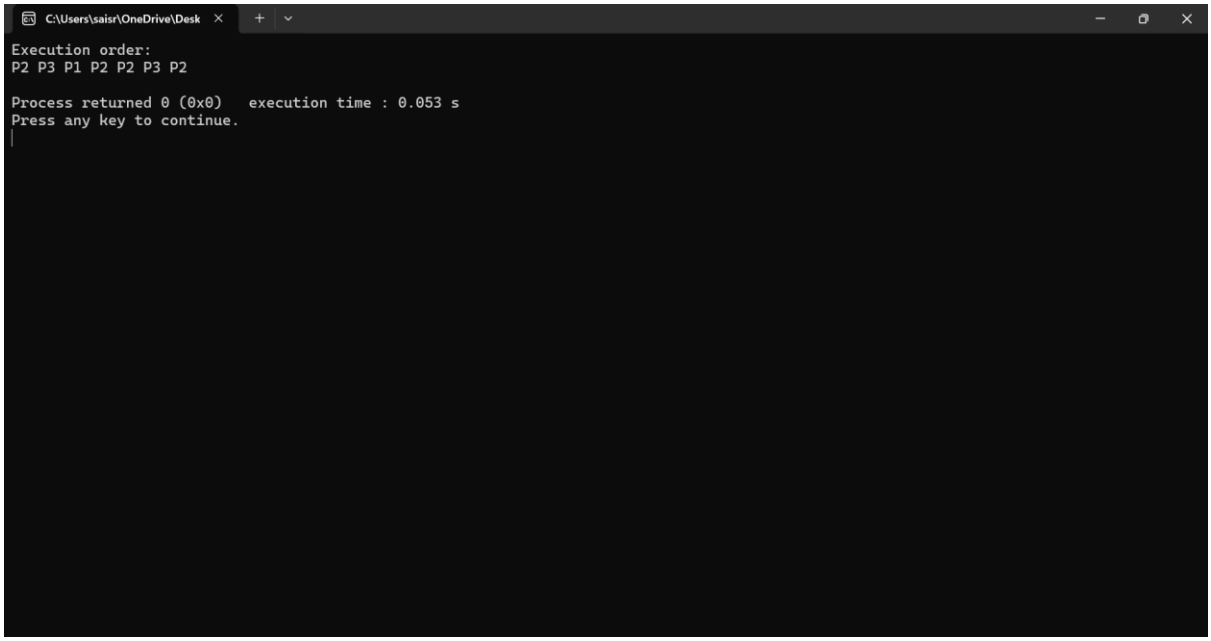
4. Write a C program to simulate Real-Time CPU Scheduling algorithms:

a) Rate- Monotonic

```
#include <stdio.h>
// Structure to represent a process
struct Process {
int execution_time;
int time_period;
};
// Function to calculate the least common multiple (LCM)
int lcm(int a, int b) {
int max = (a > b) ? a : b;
while (1) {
if (max % a == 0 && max % b == 0)
return max;
max++;
}
}
// Function to check if the set of processes is schedulable
int is_schedulable(struct Process processes[], int n) {
   float utilization = 0.0;
   for (int i = 0; i < n; i++) {
      utilization += (float)processes[i].execution_time / processes[i].time_period;
   }
   return utilization <= 1.0;
}
int main() {
   struct Process processes[] = {
      {3, 20}, // P1
      {2, 5},  // P2
      {2, 10}  // P3
   };
```

```
    int n = sizeof(processes) / sizeof(processes[0]);

    // Check if the processes are schedulable
    if (!is_schedulable(processes, n)) {
        printf("The given set of processes is not schedulable.\n");
        return 0;
    }
    // Calculate the scheduling time (LCM of time periods)
    int scheduling_time = lcm(processes[0].time_period, processes[1].time_period);
    scheduling_time = lcm(scheduling_time, processes[2].time_period);
    // Display the execution order
    printf("Execution order:\n");
    for (int t = 0; t < scheduling_time; t++) {
        if (t % processes[1].time_period == 0)
            printf("P2 ");
        if (t % processes[2].time_period == 0)
printf("P3 ");
if (t % processes[0].time_period == 0)
printf("P1 ");
}
printf("\n");
return 0;
}
```

output:



b) Earliest-deadline First

```
#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;
```

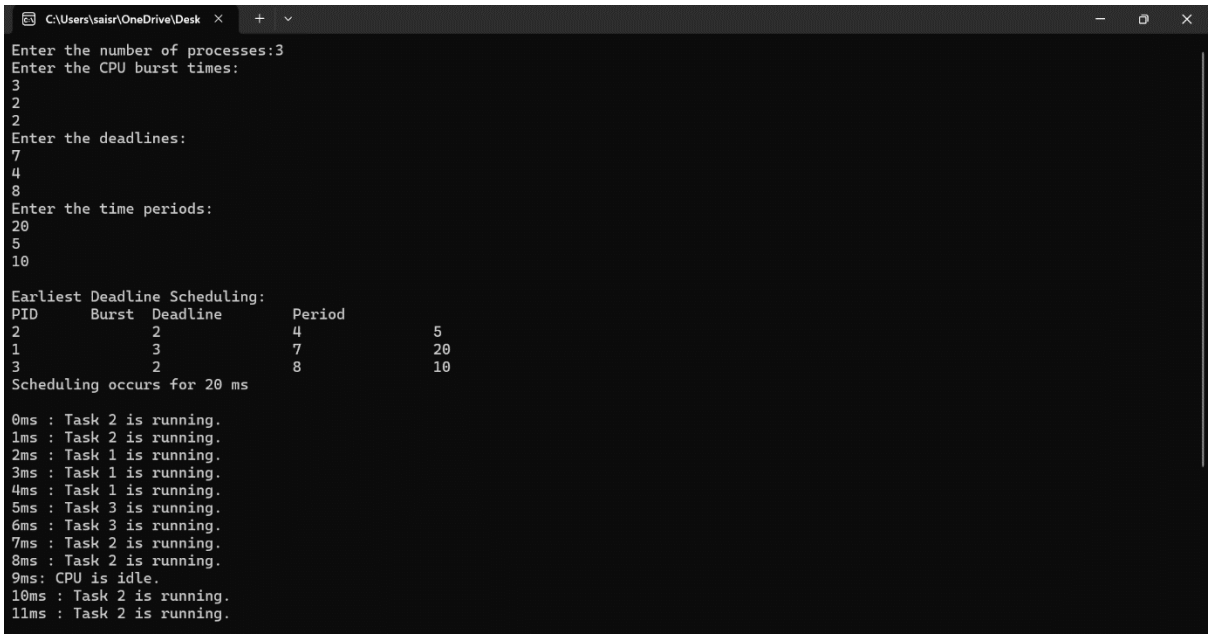
```
                    temp = b[j];
                    b[j] = b[i];
                    b[i] = temp;
                    temp = proc[i];
                    proc[i] = proc[j];
                    proc[j] = temp;
                   }
                }
        }
}
int gcd (int a, int b)
{
 int r;
 while (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;
}
```

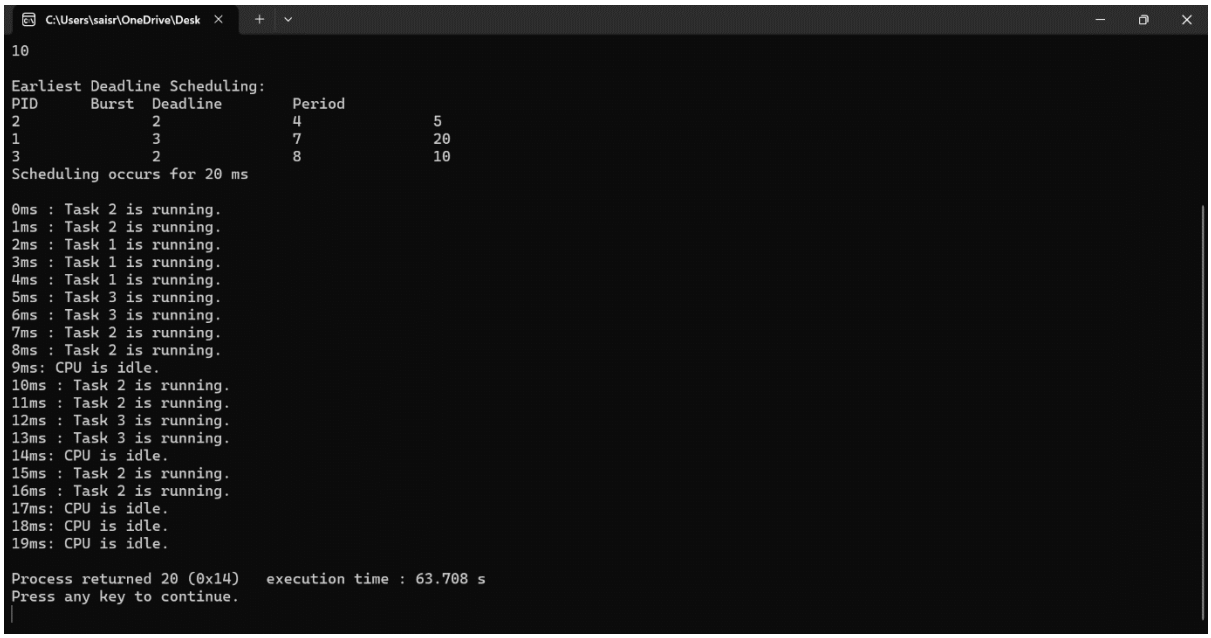
```
void main ()
{
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], d[n], rem[n];
  printf ("Enter the CPU burst times:\n");
  for (int i = 0; i < n; i++)
            {
              scanf ("%d", &b[i]);
              rem[i] = b[i];
            }
  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);
  //LCM
  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);
  //EDF
```

```
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:



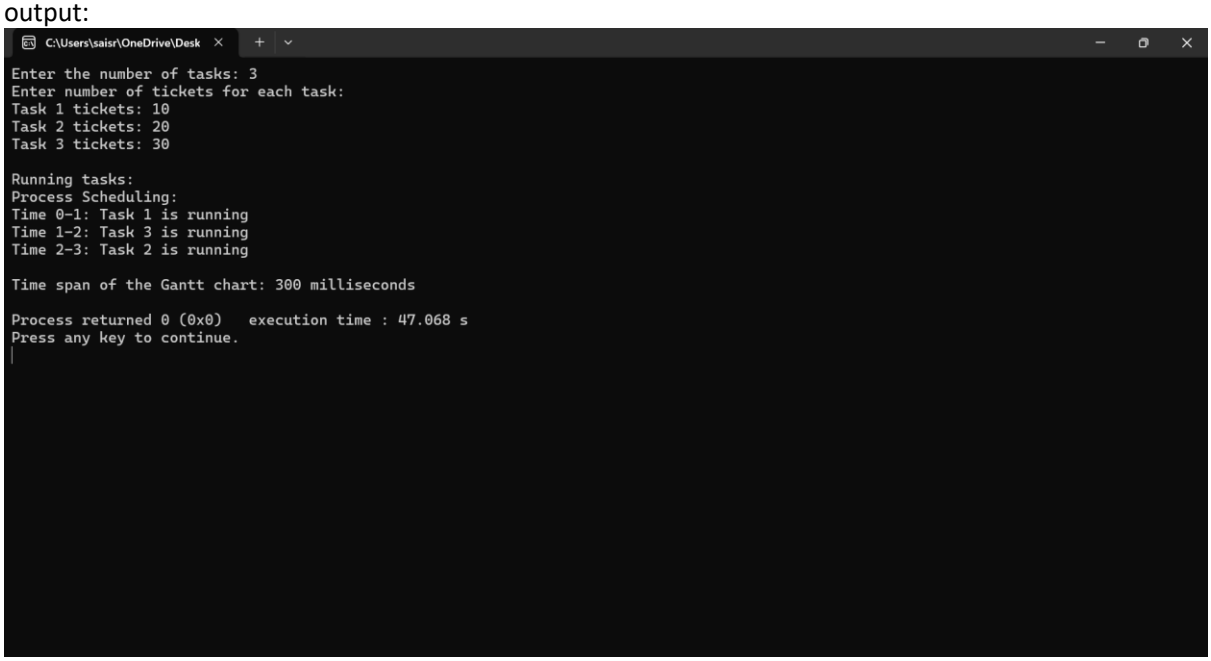


c) Proportional scheduling

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

#define MAX_TASKS 10
#define MAX_TICKETS 100
#define TIME_UNIT_DURATION_MS 100 // Duration of each time unit in milliseconds

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++;
   }

   // Calculate time span in milliseconds
   *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:

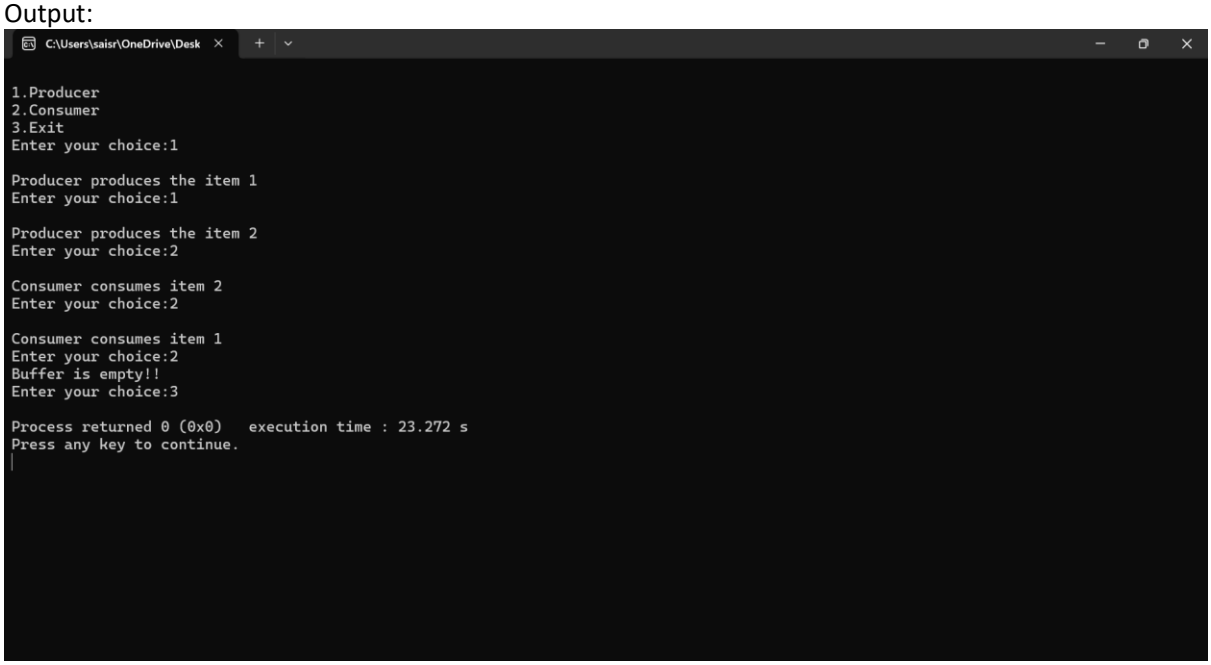


5. Write a C program to simulate producer-consumer problem using semaphores.

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

#define BUFFER_SIZE 5
#define MAX_ITEMS 20

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

void* producer(void* arg) {
   int item = 1;
   while (produced_count < MAX_ITEMS) {
      sem_wait(&empty);
      sem_wait(&mutex);
      buffer[in] = item;
      printf("Produced: %d\n", item);
      item++;
      in = (in + 1) % BUFFER_SIZE;
      produced_count++;
      sem_post(&mutex);
      sem_post(&full);
   }
   pthread_exit(NULL);
}

void* consumer(void* arg) {
   while (consumed_count < MAX_ITEMS) {
      sem_wait(&full);
      sem_wait(&mutex);
      int item = buffer[out];
      printf("Consumed: %d\n", item);
      out = (out + 1) % BUFFER_SIZE;
      consumed_count++;
      sem_post(&mutex);
      sem_post(&empty);
   }
   pthread_exit(NULL);
}

int main() {
   pthread_t producerThread, consumerThread;
   sem_init(&mutex, 0, 1);
   sem_init(&full, 0, 0);
   sem_init(&empty, 0, BUFFER_SIZE);
```

```
    pthread_create(&producerThread, NULL, producer, NULL);
    pthread_create(&consumerThread, NULL, consumer, NULL);

    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);

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

    return 0;
}
```

Output:



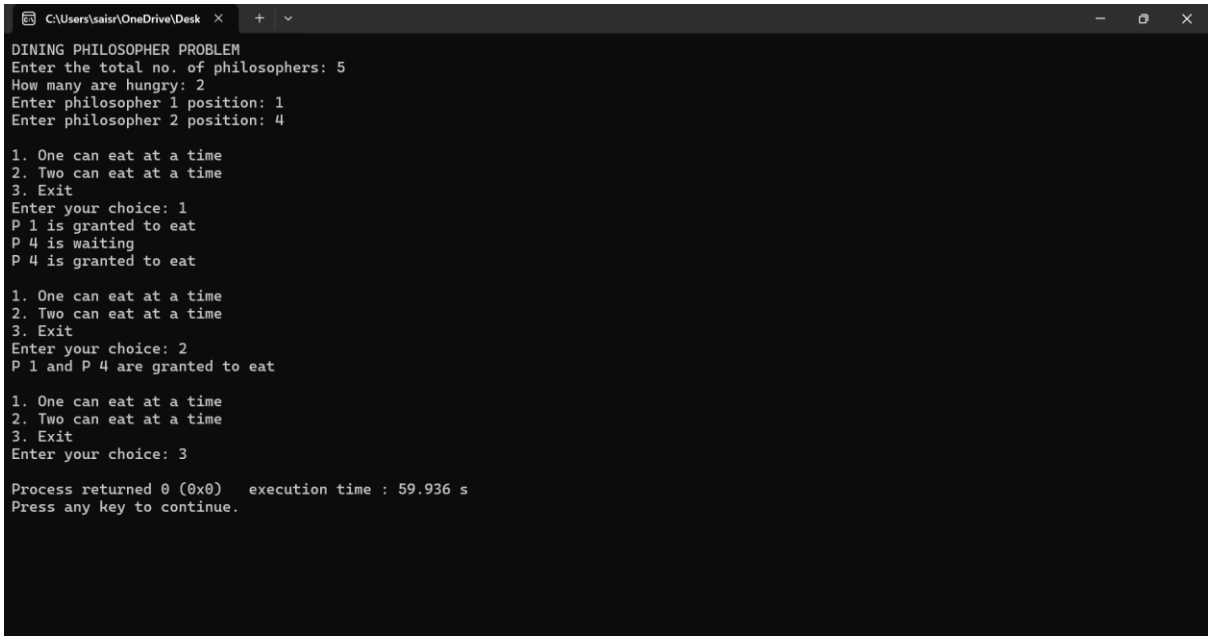
6. Write a C program to simulate the concept of Dining-Philosophers problem.

```
#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:


C:\Users\saisr\OneDrive\Desk
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: 3
Process returned 0 (0x0) execution time : 59.936 s
Press any key to continue.

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

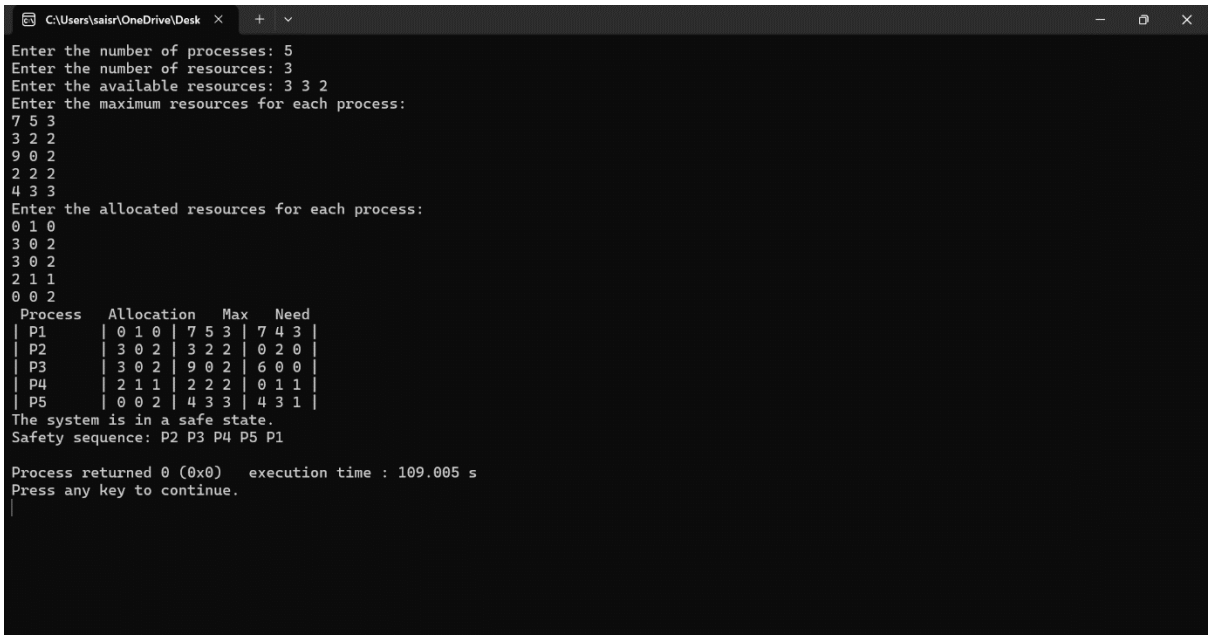
```
#include <stdio.h>

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:

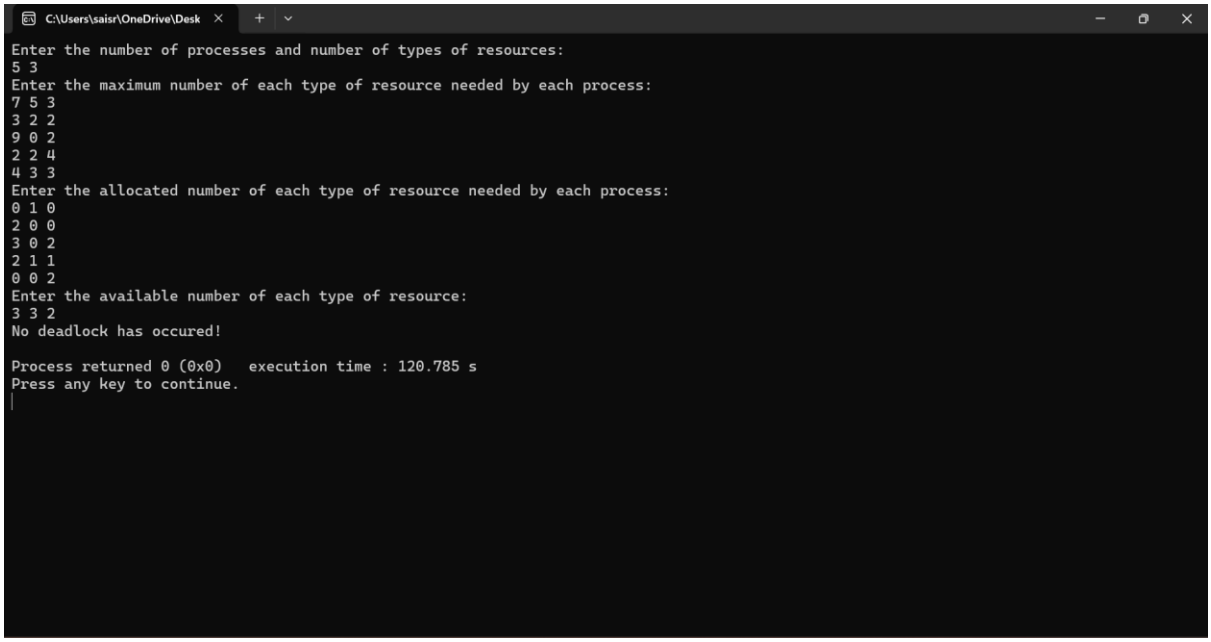


8. Write a C program to simulate deadlock detection

```
#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:


C:\Users\saisr\OneDrive\Desk
Enter the number of processes and number of types of resources:
5 3
Enter the maximum number of each type of resource needed by each process:
7 5 3
3 2 2
9 0 2
2 2 4
4 3 3
Enter the allocated number of each type of resource needed by each process:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available number of each type of resource:
3 3 2
No deadlock has occured!
Process returned 0 (0x0) execution time : 120.785 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

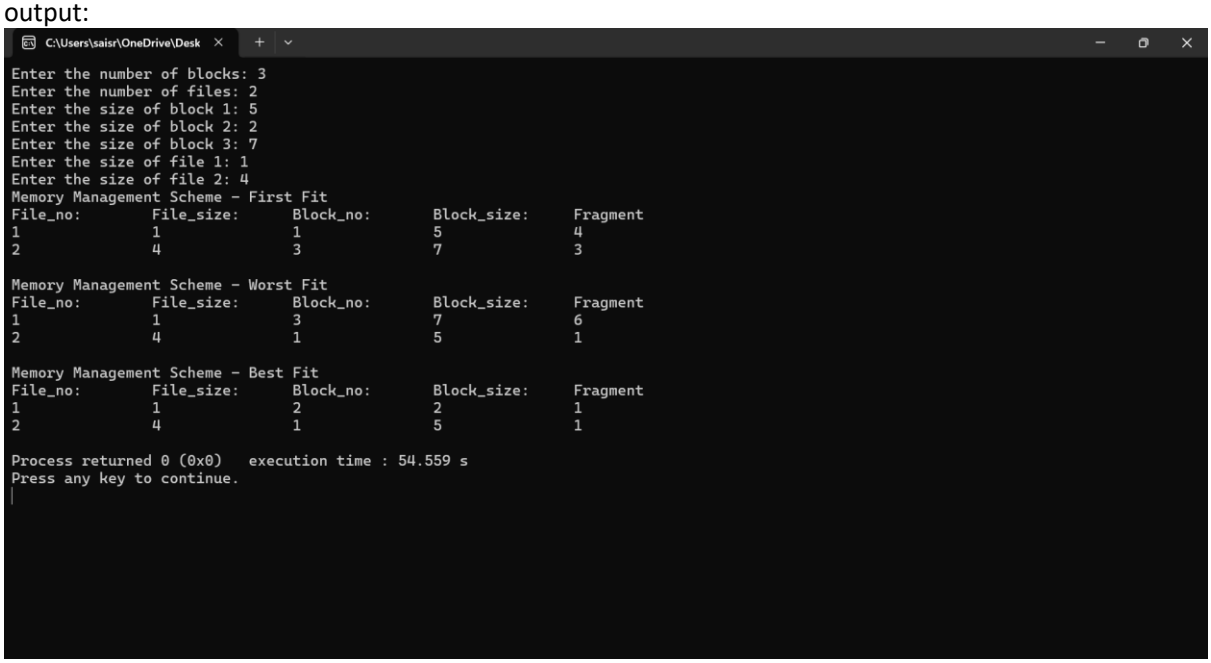
c) First-fit

```
#include <stdio.h>

struct Block {
   int block_no;
   int block_size;
   int is_free; // 1 for free, 0 for allocated
};

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

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

```
            blocks[worst_fit_block].is_free = 0;
            printf("%d\t\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; // Initialize with a large value
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment < min_fragment) {
                    min_fragment = fragment;
                    best_fit_block = j;
                }
            }
        }

        if (best_fit_block!= -1) {
            blocks[best_fit_block].is_free = 0;
            printf("%d\t\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");

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

    worstFit(blocks, n_blocks, files, n_files);
    printf("\n");

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

    bestFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

output:



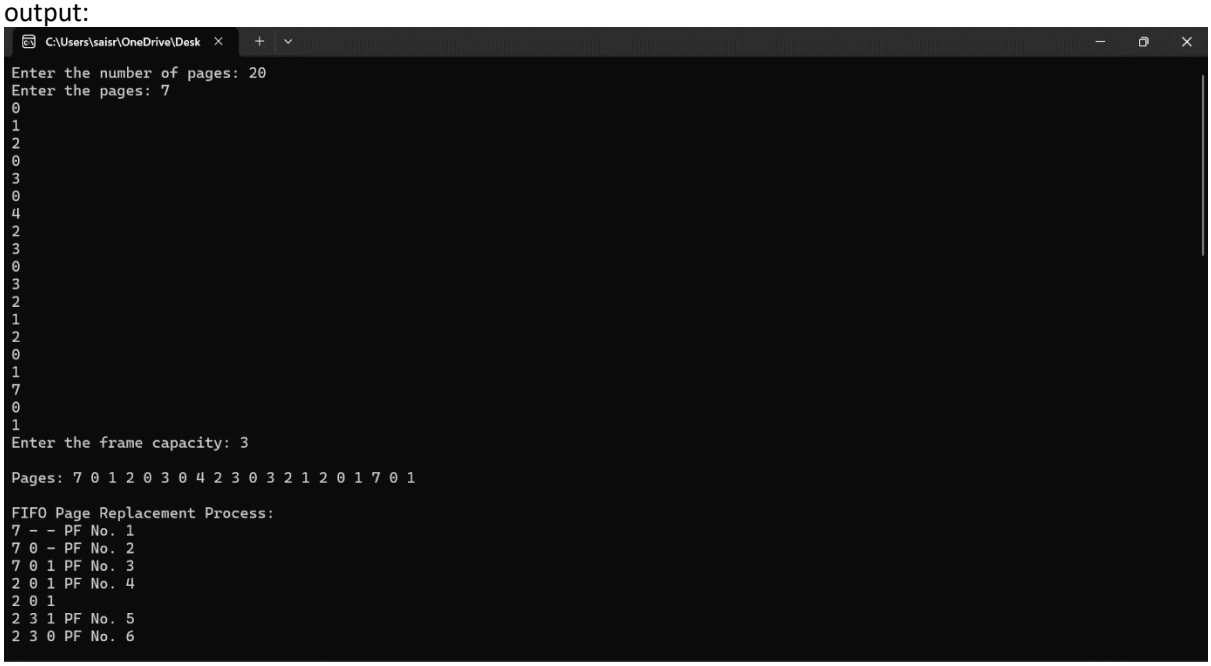
10. Write a C program to simulate paging technique of memory management.

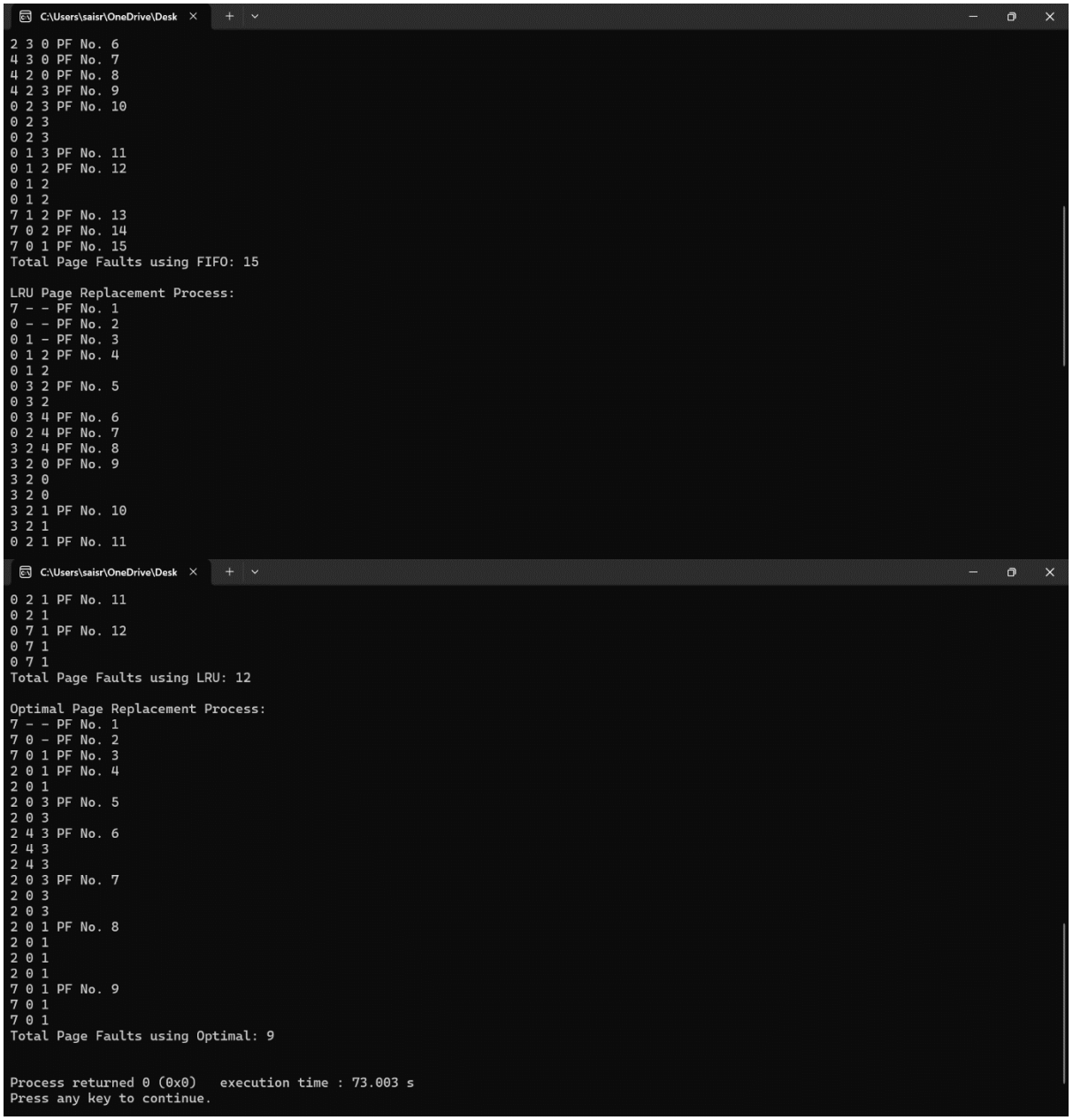
```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

void print_frames(int frame[], int capacity, int page_faults) {
   for (int i = 0; i < capacity; i++) {
      if (frame[i] == -1)
         printf("- ");
      else
         printf("%d ", frame[i]);
   }
   if (page_faults > 0)
      printf("PF No. %d", page_faults);
   printf("\n");
}

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;

   printf("FIFO Page Replacement Process:\n");
   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++;
      }
      print_frames(frame, capacity, found ? 0 : page_faults);
   }
   printf("Total Page Faults using FIFO: %d\n\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;
   }

   printf("LRU Page Replacement Process:\n");
   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++;
      }
      print_frames(frame, capacity, found ? 0 : page_faults);
    }
    printf("Total Page Faults using LRU: %d\n\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;

    printf("Optimal Page Replacement Process:\n");
    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++;
        }
        print_frames(frame, capacity, found ? 0 : page_faults);
    }
    printf("Total Page Faults using Optimal: %d\n\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:




C:\Users\saisr\OneDrive\Desk
2 3 0 PF No. 6
4 3 0 PF No. 7
4 2 0 PF No. 8
4 2 3 PF No. 9
0 2 3 PF No. 10
0 2 3
0 2 3
0 1 3 PF No. 11
0 1 2 PF No. 12
0 1 2
0 1 2
7 1 2 PF No. 13
7 0 2 PF No. 14
7 0 1 PF No. 15
Total Page Faults using FIFO: 15
LRU Page Replacement Process:
7 - - PF No. 1
0 - - PF No. 2
0 1 - PF No. 3
0 1 2 PF No. 4
0 1 2
0 3 2 PF No. 5
0 3 2
0 3 4 PF No. 6
0 2 4 PF No. 7
3 2 4 PF No. 8
3 2 0 PF No. 9
3 2 0
3 2 0
3 2 1 PF No. 10
3 2 1
0 2 1 PF No. 11
C:\Users\saisr\OneDrive\Desk
0 2 1 PF No. 11
0 2 1
0 7 1 PF No. 12
0 7 1
0 7 1
Total Page Faults using LRU: 12
Optimal Page Replacement Process:
7 - - PF No. 1
7 0 - PF No. 2
7 0 1 PF No. 3
2 0 1 PF No. 4
2 0 1
2 0 3 PF No. 5
2 0 3
2 4 3 PF No. 6
2 4 3
2 4 3
2 0 3 PF No. 7
2 0 3
2 0 3
2 0 1 PF No. 8
2 0 1
2 0 1
2 0 1
7 0 1 PF No. 9
7 0 1
7 0 1
Total Page Faults using Optimal: 9
Process returned 0 (0x0) execution time : 73.003 s
Press any key to continue.

~~Scheduling Programming~~

Scheduling Programming

Lab-1 8/5/24

(1) write a c program to simulate the following non preemptive cpu scheduling ~~algorithms~~ algorithm & find turn around time and waiting time
i) fcfs

```
#include<stdio.h>
int main(){
    int n,i;
    float atat=0, awt=0;
    printf("enter the number of process");
    scanf("%d",&n);
    int atime1[n], btime2[n], ctime3[n], tattime4[n], wtime5[n];
    printf("enter arrival time of process");
    for(i=0; i<n; i++){
        scanf("%d", &atime1[i]);
    }
    printf("enter burst time of process");
    for(i=0; i<n; i++){
        scanf("%d",&btime2[i]);
    }
    for(i=0; i<n; i++){
        if(i==0){
            ctime3[i] = atime1[i] + btime2[i];
        }
        else{
            ~~if(ctime3[i-1])~~
            if(ctime3[i-1] < atime1[i]){
                ctime3[i] = (atime1[i] - ctime3[i-1]) + ctime3[i-1] +
                            btime2[i];
            }
            else {
                ctime3[i] = ctime3[i-1] + btime2[i];
            }
        }
    }
    for(i=0; i<n; i++){
        tattime4[i] = ctime3[i] - atime1[i];
```

```
}
for(i=0; i<n; i++){
    wtimes[i] = tattime4[i] - btime2[i];
}
for(i=0; i<n; i++){
    atat = atat + tattime4[i];
}
atat = (atat/n);
for(i=0; i<n; i++){
    awt = awt + wtimes[i];
}
awt = (awt/n);
for(i=0; i<n; i++){
    printf("process id %d arrival time %d burst time %d complete
        time %d turn around time %d waiting time %d \n", i+1,
        atime1[i], btime2[i], ctime3[i], tattime4[i], wtimes[i]);
}
printf("average turn around time is %f ", atat);
printf("average working time is %f ", awt);
}
```

output:

```
enter the number of process4
enter arrival time of process 0
1
5
6
enter burst time of process 2
2
3
4
process id 1 arrival time 0 burst time 2 complete time 2 turn around time 2
waiting time 0
process id 2 arrival time 1 burst time 2 complete time 4 turn around time 3
waiting time 1
Process id 3 arrival time 5 burst time 3 complete time 8 turn around time 3
waiting time 0
process id 4 arrival time 6 burst time 4 complete time 12 turn around time 6
waiting time 2
average tu
1
```

average turn around time

~~SJF (shortest job first)~~
~~non preemptive~~
(b) SJF (Shortest Job First)
non preemptive

```
#include<stdio.h>
int main(){
  int n,i,j,temp1,temp2;
  float atat=0, awt=0;
  printf(" enter the numb
  scanf("%d", &n);
  int atime1[n], btime2
  printf(" enter proces
  for(i=0; i<n; i++){
    scanf("%d", &Pid
  }
  printf("enter arrival
  for(i=0; i<n; i++){
    scanf("%d", &at
  }
  printf(" enter burst
  for(i=0; i<n; i++){
    scanf("%d", &b
  }
  for(i=0; i<n-1; i++)
    for(j=0; j<n-i-1;
      if(btime2[j]>
        temp1 = bt
        btime2[j]
        btime2[j+
        temp2 = P
        Pid[j] =
        Pid[j+1]
      }
  }
}
```

average turn around time is 3.500000 average waiting time is 0.750000

~~sjf (shortest job first)~~
~~non preemptive~~

(b) sjf (shortest job first)
non preemptive

```
#include<stdio.h>
int main(){
  int n,i,j, temp1, temp2;
  float atat =0, awt = 0;
  printf(" enter the number of process ");
  scanf("%d", &n);
  int atime1[n], btime2[n], ctime3[n], tattime4[n], wtimes[n], pid[n];
  printf(" enter process id's ")
  printf(" enter process ids");
  for(i=0; i<n; i++){
     scanf("%d", &pid[i]);
  }
  printf("enter arrival times :");
  for(i=0; i<n; i++){
     scanf("%d", &atime1[i]);
  }
  printf(" enter burst times ");
  for(i=0; i<n; i++){
     scanf("%d", &btime2[i]);
  }
  for(i=0; i<n-1; i++){
     for(j=0; j<n-i-1; j++){
        if(btime2[j] > btime2[j+1]){
           temp1 = btime2[j];
           btime2[j] = btime2[j+1];
           btime2[j+1] = temp1;
           temp2 = pid[j];
           pid[j]
           pid[j] = pid[j+1];
           pid[j+1]
           pid[j+1] = temp2;
        }
     }
  }
```

e %d complete
time %d \n", i+1,
ne4[i], wtimes[i]);

turn around time 2
turn around time 3
turn around time 3
12 turn around time 6

```
}
for(i=0; i<n; i++){
    if(i==0){
        ctime3[i] = atime1[i] + btime2[i];
    }
    else {
        if(ctime3[i-1]< atime1[i]){
            ctime3[i] = (atime1[i]- ctime3[i-1]) + ctime3[i-1]+
                        btime2[i];
        }
        else{
            ctime3[i] = ctime3[i-1] + btime2[i];
        }
    }
}
for(i=0; i<n; i++){
    tattime4[i] = ctime3[i] - atime1[i];
}
for(i=0; i<n; i++){
    wtimes[i] = tattime4[i] - btime2[i];
}
for(i=0; i<n; i++){
    atat = atat + tattime4[i];
}
atat = (atat/n);
for(i=0;i<n);
for(i=0; i<n; i++){
    awt = awt + wtimes[i];
}
awt = (awt/n);
for(i=0; i<n; i++
for(i=0; i<n; i++){
    printf(" process id %d arrival time %d burst time %d complete
        time %d turn around time %d waiting time %d\n ",
        pid[i], atime1[i], btime2[i], ctime3[i], tattime4[i],
        wtimes[i]);
}
```

```
        wtimes[i]);
    }
    printf(" average tur
    printf(" average wa
    return 0;
}
}
output:
enter the number of pro
enter process ids:
enter process ids 1
2
3
4
enter arrival times : 0
0
0
0
enter burst times 6
3
1
2
process id 4 arrival ti
process id 1 arrival tim
process id 3 arrival time
process id 2 arrival time
average turnaroundtime
average wait
average waiting time :
#include<stdio.h>
#define max 10
void
void sif
void sif(int n, int at[
{
    int ct[max];
    int tat[max];
    int wt[max];
    int total_wt;
    int total_wt = 0,
    int total_tat = 0,
    for(int i=0; i<n; i
    {
        ct[i]= -1;
    }
}
```

```
    wtimes[i]);
}
printf("average turnaround time %.f\n", atat);
printf("average waiting time %f\n", awt);
return 0;
}
```

output:

```
enter the number of processes 4
enter process ids 1
2
3
4
enter arrival times : 0
0
0
0
enter burst times 6
8
7
3
```

Process id 4 arrival time 0 burst time 3 complete time 3 turnaround time 3 waiting time 0

Process id 1 arrival time 0 burst time 6 complete time 9 turnaround time 9 waiting time 3

Process id 3 arrival time 0 burst time 7 complete time 16 turnaround time 16 waiting time 9

Process id 2 arrival time 0 burst time 8 complete time 24 turnaround time 24 waiting time 16

average turnaroundtime 13.000000

average waiting time : 7.000000

```
#include<stdio.h>
#define max 10

void sjf(int n, int at[], int bt[])
{
    int ct[max];
    int tat[max];
    int wt[max];
    int total_wt;
    int total_tat;
    for(int i=0; i<n; i++)
    {
        ct[i]=-1;
    }
```

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

P-9
P-9

1) (c)

SJF (shortest job first)
preemptive

```
#include <stdio.h>
#define max 10
int find_min(int arr[], int n){
    int min = arr[0];
    int index = 0;
```

```
    for(int i=1; i<n; i++)
        if(arr[i] < min){
            min = arr[i];
            index = i;
        }
    }
    return index;
}
void SJF_Preemptive(int n, i
    int ct[max] = {0};
    int tat[max] = {0};
    int wt[max] = {0};
    int rt[max];
    int total_wt = 0;
    int total_tat = 0;
    for(int i=0; i<n; i++)
        rt[i] = bt[i];
    }
    int current_time = 0;
    int completed_processes
    while(completed_process
        int available_pr
        int available_co
        for(int i=0; i<n;
            if(at[i] <=
                availa
                availa
                availa
                availa
        }
    }
    if(available_cou
        current_ti
```

```
for(int i=1; i<n; i++){
    if(arr[i] < min){
        min = arr[i];
        index = i;
    }
}
return index;
}

void Sjf_Preemptive(int n, int at[], int bt[]){
    int ct[max] = {0};
    int tat[max] = {0};
    int wt[max] = {0};
    int rt[max];
    int total_wt = 0;
    int total_tat = 0;
    for(int i=0; i<n; i++){
        rt[i] = bt[i];
    }
    int current_time = 0;
    int completed_processes = 0;
    while(completed_processes<n){
        int available_process[max];
        int available_count = 0;
        for(int i=0; i<n; i++){
            if(at[i] <= current_time && rt[i] > 0){
                available_processes[available_count] = i;
                available_count ++;
            }
        }
        if(available_count == 0){
            current_time ++;
```

```
            continue;
        }
        int shortest_Job_index = available_processes[find_min(rt,
                                                    available_num)];

        rt[shortest_Job_index]--;
        current_time++;
        if(rt[shortest_Job_index]==0){
            completed_processes++;
            ct[shortest_Job_index] = current_time;
            tat[shortest_Job_index] = ct[shortest_Job_index] - at[shortest
                                                    _Job_index];
            wt[shortest_Job_index] = tat[shortest_Job_index] - bt[shortest
                                                    _Job_index];

            total_wt += wt[shortest_Job_index];
            total_tat += tat[shortest_Job_index];

        }

    }
    printf("\n process\t arrival time\t burst time\t completion time\t turnaround
            time \t waiting time \n");

    for(int i=0; i<n; i++){
        printf("%d
        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("\n average waiting time %.2f ", (float) total_wt/n);
    printf("\n average turnaround time %.2f ", (float) total_tat/n);

}
int main(){
    int n;
    printf("enter the number of processes ");
    scanf("%d", &n);
    int at[max], bt[max];
```

```
    printf("enter the burst
    for(int i=0;

    printf("enter the arri
    for(int i=0; i<n; i++
        scanf("%d", &at

    }
    printf("enter the burs
    for(int i=0; i<n; i++
        scanf("%d", &bt[

    }
    SJF_Preemptive(n, at,
}
```

output

output

enter the number of proc

Output:

enter the number of pro
enter the arrival time
2
1
4
0
2
enter the burst time
1
5
1
6
3

process	arrival time	b
1	2	
2	1	
3	4	
4	0	
5	2	

average waiting time 4.

```
printf("enter the burst time \n");
for(int i=0;

printf("enter the arrival time \n");
for(int i=0; i<n; i++){
    scanf("%d", &at[i]);

}
printf("enter the burst time \n");
for(int i=0; i<n; i++){
    scanf("%d", &bt[i]);
}
sjf_preemptive(n, at, bt);

}
```

output

output:

enter the number of processes :

Output:

enter the number of process 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	7	6	1
3	4	1	8	4	3
4	0	6	13	13	7
5	2	3	16	14	11

average waiting time 4.40

```
ct[0] = at[0] + bt[0];
tat[0] = ct[0] - at[0];
wt[0] = tat[0] - bt_copy[0];
total_wt += wt[0];
total_tat += tat[0];

for(int i=1; i<n; i++){
    ct[i] = ct[i-1] + bt[i];
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt_copy[i];
    total_wt += wt[i];
    total_tat += tat[i];
}
printf("\n Process\t arrival time\t burst time\t priority\t completion time
        \t turnaroundtime\t waiting time\n");

for(int i=0; i<n; i++){
    printf("%d\t %d\t %d\t %d\t %d\t %d\t %d\t %d\n",
        i+1, at[i], bt_copy[i], p[i], ct[i], tat[i], wt[i]);
}
printf("\n average waiting time %.2f ", (float)total_wt/n);
printf("\n average turnaroundtime %.2f ", (float)total_tat/n);
}
int main(){
    int n;
    printf("enter the number of processes ");
    scanf("%d", &n);
```

```
int at[max], bt[max], p[max];
printf("enter the arrival time\n");
for(int i=0;i<n;i++){
    scanf("%d",&at[i]);
}
printf("enter the burst time\n");
for(int i=0;i<n;i++){
    scanf("%d",&bt[i]);
}
printf("enter the priority\n");
for(int i=0;i<n;i++){
    scanf("%d",&p[i]);
}
priority_non-preemptive(n,at,bt,p);
}
```

output:

```
enter the number of process 4
enter the arrival time
0
1
2
4
enter the burst time
5
4
2
1
enter the burst time
5
4
2
1
enter the priority
10
20
30
40
```

process	arrival time	burst time
1	4	5
2	2	4
3	1	2
4	0	1

average waiting time 5.00
average turnaround time 8.00

1

enter the priority

10

20

30

40

process	arrival time	burst time	priority	tat	wt	ct
1	4	5	40	1	4	5
2	2	4	30	5	1	7
3	1	2	20	10	8	11
4	0	1	10	16	15	16

average waiting time 5.00

average turnaround time 8.00

(b)
2) round robin (experiment with different quantum sizes for rr algorithm)
round robin scheduling.

```
#include<stdio.h>
struct process{
    int pid;
    int burst_time;
    int arrival_time;
    int remaining_time;
};
void roundrobin(struct process processes[], int n, int time_quantum){
    int remaining_processes = n;
    int current_time = 0;
    int completed[n];
    int ct[n], wt[n], tat[n], rt[n];
    for(int i=0; i<n; i++){
        completed[i]=0;
    }
    while(remaining_processes>0){
        for(int i=0; i<n; i++){
            if (completed[i] == 0 && processes[i].arrival_time <= current_time){
                if(processes[i].remaining_time >0){
                    if(processes[i].remaining_time <= time_quantum){
                        current_time += processes[i].remaining_time;
                        processes[i].remaining_time = 0;
                        completed[i] = 1;
                        remaining_processes --;
                        ct[i] = current_time;
                        tat[i] = ct[i] - processes[i].arrival_time;
                    }
                    else{
                        current_time += time_quantum;
                        processes[i].remaining_time -= time_quantum;
                    }
```

```
                }
                wt[i] = ct[i] - processes[i]
                rt[i] = wt[i];
            }
        }
    }
}
printf("Pid\tat\tbt\tct\twt\tt
float avg_tat=0, avg_wt=0;
for(int i=0; i<n; i++){
    printf("%d\t%d\t%d\t%d\t%d\t
    avg_tat += tat[i];
    avg_wt += wt[i];
}
avg_tat /= n;
avg_wt /= n;
printf("\naverage turnaround tim
printf("average waiting time
}
int main(){
    int n, time_quantum;
    printf("enter the number of
    scanf("%d", &n);
    printf("enter the time quant
    scanf("%d", &time_quantum
    struct process processes[n];
}
```

```
    }
    wt[i] = ct[i] - processes[i].arrival_time - processes[i].burst_time;
    rt[i] = wt[i];
  }
 }
}
printf("Pid\tat\tbt\tct\twt\ttat\trt\n");
float avg_tat = 0, avg_wt = 0;
for(int i=0; i<n; i++){
   printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].
                                          arrival_time, processes[i].burst_time,
                                          ct[i], wt[i], tat[i], rt[i]);

   avg_tat += tat[i];
   avg_wt += wt[i];
 }

 avg_tat /= n;
 avg_wt /= n;
 printf("\naverage turnaroundtime %.2f\n", avg_tat);
 printf("average waiting time %.2f\n", avg_wt);
}

int main(){
   int n, time_quantum;
   printf("enter the number of processes");
   scanf("%d", &n);
   printf("enter the time quantum");
   scanf("%d", &time_quantum);
   struct process processes[n];
}
```

```
printf("enter arrival time and burst time for each process\n");

printf("enter arrival time and burst time for each process\n");
for(int i=0;i<n;i++){
    printf("enter arrival time for process %d ",i+1);
    scanf("%d",&proce
    scanf("%d",&processes[i].arrival_time);
    printf("enter burst time for process %d",i+1);
    printf("enter burst time for process %d ",i+1);
    scanf("%d",&processes[i].burst_time);
    processes[i].
    processes[i].pid = i+1
    processes[i].remaining_time = processes[i].burst_time;
}
roundrobin
roundrobin(processes,n,time_quantum);

}
```

Output:

```
enter the number of processes 5
enter the time quantum 2
enter arrivaltime and burst time for each pr
enter arrival time and burst time for each process
enter arrival time for process 1 0
enter burst time for process 1 5
enter arrival time for process 2 1
enter burst time for process 2 3
enter arrival time for process 3 2
enter burst time for process 3 1
enter arrival time for process 4 3
enter burst time for process 4 2
enter arrival time for process 5 4
enter burst time
enter burst time for process 5 3
```

```
pid
```

pid	at	bt	ct
1	0	5	14
2	1	3	12
3	2	1	5
4	3	2	7
5	4	3	13

```
avg
```

average turnaroundtime = 8.2

average turnaroundtime : 8 2

average wai

average waitingtime 5.40

average turnaround time 8

average waiting time 5.4

earliest deadline first sche

earlpest deadline first schedulin

```
#
#include<stdio.h>
typedef struct{
    int capacity;
    int deadline;
    int period;
}task;
int compare_tasks(const voi
    task
    task* task1 = (task*
    task* task2 = (task*
    return (task1->dead
}
int main()
int main(){
    int num_tasks;
```

pid	at	bt	ct	wt	tat	rt
1	0	5	14	9	14	9
2	1	3	12	8	11	8
3	2	1	5	2	3	2
4	3	2	7	2	4	2
5	4	3	13	6	9	6

~~average turnaroundtime = 8.20~~

average turnaroundtime = 8.20

~~average wai~~

average ~~waiting~~time 5.40

average turnaround time 8.20

average waiting time 5.40

~~earliest deadline first sche~~

earliest deadline first scheduling :

```
#include<stdio.h>
typedef struct{
    int capacity;
    int deadline;
    int period;
}task;
int compare_tasks(const void* a, const void * b){
    ~~task * ta~~
    task* task1 = (task*)a;
    task* task2 = (task*)b;
    return (task1->deadline - task2->deadline);
}
~~int main(){~~
int main(){
    int num_tasks;
```

3)
lab-3 6/6/24
Write a C program to simulate the multilevel queue scheduling ~~algo~~ algorithm
multilevel queue

```
#include<stdio.h>
void findwaitingtime(int process[], 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 process[], int n, int bt[], int wt[], int tat[])
{
  for(int i=0;i<n;i++)
  {
    tat[i]= bt[i]+wt[i];
  }
}
```

```
Void roundrobin(int proce
{
int wt[n], tat[n], ct[n],
int remaining_bt[n];
int remaining_bt[n];
int completed=0;
int time=0;
for(int
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]
   {
     if(remai
     if(remaining_bt
     {
       if(remaining_bt[
       {
         time+= remaini
         remaining_bt[i]
         ct[i]=time;
         completed++;
       }
       else
       {
         time+= quantum
         remaining_bt[i]
       }
     }
   }
  }
}
```

duling ~~algo~~ algorithm

t at[], int wt[])

int wt[], ~~int tat~~]

```
void roundrobin(int process[], 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(process, n, bt, at, wt);
findturnaroundtime(process, n, bt, wt, tat);
printf("process burst time arrival time waiting time turnaround time
        completion completiontime \n");
for(int i=0; i<n; i++)
{
  printf("p%d \t\t %d \t\t %d \t\t %d \t\t %d \n", process[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 turnaroundtime (round robin) = %f \n", (float)
                                                  total_tat/n);
}
```

```
void fcfs(int process[], int n
{
  int wt[n], tat[n], ct[n],
  findwaitingtime(
  findwaitingtime(process, n
  findturnaroundtime(process,
  printf("process burst 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
            process[i], bt[i],
    total_wt += wt[i];
    total_tat += tat[i];
  }
  printf("average waiting tim
  printf("average turnaround t
}
int main()
{
int process[] = {1,2,3,4,5};
int n = sizeof(process) / sizeof(
int bt = {10,5,8,12,15};
int at[] = {0,1,2,3,4};
int quantum = 2;
```

```
void fcfs(int process[], int n, int bt[], int at[])
{
  int wt[n], tat[n], ct[n], total_wt=0, total_tat=0;
  int wt[n], tat[n], ct[n], total_wt=0, total_tat=0;
  findwaitingtime(process, n, bt, at, wt);
  findturnaroundtime(process, n, bt, wt, tat);
  printf("process bursttime arrivaltime waitingtime turnaroundtime
        completion time \n");
  for(int i=0; i<n; i++)
  for(int i=0; i<n; i++)
  {
    ct[i]=at[i]+bt[i];
    printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
        process[i], bt[i], at[i], wt[i], tat[i], ct[i]);
    total_wt+=wt[i];
    total_tat+=tat[i];
  }
  printf("average waiting time (fcfs)=%d\n", (float)total_wt/n);
  printf("average turnaround time (fcfs)=%d\n", (float)total_tat/n);
}
int main()
{
int process[]={1,2,3,4,5};
int n=sizeof(process)/sizeof(process[0]);
int bt={10,5,8,12,15};
int at[]={0,1,2,3,4};
int quantum=2;
```

~~int quan~~

~~int quantum;~~

int quantum = 2;

roundrobin (process, n, bt, at, quantum);

fcfs (process, n, bt, at);

}

Output:

process	bt	at	wt	tat	ct
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 tt~~

Average wt (round robin) = 14.60

~~average turnaroundtime~~ 0

average turnaroundtime (round robin) = 24.60

process	bt	at	wt	tat	ct
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 wt (fcfs) = 14.60

Average tat (fcfs) = 24.6

(a) 4)

rate monotonic:

```
#include<stdio.h>
struct process{
    int execution_time;
    int time_period;
};
```

```
int lcm(int a, int b)
{
  int max = (a>b)?a:b;
  while(1)
  {
  if(max%a==0 &&
      return max;
  }
  if(max%a==0 &&
      ~~return(max);~~
      return(max);
  }
  max++;
}
}
int is_schedulable(struct
{
  float utilization = 0.0
  for(int i=0;i<n;i++)
  {
    utilization += (float)
    process[i].time_peri
  }
  return (utilization <= 1.0
}
~~int main()~~
{
  struct process process[3
int main()
{
  struct processes process
  {
    {3,20};
    {2,5};
  }
```

```
int lcm(int a, int b)
{
  int max = (a>b)? a : b;
  while(1)
  {
  if(max % a == 0 && max % b == 0){
     return max;
  }
  if(max % a == 0 && max % b == 0){
      return(max);
  }
  max++;
 }
}
int is_schedulable(struct process process[], int n)
{
  float utilization = 0.0
  for(int i=0; i<n; i++)
  {
    utilization += (float) process[i].execution_time;
    process[i].time = period;
  }
  return (utilization <= 1.0);
}
int main()
{
  struct process process[]
int main()
{
  struct processes processes[] =
  {
     {3,20};
     {2,5};
     |
```

```
    {1,10},
    {2,10}};
int n = sizeof(process)/sizeof(process[0]);
if(!is_schedulable(processes,n))
{
  printf("the given set of process is not schedule\n");
  return 0;
}
int scheduling_time = lcm(processes[0].
int scheduling_time = lcm(processes[0].time_period, process[1].time_period);
printf("execution order\n");
if(t%.processes[1].time_period==0)
  printf(
if(t%.p0
if(t%.processes[1].time_period==0){
  printf("P2");
}
if(t%.processes[2].time_period==0){
  printf("P3");
}
if(t%.processes[2].time_period==0){
  printf("P3");
}
if(t%.process[0]
if(t%.processes[0].time_period==0){
  printf("P1");
}
}
}
printf("\n");
}
output:
exe
output:
execution or
output:
execution order
P2 P3 P1
output:
execution order
P2 P3 P1 P2 P2 P3 P2
```

(b)
earliest deadline first scheduling

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
void sort(int proc[], int d[],
{
  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(
  }
  return lcm;
}
void main()
{
  int n;
  printf(" enter the numbe
  scanf("%d", &n);
  int proc[n], b[n], pt[n],
  printf(" enter the cpu bu
  for(int i=0; i<n; i++)
  {
    scanf(" %d", &b[i]);
    rem[i]= b[i];
  }
  printf(" enter the deadli
  for(int i=0; i<n; i++){
    scanf("%d", &d[i]);
  }
  printf(" enter the time pe
  for(int i=0; i<n; i++){
    proc[i]= i+1;
  }
  sort(proc, d, b, pt, n);
  int l= lcmul(pt, n);
  printf(" earliest deadline
  printf(" Pid \t burst \t dead
  for(int i=0; i<n; i++){
    printf("%d \t \t %d
          pt[i]);
}
```

```
    lcm = (lcm * pt[i])/gcd(lcm, pt[i]);
  }
  return lcm;
}
void main()
{
  int n;
  printf(" enter the number of process");
  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++){
    proc[i] = i+1;
  }
  sort(proc, d, b, pt, n);
  int l = lcmul(pt, n);
  printf(" earliest deadline scheduling \n");
  printf(" Pid \t burst \t deadline \t period \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;
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 % p[i] == 0 && time != 0)
   {
      nextdeadlines[i] = time + d[i];
      rem[i] = b[i];
   }
}
int mi
int mindeadline = l+1;
int task to
int tasktoexecute = -1;
for(int i=0; i<n; i++)
{
  if(rem[i] > 0 && nextdeadlines[i] < mindeadline)
  {
     mindeadline = nextdeadlines[i];
     task toexecute =
     tasktoexecute = i;
  }
}
if(tasktoexecute != -1)
{
  printf(" %d ms : task %d is running \n", time, proc[tasktoexecute]);
  rem[tasktoexecute]
  }
  else
  {
    printf("%d ms : cpu i
  }
  time++;
 }
}
```

output:

```
enter the number of process 3
enter cpu burst time
3
2
2
enter the deadlines
7
4
8
enter
enter the timeperiod
20
5
10
earliest deadline scheduling
Pid   burst   deadlines   pe
2     2       4
1     3       7
3     2       8
Scheduling occurs for 20ms
0ms : task 2 is running
1ms : task 2 is running
3ms : task 1 is running
2ms : task 1 is running
3ms : task 1 is running
4ms : task 1 is running
5ms : task 3 is running
6ms : task 3 is running
7ms : task 2 is running
8ms : task 2 is running
9ms : cpu is idle
```

```
        rem[task to execute]--;
    }
    else
    {
      printf("%d ms : cpu is idle to");
      printf("%d ms  cpu is idle \n", time);
    }
  }
  Enter 11;
 }
  time++;
 }
}
```

output:

enter the number of process 3
enter cpu burst time
3
2
2
enter the deadlines
7
4
8
enter
enter the timeperiod
20
5
10
earliest deadline scheduling

pid	burst	deadlines	period
2	2	4	5
1	3	7	20
3	2	8	10

Scheduling occurs for 20ms
0ms: task 2 is running
1ms: task 2 is running
3ms: task 1 is running
2ms: task 1 is running
3ms: task 1 is running
4ms: task 1 is running
5ms: task 3 is running
6ms: task 3 is running
7ms: task 2 is running
8ms: task 2 is running
9ms: CPU is idle

proc[task to execute]);

```
10 ms  task 2 is running
11 ms  task 2 is running
12 ms  task 3 is running
13 ms  task 3 is running
14 ms  cpu is idle
15 ms  task 2 is running
16 ms  task 2 is running
17 ms  cpu is idle
18 ms  cpu is idle
19 ms  cpu is idle
```

(c) proportional scheduling

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define max_tasks 10
#define max_ticket 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_ticket = 10
    int total_ticket = 0;
    for(int i=0; i<num_tasks; i++){
        total_ticket += tasks[i].tickets;
    }
    srand(time(NULL));
    int current_time = 0;
    int completed = 10
    int c
    int completed_tasks = 0;
    printf("process scheduling \n");
    while(completed_tasks < num_tasks){
        int winning_ticket = rand() % total_tickets;
        if
```

```
int cumulative_ticket =
for(int i=0; i< num_tas
    cumulative_ticke
    if
    if(winning_ticke
    if (winn
    if (winning_tic
        printf("time
            cur
        current_time
        current_tim
        break;
    }
}
completed_tasks++;
}
*time_span_ms = current
}
int main(){
    struct task tasks[max_t
    int num_tasks;
    int time_span_ms;
    printf("enter the numbe
    scanf("%d", &num_tas
    if(num_tasks <= 0 || num
        printf("invalid numb
            1 and %d
        return;
        return(1);
}
```

```
int cumulative_tickets = 0;
for(int i=0; i< num_tasks; i++){
    cumulative_tickets += tasks[i].tickets;
    if(winning_ticket < cumulative_tickets)
    if(winn
    if (winning_ticket < (cumulative_tickets){
        printf("time %d - %d task %d is running\n",
                current_time, current_time+1, tasks[i].tid);
        current_time
        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 number between
            1 and %d \n", max_tasks);
    return(1);
  }
```

```
printf("enter
printf("enter number of tickets for each task \n");
for(int i=0; i<num_tasks; i++)
for(int i = 0, i< num_tasks; i++){
    tasks[i].tid = i+1;
    printf("task %d ticket
    printf("task %d ticket", tasks[i].tid);
    scanf(
    scanf("%d", &tasks[i].tickets);
}
printf("\n running tasks \n");
schedule(tasks, num_tasks, &time_span_ms);
printf("\n time span of the gantt chart : %d milliseconds
printf("\n time span of the gantt chart %d milliseconds \n",
                                    time_span_ms);
                                    time_span_ms);
}
```

output:

```
enter the number of tasks 3
enter the number of tickets for each task
task 1 tickets 10
task 2 tickets 20
task 3 tickets 30
running tasks
process schet
process scheduling
process scheduling
process scheduling
time 0-1 task 3 is running
time 1-2 task 3 is running
time 2-3 task 1 is running
```

time span of gantt chart

time span of gantt char

12-06-2024

5) write a c program to
Semaphores

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty
int mai
int main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1. producer \n
while(1)
{
  printf"
  printf("\n enter your c
  printf("\n enter your ch
  scanf("%d",&n);
  switch(n)
  {
    case 1 : if((mu
    {
      pr
    }
    else{
    }
    break
    case 2 : if((m
    {
      co
```

time span of gantt chart

time span of gantt chart 300 milliseconds

12-06-2024

5) Write a c program to simulate producer consumer problem using semaphores

```
#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("
printf("\n enter your choice");
printf("\n enter 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;
        }
        }
        break;

    case 3 :   exit(0);
               break;

      }
   }
}
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 item %d", x);
   mutex = signal(mutex);
}
void consumer()
{
```

```
void consumer()
{
   mutex = wait(mutex);
   full = wait(full);
   empty = signal(empty);
   printf("\nconsumer co
   x--;
   mutex = signal(mutex);
}
```

output:

```
1. producer
2. consumer
3. exit
enter your choice 1
Producer produces item 1
enter your choice 1
Producer produces item 2
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
```

```
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 1

producer

Producer produces item 1

enter your choice 1

producer

produ

Producer produces item 2

enter your choice 2

enter your choice 2

consumer consumes item 2

enter your choice 2

enter your choice 2

consumer consumes item 1

enter you

ente

enter your choice 2

buffer is empty

enter yo

buffer is empty

enter your choice 3

enter your choice 3

19/06/2024

6) write a c program to simulate the concept of dining philosophers problem

```
#include<stdio.h>
#include<stdlib.h>
#define max_philosophers 10
typedef enum{thinking, hungry, eating} state_t;
state_t states[max_philosophers];
int num_philosophers;
int num_hungry
int hungry_philosophers[
int hungry_philosophers[max_philosophers];
int forks[max_philosophers];
void print_state(){
    printf("\n");
    for(i=0;i<num_philosophers;++i){
        if(states[i]==thinking)
        {
            printf("P%d is think\n",i+1);
        }
        else if(states[i]==hungry)
        {
            printf("P%d is waiting\n",i+1);
        }
        else if(states[i]==eating)
        {
            printf("P%d is eating\n",i+1);
        }
    }
}
int can_eat(int philosopher_id){
    int left_fork=philosopher_i
    int left_fork=philosopher_id;
    int right_fork=(philosopher_id+1)%num_philosophers;
    if(forks[left_fork]==0&&forks[right_fork]==0){
        forks[left_fork]=forks[right_fork]=1;
        return(1);
    }
```

```
    return 0;
}
void simulate(int allow_hu
    int eating_count=0;
    for(int i=0;i<num_hu
        int philosopher_id=
        if(states[philosophe
            if(can_eat(phil
                states[phil
                eating_coun
                printf("P%
                if(!allow_h
                {
                    break;
                }
                if(allow_hu
                    break;
                }
            }
        }
    }
}
for(int i=0;i<num_h
    int philosopher_id
    if(states[philosoph
        int left_fork
        int rig
        int left_fork
        int right_for
    }
```

philosophers

```
    return 0;
}
void simulate(int allow_two){
    int eating_count = 0;

    for(int i = 0; i< num_hungry; ++i){
        int philosopher_id = hungry_philosophers[i];
        if(states[philosopher_id] == hungry){
            if(can_eat(philosopher_id)){

                states[philosopher_id] = eating;

                eating_count++;
                printf("P %d is granted to eat \n", philosopher_id +1);
                if(!allow_two && eating_count == 1)
                {
                    break;
                }
                if(allow_two && eating_count == 2){
                    break;
                }
            }
        }
    }
}

for(int i = 0; i<num_hungry ; i++){
    int philosopher_id = hungry_philosophers[i];

    if(states[philosopher_id] == eating){
        int left_fork = philosopher_id;
        int rig
        int left_fork = philosopher_id;
        int right_fork = (philospher_id +1) % num_philosophers;
    }
```

```
    }
    forks[left_fork] = forks[right_fork] = 0;
    states[philosopher_i]
    states[philosopher_id] = thinking;
  }
 }
}
int main(){
  printf(" enter the total number of philosophers (max %d) :", max_philosophers);
  scanf("%d", &num_philosophers);
  if(num_philosophers < 2 || num_philosophers > max_philosophers){
    printf(" invalid number
    printf("invalid number of philosophers exiting \n");
    return 1;
  }
  printf(" how many are hungry \n");
  printf(" how many are hungry ");
  scanf("%d", &num_hungry);
  for(int i=0; i< num_hungry ; i++){
    printf(" enter philosopher %d position",
    printf("enter philosopher %d position", i+1);
    int position;
    scanf("%d", &position);
    hungry_philosophers[i] = position-1;
    states[hungry_philosophers[i]] = hungry;
  }
  for(int i=0; i< num_philosophers ; i++
  for(int i=0; i<num_philosophers; i++){
    forks[i] = 0;
  }
```

```
  int choice;
  do{
    print_state();
    printf("\n1. one can eat at a ti
    printf("2 two can eat at a time
    printf("3 exit\n");
    printf(" enter your choice");
    printf(" enter your choice ");
    scanf("%d", &choice);
    switch(choice){
      case 1 : simulate(0);
               break;
      case 2 : simulate(1);
               break;
      case 3 : printf("exiting\n"
               break;
      default : printf("invalid ch
               break;
    }
  }while(choice!=3);
}
output:
output-
enter total number of philosophe
how many are hungry : 2
enter philosopher 1 position
enter philosopher 2 position
1. one can eat at a time
2. two can eat at a time
3. exit
enter your choice 1
P1 is granted to eat
```

```
int choice;
do{
print_state();
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 : simulate(0);
         break;
case 2 : simulate(1);
         break;
case 3 : printf("exiting\n");
         break;
default: printf("invalid choice please try again\n");
         break;
}
}while(choice!=3);
}
```

output:

```
enter total number 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
P1 is granted to eat
```

P4 is granted to eat

1. one can eat at a time
2. two can eat at a time
3. exit

enter your choice : 2

~~P1 and P4 are grd~~

P1 and P4 are granted to eat

1. one can eat at a time
2. two can eat at a time
3. exit

enter your choice : 3

7) Write a c program to simulate bankers algorithm for the purpose of deadlock avoidance

```
#include<stdio.h>
int P,R;
void avai(int all[P][R],int tot[R],int avail[R]){
    for(int j=0;j<R;j++){
        avail[j]=0;
        for(int k=0;k<P;k++){
            avail[j]+=all[k][j];
        }
    }
    for(int j=0;j<R;j++){
        avail[j]=tot[j]-avail[j];
    }
}
```

```
}
void need(int all[P][R], int max
    for(int i=0;i<P;i++){
        for(int j=0;j<R;j++){
            needmat[i][j]= max
        }
    }
}
void safety(int P, int R, int all[P]
    int f[P], c, count=0, h=0;
    for(int i=0;i<P;i++){
        f[i]=0;
    }
    while(count<P && h<P){
        for(int i=0;i<P;i++){
            if(f[i]==0){
                c=0;
                for(int j=0;j<
                    if(needmat[
                        c=(+1);
                    }
                }
                if(c==R){
                    printf("P%d is
                    for(int k=0;k
                        avail[k]+=
                        avail[k]+=
```

the purpose of deadlock

```
}
void need(int all[P][r], int max[P][r], int needmat[P][r]){
   for(int i=0;i<P;i++){
      for(int j=0; j<r; j++){
         needmat[i][j]= max[i][j]-all[i][i];
      }
   }
}
void safety(int P, int r, int all[P][r], int avail[r], int needmat[P][r], int seq[P]){
   int f[P], c, count=0, h=0;
   for(int i=0; i<P; i++){
      f[i]=0;
   }
   while(count<P && h<P){
      for(int i=0; i<P; i++){
         if(f[i]==0){
            c=0;
            for(int j=0; j<r; j++){
               if(needmat[i][j]<=avail[j]){
                  c=c+1;
               }
            }
            if(c==r){
               printf("P %d is visited(\t", i);
               for(int k=0; k<r; k++){
                  avail[k]+=
                  avail[k]+= all[i][k];
```

```
                printf("%d",
                printf("%d", avail[k]);
                printf("%d"
                printf("%d", avail[k]);
                printf("\t");
            }
            printf(")\n");
            f[i]=1;
            count=count+1;
            seq[h]=i;
            h=h+1;
        }
    }
}
}
int main(){
    printf("enter the number of processes");
    scanf("%d",&p);
    printf("enter the number of resources");
    scanf("%d",&r);
    int tot[r], needmat[p][r], avail[r], seq[p];
    printf("enter the total instances of each resource");
    for(int i=0;i<r;i++)
    for(int i=0;i<r;i++){
        scanf("%d",&tot[i]
        scanf("%d", &tot[i]);
        scanf("%d",&tot[i]);
}
tot
int all[p][r], max[p][r];
}
```

```
printf
printf("enter details of each pro
for
for(int i=0;i<p;i++){
    for(int j=0;j<r;j++){
        scanf("%d",&all[i][j]
    }
}
printf("enter maximum matrix
for(int i=0;i<p;i++){
    for(int j=0;j<r;j++){
        scanf("%d",&max[i][j]
    }
}
avail(all,tot,avail);
avail(all,tot,avail);
need(all,max,needmat)
need(all,max,needmat);
printf("need matri
printf("
printf(" need matrix\n");
for(int i=0;i<p;i++){
    for(int j=0;j<r;j++){
        printf("%d\t",needmat
    }
    printf("\n");
    Printf("\n");
}
Safety(p,r,all,avail,needmat,seq
printf("safe sequence is
Printf("safe sequence is ");
}
```

```
printf("enter details of each process (allocation matrix)\n");
for(int i=0; i<p; i++){
    for(int j=0; j<r; j++){
        scanf("%d", &all[i][j]);
    }
}
printf("enter maximum matrix\n");
for(int i=0; i<p; i++){
    for(int j=0; j<r; j++){
        scanf("%d", &max[i][j]);
    }
}
avail(all, tot, avail);
need(all, max, needmat);
printf("
printf(" need matrix\n");
for(int i=0; i<p; i++){
    for(int j=0; j<r; j++){
        printf("%d\t", needmat[i][j]);
    }
    printf("\n");
}
Safety(p, r, all, avail, needmat, seq);
Printf("Safe sequence is ");
```

~~for(int i=0; i<p; i++){~~

~~for(int i=0~~

```
for(int i=0;i<p;i++){
    printf("p%d|",seq[i]);
}
}
```

output:

enter the number of processes 5
enter the number of resources 3
enter the available resources 3 3 2
enter maximum resources for each processes 7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
~~enter the allocated proc~~
enter the allocated resources for each processes 0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

process	allocation	max	need
P1	0 1 0	7 5 3	7 4 3
P2	3 0 2	3 2 2	0 2 0
P3	3 0 2	9 0 2	6 0 0
P4	2 1 1	2 2 2	0 1 1
P5	0 0 2	4 3 3	4 3 1

~~Safety sequence P2 P3~~

Safety sequence P2 P3 P4 P5 P1

3/7

03/07/2024

8) write a c program to simulate deadlock detection

~~#includecs~~

```
#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);
scanf("%d %d",&n,&m);
int max[n][m], need[n][m
c=0;
printf("enter the maximum
      each process \n");

for(i=0;i<n;i++)
{
  for(j=0;j<m;j++)
  {
    scanf("%d",&max[i
  }
}
printf("enter the allocated
      each process \n");
for(i=0;i<n;i++)
{
  for(j=0;j<m;j++)
  {
    scanf("%d",&all[i
  }
}
printf("enter the available 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
  }
}
}
```

```
scanf("%d %d",&n,&m);
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=c+1;
                if(c==m)
                {
                    for(j=0;j<m;j++)
                    {
                        ava[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=j+1;
        flag=1;
    }
}
if(flag==1)
{
    printf(" deadlock has occur
    printf(" the deadlock proces
    printf(" the deadlock proces
    for(i=0;i<n;i++)
    {
        printf("%d", dead[i
    }
}
else
{
    printf(" no deadlock has
}
}
```

Output:

```
enter the number of proces
s 3
enter the maximum number
each process
```

```
}
j=0;
flag=0;
for(i=0;i<n;i++)
{
  if(finish[i]==0)
  {
     dead[j]=i;
     j=j+1;
     flag=1;
  }
}
if(flag==1)
{
  printf("deadlock has occured\n");
  printf("the deadlock processes are\n");
  for(i=0;i<n;i++)
  {
     printf("%d", dead[i]);
  }
}
else
{
  printf("no deadlock has occured\n");
}
}
```

Output:

enter the number of process and number of types of resources
5 3
enter the maximum number of each type of resource needed by each process

7 5 3
3 2 2
9 0 2
2 2 4
4 3 3

enter the allocated number of eac

enter the allocated number of each type of resource needed by each process

0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

enter the available number of each type of resource

3 3 2

no deadlock has occurred

9) write a c program to simulate the following contiguous memory allocation techniques
(a) worst-fit
(b) best-fit
(c) first-fit

```
#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 ma
printf("file_no\t fi
for(int i=0; i<n_file
    for(int j=0; j<
        if(blocks[j
    }
}
}
void worstfit(struct block bl

printf("memory manag
printf("file_no\t file_
for(int i=0; i<n_files
    int worst_fit_block
    int max_fragment
    for(int j=0; j<n_
        if(blocks[j].i
            int fragm
            if(fragm
```

```
printf("memory management scheme - first fit\n");
printf("file_no\t file_size\t block_no\t block_size\t fragment\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\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\t file_size\t block_no\t block_size\t fragment\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\t file_size\t block_no\t block_size\t fragment\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 frag;
                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_fi]. bloc
        printf("%d\t\t%d\t
            files[i].file_si
            blocks[best_
        }
    }
}
int main(){
    int n_blocks, n_files;
    printf("enter number of
    scanf("%d", &n_blocks)
    printf("
    printf("enter number of
    scanf("%d", &n_files);
    struct block blocks[n_bl
    for(int i=0; i<n_blocks
    {
        blocks[i].block_no =
        printf("enter size of
        blocks[i].is_free = 1;
    }
    struct file files[
    struct file files[n_files]
    for(int i=0; i<n_files;
        printf("enter size of
        scanf("%d", &files[
    }
    firstfit(blocks, n_blocks, fi
```

```
        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].file_size, min_fragment);
    }
}

int main(){
    int n_blocks, n_files;
    printf("enter number of blocks ");
    scanf("%d", &n_blocks);
    printf("enter 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 size of block %d", i+1);
        blocks[i].is_free = 1;
    }
    struct file files[n_files];
    for(int i = 0; i<n_files; i++){
        printf("enter 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);
}
bestfit(blocks, n_blocks, files, n_files);
}
```

Output:

```
enter the number of blocks 3
enter the number of files 2
enter the size of block1 5
enter the size of block2 2
enter the size of block3 7
enter the size of file1 1
enter the size of file2 4
memory management scheme- first fit
```

file no	file size	block no	block size	fragment
1	1	1	5	4
2	4	3	7	3

memory management scheme worst-fit

file no	file size	block no	block size	fragment
1	1	3	7	6
2	4	1	5	1

memory management scheme best-fit

file no	file size	block no	block size	fragment
1	1	2	2	1
2	4	1	5	

10/07/2024

10) write a c program to simulate pag
memory management

```
#include<stdio.h>
#include<limits.h>
#include<stdlib.h>
void print_frames(int frame[], int capacity,
    for(int i=0; i< capacity; i++){
        if(frame[i]==-1)
        {
            printf("-");
        }
        else
        {
            printf("%d", frame[i]);
        }
    }
    if(page_faults>0)
    {
        printf("PF no %d", page_faults
        printf("PF no %d", page_faults);
    }
    printf("\n");
}
void fifo(int pages[], int n, int capacity)
    int frame[capacity], index=0, page_fa
    for(int i=0; i< capacity; i++)
    {
        frame[i]=-1;
    }
    printf("Fifo page replacement Process
    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;
            }
    }
```

10/07/2024

10) write a c program to simulate paging technique of ~~memory management~~ memory management

```
#include<stdio.h>
#include<limits.h>
#include<stdlib.h>
void Print_Frames(int frame[], int capacity, int Page_faults){
   for(int i=0; i< capacity ; i++){
       if(frame[i]==-1)
       {
         Printf("-");
       }
       else
       {
         printf("%d", frame[i]);
       }
   }
   if(Page_faults > 0)
   {
     ~~printf(" PF no %d", Page_faults~~
     printf(" PF no %d", Page_faults);
   }
   Printf("\n");
}
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;
   }
   printf(" Fifo page replacement Processes \n");
   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;
           }
   }
```

Fragment
4
3

Fragment
6
1

Fragment
1

```
        if(!found){
            frame[index] = pages[i];
            index = (index+1) % capacity;
            page_faults++;
        }
        print_frames(frame, capacity, found?0: page_faults);
    }
    printf("total page faults %d\n\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;
    }
    printf("lru page replacement process\n");
    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++;
        }
        print_frames(frame, capacity, found?0: page_faults);
    }
    printf("total page faults using
}
void optimal(int pages[], int n,
    int frame[capacity], page_
    for(int i=0; i<capacity; i+
    {
        frame[i]=-1;
    }
    printf("optimal page repl
    for(int i=0; i<n; i++){
        int found=0;
        for(int j=0, j<capaci
            if(frame[j]==
                found=1;
                break;
                break;
            }
        }
        if(!found){
            int farthest=i+1
            for(int j=0; j<co
                int k;
                for(k=i+1; k
                    if(frame
                    {
                        brea
                    }
```

```
            min_index = j;
        }
    }
    frame[min_index] = pages[i];
    counter[min_index] = time++;
    page_faults++;
}
print_frames(frame, capacity, found ? 0 : page_faults);
}
printf("total page faults using lru %d\n\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;
    }
    printf("optimal page replacement process \n");
    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;~~
                ~~}~~;
                break;
            }
        }
        if(!found){
            int farthest = i+1, index = -1;
            ~~for(int j=0; j<capacity; j~~++
            for(int j=0; j< capacity; j++){
                int k;
                ~~for(k=i+1;~~
                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++;
    }
    print_frames(frame, capacity, found ? 0 : page_faults);
}
printf("total page faults using optimal %d \n\n", page_faults);
printf("total page faults using Ophimal %d \n\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)
    for(int i=0; i<n; i++){
        scanf("%d", &pages[i]);
    }
    printf("enter the frame capacity ");
    scanf("%d",
    scanf("%d"
    Scanf("%d", &capacity);
    printf("\n Pages ");
    for(int i=0; i<n; i++){
        printf("%d", pages[i]);
    }
    printf("\n\n");
    fifo(pages, n, capacity);
    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    ophimal(pages, n, capacity);
    Secfo
    free(pages);
}
```

output :

```
enter the number of pages 20
enter the pages 7,0,1,2,0,3,0
enter the frame capacity 3
pages: 7 0 1 2 0 3 0 4 2 3 0
fifo page
```

fifo page replacement process

```
7 - - PF no 1
7 0 - PF no 2
7 0 1 - PF no 3
2 0 1 - PF no 4
2 0 1
2 3 1 - PF no 5
2 3 0 - PF no 6
4 3 0 - PF no 7
4 2 0 - PF no 8
4 2 3 - PF no 9
0 2 3 - PF no 10
0 2 3
0 2 3
0 1 3 - PF no 11
0 1 2 - PF no 12
0 1 2
0 1 2
7 1 2 - PF no 13
7 0 2 - PF no 14
7 0 1 - PF no 15
```

total page faults using fifo 15

```
    for(int i=0; i<n; i++){
        printf("%d", pages[i]);
    }
    printf("\n\n");
    ~~fifo(pages, n, capacity);~~
    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    optimal(pages, n, capacity);
    ~~fcfs();~~
    free(pages);
}
```

output :

enter the number of pages 20
enter the pages 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
enter the frame capacity 3
pages: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

~~fifo page~~ fifo page replacement process

7 - - pf no 1
7 0 - pf no 2
7 0 1 - pf no 3
2 0 1 - pf no 4
2 0 1
2 3 1 - pf no 5
2 3 0 - pf no 6
~~4 3 0~~
~~4 2 0~~
4 3 0 - ~~pf no 7~~ pf no 7
4 2 0 - pf no 8
4 2 3 - pf no 9
0 2 3 - pf no 10
0 2 3
0 2 3
0 1 3 - pf no 11
0 1 2 - pf no 12
0 1 2
0 1 2
7 1 2 - pf no 13
7 0 2 - pf no 14
7 0 1 - pf no 15
total page faulb using fifo 15

~~lru page replacement process~~:

lru page replacement process:

7 – –	pf no 1
0 – –	pf no 2
0 1 –	pf no 3
0 1 2	pf no 4
0 3 2	pf no 5
0 3 2	~~pf no 5~~
0 3 4	pf no 6
0 2 4	~~pf no 6~~
3 2 4	pf no 7
3 2 0	pf no 8
3 2 0	pf no 9
3 2 1	
3 2 1	
0 2 1	pf no 10
0 2 1	
0 7 1	
0 7 1	
0 7 1	

7 – –	pf no 1
0 – –	pf no 2
0 1 –	pf no 3
0 1 2	pf no 4
0 1 2	
0 3 2	pf no 5
0 3 2	
0 3 4	pf no 6
0 2 4	pf no 7
3 2 4	pf no 8
3 2 0	pf no 9
3 2 0	
3 2 0	
3 2 1	pf no 10
3 2 1	~~pf no 10~~
0 2 1	pf no 11
0 2 1	~~pf no 11~~
0 7 1	pf no 12
0 7 1	
0 7 1	

total page faults using lru 12

optimal page replacement process

7 – –	pf no 1
7 0 –	pf no 2
7 0 1	pf no 3
2 0 1	pf no 4
2 0 1	
2 0 3	~~pf no 5~~ pf no 5
2 0 3	
2 4 3	pf no 6
2 4 3	
2 4 3	
2 0 3	pf no 7
2 0 3	
2 0 3	
2 0 1	pf no 8
2 0 1	
2 0 1	
2 0 1	
7 0 1	pf no 9
7 0 1	
7 0 1	

8/7 complete

~~total page faults of~~

total page faults using optimal 9

Colour Me