VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

YASHASVINI MR (1BM21CS252)

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
June-2023 to September-2023

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" carried out by **YASHASVINI M R(1BM21CS252),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to September-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS **(22CS4PCOPS)** work prescribed for the said degree.

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

Index Sheet

	b) Best-fit **c) First-fit**	
11	**Write a C program to simulate page replacement algorithms** **a) FIFO** **b) LRU** **c) Optimal**	61
12	**Write a C program to simulate disk scheduling algorithms** **a) FCFS** **b) SCAN** **c) C-SCAN**	66
13	**Write a C program to simulate disk scheduling algorithms** **a) SSTF** **b) LOOK** **c) c-LOOK**	71
14	**Write a C program to simulate paging technique of memory management**	76

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-pre-emptive)

```
#include<stdio.h>
typedef struct {
 int pID,aT,bT,sT,cT,taT,wT;
} Process;
void calculateTimes(Process p[], int n) {
 int currT = 0;
 for (int i = 0; i < n; i++) {
 p[i].sT = currT;
 p[i].cT = currT + p[i].bT;
 p[i].taT = p[i].cT - p[i].aT;
 p[i].wT = p[i].taT - p[i].bT;
 currT = p[i].cT;
 }
}
void displayp(Process p[], int n) {
 printf("Process\tArrival Time\tCPU Time\tStart 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\t\t%d\n", p[i].pID, p[i].aT,
 p[i].bT, p[i].sT, p[i].cT,
 p[i].taT, p[i].wT);
 }
}
int main() {
 int n;
 double sum;
 printf("Enter the number of processes: ");
 scanf("%d", &n);
 Process p[n];
 for (int i = 0; i < n; i++) {
 printf("Enter the arrival time and CPU time for process %d: ", i + 1);
 scanf("%d %d", &p[i].aT, &p[i].bT);
 p[i].pID = i + 1;
 }
 calculateTimes(p, n);
 displayp(p, n);
```

```
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (p[j].aT > p[j + 1].aT) {
Process temp = p[j];
p[j] = p[j + 1];
p[j + 1] = temp;
}
}
}
calculateTimes(p, n);
displayp(p, n);
printf("\nAverage waiting time:\n");
for(int i=0;i<n;i++)
{
sum+=p[i].bT;
}
printf("\n%f",sum/4);
return 0;
}
```

OUTPUT

```
Enter the number of processes: 4
Enter the arrival time and CPU time for process 1: 0 3
Enter the arrival time and CPU time for process 2: 1 6
Enter the arrival time and CPU time for process 3: 4 4
Enter the arrival time and CPU time for process 4: 6 2
Process Arrival Time    CPU Time        Start Time      Completion Time Turnaround Time Waiting Time
1       0               3               0               3               3               0
2       1               6               3               9               8               2
3       4               4               9               13              9               5
4       6               2               13              15              9               7
Process Arrival Time    CPU Time        Start Time      Completion Time Turnaround Time Waiting Time
1       0               3               0               3               3               0
2       1               6               3               9               8               2
3       4               4               9               13              9               5
4       6               2               13              15              9               7

Average waiting time:

3.750000
```

SJF

```
#include <stdio.h>

#include <stdbool.h>

#include<stdlib.h>


#define MAX_PROCESSES 10
```

```
struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

void sjf_nonpreemptive(struct Process processes[], int n) {
    // Sort the processes based on burst time in ascending order
    int i,j,count=0,m;
    for(i=0;i<n;i++)
    {
    if(processes[i].arrival_time==0)
    count++;
}
if(count==n||count==1)
{
if(count==n)
{
for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
```

```
            }
        }
    }
}
else
{
for (i = 1; i < n - 1; i++) {
        for (j = 1; j <= n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}



}

    int total_time = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    for (i = 0; i < n; i++) {
        total_time += processes[i].burst_time;
        processes[i].turnaround_time = total_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
```

```
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
    }

    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void sjf_preemptive(struct Process processes[], int n) {
    int total_time = 0,i;
    int completed = 0;

    while (completed < n) {
        int shortest_burst = -1;
        int next_process = -1;

        for (i = 0; i < n; i++) {
            if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0) {
                if (shortest_burst == -1 || processes[i].remaining_time < shortest_burst) {
                    shortest_burst = processes[i].remaining_time;
                    next_process = i;
                }
```

```
            }
        }

        if (next_process == -1) {
            total_time++;
            continue;
        }

        processes[next_process].remaining_time--;
        total_time++;

        if (processes[next_process].remaining_time == 0) {
            completed++;
            processes[next_process].turnaround_time = total_time - processes[next_process].arrival_time;
            processes[next_process].waiting_time = processes[next_process].turnaround_time - processes[next_process].burst_time;
        }
    }

    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time, processes[i].waiting_time);

        total_turnaround_time += processes[i].turnaround_time;
```

```
        total_waiting_time += processes[i].waiting_time;
    }

    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void priority_nonpreemptive(struct Process processes[], int n) {
    // Sort the processes based on priority in ascending order
    int i,j,count=0,m;
    for(i=0;i<n;i++)
    {
    if(processes[i].arrival_time==0)
    count++;
}
if(count==n||count==1)
{
if(count==n)
{
for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
```

```
}

else
{
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j <= n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
}

    int total_time = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    for (i = 0; i < n; i++) {
        total_time += processes[i].burst_time;
        processes[i].turnaround_time = total_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }
```

```
    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
    }

    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}
int main() {
    int n, i,choice;
    struct Process processes[MAX_PROCESSES];

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

    for (i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time, burst time, priority: ");
        scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time,
&processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
    printf("\nSelect a scheduling algorithm:\n");
    printf("1. SJF Non-preemptive\n");
```

```
    printf("2. SJF Preemptive\n");
    while(1)
    {
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nSJF Non-preemptive Scheduling:\n");
                sjf_nonpreemptive(processes, n);
                break;
            case 2:
                printf("\nSJF Preemptive Scheduling:\n");
                sjf_preemptive(processes, n);
                break;
 case 3: exit(0); break;
            default:
                printf("Invalid choice!\n");
                return 1;
        }
    }
    return 0;
}
```

OUTPUT

```
Enter the number of processes: 4
Process 1
Enter arrival time, burst time, priority: 0 8 0
Process 2
Enter arrival time, burst time, priority: 1 4 0
Process 3
Enter arrival time, burst time, priority: 2 9 0
Process 4
Enter arrival time, burst time, priority: 3 5 0

Select a scheduling algorithm:
1. SJF Non-preemptive
2. SJF Preemptive
```

```
Enter your choice: 2

SJF Preemptive Scheduling:
Process Turnaround Time Waiting Time
1       17              9
2       4               0
3       24              15
4       7               2
Average Turnaround Time: 13.00
Average Waiting Time: 6.50
Enter your choice: 1

SJF Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1       8               0
2       11              7
4       14              9
3       24              15
Average Turnaround Time: 14.25
Average Waiting Time: 7.75
```

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

Priority (only Non-pre-emptive)

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

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

#define MAX_PROCESSES 10

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};
void priority_nonpreemptive(struct Process processes[], int n) {
    // Sort the processes based on priority in ascending order
    int i,j,count=0,m;
    for(i=0;i<n;i++)
    {
    if(processes[i].arrival_time==0)
    count++;
}
if(count==n||count==1)
{
if(count==n)
{
for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

else
{
```

```
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j <= n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
}

    int total_time = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    for (i = 0; i < n; i++) {
        total_time += processes[i].burst_time;
        processes[i].turnaround_time = total_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
    }

    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}
int main() {
    int n, quantum,i,choice;
    struct Process processes[MAX_PROCESSES];

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

    for (i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time, burst time, priority: ");
        scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time,
```

```
&processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
 printf("\nPriority Non-preemptive Scheduling:\n");
              priority_nonpreemptive(processes, n);
return 0;
}
```

OUTPUT

```
Enter the number of processes: 5
Process 1
Enter arrival time, burst time, priority: 0 10 3
Process 2
Enter arrival time, burst time, priority: 0 1 1
Process 3
Enter arrival time, burst time, priority: 0 2 5
Process 4
Enter arrival time, burst time, priority: 0 1 4
Process 5
Enter arrival time, burst time, priority: 0 5 2
Priority Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
2       1               0
5       6               1
1       16              6
4       17              16
3       19              17
Average Turnaround Time: 11.80
Average Waiting Time: 8.00
```

Round Robin

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

struct P{
int AT,BT,ST[20],WT,FT,TAT,pos;
};

int quant;
int main()
{
   int n,i,j;
   printf("Enter the no. of processes :");
   scanf("%d",&n);
   struct P p[n];
   printf("Enter the quantum \n");
```

```
scanf("%d",&quant);
printf("Enter the process numbers \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].pos));
printf("Enter the Arrival time of processes \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].AT));
printf("Enter the Burst time of processes \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].BT));
int c=n,s[n][20];
float time=0,mini=INT_MAX,b[n],a[n];
int index=-1;
for(i=0;i<n;i++)
   {
   b[i]=p[i].BT;
   a[i]=p[i].AT;
   for(j=0;j<20;j++)
   {
      s[i][j]=-1;
   }
}

int tot_wt,tot_tat;
tot_wt=0;
tot_tat=0;
bool flag=false;
while(c!=0)
{
   mini=INT_MAX;
   flag=false;
   for(i=0;i<n;i++)
   {
      float p=time+0.1;
      if(a[i]<=p && mini>a[i] && b[i]>0)
      {
         index=i;
         mini=a[i];
         flag=true;

      }
   }

   if(!flag)
   {
```

```
        time++;
        continue;
    }
    j=0;
    while(s[index][j]!=-1)
    {
        j++;
    }
    if(s[index][j]==-1)
    {
        s[index][j]=time;
        p[index].ST[j]=time;
    }

    if(b[index]<=quant)
    {
    time+=b[index];

    b[index]=0;
    }
    else
    {
        time+=quant;
        b[index]-=quant;
    }
    if(b[index]>0)
    {
        a[index]=time+0.1;
    }
    if(b[index]==0)
    {
        c--;
        p[index].FT=time;
        p[index].WT=p[index].FT-p[index].AT-p[index].BT;
        tot_wt+=p[index].WT;
        p[index].TAT=p[index].BT+p[index].WT;
        tot_tat+=p[index].TAT;
    }

}

printf("Process number ");
printf("Arrival time ");
printf("Burst time ");
printf("\tStart time");
```

```
    j=0;
    while(j!=10)
    {
       j+=1;
       printf(" ");
    }
    printf("\t\tFinal time");
    printf("\tWait Time ");
    printf("\tTurnAround Time \n");
    for(i=0;i<n;i++)
    {
       printf("%d \t\t",p[i].pos);
       printf("%d \t\t",p[i].AT);
       printf("%d \t",p[i].BT);
       j=0;
       int v=0;
       while(s[i][j]!=-1)
       {
          printf("%d ",p[i].ST[j]);
          j++;
          v+=3;
       }
       while(v!=40)
       {
          printf(" ");
          v+=1;
       }
       printf("%d \t\t",p[i].FT);
       printf("%d \t\t",p[i].WT);
       printf("%d \n",p[i].TAT);

    }
    double avg_wt,avg_tat;
    avg_wt=tot_wt/(float)n;
    avg_tat=tot_tat/(float)n;
    printf("The average wait time is : %lf\n",avg_wt);
    printf("The average TurnAround time is : %lf\n",avg_tat);
    return 0;
}
```

OUTPUT

```
Enter the process numbers
1
2
3
4
5
Enter the Arrival time of processes
0
1
3
4
2
Enter the Burst time of processes
8
1
2
1
5
Process number Arrival time Burst time  Start time                    Final time      Wait Time       TurnAround Time
1              0             8          0 12                          16              8               16
2              1             1          4                             5               3               4
3              3             2          9                             11              6               8
4              4             1          11                             12             7               8
5              2             5          5 16                          17              10              15
The average wait time is : 6.800000
The average TurnAround time is : 10.200000
```

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

```
#include<stdio.h>
void swap(int *a,int *b)
{
   int temp;
   temp=*a;
   *a=*b;
   *b=temp;
}
void main()
{
   int n,pid[10],burst[10],type[10],arr[10],wt[10],ta[10],ct[10],i,j;
   float avgwt=0,avgta=0;
   int sum = 0;
   printf("Enter the total number of processes\n");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      printf("Enter the process id, type of process(user-1 and system-0), arrival time and burst time\n");
      scanf("%d",&pid[i]);
      scanf("%d",&type[i]);
      scanf("%d",&arr[i]);
      scanf("%d",&burst[i]);
   }
   //sorting the processes according to arrival time
```

```
for(i=0;i<n-1;i++)
{
   for(j=0;j<n-i-1;j++)
   {
      if(arr[j]>arr[j+1])
      {
         swap(&arr[j],&arr[j+1]);
         swap(&pid[j],&pid[j+1]);
         swap(&burst[j],&burst[j+1]);
         swap(&type[j],&type[j+1]);

      }
   }
}
//assuming only two process can have same arrival time and different priority
for(i=0;i<n-1;i++)
{
   for(j=0;j<n-i-1;j++)
   {
      if(arr[j]==arr[j+1] && type[j]<type[j+1])
      {
         swap(&arr[j],&arr[j+1]);
         swap(&pid[j],&pid[j+1]);
         swap(&burst[j],&burst[j+1]);
         swap(&type[j],&type[j+1]);
      }
   }
}
```

```
    //calculating completion time, arrival time and waiting time
    sum = sum + arr[0];
    for(i = 0;i<n;i++){
        sum = sum + burst[i];
        ct[i] = sum;
        ta[i] = ct[i] - arr[i];
        wt[i] = ta[i] - burst[i];
        if(sum<arr[i+1]){
            int t = arr[i+1]-sum;
            sum = sum+t;
        }
    }

    printf("Process id\tType\tarrival time\tburst time\twaiting time\tturnaround time\n");
    for(i=0;i<n;i++)
    {
        avgta+=ta[i];
        avgwt+=wt[i];
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",pid[i],type[i],arr[i],burst[i],wt[i],ta[i]);
    }
    printf("average waiting time =%f\n",avgwt/n);
    printf("average turnaround time =%f",avgta/n);

}
```

OUTPUT

```
Enter the total number of processes
6
Enter the process id, type of process(user-1 and system-0), arrival time and burst time
1 0 0 3
Enter the process id, type of process(user-1 and system-0), arrival time and burst time
2 0 2 2
Enter the process id, type of process(user-1 and system-0), arrival time and burst time
3 1 4 4
Enter the process id, type of process(user-1 and system-0), arrival time and burst time
4 1  4 2
Enter the process id, type of process(user-1 and system-0), arrival time and burst time
5 0 8 2
Enter the process id, type of process(user-1 and system-0), arrival time and burst time
6 1 10 3
Process id      Type    arrival time    burst time      waiting time    turnaround time
1               0               0               3               0               3
2               0               2               2               1               3
3               1               4               4               1               5
4               1               4               2               5               7
5               0               8               2               3               5
6               1               10              3               3               6
average waiting time =2.166667
average turnaround time =4.833333
```

4.Simulate Rate Monotonic Scheduling for the following and show the order of execution of processes in CPU timeline:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#define MAX_PROCESS 10
int num_of_process = 3, count, remain, time_quantum;
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
   remain_time[MAX_PROCESS], deadline[MAX_PROCESS],
   remain_deadline[MAX_PROCESS];
int burst_time[MAX_PROCESS], wait_time[MAX_PROCESS],
   completion_time[MAX_PROCESS], arrival_time[MAX_PROCESS];
// collecting details of processes
void get_process_info(int selected_algo)
{
   printf("Enter total number of processes (maximum %d): ",
         MAX_PROCESS);
   scanf("%d", &num_of_process);
   if (num_of_process < 1)
   {
      printf("Do you really want to schedule %d processes? -_-",
            num_of_process);
      exit(0);
   }
   for (int i = 0; i < num_of_process; i++)
   {
      printf("\nProcess %d:\n", i + 1);
      printf("==> Execution time: ");
```

```
        scanf("%d", &execution_time[i]);
        remain_time[i] = execution_time[i];

        printf("==> Period: ");
        scanf("%d", &period[i]);
    }
}
// get maximum of three numbers
int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else if (c >= a && c >= b)
        max = c;
    return max;
}
// calculating the observation time for scheduling timeline
int get_observation_time(int selected_algo)
{

    return max(period[0], period[1], period[2]);
}
// print scheduling sequence
void print_schedule(int process_list[], int cycles)
{
```

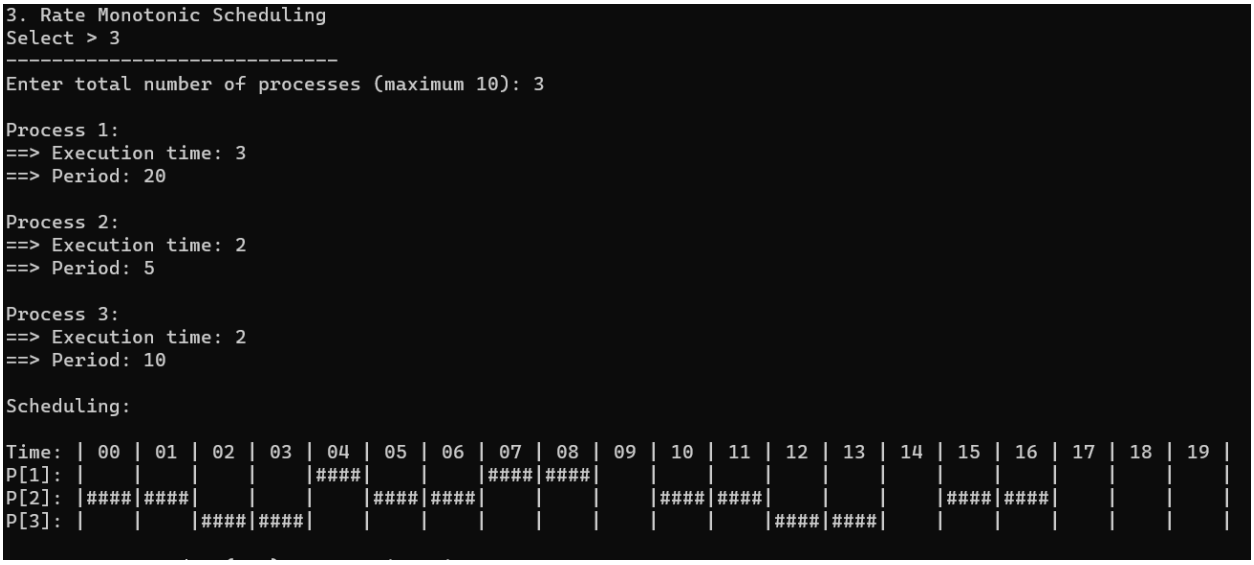
```
    printf("\nScheduling:\n\n");
    printf("Time: ");
    for (int i = 0; i < cycles; i++)
    {
        if (i < 10)
            printf("| 0%d ", i);
        else
            printf("| %d ", i);
    }
    printf("|\n");
    for (int i = 0; i < num_of_process; i++)
    {
        printf("P[%d]: ", i + 1);
        for (int j = 0; j < cycles; j++)
        {
            if (process_list[j] == i + 1)
                printf("|####");
            else
                printf("|    ");
        }
        printf("|\n");
    }
}
void rate_monotonic(int time)
{
    int process_list[100] = {0}, min = 999, next_process = 0;
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
```

```
{
   utilization += (1.0 * execution_time[i]) / period[i];
}
int n = num_of_process;
if (utilization > n * (pow(2, 1.0 / n) - 1))
{
   printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");

   exit(0);
}
for (int i = 0; i < time; i++)
{
   min = 1000;
   for (int j = 0; j < num_of_process; j++)
   {
      if (remain_time[j] > 0)
      {
         if (min > period[j])
         {
            min = period[j];
            next_process = j;
         }
      }
   }
   if (remain_time[next_process] > 0)
   {
      process_list[i] = next_process + 1; // +1 for catering 0 array index.
      remain_time[next_process] -= 1;
```

```
        }
        for (int k = 0; k < num_of_process; k++)
        {
            if ((i + 1) % period[k] == 0)
            {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }
    print_schedule(process_list, time);
}
int main(int argc, char *argv[])
{
    int option = 0;
    printf("3. Rate Monotonic Scheduling\n");
    printf("Select > ");
    scanf("%d", &option);
    printf("-----------------------------\n");
    get_process_info(option); // collecting processes detail
    int observation_time = get_observation_time(option);
    if (option == 3)
        rate_monotonic(observation_time);
    return 0;
}
```

OUTPUT


3. Rate Monotonic Scheduling
Select > 3
Enter total number of processes (maximum 10): 3
Process 1:
==> Execution time: 3
==> Period: 20
Process 2:
==> Execution time: 2
==> Period: 5
Process 3:
==> Execution time: 2
==> Period: 10
Scheduling:
Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: | | | | | |####| | | |####|####| | | | | | | | | | | | |
P[2]: |####|####| | | | |####|####| | | | |####|####| | | | |####|####| | | | |
P[3]: | | | |####|####| | | | | | | | | |####|####| | | | | | | |

5.Simulate Earliest Deadline First for the following and show the order of execution of processes in CPU timeline:

```
#include <stdio.h>
#define arrival             0
#define execution           1
#define deadline            2
#define period              3
#define abs_arrival         4
#define execution_copy  5
#define abs_deadline  6


typedef struct
{
        int T[7],instance, alive;

}task;

#define IDLE_TASK_ID 1023
#define ALL 1
#define CURRENT 0

void get_tasks(task *t1,int n);
int hyperperiod_calc(task *t1,int n);
float cpu_util(task *t1,int n);
int gcd(int a, int b);
int lcm(int *a, int n);
int sp_interrupt(task *t1,int tmr,int n);
int min(task *t1,int n,int p);
void update_abs_arrival(task *t1,int n,int k,int all);
void update_abs_deadline(task *t1,int n,int all);
void copy_execution_time(task *t1,int n,int all);


int timer = 0;

int main()
{
        task *t;
```

```
int n, hyper_period, active_task_id;
float cpu_utilization;
printf("Enter number of tasks\n");
scanf("%d", &n);
t = malloc(n * sizeof(task));
get_tasks(t, n);
cpu_utilization = cpu_util(t, n);
printf("CPU Utilization %f\n", cpu_utilization);

if (cpu_utilization < 1)
        printf("Tasks can be scheduled\n");
else
        printf("Schedule is not feasible\n");

hyper_period = hyperperiod_calc(t, n);
copy_execution_time(t, n, ALL);
update_abs_arrival(t, n, 0, ALL);
update_abs_deadline(t, n, ALL);

while (timer <= hyper_period)
{

        if (sp_interrupt(t, timer, n))
        {
                active_task_id = min(t, n, abs_deadline);
        }

        if (active_task_id == IDLE_TASK_ID)
        {
                printf("%d  Idle\n", timer);
        }

        if (active_task_id != IDLE_TASK_ID)
        {

                if (t[active_task_id].T[execution_copy] != 0)
                {
                        t[active_task_id].T[execution_copy]--;
                        printf("%d  Task %d\n", timer, active_task_id + 1);
                }
```

```
                        if (t[active_task_id].T[execution_copy] == 0)
                        {
                                t[active_task_id].instance++;
                                t[active_task_id].alive = 0;
                                copy_execution_time(t, active_task_id, CURRENT);
                                update_abs_arrival(t, active_task_id, t[active_task_id].instance,
CURRENT);
                                update_abs_deadline(t, active_task_id, CURRENT);
                                active_task_id = min(t, n, abs_deadline);
                        }
                }
                ++timer;
        }
        free(t);
        return 0;
}
void get_tasks(task *t1, int n)
{
        int i = 0;
        while (i < n)
        {
                printf("Enter Task %d parameters\n", i + 1);
                printf("Arrival time: ");
                scanf("%d", &t1->T[arrival]);
                printf("Execution time: ");
                scanf("%d", &t1->T[execution]);
                printf("Deadline time: ");
                scanf("%d", &t1->T[deadline]);
                printf("Period: ");
                scanf("%d", &t1->T[period]);
                t1->T[abs_arrival] = 0;
                t1->T[execution_copy] = 0;
                t1->T[abs_deadline] = 0;
                t1->instance = 0;
                t1->alive = 0;
                t1++;
                i++;
        }
}
```

```
int hyperperiod_calc(task *t1, int n)
{
	int i = 0, ht, a[10];
	while (i < n)

	{
		a[i] = t1->T[period];
		t1++;
		i++;
	}
	ht = lcm(a, n);

	return ht;
}

int gcd(int a, int b)
{
	if (b == 0)
		return a;
	else
		return gcd(b, a % b);
}

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

int sp_interrupt(task *t1, int tmr, int n)
{
	int i = 0, n1 = 0, a = 0;
	task *t1_copy;
	t1_copy = t1;
	while (i < n)
```

```
	{
		if (tmr == t1->T[abs_arrival])
		{
			t1->alive = 1;
			a++;
		}
		t1++;
		i++;
	}

	t1 = t1_copy;
	i = 0;

	while (i < n)
	{
		if (t1->alive == 0)
			n1++;
		t1++;
		i++;
	}

	if (n1 == n || a != 0)
	{
		return 1;
	}

	return 0;
}

void update_abs_deadline(task *t1, int n, int all)
{
	int i = 0;
	if (all)
	{
		while (i < n)
		{
			t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
			t1++;
			i++;
		}
```

```
	}
	else
	{
		t1 += n;
		t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
	}
}

void update_abs_arrival(task *t1, int n, int k, int all)
{
	int i = 0;
	if (all)
	{
		while (i < n)
		{
			t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
			t1++;
			i++;
		}
	}
	else
	{
		t1 += n;
		t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
	}
}

void copy_execution_time(task *t1, int n, int all)
{
	int i = 0;
	if (all)
	{
		while (i < n)
		{
			t1->T[execution_copy] = t1->T[execution];
			t1++;
			i++;
		}
	}
	else
```

```
	{
		t1 += n;
		t1->T[execution_copy] = t1->T[execution];
	}
}

int min(task *t1, int n, int p)
{
	int i = 0, min = 0x7FFF, task_id = IDLE_TASK_ID;
	while (i < n)
	{
		if (min > t1->T[p] && t1->alive == 1)
		{
			min = t1->T[p];
			task_id = i;
		}
		t1++;
		i++;
	}
	return task_id;
}

float cpu_util(task *t1, int n)
{
	int i = 0;
	float cu = 0;
	while (i < n)
	{
		cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];
		t1++;
		i++;
	}
	return cu;
}
```

OUTPUT

```
Enter number of tasks
3
Enter Task 1 parameters
Arrival time: 0
Execution time: 3
Deadline time: 7
Period: 20
Enter Task 2 parameters
Arrival time: 0
Execution time: 2
Deadline time: 4
Period: 5
Enter Task 3 parameters
Arrival time: 0
Execution time: 2
Deadline time: 8
Period: 10
CPU Utilization 1.178571
Schedule is not feasible
0  Task 2
1  Task 2
2  Task 1
3  Task 1
4  Task 1
5  Task 3
6  Task 3
7  Task 2
8  Task 2
9  Idle
10  Task 2
11  Task 2
12  Task 3
13  Task 3
14  Idle
15  Task 2
16  Task 2
17  Idle
18  Idle
19  Idle
20  Task 2
```

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

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 5, x = 0;
void producer()
{
   --mutex;
   ++full;
   --empty;
   x++;
   printf("\nProducer produces "
         "item %d",
         x);
   ++mutex;
}
void consumer()
{
   --mutex;
   --full;
   ++empty;
   printf("\nConsumer consumes "
         "item %d",
         x);
   x--;
   ++mutex;
}
int main()
{
   int n, i;
   printf("\n1. Press 1 for Producer"
         "\n2. Press 2 for Consumer"
         "\n3. Press 3 for Exit");
   for (i = 1; i > 0; i++) {

      printf("\nEnter your choice:");
      scanf("%d", &n);
      switch (n) {
```

```
        case 1:
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }
            else {
                printf("Buffer is full!");
            }
            break;

        case 2:
            if ((mutex == 1)
                && (full != 0)) {
                consumer();
            }
            else {
                printf("Buffer is empty!");
            }
            break;
        case 3:
            exit(0);
            break;
        }
    }
}
```

OUTPUT

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer produces item 1
Enter your choice:1

Producer produces item 2
Enter your choice:1

Producer produces item 3
Enter your choice:1

Producer produces item 4
Enter your choice:1

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

Consumer consumes item 5
Enter your choice:2

Consumer consumes item 4
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
```

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

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
        if (state[phnum] == HUNGRY
                && state[LEFT] != EATING
                && state[RIGHT] != EATING) {
                // state that eating
                state[phnum] = EATING;

                sleep(2);

                printf("Philosopher %d takes fork %d and %d\n",
                                        phnum + 1, LEFT + 1, phnum + 1);

                printf("Philosopher %d is Eating\n", phnum + 1);

                // sem_post(&S[phnum]) has no effect
                // during takefork
                // used to wake up hungry philosophers
                // during putfork
                sem_post(&S[phnum]);
        }
}

// take up chopsticks
void take_fork(int phnum)
{

        sem_wait(&mutex);
```

```
        // state that hungry
        state[phnum] = HUNGRY;

        printf("Philosopher %d is Hungry\n", phnum + 1);

        // eat if neighbours are not eating
        test(phnum);

        sem_post(&mutex);

        // if unable to eat wait to be signalled
        sem_wait(&S[phnum]);

        sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

        sem_wait(&mutex);

        // state that thinking
        state[phnum] = THINKING;

        printf("Philosopher %d putting fork %d and %d down\n",
                phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is thinking\n", phnum + 1);

        test(LEFT);
        test(RIGHT);

        sem_post(&mutex);
}

void* philosopher(void* num)
{

        while (1) {

                int* i = num;

                sleep(1);

                take_fork(*i);
```

```
                sleep(0);

                put_fork(*i);
        }
}

int main()
{

        int i;
        pthread_t thread_id[N];

        // initialize the semaphores
        sem_init(&mutex, 0, 1);

        for (i = 0; i < N; i++)

                sem_init(&S[i], 0, 0);

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

                // create philosopher processes
                pthread_create(&thread_id[i], NULL,
                                philosopher, &phil[i]);

                printf("Philosopher %d is thinking\n", i + 1);
        }

        for (i = 0; i < N; i++)

                pthread_join(thread_id[i], NULL);
}
```

OUTPUT

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 3 is Hungry
Philosopher 2 is Hungry
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 5 is Hungry
Philosopher 4 putting fork 3 and 4 down
```

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

```
#include <stdio.h>
void algo(int alloc[20][20],int max[20][20], int avail[20], int n, int m)
{
   int i,j, k;
   int fin[n], ans[n], ind = 0;
      for (k = 0; k < n; k++) {
            fin[k] = 0;
      }
      int need[n][m];
      for (i = 0; i < n; i++) {
            for (j = 0; j < m; j++)
                  need[i][j] = max[i][j] - alloc[i][j];
      }
      int y = 0;
      for (k = 0; k < n; k++) {
            for (i = 0; i < n; i++) {
                  if (fin[i] == 0) {

                        int flag = 0;
                        for (j = 0; j < m; j++) {
                              if (need[i][j] > avail[j]){
                                    flag = 1;
                                    break;
                              }
                        }

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

      int flag = 1;

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

```
                printf("The following system is not safe");
                break;
        }
        }

        if(flag==1)
        {
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
                printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);
        }
}

int main()
{

        int n, m, i, j, k,t, ch, alloc[20][20],  max[20][20], add[20];
        int avail[20];
   printf("Enter the number of processes:\n");
   scanf("%d",&n);
   printf("Enter the number of resources:\n");
   scanf("%d",&m);
   for(i=0;i<n;i++)
   {
      printf("Enter the allocated resources for each process P%d:\n",i);
      {
         for(j=0;j<m;j++)
         {
            scanf("%d",&alloc[i][j]);
         }
      }
   }
   for(i=0;i<n;i++)
   {
      printf("Enter the max resources for each process P%d:\n",i);
      {
         for(j=0;j<m;j++)
         {
            scanf("%d",&max[i][j]);
         }
      }
   }
   printf("Enter the available resources:\n");
   for(j=0;j<m;j++)
   {
```

```
        scanf("%d", &avail[j]);
    }
    algo(alloc, max, avail,n, m);
    printf("Does any process want to request for additional resources 1 for yes 0 for no?\n");
    scanf("%d",&ch);
     if(ch==1)
    {
        printf("Enter the process number for which there is an additional request");
        scanf("%d",&t);
        printf("Enter the number of instances required for each resource");
        for(i=0;i<m;i++)
        {
            scanf("%d",&add[i]);
        }
        for(i=0;i<m;i++)
        {
            alloc[t][i]+=add[i];
        }
        if(max[t][0]<alloc[t][0]||max[t][1]<alloc[t][1]||max[t][2]<alloc[t][2])
            printf("It is not a valid request");
        else
        {

            for(i=0;i<m;i++)
        {
            avail[i]-=add[i];
        }
         algo(alloc, max, avail,n, m);
        }
    }
    else
         algo(alloc, max, avail,n, m);


          return (0);

  }
```

OUTPUT

```
Enter the number of processes:
5
Enter the number of resources:
3
Enter the allocated resources for each process P0:
0 1 0
Enter the allocated resources for each process P1:
2 0 0
Enter the allocated resources for each process P2:
3 0 2
Enter the allocated resources for each process P3:
2 1 1
Enter the allocated resources for each process P4:
0 0 2
Enter the max resources for each process P0:
7  5 3
Enter the max resources for each process P1:
3 2 2
Enter the max resources for each process P2:
9 0 2
Enter the max resources for each process P3:
2 2 2
Enter the max resources for each process P4:
4 3 3
Enter the available resources:
3 3 2
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2Does any process want to request for additional resources 1 for yes 0 for no?
1
Enter the process number for which there is an additional request1
Enter the number of instances required for each resource1 0 2
Following is the SAFE Sequence
 P0 -> P1 -> P2 -> P3 -> P4
```

9.Write a C program to simulate deadlock detection

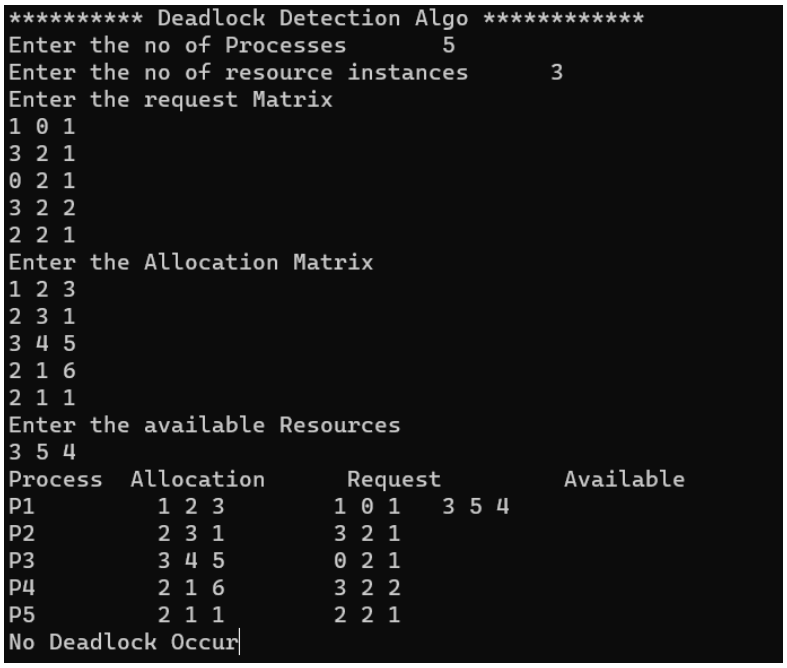
```
#include <stdio.h>
#include <conio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;
void input();
void show();
void cal();
int main()
{
   int i, j;
   printf("********** Deadlock Detection Algo **********\n");
   input();
   show();
   cal();
   getch();
   return 0;
}
void input()
{
   int i, j;
   printf("Enter the no of Processes\t");
   scanf("%d", &n);
   printf("Enter the no of resource instances\t");
   scanf("%d", &r);
   printf("Enter the request Matrix\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < r; j++)
      {
         scanf("%d", &max[i][j]);
      }
   }
   printf("Enter the Allocation Matrix\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < r; j++)
      {
         scanf("%d", &alloc[i][j]);
      }
   }
   printf("Enter the available Resources\n");
   for (j = 0; j < r; j++)
```

```
    {
        scanf("%d", &avail[j]);
    }
}
void show()
{
    int i, j;
    printf("Process\t Allocation\t Request\t Available\t");
    for (i = 0; i < n; i++)
    {
        printf("\nP%d\t   ", i + 1);
        for (j = 0; j < r; j++)
        {
            printf("%d ", alloc[i][j]);
        }
        printf("\t");
        for (j = 0; j < r; j++)
        {
            printf("%d ", max[i][j]);
        }
        printf("\t");
        if (i == 0)
        {
            for (j = 0; j < r; j++)
                printf("%d ", avail[j]);
        }
    }
}
void cal()
{
    int finish[100], temp, need[100][100], flag = 1, k, c1 = 0;
    int dead[100];
    int safe[100];
    int i, j;
    for (i = 0; i < n; i++)
    {
        finish[i] = 0;
    }
    // find need matrix
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < r; j++)
        {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
```

```
while (flag)
{
   flag = 0;
   for (i = 0; i < n; i++)
   {
      int c = 0;
      for (j = 0; j < r; j++)
      {
         if ((finish[i] == 0) && (need[i][j] <= avail[j]))
         {
            c++;
            if (c == r)
            {
               for (k = 0; k < r; k++)
               {
                  avail[k] += alloc[i][j];
                  finish[i] = 1;
                  flag = 1;
               }
               // printf("\nP%d",i);
               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("\n\nSystem is in Deadlock and the Deadlock process are\n");
   for (i = 0; i < n; i++)
   {
      printf("P%d\t", dead[i]);
```

```
    }
  }
  else
  {
    printf("\nNo Deadlock Occur");
  }
}
```

OUTPUT



```
********** Deadlock Detection Algo ************
Enter the no of Processes       5
Enter the no of resource instances      3
Enter the request Matrix
5 4 6
7 8 5
4 6 7
5 9
8
7 9 8
Enter the Allocation Matrix
1 2 3
1 4 2
1 3 1
1 1 3
1 1 1
Enter the available Resources
1 1 0
Process  Allocation      Request         Available
P1         1 2 3        5 4 6   1 1 0
P2         1 4 2        7 8 5
P3         1 3 1        4 6 7
P4         1 1 3        5 9 8
P5         1 1 1        7 9 8

System is in Deadlock and the Deadlock process are
P0      P1      P2      P3      P4
```

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

d) Worst-fit

e) Best-fit

f) First-fit

```
#include <stdio.h>
#include <stdlib.h>
#define max 25

void readInput(int *nb, int *nf, int b[], int f[]);
void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[]);
void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[]);
void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[]);
void displayResults(int nf, int f[], int ff[], int b[]);

int main()
{
   int nb, nf, ch;
   int b[max], f[max], bf[max] = {0}, ff[max] = {0};
   readInput(&nb, &nf, b, f);
   printf("1.Best Fit 2.Worst Fit 3.First Fit 4. Exit\n");
   scanf("%d", &ch);
   switch (ch)
   {
   case 1:
      bestFit(nb, nf, b, f, bf, ff);
      break;
   case 2:
      worstFit(nb, nf, b, f, bf, ff);
      break;
   case 3:
      firstFit(nb, nf, b, f, bf, ff);
      break;
   case 4:
      exit(0);
      break;
   default:
      printf("Invalid choice\n");
      break;
   }
   displayResults(nf, f, ff, b);
   return 0;
}

void readInput(int *nb, int *nf, int b[], int f[])
```

```
{
   int i;
   printf("Enter the number of Holes:");
   scanf("%d", nb);

   printf("Enter the number of requests:");
   scanf("%d", nf);

   printf("\nEnter the size of the holes:\n");
   for (i = 1; i <= *nb; i++)
   {
      printf("Hole %d:", i);
      scanf("%d", &b[i]);
   }

   printf("Enter the size of the requests:\n");
   for (i = 1; i <= *nf; i++)
   {
      printf("Request %d:", i);
      scanf("%d", &f[i]);
   }
}

void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[])
{
   int i, j, temp, lowest = 10000;

   for (i = 1; i <= nf; i++)
   {
      for (j = 1; j <= nb; j++)
      {
         if (bf[j] != 1) //if bf[j] is not allocated
         {
            temp = b[j] - f[i];
            if (temp >= 0)
            {
               if (lowest > temp)
               {
                  ff[i] = j;
                  lowest = temp;
               }
            }
         }
      }
      bf[ff[i]] = 1;
      lowest = 10000;
```

```
    }
}

void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[])
{
    int i, j, temp, lowest = 10000;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1)
            {
                temp = b[j] - f[i];
                if (temp >= 0)
                {
                    if (lowest == 10000 || temp > lowest)
                    {
                        ff[i] = j;
                        lowest = temp;
                    }
                }
            }
        }
        bf[ff[i]] = 1;
        lowest = 10000;
    }
}

void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[])
{
    int i, j, temp;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1)
            {
                temp = b[j] - f[i];
                if (temp >= 0)
                {
                    ff[i] = j;
                    break;
                }
            }
```

```
        }
        bf[ff[i]] = 1;
    }
}

void displayResults(int nf, int f[], int ff[], int b[])
{
    int i;

    printf("\nFile_no\t\tFile_size\tBlock_no\tBlock_size");
    for (i = 1; i <= nf; i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]]);
    }
}
```

OUTPUT

```
Enter the number of Holes:8
Enter the number of requests:3

Enter the size of the holes:
Hole 1:10
Hole 2:4
Hole 3:20
Hole 4:18
Hole 5:7
Hole 6:9
Hole 7:12
Hole 8:15
Enter the size of the requests:
Request 1:12
Request 2:10
Request 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
1

File_no         File_size       Block_no        Block_size
1               12              7               12
2               10              1               10
3               9               6               9
```

```
Enter the number of Holes:8
Enter the number of requests:3

Enter the size of the holes:
Hole 1:10
Hole 2:4
Hole 3:20
Hole 4:18
Hole 5:7
Hole 6:9
Hole 7:12
Hole 8:15
Enter the size of the requests:
Request 1:12
Request 2:10
Request 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
2

File_no         File_size       Block_no        Block_size
1               12              3               20
2               10              4               18
3               9               8               15
```

11.Write a C program to simulate page replacement algorithms

d) FIFO

e) LRU

f) Optimal

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


void printFrames(int frames[], int n) {
    for (int i = 0; i < n; i++) {
        if (frames[i] == -1)
            printf(" - ");
        else
            printf(" %d ", frames[i]);
    }
    printf("\n");
}

int findIndex(int arr[], int n, int element) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == element)
            return i;
    }
    return -1;
}

int findOptimal(int pages[], int n, int frames[], int start, int NUM_FRAMES) {
    int res = -1, farthest = start;
    for (int i = 0; i < NUM_FRAMES; i++) {
        int j;
        for (j = start; j < n; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    res = i;
                }
                break;
            }
        }
        if (j == n)
            return i;
    }
    return (res == -1) ? 0 : res;
}
```

```
void fifo(int pages[], int n, int NUM_FRAMES) {
    int frames[NUM_FRAMES];
    int frameIndex = 0;
    int pageFaults = 0;

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

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

    for (int i = 0; i < n; i++) {
        int page = pages[i];

        if (findIndex(frames, NUM_FRAMES, page) == -1) {
            frames[frameIndex] = page;
            frameIndex = (frameIndex + 1) % NUM_FRAMES;
            pageFaults++;
        }

        printf("Page %d -> ", page);
        printFrames(frames, NUM_FRAMES);
    }

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

void lru(int pages[], int n, int NUM_FRAMES) {
    int frames[NUM_FRAMES];
    int pageFaults = 0;

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

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

    for (int i = 0; i < n; i++) {
        int page = pages[i];

        if (findIndex(frames, NUM_FRAMES, page) == -1) {
            int emptyIndex = findIndex(frames, NUM_FRAMES, -1);
            if (emptyIndex != -1) {
                frames[emptyIndex] = page;
            } else {
                int lruIndex = i;
                for (int j = 0; j < NUM_FRAMES; j++) {
```

```
                if (findIndex(pages, n, frames[j]) < lruIndex) {
                    lruIndex = findIndex(pages, n, frames[j]);
                }
            }
            frames[lruIndex] = page;
        }
        pageFaults++;
    }

    printf("Page %d -> ", page);
    printFrames(frames, NUM_FRAMES);
  }

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

void optimal(int pages[], int n, int NUM_FRAMES) {
  int frames[NUM_FRAMES];
  int pageFaults = 0;

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

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

  for (int i = 0; i < n; i++) {
    int page = pages[i];

    if (findIndex(frames, NUM_FRAMES, page) == -1) {
      int emptyIndex = findIndex(frames, NUM_FRAMES, -1);
      if (emptyIndex != -1) {
        frames[emptyIndex] = page;
      } else {
        int optimalIndex = findOptimal(pages, n, frames, i + 1, NUM_FRAMES);
        frames[optimalIndex] = page;
      }
      pageFaults++;
    }

    printf("Page %d -> ", page);
    printFrames(frames, NUM_FRAMES);
  }

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

```
int main() {
   int pages[50],NUM_PAGES, NUM_FRAMES, i;

   int choice;
   printf("Enter the number of pages:\n");
   scanf("%d", &NUM_PAGES);
   printf("Enter the number of frames:\n");
   scanf("%d",&NUM_FRAMES);
   printf("Enter the pages:\n");
   for(i=0;i<NUM_PAGES;i++)
   {
      scanf("%d",&pages[i]);
   }
   printf("Choose Page Replacement Algorithm:\n");
   printf("1. FIFO\n2. LRU\n3. Optimal\n");
   scanf("%d", &choice);

   switch (choice) {
      case 1:
         fifo(pages, NUM_PAGES, NUM_FRAMES);
         break;
      case 2:
         lru(pages, NUM_PAGES, NUM_FRAMES);
         break;
      case 3:
         optimal(pages, NUM_PAGES, NUM_FRAMES);
         break;
      default:
         printf("Invalid choice!\n");
         break;
   }

   return 0;
}
```

OUTPUT

```
Enter the number of pages:
10
Enter the number of frames:
3
Enter the pages:
10 45 78 21 32 45 87 90 44 18
Choose Page Replacement Algorithm:
1. FIFO
2. LRU
3. Optimal
2
LRU Page Replacement:
Page 10 ->  10  -  -
Page 45 ->  10  45  -
Page 78 ->  10  45  78
Page 21 ->  21  45  78
Page 32 ->  21  32  78
Page 45 ->  21  32  45
Page 87 ->  21  87  45
Page 90 ->  21  90  45
Page 44 ->  21  44  45
Page 18 ->  21  18  45
Total Page Faults: 10
```

```
Enter the number of pages:
10
Enter the number of frames:
3
Enter the pages:
10 45 78 21 32 45 87 90 44 18
Choose Page Replacement Algorithm:
1. FIFO
2. LRU
3. Optimal
1
FIFO Page Replacement:
Page 10 ->  10  -  -
Page 45 ->  10  45  -
Page 78 ->  10  45  78
Page 21 ->  21  45  78
Page 32 ->  21  32  78
Page 45 ->  21  32  45
Page 87 ->  87  32  45
Page 90 ->  87  90  45
Page 44 ->  87  90  44
Page 18 ->  18  90  44
Total Page Faults: 10
```

12.Write a C program to simulate disk scheduling algorithms

d) FCFS

e) SCAN

f) C-SCAN

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


void FCFS(int arr[],int head, int n)
{
   int seek_count = 0;
     int cur_track, distance;

   for(int i=0;i<n;i++)
      {
       cur_track = arr[i];
       distance = fabs(head - cur_track);
       seek_count += distance;
       head = cur_track;
   }

   printf("Total number of seek operations: %d\n",seek_count);
   printf("Seek Sequence is\n");

     for (int i = 0; i < n; i++) {
       printf("%d\n",arr[i]);
   }
}
void CSCAN(int arr[],int head, int n, int range)
{
   int headm=0,i,dir,temp,seek_count=0;
   printf("Enter the direction, upward/right=1, downward/left=-1");
   scanf("%d",&dir);
   if(dir==1)
   {
      for(i=0;i<n;i++)
      {
         if(arr[i]<head)
         {
            seek_count++;
            continue;
         }
         else if(i==seek_count)
            headm=headm+(arr[i]-head);
```

```
            else
                headm=headm+(arr[i]-arr[i-1]);
        }
        headm=headm+(range-arr[i-1]);
        for(i=seek_count-1;i>0;i--)
        {
            headm+=(arr[i]-arr[i-1]);
        }
        headm+=(arr[i]-0);
    }
    else
    {
        for(i=0;i<n;i++)
        {
            if(arr[i]>head)
            break;
            else
            seek_count++;
        }
        headm+=(head-arr[seek_count-1]);
        for(i=seek_count-1;i>0;i--)
        {
            headm+=(arr[i]-arr[i-1]);
        }
        headm+=(arr[0]-0);
        for(i=seek_count;i<n-1;i++)
        {
            headm+=(arr[i+1]-arr[i]);
        }
        headm=headm+(range-arr[i]);

    }
    printf("CSCAN-Total head movement=%d",headm);
     printf("Seek Sequence is\n");

          for (int i = 0; i < n; i++) {
            printf("%d\n",arr[i]);
        }
}
void SCAN(int arr[],int head, int n, int range)
{
int headm=0,i,dir,temp,cnt=0;
printf("Enter the direction, upward/right=1, downward/left=-1");
scanf("%d",&dir);
if(dir==1)
{
```

```
for(i=0;i<n;i++)
{
if(arr[i]<head)
{
cnt++;
continue;
}
else if(i==cnt)
headm=headm+(arr[i]-head);
else
headm=headm+(arr[i]-arr[i-1]);
}
headm=headm+(range-arr[i-1]);
headm+=(range-arr[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(arr[i]-arr[i-1]);
}
}
else
{
for(i=0;i<n;i++)
{
if(arr[i]>head)
break;
else
cnt++;
}
headm+=(head-arr[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(arr[i]-arr[i-1]);
}
headm+=(arr[0]-0);
headm+=(arr[cnt]-0);
for(i=cnt;i<n-1;i++)
{
headm+=(arr[i+1]-arr[i]);
}
}
printf("SCAN-Total head movement=%d",headm);
printf("Seek Sequence is\n");

        for (int i = 0; i < n; i++) {
          printf("%d\n",arr[i]);
       }
```

```
}
int main()
{
   int arr[30], head, ch, size, move,n,i,range ;
   printf("1.FCFS 2.SCAN 3.C-Scan\n");
   scanf("%d",&ch);
   printf("Enter the number of disks:\n");
   scanf("%d",&n);
   printf("Enter the sequence:\n");
   for(i=0;i<n;i++)
   {
      scanf("%d",&arr[i]);
   }
   printf("Enter the head:\n");
   scanf("%d",&head);
   switch(ch)
   {
      case 1:FCFS(arr,head,n);
            break;
      case 2:printf("Enter the range:\n");
            scanf("%d", &range);
            CSCAN(arr,head,n,range);
            break;
      case 3:printf("Enter the range:\n");
            scanf("%d", &range);
            SCAN(arr, head,n,range);
            break;
      case 4:exit(0);
      break;
      default:("Invalid choice");
            break;

   }


   return 0;
}
```

OUTPUT

```
1.FCFS 2.SCAN 3.C-Scan
3
Enter the number of disks:
8
Enter the sequence:
20 10 45 63 88 74 80 97
Enter the head:
55
Enter the range:
199
Enter the direction, upward/right=1, downward/left=-11
SCAN-Total head movement=323
```

```
1.FCFS 2.SCAN 3.C-Scan
1
Enter the number of disks:
8
Enter the sequence:
20 10 45 63 88 74 80 97
Enter the head:
55
Total number of seek operations: 160
```

13.Write a C program to simulate disk scheduling algorithms

d) SSTF

e) LOOK

f) c-LOOK

```
#define MAX_REQUESTS 100

void sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int absDiff(int a, int b) {
    return (a > b) ? a - b : b - a;
}

void sstf(int requests[], int n, int start) {
    int totalSeek = 0;
    int current = start;
    int visited[MAX_REQUESTS] = {0};

    printf("SSTF Disk Scheduling:\n");

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

        for (int j = 0; j < n; j++) {
            if (!visited[j]) {
                int distance = absDiff(current, requests[j]);
                if (distance < minDist) {
                    minDist = distance;
                    nextIndex = j;
                }
            }
        }
```

```
        visited[nextIndex] = 1;
        totalSeek += minDist;
        printf("Move from %d to %d\n", current, requests[nextIndex]);
        current = requests[nextIndex];
    }

    printf("Total Seek Distance: %d\n", totalSeek);
}

void look(int requests[], int n, int start, int direction) {
    int totalSeek = 0;
    int current = start;

    sort(requests, n);

    printf("LOOK Disk Scheduling:\n");

    if (direction == 1) { // Upward direction
        for (int i = 0; i < n; i++) {
            if (requests[i] >= current) {
                totalSeek += absDiff(current, requests[i]);
                printf("Move from %d to %d\n", current, requests[i]);
                current = requests[i];
            }
        }

        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] < current) {
                totalSeek += absDiff(current, requests[i]);
                printf("Move from %d to %d\n", current, requests[i]);
                current = requests[i];
            }
        }
    } else { // Downward direction
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] <= current) {
                totalSeek += absDiff(current, requests[i]);
                printf("Move from %d to %d\n", current, requests[i]);
                current = requests[i];
            }
        }

        for (int i = 0; i < n; i++) {
            if (requests[i] > current) {
                totalSeek += absDiff(current, requests[i]);
                printf("Move from %d to %d\n", current, requests[i]);
```

```
            current = requests[i];
        }
    }
  }

  printf("Total Seek Distance: %d\n", totalSeek);
}

void cLook(int requests[], int n, int start) {
  int totalSeek = 0;
  int current = start;

  sort(requests, n);

  printf("C-LOOK Disk Scheduling:\n");

  int index = 0;
  while (index < n && requests[index] <= current)
    index++;

  for (int i = index; i < n; i++) {
    totalSeek += absDiff(current, requests[i]);
    printf("Move from %d to %d\n", current, requests[i]);
    current = requests[i];
  }

  for (int i = 0; i < index; i++) {
    totalSeek += absDiff(current, requests[i]);
    printf("Move from %d to %d\n", current, requests[i]);
    current = requests[i];
  }

  printf("Total Seek Distance: %d\n", totalSeek);
}

int main() {
  int requests[MAX_REQUESTS];
  int n, start, direction;

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

  if (n > MAX_REQUESTS) {
    printf("Maximum number of requests exceeded.\n");
    return 1;
  }
```

```
    printf("Enter the requests: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &requests[i]);

    printf("Enter the starting position: ");
    scanf("%d", &start);

    printf("Enter the direction (1 for upward, 0 for downward): ");
    scanf("%d", &direction);

    int choice;
    printf("Choose Disk Scheduling Algorithm:\n");
    printf("1. SSTF\n2. LOOK\n3. C-LOOK\n");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            sstf(requests, n, start);
            break;
        case 2:
            look(requests, n, start, direction);
            break;
        case 3:
            cLook(requests, n, start);
            break;
        default:
            printf("Invalid choice!\n");
            break;
    }

    return 0;
}
```

OUTPUT

```
Enter the number of requests: 8
Enter the requests: 20 10 45 63 88 74 80 97
Enter the starting position: 0
Enter the direction (1 for upward, 0 for downward): 1
Choose Disk Scheduling Algorithm:
1. SSTF
2. LOOK
3. C-LOOK
2
LOOK Disk Scheduling:
Move from 0 to 10
Move from 10 to 20
Move from 20 to 45
Move from 45 to 63
Move from 63 to 74
Move from 74 to 80
Move from 80 to 88
Move from 88 to 97
Move from 97 to 88
Move from 88 to 80
Move from 80 to 74
Move from 74 to 63
Move from 63 to 45
Move from 45 to 20
Move from 20 to 10
Total Seek Distance: 184
```

```
Enter the number of requests: 8
Enter the requests:  20 10 45 63 88 74 80 97
Enter the starting position: 0
Enter the direction (1 for upward, 0 for downward): 1
Choose Disk Scheduling Algorithm:
1. SSTF
2. LOOK
3. C-LOOK
1
SSTF Disk Scheduling:
Move from 0 to 10
Move from 10 to 20
Move from 20 to 45
Move from 45 to 63
Move from 63 to 74
Move from 74 to 80
Move from 80 to 88
Move from 88 to 97
Total Seek Distance: 97
```

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

```
#include<stdio.h>
#include<conio.h>
main()
{
   int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
   int s[10], fno[10][20];
   printf("\nEnter the memory size -- ");
   scanf("%d",&ms);
   printf("\nEnter the page size -- ");
   scanf("%d",&ps);
   nop = ms/ps;
   printf("\nThe no. of pages available in memory are -- %d ",nop);
   printf("\nEnter number of processes -- ");
   scanf("%d",&np);
   rempages = nop;
   for(i=1;i<=np;i++)
   {

      printf("\nEnter no. of pages required for p[%d]-- ",i);
       scanf("%d",&s[i]);

      if(s[i] >rempages)
      {

         printf("\nMemory is Full");
         break;
      }
      rempages = rempages - s[i];

      printf("\nEnter page table for p[%d] --- ",i);
      for(j=0;j<s[i];j++)
      scanf("%d",&fno[i][j]);
   }

   printf("\nEnter Logical Address to find Physical Address ");
   printf("\nEnter process no. and page number and offset -- ");

   scanf("%d %d %d",&x,&y, &offset);



   if(x>np || y>=s[i] || offset>=ps)
   printf("\nInvalid Process or Page Number or offset");
```

```
    else
    {
       pa=fno[x][y]*ps+offset;
       printf("\nThe Physical Address is -- %d",pa);

    }
    getch();
}
```

OUTPUT

```
Enter the memory size -- 1000

Enter the page size -- 100

The no. of pages available in memory are -- 10
Enter number of processes -- 3

Enter no. of pages required for p[1]-- 4

Enter page table for p[1] --- 8 6 9 5

Enter no. of pages required for p[2]-- 5

Enter page table for p[2] --- 1 4 5 7 3

Enter no. of pages required for p[3]-- 5

Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and page number and offset -- 2 3 60

The Physical Address is -- 760
```