LAB1:

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

void findWaitingTimeFCFS(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    for (int i = 0; i < n; i++) {
        wt[i] = ct[i] - at[i] - bt[i];
    }
}

void findWaitingTimeSJFPreemptive(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((at[j] <= t) && (rt[j] < minm) && (rt[j] > 0)) {
                minm = rt[j];
                shortest = j;
            }
        }
        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;
        if (rt[shortest] == 0) {
            complete++;
            finish_time = t + 1;
            wt[shortest] = finish_time - bt[shortest] - at[shortest];
            if (wt[shortest] < 0)
                wt[shortest] = 0;
            ct[shortest] = finish_time;
        }
        t++;
    }
}


void findWaitingTimeSJFNonPreemptive(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];

    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
```

```
                minm = rt[j];
                shortest = j;
            }
        }
        t += rt[shortest];
        finish_time = t;
        wt[shortest] = finish_time - bt[shortest] - at[shortest];
        if (wt[shortest] < 0)
            wt[shortest] = 0;
        rt[shortest] = INT_MAX;
        complete++;
        ct[shortest] = finish_time;
        minm = INT_MAX;
    }
}


void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[], int ct[], int at[]) {
    for (int i = 0; i < n; i++)
        tat[i] = ct[i] - at[i];
}
void findAverageTimeFCFS(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeFCFS(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct, at);

    printf("FCFS Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf(" %d ", processes[i]);
        printf("          %d ", at[i]);
        printf("          %d ", bt[i]);
        printf("           %d", wt[i]);
        printf("                    %d", tat[i]);
        printf("                    %d\n", ct[i]);
    }

    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
```

```
    printf("Average turn around time = %f\n", avg_tat);
}

void findAverageTimeSJFNonPreemptive(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeSJFNonPreemptive(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct , at);

    printf("\nSJF (Non-preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf(" %d ", processes[i]);
        printf("         %d ", at[i]);
        printf("         %d ", bt[i]);
        printf("          %d", wt[i]);
        printf("                  %d", tat[i]);
        printf("                  %d\n", ct[i]);
    }
```

```
    float avg_tat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
    printf("Average turn around time = %f\n", avg_tat);
}


void findAverageTimeSJFPreemptive(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeSJFPreemptive(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct , at);

    printf("\nSJF (Preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf(" %d ", processes[i]);
        printf("         %d ", at[i]);
        printf("         %d ", bt[i]);
        printf("          %d", wt[i]);
        printf("                  %d", tat[i]);
```

```
    float avg_tat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
    printf("Average turn around time = %f\n", avg_tat);
}

int main() {
    int processes[10], burst_time[10], arrival_time[10], completion_time[10];
    int n;

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

    printf("Enter arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Arrival time of process[%d]: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Burst time of process[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        processes[i] = i + 1;
    }

    completion_time[0] = arrival_time[0] + burst_time[0];
    for (int i = 1; i < n; i++) {
        if (arrival_time[i] > completion_time[i - 1]) {
```

```
    for (int i = 0; i < n; i++) {
        printf("Arrival time of process[%d]: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Burst time of process[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        processes[i] = i + 1;
    }

    completion_time[0] = arrival_time[0] + burst_time[0];
    for (int i = 1; i < n; i++) {
        if (arrival_time[i] > completion_time[i - 1]) {
            completion_time[i] = arrival_time[i] + burst_time[i];
        } else {
            completion_time[i] = completion_time[i - 1] + burst_time[i];
        }
    }

    findAverageTimeFCFS(processes, n, burst_time, arrival_time, completion_time);
    findAverageTimeSJFNonPreemptive(processes, n, burst_time, arrival_time, completion_time);
    findAverageTimeSJFPreemptive(processes, n, burst_time, arrival_time, completion_time);

    return 0;
}
```

OUTPUT:

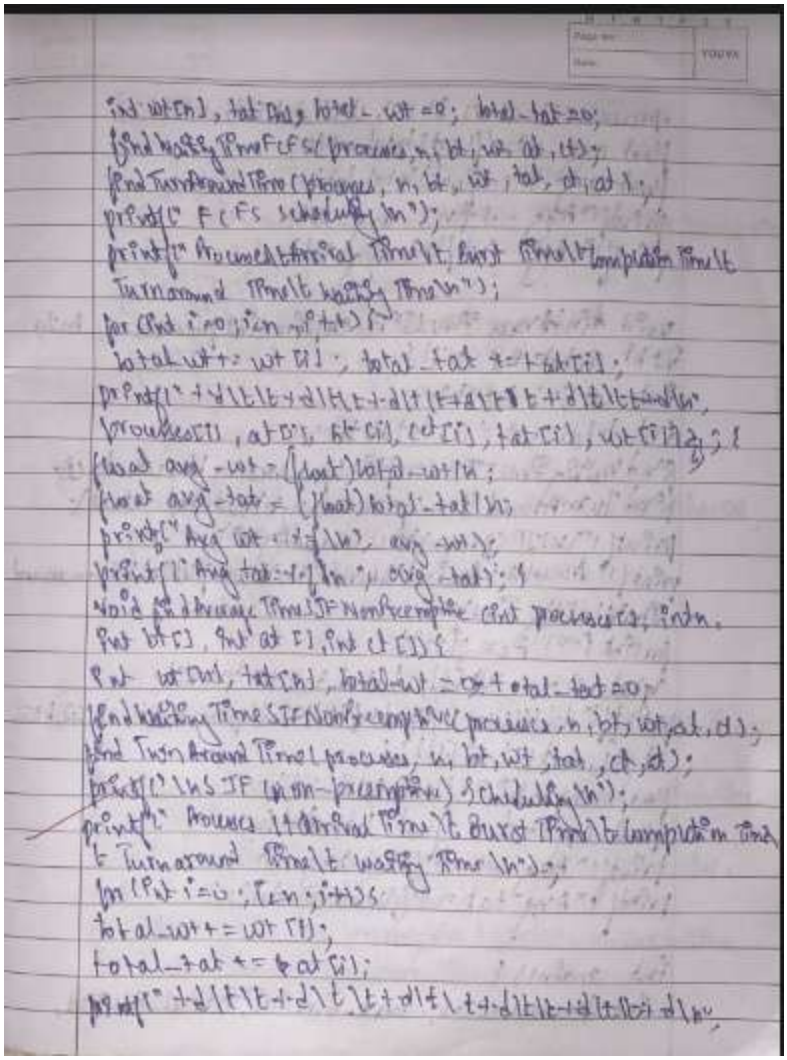
```
Enter the number of processes: 3
Enter arrival time and burst time for each process:
Arrival time of process[1]: 0
Burst time of process[1]: 2
Arrival time of process[2]: 1
Burst time of process[2]: 4
Arrival time of process[3]: 2
Burst time of process[3]: 3
FCFS Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0          2          0                2                  2
 2          1          4          1                5                  6
 3          2          3          4                7                  9
Average waiting time = 1.666667
Average turn around time = 4.666667

SJF (Non-preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0          2          0                2                  2
 2          1          4          4                8                  9
 3          2          3          0                3                  5
Average waiting time = 1.333333
Average turn around time = 4.333333

SJF (Preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0          2          0                2                  2
 2          1          4          4                8                  9
 3          2          3          0                3                  5
Average waiting time = 1.333333
Average turn around time = 4.333333
```

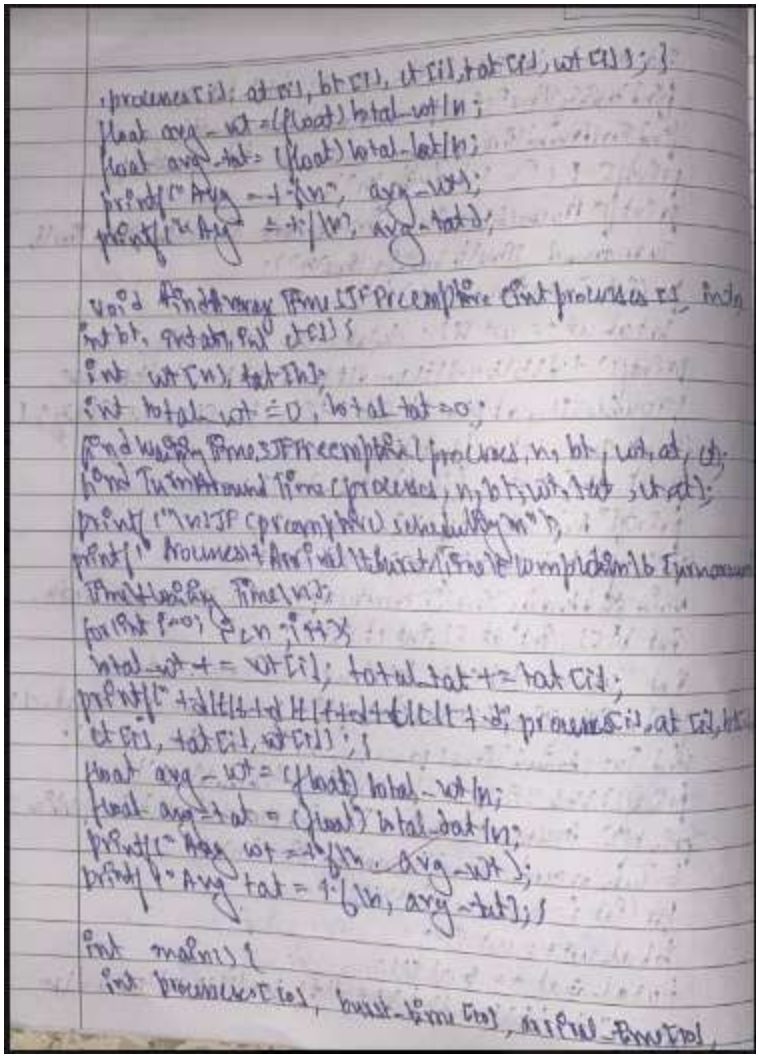
x: 15/2/21 LAB PROGRAM - 01

Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time
→ FCFS
→ SJF(Pre-emptive and non-pre-emptive)

#include<stdio.h>
#include<conio.h>
void findwaitingtime(int processes[], int n, int bt[], int wt[], int at[], int ct[])
{
for(int i=0; i<n; i++)
wt[i] = ct[i] - at[i] - bt[i]; }
void findturnaroundtime(int processes[], int n, int bt[], int wt[], int at[], int tat[])
{
int rt[n];
for(int i=0; i<n; i++)
rt[i] = bt[i];
int complete=0, t=0, minm=INT_MAX, shortest=0, finish_time;
while(complete!=n){
for(int j=0; j<n; j++){
if((at[j]<=t) && (rt[j]<minm) && (rt[j]>0)){
minm=rt[j]; shortest=j; }
rt[shortest]--; minm=rt[shortest];
if(minm==0)
minm=INT_MAX;
if(rt[shortest]==0){
complete++;

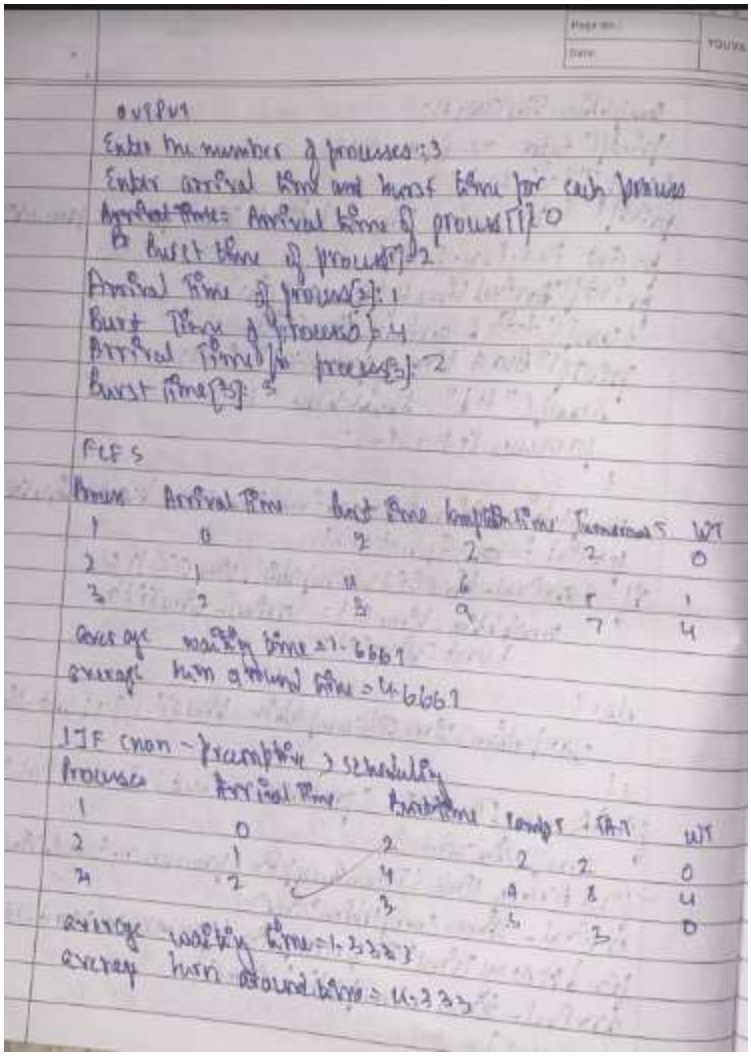
```
finish_time=t+1;
wt[shortest]=finish_time - bt[shortest]-at[shortest];
if (wt[shortest]<0)
    wt[shortest]=0;
ct[shortest]=finish_time;
h++;}
void findAverageTimeSJF(int processes[], int n, int bt[],
int wt[], int at[], int ct[])
{int rt[n];
for(int i=0;i<n;i++)
  rt[i]=bt[i];
int complete=0, t=0, minm=INT_MAX, shortest=0, finish_time;
while(complete!=n){
  for(int j=0;j<n;j++){
    if((at[j]<=t)&&(rt[j]<minm)&&ct[j]!=0){
      minm=rt[j]; shortest=j;}}
  t+=rt[shortest];
  finish_time=t;
  wt[shortest]=finish_time-bt[shortest]-at[shortest];
  if(wt[shortest]<0)
    wt[shortest]=0;
  rt[shortest]=INT_MAX; complete++;
  ct[shortest]=finish_time;
  minm=INT_MAX;}}
void findAverageTimeFCFS(int processes[], int n, int bt[],
int at[], int wt[]){
```





```
process[i], at[i], bt[i], ct[i], tat[i], wt[i]);
float avg_wt = (float) total_wt/n;
float avg_tat = (float) total_tat/n;
printf("Avg wt = %f\n", avg_wt);
printf("Avg tat = %f\n", avg_tat);

void findavgTime(int processes[], int n, int bt[],
int wt[], int tat[], int ct[])
{
int wt[n], tat[n];
int total_wt = 0, total_tat = 0;
findWaitingTime(processes, n, bt, wt, at, ct);
findTurnAroundTime(processes, n, bt, wt, tat, ct);
printf("\nSJF (preemptive) scheduling\n");
printf("Process\tArrival\tBurst\tFinish\tWaiting\tTurnaround\n");
Time\tWaiting Time\n");
for(int i=0; i<n; i++)
{
total_wt += wt[i]; total_tat += tat[i];
printf("%d\t%d\t%d\t%d\t%d\t%d\n", processes[i], at[i], bt[i],
ct[i], tat[i], wt[i]);
}
float avg_wt = (float) total_wt/n;
float avg_tat = (float) total_tat/n;
printf("Avg wt = %f\n", avg_wt);
printf("Avg tat = %f\n", avg_tat);
}

int main()
{
int processes[10], burst_time[10], arrival_time[10],
```

```
completion_time[10], n;
printf("Enter no. of processes");
scanf("%d", &n);
printf("Enter arrival time and burst time for each process");
for(int i=0; i<n; i++){
printf("Arrival time of process-%d", i+1);
scanf("%d", &arrival_time[i]);
printf("Burst time of process-%d", i+1);
scanf("%d", &burst_time[i]);
process[i] = i+1;
}
completion_time[0] = arrival_time[0] + burst_time[0];
for(int i=0; i<n; i++){
if (arrival_time[i] > completion_time[i-1]){
completion_time[i] = arrival_time[i] +
burst_time[i];
}
else{
completion_time[i] = completion_time[i-1] + burst_time[i];
}
findAverageTimeFCFS(processes, n, burst_time, arrival_time,
completion_time);
findAverageTimeSJFNonPreemption(processes, n, burst_time,
arrival_time, completion_time);
findAverageTimeSJFPreemptive(processes, n, burst_time,
arrival_time, completion_time);
return 0;
}
```



OUTPUT

Enter the number of processes: 3
Enter arrival time and burst time for each process
Arrival Time of process[1]: 0
Burst time of process[1]: 2
Arrival time of process[2]: 1
Burst time of process[2]: 4
Arrival time of process[3]: 2
Burst time[3]: 3

FCFS

Process	Arrival Time	Burst Time	Completion Time	Turnaround	WT
1	0	2	2	2	0
2	1	4	6	5	1
3	2	3	9	7	4

Average waiting time = 1.6667
Average turn around time = 4.6667

NPF (non-preemptive) scheduling

Process	Arrival Time	Burst Time	Completion Time	TAT	WT
1	0	2	2	2	0
2	1	4	6	5	4
3	2	3	9	7	4

average waiting time = 1.3333
average turn around time = 4.3333

SJF (preemptive)

Process	Arrival Time	Burst Time	Completion Time	TAT	WT
1	0	2	2	2	0
2	1	4	9	8	4
3	2	3	5	3	0

Average waiting time = 1.333

Average turn around time = 4.3333

LAB2:Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. → Priority →Round Robin (Experiment with different quantum sizes for RR algorithm)

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

void priorityNonPreemptive(int n, int bt[], int priority[]) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (priority[j] > priority[j + 1]) {

                int temp = priority[j];
                priority[j] = priority[j + 1];
                priority[j + 1] = temp;

                temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;
            }
        }
    }

    wt[0] = 0;


    for (int i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
        total_wt += wt[i];
    }

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


    printf("\nProcess\t Burst Time\t Priority\t Waiting Time\t Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t %d\t\t %d\t\t %d\t\t %d\n", i + 1, bt[i], priority[i], wt[i], tat[i]);
    }
    printf("Average waiting time: %.2f\n", (float)total_wt / n);
    printf("Average turnaround time: %.2f\n", (float)total_tat / n);
}

void roundRobinScheduling(int n, int bt[], int quantum) {
    int remaining_bt[n], wt[n], tat[n], total_wt = 0, total_tat = 0;
```

INPUT:

```
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }

    int t = 0;


    int completion_time[n];
    for (int i = 0; i < n; i++) {
        completion_time[i] = 0;
    }


    while (1) {
        int done = 1;


        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0) {
                done = 0;


                if (remaining_bt[i] > quantum) {
                    t += quantum;
                    remaining_bt[i] -= quantum;
                } else {
                    t += remaining_bt[i];
                    remaining_bt[i] = 0;


                    wt[i] = t - bt[i];


                    tat[i] = t;
                }
            }
        }
    if (done == 1)
            break;
    }


    printf("\nProcess\t Burst Time\t Waiting Time\t Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t %d\t\t %d\t\t %d\n", i + 1, bt[i], wt[i], tat[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
```

```
    printf("Average turnaround time: %.2f\n", (float)total_tat / n);
}

int main() {
    int choice, n;

    do {
        printf("\nChoose the scheduling algorithm:\n");
        printf("1. Priority Scheduling (Non-preemptive)\n");
        printf("2. Round Robin Scheduling\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("\nEnter the number of processes for Priority Scheduling: ");
                scanf("%d", &n);

                int *bt1 = (int *)malloc(n * sizeof(int));
                int *priority1 = (int *)malloc(n * sizeof(int));

                printf("Enter burst time and priority for each process:\n");
                for (int i = 0; i < n; i++) {
                    printf("Process %d: ", i + 1);
                    scanf("%d %d", &bt1[i], &priority1[i]);
                }

                priorityNonPreemptive(n, bt1, priority1);

                free(bt1);
                free(priority1);
                break;
            case 2:
                printf("\nEnter the number of processes for Round Robin Scheduling: ");
                scanf("%d", &n);

                int *bt2 = (int *)malloc(n * sizeof(int));
                int quantum;

                printf("Enter burst time for each process:\n");
                for (int i = 0; i < n; i++) {
                    printf("Process %d: ", i + 1);
                    scanf("%d", &bt2[i]);
                }

                printf("Enter time quantum for Round Robin Scheduling: ");
```

```
            for (int i = 0; i < n; i++) {
                printf("Process %d: ", i + 1);
                scanf("%d", &bt2[i]);
            }

            printf("Enter time quantum for Round Robin Scheduling: ");
            scanf("%d", &quantum);

            roundRobinScheduling(n, bt2, quantum);


            free(bt2);
            break;
        case 3:
            printf("\nExiting...\n");
            break;
        default:
            printf("\nInvalid choice. Please enter a valid option.\n");
    }
} while (choice != 3);

return 0;
```

OUTPUT:

```
Choose the scheduling algorithm:
1. Priority Scheduling
2. Round Robin Scheduling
3. Exit
Enter your choice: 1

Enter the number of processes for Priority Scheduling: 5
Enter burst time and priority for each process:
Process 1: 10
3
Process 2: 1
1
Process 3: 2
4
Process 4: 1
5
Process 5: 5
2

Process  Burst Time      Priority        Waiting Time    Turnaround Time
1               1               1               0       1
2               5               2               1       6
3               10              3               6       16
4               2               4               16      18
5               1               5               18      19
Average waiting time: 8.20
Average turnaround time: 12.00
```

```
Choose the scheduling algorithm:
1. Priority Scheduling
2. Round Robin Scheduling
3. Exit
Enter your choice: 2

Enter the number of processes for Round Robin Scheduling: 3
Enter burst time for each process:
Process 1: 1
Process 2: 6
Process 3: 10
Enter time quantum for Round Robin Scheduling: 2

Process  Burst Time      Waiting Time    Turnaround Time
1               1               0               1
2               6               5               11
3               10              7               17
Average waiting time: 4.00
Average turnaround time: 9.67
```

15/05/24 LAB-2

Write a c program to simulate an scheduling algorithm to find turn around time and waiting time.

→ Priority

→ Round Robin

PRIORITY

```
#include<stdio.h>
#include<stdlib.h>
void priorityScheduling(int n, int bt[], int priority[]){
int wt[n], tat[n], total_wt=0, total_tat=0;
for(int i=0; i<n-1; i++){
  for(int j=0; j<n-i-1; j++){
    if(priority[j] > priority[j+1]){
      int temp = priority[j];
      priority[j] = priority[j+1];
      priority[j+1] = temp;
      temp = bt[j];
      bt[j] = bt[j+1];
      bt[j+1] = temp; }}}
wt[0] = 0;
for(int i=1; i<n; i++){
  wt[i] = wt[i-1] + bt[i-1];
  total_wt += wt[i]; }
for(int i=0; i<n; i++){
  tat[i] = bt[i] + wt[i];
  total_tat += tat[i]; }
```

printf("Process\t Burst Time\t Priority\t Waiting Time\t Turnaround Time\n");
for(int i=0; i<n; i++){
printf("%d\t %d\t %d\t %d\t %d\n", i+1, bt[i], priority[i], wt[i], tat[i]);}
printf("average waiting time = %.2f\n", (float) total_wt/n);
printf("average turn around time = %.2f\n", (float total_tat/n);

void roundRobinScheduling(int n, int bt[], int quantum){
int remaining_bt[n], wt[n], tat[n], total_wt=0; total_tat=0;
for(int i=0; i<n; i++){
remaining_bt[i] = bt[i];}
int t=0;
while(1){
int done=1;
for(int i=0; i<n; i++){
if(remaining_bt[i]>0){
done=0;
if(remaining_bt[i] > quantum){
t+= quantum;
remaining_bt[i] -= quantum;
wt[i] = t - bt[i];}
else{
t=t + remaining_bt[i];
wt[i] = t - bt[i];
remaining_bt[i] = 0;

```
tat[i]=t; } }
if(done==1)
break;}
printf("Process\t Burst time\t waiting time\t turnaround time\n");
for(int i=0;i<n;i++){
printf("%d\t\t%d\t\t%d\t\t%d\n", i+1, bt[i], wt[i],
tat[i]);
total_wt+=wt[i];
total_tat=tat[i];}
printf("Average waiting time = %.2f\n"(float) total_wt/n;
printf("Average turnaround time = %.2f\n", (float) total_tat/n
);}
int main(){
int choice n;
do {
printf("\n choose scheduling algorithm:\n");
printf("1. Priority Scheduling\n2. Round Robin Scheduling
3. Exit\n");
printf("Enter your choice");
scanf("%d", &choice);
switch(choice){
case 1:
printf("Enter no. of process for priority scheduling:");
scanf("%d",&n);
int *bt1 = (int*)malloc(n* sizeof(int));
int *priority1=(int*) malloc (n* sizeof(int));
```

```
printf("Enter burst time and priority for each process:\n");
for(int i=0; i<n; i++){
printf("Process %d", i+1);
scanf("%d %d", &bt[i], &priority[i]); }
priorityScheduling(n, bt, priority);
free(bt);
free(priority);
break;
case 2:
printf("\nEnter the number of process for round Robin
Scheduling");
scanf("%d", &n);
int *bt2 = (int*) malloc(n* sizeof(int));
int quantum;
printf("Enter burst time for each process:\n");
for(int i=0; i<n; i++){
  printf("Process %d", i+1);
  scanf("%d", &bt2[i]);
}
printf("Enter time quantum for Round Robin scheduling:");
scanf("%d", &quantum);
free(bt2);
break;
case 3:
    printf("Exit");
    break;
```

default:
printf("Invalid choice. Please Enter a valid option\n");
} while (choice!=3);
return 0;}

OUTPUT

choose the scheduling algorithm
1. priority scheduling
2. Round Robin scheduling
Enter your choice: 1
Enter the number of processes for priority scheduling:5
Enter burst time and priority for each process
Process 1: 10 3
Process 2: 1 1
Process 3: 2 4
Process 4: 1 5
Process 5: 5 2

Process	Burst time	Priority	Waiting Time	Turnaround Time
1	1	1	6	1
2	5	2	1	6
3	10	3	6	16
4	2	4	16	18
5	1	5	10	19

average waiting time: 8.20
average turnaround time: 12.00

Date: YOUVA

Enter your choice: 2
Enter the number of processes for Round Robin scheduling: 3
Enter burst time for each process:
Process 1: 1
Process 2: 6
Process 3: 10
Enter time quantum for Round Robin scheduling: 2

Process	Burst Time	Waiting Time	Turnaround Time
1	1	0	1
2	6	5	11
3	10	7	17

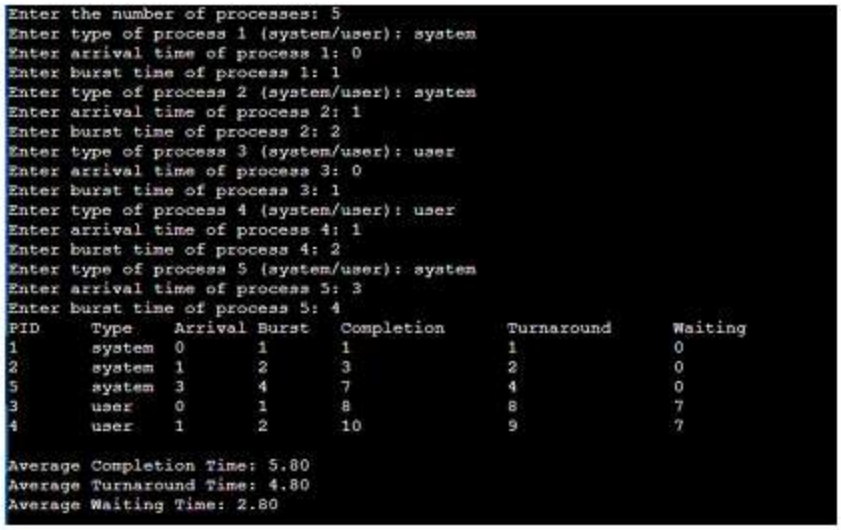
average waiting time: 4.00
average turnaround time: 9.67

SJF

LAB03: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
INPUT:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_PROCESSES 100
typedef struct {
    int pid;
    char type[10];
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
} Process;
typedef struct {
    Process processes[MAX_PROCESSES];
    int front;
    int rear;
} Queue;
void initQueue(Queue *q) {
    q->front = 0;
    q->rear = -1;
}
int isEmpty(Queue *q) {
    return q->rear < q->front;
}
void enqueue(Queue *q, Process p) {
    if (q->rear < MAX_PROCESSES - 1) {
        q->processes[++q->rear] = p;
    } else {
        printf("Queue is full\n");
    }
}
Process dequeue(Queue *q) {
    if (!isEmpty(q)) {
        return q->processes[q->front++];
    } else {
        printf("Queue is empty\n");
        Process emptyProcess = {0, "", 0, 0, 0, 0, 0};
        return emptyProcess;
    }
}
void multiLevelQueueScheduling(Queue *systemQueue, Queue *userQueue,
int *currentTime, int *totalCompletionTime, int *totalTurnaroundTime, int *totalWaitingTime, int *totalProcesses) {
    printf("PID\tType\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");
    while (!isEmpty(systemQueue)) {
        Process p = dequeue(systemQueue);
        if (*currentTime < p.arrival_time) {
            *currentTime = p.arrival_time;
        }
        p.completion_time = *currentTime + p.burst_time;
        p.turnaround_time = p.completion_time - p.arrival_time;
        p.waiting_time = p.turnaround_time - p.burst_time;
        *currentTime = p.completion_time;
        *totalCompletionTime += p.completion_time;
        *totalTurnaroundTime += p.turnaround_time;
        *totalWaitingTime += p.waiting_time;
        (*totalProcesses)++;
        printf("%d\t%s\t%d\t%d\t%d\t\t%d\t\t%d\n",
```

OUTPUT:


Enter the number of processes: 5
Enter type of process 1 (system/user): system
Enter arrival time of process 1: 0
Enter burst time of process 1: 1
Enter type of process 2 (system/user): system
Enter arrival time of process 2: 1
Enter burst time of process 2: 2
Enter type of process 3 (system/user): user
Enter arrival time of process 3: 0
Enter burst time of process 3: 1
Enter type of process 4 (system/user): user
Enter arrival time of process 4: 1
Enter burst time of process 4: 2
Enter type of process 5 (system/user): system
Enter arrival time of process 5: 3
Enter burst time of process 5: 4
PID Type Arrival Burst Completion Turnaround Waiting
1 system 0 1 1 1 0
2 system 1 2 3 2 0
5 system 3 4 7 4 0
3 user 0 1 8 8 7
4 user 1 2 10 9 7
Average Completion Time: 5.80
Average Turnaround Time: 4.80
Average Waiting Time: 2.80

Date:

3/6/24

LAB-3

Write a C program to simulate multilevel queue scheduling algorithm considering the following scenario. All process in the system are divided into two categories: system processes and user processes. System processes are given higher priority than user processes. use FCFS scheduling for the processes in each queue

INPUT

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX_PROCESSES 100
typedef struct{
int pid, arrival_time, burst_time, completion_time,
turnaround_time, waiting_time;
char type[10]; } process
typedef struct{
process processes[MAX_PROCESSES];
int front, rear; } Queue;
void initQueue(Queue *q){
q->front=0; q->rear=-1; }
int isEmpty(Queue *q){
return q->rear<q->front; }
void enqueue(Queue *q, process p){
if(q->rear<MAX_PROCESSES-1){
q->processes[++q->rear]=p; }
else printf("Queue is full"); }
```

Date: YOUVA

```
Process dequeue(Queue *q){
if(!isEmpty(q)){
return q->processes[q->front++];
else printf("Queue is Empty\n");
    Process emptyProcess={0,"",0,0,0,0,0};
    return emptyProcess;}
void multilevelQueueScheduling(Queue *systemQueue, Queue *
userQueue, int *currentTime, int *totalCompletionTime,
int *totalTurnaroundTime, int *totalWaitingTime, int *total
Processes){
while(!isEmpty(systemQueue)){
  Process p=dequeue(systemQueue);
  if(*currentTime<p.arrivalTime){
   *currentTime=p.arrivalTime;}
  p.completionTime=*currentTime+p.burstTime;
  p.turnaroundTime=p.completionTime-p.arrivalTime;
  p.waitingTime=p.turnaroundTime-p.burstTime;
  *totalCompletionTime+=p.completionTime;
  *totalTurnaroundTime+=p.turnaroundTime;
  *totalWaitingTime+=p.waitingTime;
   (*totalProcesses)++;}
while(!isEmpty(userQueue)){
process p=dequeue(userQueue);
if(*currentTime<p.arrivalTime){
  *currentTime=p.arrivalTime;}
p.completionTime=*currentTime+p.burstTime;
```

```
p.turnaround_time = p.completion_time - p.arrival_time;
p.waiting_time = p.turnaround_time - p.burst_time;
*total_completion_time += p.completion_time;
*total_turnaround_time += p.turnaround_time;
*total_waiting_time += p.waiting_time;
(*total_processes)++; }
Put main()
Queue systemQueue, userQueue;
initQueue(&systemQueue);
initQueue(&userQueue);
int n;
printf("Enter no. of processes");
scanf("%d", &n);
if (n > MAX_PROCESSES){
printf("limit exceeded\n"); }
for(int i=0; i<n; i++){
process p;
p.pid = i+1;
printf("Enter type of process %d", p.pid);
scanf("%d", &p.type);
printf("Enter arrival time");
scanf("%d", &p.arrival_time);
printf("Enter burst time");
scanf("%d", &p.burst_time);
if(isCompatible(p.type, SYSTEM) == 0){
enqueue(&systemQueue, p);
```

```
if(strcmp(p.type, "user")==0){
    enqueue(&userQueue, p);}else{
printf("Invalid process");}}
int waitingTime=0, totalCompletionTime=0, totalTurnaroundTime=0,
totalWaitingTime=0, totalProcess=0
multilevelQueueScheduling(systemQueue, &userQueue, &waitingTime,
&totalCompletionTime, &totalTurnaroundTime, &totalWaitingTime,
&totalProcesses);
if(totalProcesses>0){
printf("average CT=%.2f\n, avg TAT=%.2f\n average
WT=%.2f\n", (float)totalCompletionTime/totalProcesses,
(float)totalTurnaroundTime/totalProcesses, (float)
waitingTime/totalProcesses);}
return 0;
}
```

OUTPUT

Enter the number of processes: 5
Enter type of process 1 (System/user): System
Enter arrival time of process 1: 0
Enter burst time of process 1: 1
Enter type of process 2 (System/user): System
Enter arrival time of process 2: 1
Enter burst time of process 2: 2
Enter type of process 3 (System/user): user
Enter arrival time of process 3: 0
Enter burst time of process 3: 1
Enter type of process 4 (System/user): user
Enter arrival time of process 4: 1
Enter burst time of process 4: 2
Enter type of process 5 (System/user): System
Enter arrival time of process 5: 3
Enter burst time of process 5: 4

P ID	Type	Arrival	Burst	Completion	Turnaround	Waiting
1	System	0	1	1	1	0
2	System	1	2	3	2	0
5	System	3	4	7	4	0
3	user	0	1	8	8	7
4	user	1	2	10	9	7

Average completion Time: 5.80 ms
Average turnaround Time: 4.80 ms
Average waiting Time = 2.80

LAB04:

Write a C program to simulate Real-Time CPU Scheduling algorithms:
a) Rate- Monotonic
b) Earliest-deadline First
c) Proportional scheduling
Input:

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

#define MAX_TASKS 10

typedef struct {
    int id;
    int period;
    int deadline;
    int computation_time;
    int remaining_time;
    int completion_time;
} Task;

void rate_monotonic(Task tasks[], int num_tasks);
void earliest_deadline_first(Task tasks[], int num_tasks);
void proportional_scheduling(Task tasks[], int num_tasks);

int main() {
    int num_tasks;
    Task tasks[MAX_TASKS];

    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);

    printf("Enter the details of each task (id, period, deadline, computation time):\n");
    for (int i = 0; i < num_tasks; i++) {
        printf("Task %d: ", i + 1);
        scanf("%d %d %d %d", &tasks[i].id, &tasks[i].period, &tasks[i].deadline, &tasks[i].computation_time);
        tasks[i].remaining_time = tasks[i].computation_time;
        tasks[i].completion_time = 0; // Initialize completion time
    }

    printf("\nUser Input:\n");
    for (int i = 0; i < num_tasks; i++) {
        printf("Task %d: ID=%d, Period=%d, Deadline=%d, Computation Time=%d\n",
               i + 1, tasks[i].id, tasks[i].period, tasks[i].deadline, tasks[i].computation_time);
    }
```

```
    rate_monotonic(tasks, num_tasks);
    earliest_deadline_first(tasks, num_tasks);
    proportional_scheduling(tasks, num_tasks);

    printf("\nCompletion times for the first two tasks:\n");
    if (num_tasks >= 1) {
        printf("Task 1 (ID=%d) Completion Time: %d\n", tasks[0].id, tasks[0].completion_time);
    }
    if (num_tasks >= 2) {
        printf("Task 2 (ID=%d) Completion Time: %d\n", tasks[1].id, tasks[1].completion_time);
    }

    return 0;
}

void rate_monotonic(Task tasks[], int num_tasks) {
    printf("\nRate-Monotonic Scheduling:\n");

    for (int i = 0; i < num_tasks - 1; i++) {
        for (int j = 0; j < num_tasks - i - 1; j++) {
            if (tasks[j].period > tasks[j + 1].period) {
                Task temp = tasks[j];
                tasks[j] = tasks[j + 1];
                tasks[j + 1] = temp;
            }
        }
    }

    int current_time = 0;
    for (int i = 0; i < num_tasks; i++) {
        tasks[i].completion_time = current_time + tasks[i].computation_time;
        current_time = tasks[i].completion_time;
        printf("Task %d scheduled (Completion Time: %d)\n", tasks[i].id, tasks[i].completion_time);
    }
}

void earliest_deadline_first(Task tasks[], int num_tasks) {
```

```
    }
}

void earliest_deadline_first(Task tasks[], int num_tasks) {
    printf("\nEarliest-Deadline First Scheduling:\n");

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

    int current_time = 0;
    for (int i = 0; i < num_tasks; i++) {
        tasks[i].completion_time = current_time + tasks[i].computation_time;
        current_time = tasks[i].completion_time;
        printf("Task %d scheduled (Completion Time: %d)\n", tasks[i].id, tasks[i].completion_time);
    }
}

void proportional_scheduling(Task tasks[], int num_tasks) {
    printf("\nProportional Scheduling:\n");

    float total_period_inverse = 0;
    for (int i = 0; i < num_tasks; i++) {
        total_period_inverse += 1.0 / tasks[i].period;
    }

    for (int i = 0; i < num_tasks; i++) {
        float proportion = (1.0 / tasks[i].period) / total_period_inverse;
        printf("Task %d gets %.2f%% of CPU time\n", tasks[i].id, proportion * 100);
    }
}
```

Output:

```
Enter the number of tasks: 3
Enter the details of each task (id, period, deadline, computation time):
Task 1: 1
5
5
2
Task 2: 2
10
10
1
Task 3: 3
7
7
3

User Inputs:
Task 1: ID=1, Period=5, Deadline=5, Computation Time=2
Task 2: ID=2, Period=10, Deadline=10, Computation Time=1
Task 3: ID=3, Period=7, Deadline=7, Computation Time=3

Rate-Monotonic Scheduling:
Task 1 scheduled (Completion Time: 2)
Task 3 scheduled (Completion Time: 5)
Task 2 scheduled (Completion Time: 6)

Earliest-Deadline First Scheduling:
Task 1 scheduled (Completion Time: 2)
Task 3 scheduled (Completion Time: 5)
Task 2 scheduled (Completion Time: 6)

Proportional Scheduling:
Task 1 gets 45.16% of CPU time
Task 3 gets 32.26% of CPU time
Task 2 gets 22.58% of CPU time
```

09/08/24 LAB-4

Write a C program to simulate Real Time OS scheduling algorithms:

a) Rate-monotonic

b) Earliest deadline first

c) Proportional scheduling

INPUT:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TASKS 10
typedef struct {
int id;
int period;
int deadline;
int computation_time;
int remaining_time;
} Task;
void rate_monotonic(Task task[], int num_tasks);
void earliest_deadline_first(Task task[], int num_tasks);
void proportional_scheduling(Task tasks[], int num_tasks);
int main() {
int num_tasks;
Task tasks[MAX_TASKS];
printf("Enter the number of tasks:");
scanf("%d", &num_tasks);
printf("Enter details of each task (id, period, deadline, computation time):\n");
```

```
for(int i=0; i<num_tasks; i++){
  printf("Task %d", i+1);
  scanf("%d %d %d %d", &task[i].id, &task[i].period,
&tasks[i].deadline, &task[i].computation_time);
  task[i].remaining_time = task[i].computation_time;
  task[i].completion_time=0;}
printf("\nUser Input:\n");
for(int i=0; i<num_tasks; i++){
  printf("Task %d: ID=%d Period=%d, Deadline=
%d, Computation Time=%d\n", i+1, task[i].id, task[i].
period, tasks[i].computation_time);}
rate_monotonic(tasks, num_tasks);
earliest_deadline_first(tasks, num_tasks);
proportional_scheduling(tasks, num_tasks);
printf("\n Completion times for the first two tasks:\n");
if (num_tasks>=1){
  printf("Task 1 (ID=%d) Completion Time:%d\n",
  tasks[0].id, tasks[0].completion_time);}
if (num_tasks>=2){
  printf("Task 2 (ID=%d) Completion Time:%d
  Completion Time: %d\n", task[1].id, task[1].comp
  } return 0;}
void rate_monotonic(Task tasks[], int num_tasks){
  printf("Rate Monotonic Scheduling\n");
for(int i=0; i<num_tasks-1; i++){
  for(int j=0; j<num_tasks-i-1; i++){
```

```
if (tasks[j].period > tasks[j+1].period){
    Task temp = tasks[j];
    tasks[j] = tasks[j+1];
    tasks[j+1] = temp;}}
int current_time = 0
for(int i=0; i<num_tasks; i++){
tasks task[i].completion_time = current_time + task[i].
computation_time;
current_time = task[i].completion_time;
printf("Task %d Scheduled Completion time = %d\n",
task[i].id, tasks[i].completion_time);}
void earliest_deadline_first(Task tasks[], int num_tasks){
void earliest
printf("Earliest Deadline First Scheduling:\n");
for(int i=0; i<num_tasks-1; i++){
for(int j=0; j<num_tasks-i-1; j++){
if (tasks[j].deadline > tasks[j+1].deadline){
Task temp = tasks[j];
tasks[j] = tasks[j+1];
tasks[j+1] = temp;}}
int current_time = 0
for(int i=0; i<num_tasks; i++){
tasks[i].completion_time = current_time + tasks[i].
computation_time;
current_time = tasks[i].completion_time;
printf("Task %d Scheduled (Completion Time: %d)\n",
tasks[i].id, tasks[i].completion_time);}
```

```
void proportional_scheduling(Task tasks[], int num_tasks)
printf("\nProportional Scheduling:");
float total_period_inverse=0;
for(int i=0; i<num_tasks;i++){
    total_period_inverse+=1.0/tasks[i].period;}
for(int i=0; i<num_tasks; i++)
    total_period_inverse+=
for(int i=0; i<num_tasks; i++){
float proportion = (1.0/tasks[i].period)/total_period_inverse;
printf("Task %d gets %.2f%% of CPU time\n", tasks[i].id
, proportion * 100);}
```

OUTPUT

Enter the number of tasks: 3

Enter the details of each task(id, period, deadline, computation time):

Task 1 : 1

5

5

2

Task 2 : 2

10

10

1

Task 3 : 3

7

7
3

User Inputs:
Task 1: ID = 1, period = 5, deadline = 5, computation time = 2
Task 2: ID = 2, period = 10, deadline = 10, computation time = 1
Task 3: ID = 3, period = 7, Deadline = 7, computation time = 3

Rate - Monotonic Scheduling
Task 1 scheduled (computation time = 2)
Task 2 scheduled (computation time = 5)
Task 3 scheduled (computation time = 6)

Earliest - Deadline First Scheduling
Task 1 Scheduled (computation time = 2)
Task 2 scheduled (computation time = 5)
Task 3 scheduled (computation time = 0)

Proportional Scheduling:
Task 1 gets 15.16% of CPU time
Task 2 gets 32.26% of CPU time
Task 3 gets 22.58% of CPU time

5/6/24

OSLAB05:Write a C program to simulate producer-consumer problem using semaphores.

INPUT:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define BUF_SIZE 10
#define MAX_ITMS 20

int buf[BUF_SIZE];
int cnt = 0;
int in = 0;
int out = 0;
int prod_cnt = 0;
int cons_cnt = 0;

pthread_mutex_t mtx;
pthread_cond_t cond_prod;
pthread_cond_t cond_cons;

void display_buffer() {
    printf("Buffer: [");
    for (int i = 0; i < cnt; ++i) {
        printf("%d ", buf[(out + i) % BUF_SIZE]);
    }
    printf("]\n");
}

void* prod(void* param) {
    int* prod_amount = (int*)param;
    int items_to_produce = *prod_amount;

    for (int i = 0; i < items_to_produce; ++i) {
        int item = rand() % 100;

        pthread_mutex_lock(&mtx);

        while (cnt == BUF_SIZE) {
            pthread_cond_wait(&cond_prod, &mtx);
        }

        buf[in] = item;
        in = (in + 1) % BUF_SIZE;
        cnt++;
        prod_cnt++;
        printf("Produced: %d\n", item);

        display_buffer();

        pthread_cond_signal(&cond_cons);
```

OUTPUT:

```
Menu:
1. Start Production and Consumption
2. Exit
Enter your choice: 1
Enter the number of items to produce: 5
Produced: 41
Buffer: [41 ]
Buffer: [41 ]
Consumed: 41
Buffer: []
Produced: 34
Buffer: [34 ]
Buffer: [34 ]
Consumed: 34
Buffer: []
Produced: 69
Buffer: [69 ]
Produced: 78
Buffer: [69 78 ]
Produced: 62
Buffer: [69 78 62 ]
Buffer: [69 78 62 ]
Consumed: 69
Buffer: [78 62 ]
Buffer: [78 62 ]
Consumed: 78
Buffer: [62 ]
Buffer: [62 ]
Consumed: 62
Buffer: []
```

12/06/24 LAB-05

Write a C program to stimulate producer-consumer problem using Semaphores

INPUT:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define BUF_SIZE 10
#define MAX_ITEMS 20
int buf[BUF_SIZE], cnt=0, pin=0, out=0, prod_cnt=0,
cons_cnt=0;
pthread_mutex_t mtx;
pthread_cond_t cond_prod;
pthread_cond_t cond_cons;
void display_buffer(){
    printf("Buffer:[");
    for(int i=0; i<cnt; ++i){
        printf("%d", buf[(out+i)%BUF_SIZE]);
    }
    printf("]\n");
}
void *prod(void *param){
int *prod_amount = (int*) param;
int items_to_produce = *prod_amount;
for(int i=0; i<items_to_produce; ++i){
```

```
int item = rand() % 100;
pthread_mutex_lock(&mtx);
while (cnt == BUF_SIZE) {
    pthread_cond_wait(&cond_prod, &mtx);
}
buf[cnt] = item;
in = (in+1) % BUF_SIZE
cnt++;
prod_cnt++;
printf("Produced: %d\n", item);
display_buffer();
pthread_cond_signal(&cond_cons);
pthread_mutex_unlock(&mtx);
sleep(rand() % 2);
}
return NULL;
};
void* cons(void * param){
while(1){
    pthread_mutex_lock(&mtx);
    if(cons_cnt >= MAX_ITEMS){
        pthread_mutex_unlock(&mtx);
        break;
    }
    while(cnt==0){
        pthread_cond_wait(&cond_cons, &mtx);}
    display_buffer();
```

```
int item = buf[out];
out = (out+1) % BUF_SIZE;
cnt--;
cons_cnt++;
printf("Consumed: %d\n", item);
display_buffer();
pthread_cond_signal(&cond_prod);
pthread_mutex_unlock(&mutex);
sleep(rand() % 2);
return NULL;
}
int main() {
pthread_t tid_prod, tid_cons;
char choice;
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_prod, NULL);
pthread_cond_init(&cond_cons, NULL);
do {
printf("\n\n1. Start production and consumption\n2. Exit\n");
printf("Enter your choice: ");
scanf("%c", &choice);
switch(choice) {
case '1': {
int prod_amount;
printf("Enter the number of items to produce:");
```

scanf("%d", &prod_amount);
p_thread_create(&tid_prod, NULL, prod, &prod_amount);
p_thread_create(&tid_cons, NULL, cons, NULL);
p_thread_join(tid_prod, NULL);
p_thread_join(tid_cons, NULL);
printf("Production & Consumption completed\n");
break;
case 2:
printf("Exiting...\n");
break;
default:
printf("Invalid choice. Please try again.\n");
}
} while(choice != 2);
p_thread_mutex_destroy(&mutex);
p_thread_cond_destroy(&cond_prod);
p_thread_cond_destroy(&cond_cons);
return 0;
}

OUTPUT

Menu
1. Start Production and Consumption
2. Exit
Enter your choice: 1
Enter the number of items to produce: 5

Produced: 41
Buffer: [41]
Buffer: [41]
Consumed: 41
Buffer: []
Produced: 34
Buffer: [34]
Buffer: [34]
Consumed: 34
Buffer: []
Produced: 69
Buffer: [69]
Produced: 78
Buffer [69, 78]
Produced: 62
Buffer: [69 78 62]
Buffer: [69 78 62]
Consumed: 69
Buffer: [78 62]
Buffer: [78 62]
Consumed: 78
Buffer: [62]
Buffer: [62]
Consumed: 62
Buffer: []

Sol. 1
12/6/24

LAB06:
Write a C program to simulate the concept of dining philosophers
INPUT:

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + N - 1) % N
#define RIGHT (phnum + 1) % N
int N;
int *state;
int *phil;
int running_time = 10;
time_t start_time;
sem_t mutex;
sem_t *S;
void test(int phnum) {
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != 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]);
    }
}
void take_fork(int phnum) {
    sem_wait(&mutex);
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
```

```
40
41  void put_fork(int phnum) {
42      sem_wait(&mutex);
43      state[phnum] = THINKING;
44      printf("Philosopher %d putting fork %d and %d down\n",
45              phnum + 1, LEFT + 1, phnum + 1);
46      printf("Philosopher %d is thinking\n", phnum + 1);
47      test(LEFT);
48      test(RIGHT);
49      sem_post(&mutex);
50  }
51
52  void* philosopher(void* num) {
53      int* i = num;
54      while (1) {
55          if (difftime(time(NULL), start_time) >= running_time) {
56              break;
57          }
58          sleep(1);
59          take_fork(*i);
60          sleep(0);
61          put_fork(*i);
62      }
63      return NULL;
64  }
65
66  int main() {
67      printf("Enter the number of philosophers: ");
68      scanf("%d", &N);
69      state = (int *)malloc(N * sizeof(int));
70      phil = (int *)malloc(N * sizeof(int));
71      S = (sem_t *)malloc(N * sizeof(sem_t));
72      for (int i = 0; i < N; i++) {
73          phil[i] = i;
74      }
75      pthread_t thread_id[N];
76      sem_init(&mutex, 0, 1);
77      for (int i = 0; i < N; i++) {
```

```
            break;
        }
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
    return NULL;
}

int main() {
    printf("Enter the number of philosophers: ");
    scanf("%d", &N);
    state = (int *)malloc(N * sizeof(int));
    phil = (int *)malloc(N * sizeof(int));
    S = (sem_t *)malloc(N * sizeof(sem_t));
    for (int i = 0; i < N; i++) {
        phil[i] = i;
    }
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);
    for (int i = 0; i < N; i++) {
        sem_init(&S[i], 0, 0);
    }
    start_time = time(NULL);
    for (int i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
    for (int i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }
    free(state);
    free(phil);
    free(S);
    return 0;
}
```

OUTPUT:

```
Enter the number of philosophers: 5
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 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 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
```

19/6/24 LAB-06

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

INPUT:

```
#include<pthread.h>
#include<semaphore.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum+N-1)%N
#define RIGHT (phnum+1)%N
int N, *state, *phil, running=1;
time_t start_time;
sem_t mutex, *S;
void test(int phnum){
if(state[phnum]==HUNGRY && state[LEFT]!=EATING
&& state[RIGHT]!=EATING){ state[phnum]=EATING;
sleep(2);
printf("Philosopher %d takes fork %d and %d\n", phnum,
LEFT+1, phnum+1);
printf("Philosopher %d is eating\n", phnum+1);
sem_post(&S[phnum]);}
void take_fork(int phnum){
```

```
sem_wait(&mutex);
state[phnum] = HUNGRY;
printf("Philosopher %d is Hungry\n", phnum+1);
test(phnum);
sem_post(&mutex);
sem_wait(&s[phnum]);
sleep(1); }
void put_fork(int phnum){
sem_wait(&mutex);
state[phnum] = THINKING;
printf("Philosopher %d is putting fork %d and %d down\n",
phnum+1, LEFT+1, phnum+1);
printf("Philosopher %d is thinking", phnum+1);
test(LEFT); test(RIGHT); sem_post(&mutex); }
void* philosopher(void* num){
int* i = num;
while(1){
if(difftime(time(NULL), start_time) > running_time
break; } sleep(1); take_fork(*i); sleep(0);
put_fork(*i); } }
return NULL; }
int main(){
printf("Enter the number of philosophers: ");
scanf("%d", &N);
state = (int*) malloc(N * sizeof(int));
phil = (int*) malloc(N * sizeof(int));
```

```
s = (sem_t*) malloc (N* sizeof (sem_t));
for(int i=0; i<N; i++) {
    phil[i] = i; }
pthread_t thread_id[N]; sem_init(&mutex,0,1);
for(int i=0; i<N; i++){
    sem_init(&S[i],0,0); }
start_time = time(NULL);
for(int i=0; i<N; i++){
    pthread_create(&thread_id[i],NULL,philosopher,&phil[i]);
    printf("Philosopher %d is thinking", i+1); }
for(int i=0; i<N; i++){
    pthread_join(thread_id[i], NULL); }
free(state); free(phil); pause();
return 0;
}
```

OUTPUT:

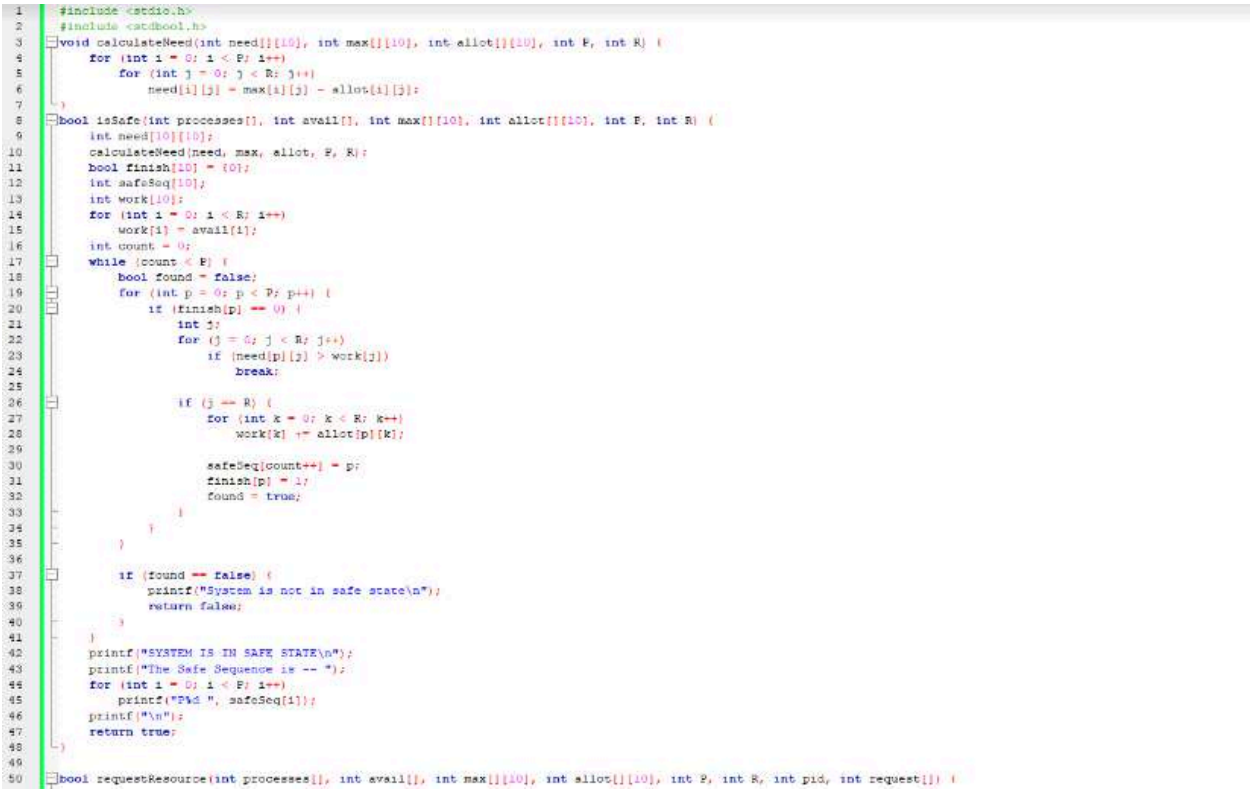
```
Enter the number of philosophers: 5
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is hungry
Philosopher 4 is hungry
Philosopher 2 is hungry
Philosopher 1 is hungry
```

Philosopher 5 is hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 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 4 takes fork 5 and 1
Philosopher 4 is eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is eating
Philosopher 5 is hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is eating
Philosopher 3 putting putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is eating
Philosopher 5 putting fork 4 and 5 down
philosopher 5 is thinking
philosopher 2 putting fork 1 and 2 down
philosopher 2 is thinking

LAB07:

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

INPUT:



```c
#include <stdio.h>
#include <stdbool.h>
void calculateNeed(int need[][10], int max[][10], int allot[][10], int P, int R) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allot[i][j];
}
bool isSafe(int processes[], int avail[], int max[][10], int allot[][10], int P, int R) {
    int need[10][10];
    calculateNeed(need, max, allot, P, R);
    bool finish[10] = {0};
    int safeSeq[10];
    int work[10];
    for (int i = 0; i < R; i++)
        work[i] = avail[i];
    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0; k < R; k++)
                        work[k] += allot[p][k];

                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = true;
                }
            }
        }

        if (found == false) {
            printf("System is not in safe state\n");
            return false;
        }
    }
    printf("SYSTEM IS IN SAFE STATE\n");
    printf("The Safe Sequence is -- ");
    for (int i = 0; i < P; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n");
    return true;
}

bool requestResource(int processes[], int avail[], int max[][10], int allot[][10], int P, int R, int pid, int request[]) {
```

```
        processes[i] = i;
    }
    int avail[10];
    int max[10][10];
    int allot[10][10];
    for (int i = 0; i < P; i++) {
        printf("Enter allocation for P%d: ", i);
        for (int j = 0; j < R; j++) {
            scanf("%d", &allot[i][j]);
        }
        printf("Enter max for P%d: ", i);
        for (int j = 0; j < R; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter available resources: ");
    for (int i = 0; i < R; i++) {
        scanf("%d", &avail[i]);
    }
    int need[10][10];
    calculateNeed(need, max, allot, P, R);
    printf("\nProcess\tAllocation\tMax\t\tNeed\n");
    for (int i = 0; i < P; i++) {
        printf("P%d\t", i);
        for (int j = 0; j < R; j++) {
            printf("%d ", allot[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < R; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < R; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
    isSafe(processes, avail, max, allot, P, R);
    int pid, request[10];
    printf("Enter PID for new request: ");
    scanf("%d", &pid);
    printf("Enter request for resources: ");
    for (int i = 0; i < R; i++) {
        scanf("%d", &request[i]);
    }
    requestResource(processes, avail, max, allot, P, R, pid, request);

    return 0;
}
```

OUTPUT:

```
Enter number of processes: 5
Enter number of resources: 3
Enter allocation for P0: 0
1
0
Enter max for P0: 7
5
3
Enter allocation for P1: 2
0
0
Enter max for P1: 3
2
2
Enter allocation for P2: 3
0
2
Enter max for P2: 9
0
2
Enter allocation for P3: 2
1
1
Enter max for P3: 2
2
2
Enter allocation for P4: 0
0
2
Enter max for P4: 4
3
3
Enter available resources: 3
3
2

Process Allocation      Max             Need
P0      0 1 0           7 5 3           7 4 3
P1      2 0 0           3 2 2           1 2 2
P2      3 0 2           9 0 2           6 0 0
P3      2 1 1           2 2 2           0 1 1
P4      0 0 2           4 3 3           4 3 1
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- P1 P3 P4 P0 P2
Enter PID for new request: 1
Enter request for resources: 1
0
2
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- P1 P3 P4 P0 P2
Resources have been allocated successfully
```

LAB-07

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

```
#include<stdio.h>
#include<stdbool.h>
void calculateNeed(int need[][10], int max[][10], int allot[][10],
int P, int R){
    for(int i=0; i<P; i++)
        for(int j=0; j<R; j++)
            need[i][j] = max[i][j] - allot[i][j]; }
bool isSafe(int processes[], int avail[], int max[][10], int allot[]
[10], int P, int R){
int need[10][10];
calculateNeed(need, max, allot, P, R);
bool finish[10] = {0};
int safeSeq[10]; int work[10];
for(int i=0; i<R; i++)
        work[i] = avail[i];
int count=0;
while(count<P){
    bool found = false;
    for(int p=0; p<P; p++){
        if(finish[p]==0){
            int j;
            for(j=0; j<R; j++)
                if(need[p][j] > work[j])
```

```
        break;
    if (j==R){
        for (int k=0; k<R; k++)
        work[k] += allot[p][k];
        safeSeq[count++]=p
        finish[p]=1;
        found=true;}}
    if (found==false){
        printf("system is not in safe state\n");
        return false;}
    printf("system is in safe state\n");
    printf("The safe sequence is--");
    for(int i=0;i<P;i++)
        printf("%d", safeSeq[i]);
    printf("\n"); return true;}
bool requestResource(int process[], int avail[], int max[][10],
int allot[][10], int P, int R, int pid, int request[]);
    int need[10][10];
    calculateNeed(need, max, allot, P, R);
    for(int i=0; i<R; i++){
    if(request[i] > need[pid][i]){
        printf("Error: Process has exceeded its maximum claim");
        return false;}}
    for(int i=0; i<R; i++){
    if(request[i] > avail[i]){
        printf("Resources are not available, process must wait");
```

```
return false; } }
for (int i=0 ; i< k; i++){
avail[ri][rj] -= request[i];
allot[pid][ri] += request[i];
need[pid][ri] -= request[i]; }
P[ri]=safeprocesses, avail, max, allot, P, R)==false){
avail[ri] += request[ri];
allot[pid][ri] -= request[i];
need[pid][ri] += request[i]; }
printf("System is not in safe state after allocation\n");
return false; }
printf("Resources have been allocated successfully\n");
return true; }

int main(){
int P, R;
printf("Enter number of process and resources\n");
scanf("%d %d", &P, &R);
int process[10], avail[10], max[10][10], allot[10][10],
need[10][10], pid, request[10];
for(i=0; i<P; i++){
printf("Enter allocation for %d: ", i);
for(j=0; j<R; j++){
scanf("%d", &allot[i][j]); }
printf("Enter max for P%d ", i);
for(int j=0; j<R; j++){
scanf("%d", &max[i][j]); }
```

```
printf("Enter available resources");
for (i=0; i<R; i++)
    scanf("%d", &avail[i]);
calculate need (need, max, allot, P, R);
printf("Process\tAllocation\tMax\tNeed\n");
for (i=0; i<P; i++)
    printf("P%d\t", i);
    for (j=0; j<R; j++)
        printf("%d", alloc[i][j]);
    for (j=0; j<R; j++)
        printf("%d", max[i][j]);
    for (j=0; j<R; j++)
        printf("%d", need[i][j]);
    printf("\n");
isSafe(process, avail, max, alloc, P, R);
int pid, request[R];
printf("Enter PID and request for for new request:");
scanf("%d", &pid);
printf("Enter request for resources:");
for (int i=0; i<K; i++)
    scanf("%d", &request[i]);
requestResource(process, avail, max, alloc, P, R, pid, request
return 0;
```

Date:

Output

Enter number of processes: 5
Enter number of resources: 3
Enter allocation for P0 = 0
1
0

Enter max for P0 : 7
5
3

Enter allocation for P1: 2
0
0

Enter max for P1: 3
2
2

Enter allocation for P2 = 3
0
2

Enter max for P2 :
0
2

Enter allocation for P3 : 2
1
1

Enter max for P3 :

1

Enter allocation for P4: 0

0

2

Enter max for P4: 4

3

3

Enter available resources: 3

3

2

Process	Allocation	MAX	Need
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

SYSTEM IS IN SAFE STATE

The Safe Sequence is -- P1 P3 P4 P0 P2

Enter P1 0 for new request: 1

Enter request for resource: 1

2

SYSTEM IS IN SAFE STATE

The Safe Sequence is -- P1 P3 P4 P0 P2

Resources have been allocated successfully.

LAB08:
Write a C program to simulate deadlock detection
INPUT:

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

bool isLessThanOrEqual(int request[], int work[], int R) {
    for (int i = 0; i < R; i++) {
        if (request[i] > work[i]) {
            return false;
        }
    }
    return true;
}

void addVectors(int work[], int allocation[], int R) {
    for (int i = 0; i < R; i++) {
        work[i] += allocation[i];
    }
}

void printState(bool finish[], int P) {
    printf("Finish vector: ");
    for (int i = 0; i < P; i++) {
        printf("%s ", finish[i] ? "true" : "false");
    }
    printf("\n");
}

void deadlockDetection(int allocation[][3], int request[][3], int available[], int P, int R) {
    int work[R];
    bool finish[P];
    int sequence[P];
    int index = 0;

    for (int i = 0; i < R; i++) {
        work[i] = available[i];
    }
    for (int i = 0; i < P; i++) {
        bool nonzero = false;
        for (int j = 0; j < R; j++) {
            if (allocation[i][j] != 0) {
                nonzero = true;
                break;
            }
        }
        finish[i] = !nonzero;
    }

    while (true) {
        bool found = false;
        for (int i = 0; i < P; i++) {
```

```
        for (int i = 0; i < P; i++) {
            if (!finish[i] && isLessThanOrEqual(request[i], work, R)) {
                addVectors(work, allocation[i], R);
                finish[i] = true;
                sequence[index++] = i;
                found = true;
                break;
            }
        }
        if (!found) {
            break;
        }
    }

    bool deadlock = false;
    for (int i = 0; i < P; i++) {
        if (!finish[i]) {
            printf("System is in deadlock, process P%d is deadlocked.\n", i);
            deadlock = true;
        }
    }
    if (!deadlock) {
        printf("System is not in deadlock.\nSafe Sequence: ");
        for (int i = 0; i < P; i++) {
            printf("P%d ", sequence[i]);
        }
        printf("\n");
    }
}

int main() {
    int P, R;

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

    printf("Enter the number of resource types: ");
    scanf("%d", &R);

    int allocation[P][R], request[P][R], available[R];

    printf("Enter the Allocation Matrix:\n");
    for (int i = 0; i < P; i++) {
        printf("Process P%d:\n", i);
        for (int j = 0; j < R; j++) {
            printf("Resource %c: ", 'A' + j);
            scanf("%d", &allocation[i][j]);
        }
    }
```

```
        }
    }

    printf("Enter the Request Matrix:\n");
    for (int i = 0; i < P; i++) {
        printf("Process P%d:\n", i);
        for (int j = 0; j < R; j++) {
            printf("Resource %c: ", 'A' + j);
            scanf("%d", &request[i][j]);
        }
    }

    printf("Enter the Available Resources:\n");
    for (int i = 0; i < R; i++) {
        printf("Resource %c: ", 'A' + i);
        scanf("%d", &available[i]);
    }

    deadlockDetection(allocation, request, available, P, R);

    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 5
Enter the number of resource types: 3
Enter the Allocation Matrix:
Process P0:
Resource A: 0
Resource B: 1
Resource C: 0
Process P1:
Resource A: 2
Resource B: 0
Resource C: 0
Process P2:
Resource A: 3
Resource B: 0
Resource C: 3
Process P3:
Resource A: 2
Resource B: 1
Resource C: 1
Process P4:
Resource A: 0
Resource B: 0
Resource C: 2
Enter the Request Matrix:
Process P0:
Resource A: 0
Resource B: 0
Resource C: 0
Process P1:
Resource A: 2
Resource B: 0
Resource C: 2
Process P2:
Resource A: 0
Resource B: 0
Resource C: 0
Process P3:
Resource A: 1
Resource B: 0
Resource C: 0
Process P4:
Resource A: 0
Resource B: 0
Resource C: 2
Enter the Available Resources:
Resource A: 0
Resource B: 0
Resource C: 0
System is not in deadlock.
Safe Sequence: P0 P2 P1 P3 P4
```

3/07/24

Page No.:
Date:
YOUVA

Write a C program to simulate deadlock detection
INPUT:

```
#include<stdio.h>
#include<stdbool.h>
bool isLessThanOrEqual(int request[], int work[], int R){
    for(int i=0; i<R; i++){
        if(request[i] > work[i]){
            return false; }}
    return true; }
void addResources(int work[], int allocation, int R){
    for(int i=0; i<R; i++){
        work[i] += allocation[i]; }}
void printState(bool finish[], int P){
    printf("Finish vector: ");
    for(int i=0; i<P; i++){
        printf("%s ", finish[i] ? "true" : "false"); }
    printf("\n"); }
void deadlockDetection(int allocation[][3], int request[][3],
int available[], int P, int R){
    int work[R], sequence[P], index=0;
    bool finish[P];
    for(int i=0; i<R; i++){
        work[i] = available[i]; }
    for(int i=0; i<P; i++){
        bool nonzero=false;
        for(int j=0; j<R; j++){
```

```
if (allocation[i][j] != 0){
    nonZero = true;
    break; }}
finish[i] = !nonZero; }
while (true) {
    bool found = false;
    for (int i=0; i<P; i++) {
        if (!finish[i] && isLessThanOrEqual(request[i], work,
        R)) {
            addVectors(work, allocation[i], R);
            finish[i] = true;
            sequence[index++] = i;
            found = true; } }
    if (!found){
        break; } }
bool deadlock = false;
for (int i=0; i<P; i++){
    if (!finish[i]){
        printf("System is not in deadlock\n Safe sequen
        for (int i=0; i<P; i++){
            printf("P%d", sequence[i]); }
            printf("\n"); } }

int main(){
    int P, R;
    printf("Enter the number of processes: ");
```

```
scanf("%d", &P);
printf("Enter the number of resource types:");
scanf("%d", &R);
int allocation[P][R], request[P][R], available[R];
printf("Enter the allocation Matrix\n");
for(int i=0; i<P; i++){
    printf("Process P%d:\n", i);
    for(int j=0; j<R; j++){
        printf("Resource type: %c:", 'A'+j);
        scanf("%d", &allocation[i][j]); }}
printf("Enter the Request matrix:\n");
for(int i=0; i<P; i++){
    printf("Process %c:", 'A'+j);
    scanf("%d", &request[i][j]); }}
printf("Enter the available Resources\n");
for(int i=0; i<R; i++){
    printf("Resource type: %c: ", 'A'+i);
    scanf("%d", &available[i]); }
deadlockDetection(allocation, request, available, P, R);
return 0;
}
```

Date: YOUVA

Enter the number of processes: 5
Enter the number of resource types: 3
Enter the Allocation Matrix:
Process P0:
Resource A: 0
Resource B: 1
Resource C: 0
Process P1:
Resource A: 2
Resource B: 0
Resource C: 0
Process P2:
Resource A: 3
Resource B: 0
Resource C: 3
Process P3:
Resource A: 2
Resource B: 1
Resource C: 1
Process P4:
Resource A: 0
Resource B: 0
Resource C: 2
Enter the Request Matrix
Process P0:
Resource A: 0

Resource B: 0
Resource C: 0
Process P_1:
Resource A: 2
Resource B: 0
Resource C: 2
Process P_2:
Resource A: 0
Resource B: 0
Resource C: 0
Process P_3:
Resource A: 1
Resource B: 0
Resource C: 0
Process P_4:
Resource A: 0
Resource B: 0
Resource C: 2
Enter the Available Resources
Resource A: 0
Resource B: 0
Resource C: 0
System is not in deadlock.
Safe sequence: P_0 P_2 P_1 P_3 P_4

LAB09: Write a C program to simulate the following contiguous memory techniques
a)First Fit
b)Best Fit
c)Worst Fit

Input:

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

void firstFit(int blockSize[], int m, int fileSize[], int n) {
    int allocation[n];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

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

    printf("Memory Management Scheme - First Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t", i+1, fileSize[i]);
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\n", allocation[i]+1, blockSize[allocation[i]], blockSize[allocation[i]]);
        } else {
            printf("Not Allocated\n");
        }
    }
    printf("\n");
}

void bestFit(int blockSize[], int m, int fileSize[], int n) {
    int allocation[n];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

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

```
50
51          if (bestIdx != -1) {
52              allocation[i] = bestIdx;
53              blockSize[bestIdx] -= fileSize[i];
54          }
55      }
56
57      printf("Memory Management Scheme - Best Fit\n");
58      printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");
59      for (int i = 0; i < n; i++) {
60          printf("%d\t%d\t\t", i+1, fileSize[i]);
61          if (allocation[i] != -1) {
62              printf("%d\t\t%d\t\t%d\n", allocation[i]+1, blockSize[allocation[i]], blockSize[allocation[i]]);
63          } else {
64              printf("Not Allocated\n");
65          }
66      }
67      printf("\n");
68  }
69
70  void worstFit(int blockSize[], int m, int fileSize[], int n) {
71      int allocation[n];
72
73      for (int i = 0; i < n; i++) {
74          allocation[i] = -1;
75      }
76
77      for (int i = 0; i < n; i++) {
78          int worstIdx = -1;
79          for (int j = 0; j < m; j++) {
80              if (blockSize[j] >= fileSize[i]) {
81                  if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
82                      worstIdx = j;
83                  }
84              }
85          }
86
87          if (worstIdx != -1) {
88              allocation[i] = worstIdx;
89              blockSize[worstIdx] -= fileSize[i];
90          }
91      }
92
93      printf("Memory Management Scheme - Worst Fit\n");
94      printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");
95      for (int i = 0; i < n; i++) {
96          printf("%d\t%d\t\t", i+1, fileSize[i]);
97          if (allocation[i] != -1) {
98              printf("%d\t\t%d\t\t%d\n", allocation[i]+1, blockSize[allocation[i]], blockSize[allocation[i]]);
```

```
            printf("%d\t\t%d\t\t%d\n", allocation[i]+1, blockSize[allocation[i]], blockSize[allocation[i]]);
        } else {
            printf("Not Allocated\n");
        }
    }
    printf("\n");
}

int main() {
    int m, n;

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

    printf("Enter the number of files: ");
    scanf("%d", &n);
    int fileSize[n];
    printf("Enter the size of the files:\n");
    for (int i = 0; i < n; i++) {
        printf("File %d: ", i+1);
        scanf("%d", &fileSize[i]);
    }

    int blockSizeCopy[m];
    for (int i = 0; i < m; i++) {
        blockSizeCopy[i] = blockSize[i];
    }

    firstFit(blockSizeCopy, m, fileSize, n);

    for (int i = 0; i < m; i++) {
        blockSizeCopy[i] = blockSize[i];
    }

    bestFit(blockSizeCopy, m, fileSize, n);

    for (int i = 0; i < m; i++) {
        blockSizeCopy[i] = blockSize[i];
    }

    worstFit(blockSizeCopy, m, fileSize, n);
    return 0;
}
```

Output:

```
Enter the number of blocks: 5
Enter the size of the blocks:
Block 1: 400
Block 2: 700
Block 3: 200
Block 4: 300
Block 5: 600
Enter the number of files: 4
Enter the size of the files:
File 1: 212
File 2: 517
File 3: 312
File 4: 526
Memory Management Scheme - First Fit
File_no File_size       Block_no        Block_size      Fragment
1       212             1               188             188
2       517             2               183             183
3       312             5               288             288
4       526             Not Allocated

Memory Management Scheme - Best Fit
File_no File_size       Block_no        Block_size      Fragment
1       212             4               88              88
2       517             5               83              83
3       312             1               88              88
4       526             2               174             174

Memory Management Scheme - Worst Fit
File_no File_size       Block_no        Block_size      Fragment
1       212             2               176             176
2       517             5               83              83
3       312             2               176             176
4       526             Not Allocated
```

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

```
#Include <stdio.h>
#Include <limits.h>
void firstFit(int blockSize[], int m, int fileSize[], int n) {
int allocation[n];
for(int i=0; i<n; i++){
    allocation[i] = -1; }
for(int i=0 ; i<n ; i++){
  for(int j=0; j<m; j++){
    if(blockSize[j] >= fileSize[i]){
      allocation[i] = j;
      blockSize[j] -= fileSize[i];
      break; } }
printf("\n Memory Management Scheme - First Fit\n");
printf("\n File - no\t File Size \t Block - no\t Block Size \t
Fragment \n");
for(int i=0 ; i<n ; i++){
    printf("%d \t %d \t %d \t %d \t %d \n", allocation[i]+1,
blockSize[allocation[i]], allocation[i]+1, blockSize[allocation[i]]);
else {
    printf("\n Not allocated \n"); } }
printf("\n"); }
```

```
void bestfit(int block_size[], int m, int file_size[], int n){
int allocation[n];
for(int i=0; i<n; i++){
    allocation[i]=-1;}
for(int i=0; i<n; i++){
    int bestIndex=-1;
    for(int j=0; j<m; j++){
    if(block_size[j]>=file_size[i]){
        if(bestIndex==-1||block_size[bestIndex]>block_size[j]){
            bestIndex=j;}}
    if(bestIndex!=-1){
        allocation[i]=bestIndex;
        block_size[bestIndex]-=file_size[i];}}
printf("Memory Manage ment scheme - best fit\n");
printf("File_no\tFile_size\tBlock_no\tBlock_size\t
Fragment\n");
for(int i=0; i<n; i++){
    printf("%d\t%d\t",i+1,file_size[i]);
    if(allocation[i]!=-1){
        printf("%d\t%d\t%d\n",allocation[i]+
1,block_size[allocation[i]],block_size[allocation[i]]);
    }else{
        printf("Not allocated\n");}
    printf("\n");
}
```

```
void worstFit(int blockSize[], int m, int fileSize[], int n){
int allocation[n];
for(int i=0; i<n; i++){
    allocation[i] = -1; }
for(int i=0; i<n; i++){
  int worstIndex = -1;
  for(int j=0; j<m; j++){
    if(blockSize[j] >= fileSize[i]){
      if(worstIndex == -1 || blockSize[worstIndex] < blockSize
        [j]){ worstIndex = j; }
    }
  }
  if(worstIndex != -1){
    allocation[i] = worstIndex;
    blockSize[worstIndex] -= fileSize[i];
  }
printf("Memory Management Scheme - Worst Fit\n
printf("File No\tFile Size\tBlock No\tBlock Size\t
Fragment\n");
for(int i=0; i<n; i++){
  printf("%d\t\t%d\t\t", i+1, fileSize[i]);
  if(allocation[i] != -1){
    printf("%d\t\t%d\t\t%d\n", allocation[i]+1,
blockSize[allocation[i]], blockSize[allocation[i]]);
  }else{
    printf("Not allocated\n");
  }
printf("\n");
}
```

```
int main(){
    int m, n;
    printf("Enter the number of blocks");
    scanf("%d", &m);
    int blockSize[m];
    printf("Enter the size of the blocks:\n");
    for(int i=0; i<m; i++){
        printf("Block %d:", i+1);
        scanf("%d", &blockSize[i]);}
    printf("Enter the number of files:");
    scanf("%d", &n);
    int fileSize[n];
    printf("Enter the size of the files:\n");
    for(int p=0; i<n; p++){
        printf("File %d:", i+1);
        scanf("%d", &fileSize[i]);}
    int blockSizecopy[m];
    for(int i=0; i<m; i++){
        blockSizecopy[i] = blockSize[i];}

    bestFit(blockSizecopy, m, fileSize, n);
    for(int i=0; i<m; i++){
        blockSizecopy[i] = blockSize[i];}
    bes+Fit(blockSizecopy, m, fileSize, n);
    for(int p=0; p<m; p++){
        blockSizecopy[i] = blockSize[i];}
```

worst Fit (block sz[ ], m, file sz[ ], n);
return no;)

OUTPUT

Enter no number of blocks: 5
Enter no size of the blocks:
Block1: 400
Block2: 700
Block3: 200
Block4: 300
Block5: 600
Enter no number of files: 4
Enter the size of the file
File 1: 212
File 2: 517
File 3: 312
File 4: 526

Memory Management Scheme - First Fit

File-no	File-Size	Block no	Block size	Fragment
1	212	1	188	400
2	517	2	183	700
3	312	5	288	600
4	526	Not Allocated		

Memory Management Scheme - First Fit

File no	File size	Block no	Fragment
1	212	1	188
2	517	2	183
3	312	5	288
4	526	Not allocated	

Memory Management scheme - Best Fit

File no	File size	Block no	Fragment
1	212	4	88
2	517	5	83
3	312	1	88
4	526	2	174

Memory Management scheme - Worst Fit

File no	File size	Block no	Fragment
1	212	2	176
2	512	5	88
3	312	1	176
4	526	Not allocated	

3/7/24