Q) write a program to implement First come First serve sheduling

```
#include <stdio.h>
int main ()
{
    int n;
    printf ("enter total no. of processes");
    scanf ("%d", &n);
    int arr_time[n];
    int burst_time[n];
    printf ("enter arrival time");
    for(int i=0; i<n; i++)
    {
        scanf ("%d", &arr_time[i]);
    }
    printf ("enter burst time");
    for (int i=0; i<n; i++)
    {
    scanf ("%d", &burst_time[i]);
    }
int temp1, temp2;
for (int i=0; i<n; i++)
    {
        for(int j= i+1; j<n; j++)
        {
            if (arr_time[i]>arr_time[j])
            {
                temp1 = arr_time[i];
                arr_time[i] = arr_time[j];
                arr_time[j] = temp1;
                temp2 = burst_time[i];
                burst_time[i] = burst_time[j];
                burst_time[j] = temp2;
            }
        }
```

```
int comp_time[n];
comp_time[0] = burst_time[0] + arr_time[0];
for(int i=1; i<n; i++)
    {
      if (arr_time[i] <= comp_time[i-1])
        {
          comp_time[i] = ~~burst_time[i] + arr_time~~
                         comp_time[i-1] + burst_time[
        }
      else
        {
          comp_time[i] = burst_time[i] + arr_time[i];
        }
    }
for( int i=0; i<n; i++)
    {
    printf("completion_time[%d] : %d\n", (i+1),
                                  comp_time[i]);
    }
   int Tat[n];
for(int i=0; i<n; i++)
    {
      Tat[i] = comp_time[i] - arr_time[i];
    }
   float sum_tat = 0;
   float avg_tat =
   for (int i=0; i<n; i++)
     {
       sum_tat += Tat[i];
     }
   avg_tat = sum_tat / n;
   printf("average Tat = %f\n", avg_tat);
```

```
int wt[n];
for(int i=0;i<n;i++)
  {
    wt[i]= (Tat[i] - burst_time[i]);
  }
  float sum_wt=0;
  float avg_wt;
for(int i=0; i<n;i++)
  {
    sum_wt +=wt[i];
  }
  avg_wt = sum_wt / n;
  printf("average waiting time=%f\n", avg_wt);
  }
```

Output:

```
enter total no. of processes: 4
enter arrival times:0
6
1
5
enter burst times:2
4
2
3
```

completion_time [1] = 2
completion_time [2] = 4
completion_time [3] = 8
completion_time [4] = 12

average TAT = 3.5000
average waiting time = 0.7500

8/5/24

SJF

```
#include <stdio.h>
typedef struct {
    int pid;
    int arr;
    int burst;
    int comp;
    int turn;
    int wait;
    int processed;
} proc;
void sjf_np(proc procs[], int n) {
    int curr_time = 0;
    int total_comp_time = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;
    while(1) {
        int shortest_job = -1;
        int shortest_burst = 9999;
        for (int i=0; i<n; i++) {
            if (procs[i].arr <= curr_time && procs[i].proce
                - ssed == 0)
            {
                if (procs[i].burst < shortest_burst) {
                    shortest_burst = procs[i].burst;
                    shortest_job = i;
                }
            }
        }
    }
```

```
if (shortest_job == -1)
{ break;
}
procs[shortest_job].comp = curr_time + procs[shortest_
                           job].burst;
procs[shortest_job].turn = procs[shortest_job].comp
                           - procs[shortest_job].arr;
procs[shortest_job].wait = procs[shortest_job].turn
                           - procs[shortest_job].burst;
if (procs[shortest_job].wait < 0)
    procs[shortest_job].wait = 0;
}
double avg_turnaround_time = (double)
                             total_turnaround_time/n;
double avg_waiting_time = (double) total_wait_time/n;
printf ("process ID \t Arrival Time \t Burst Time \t
    completion Time \n 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 \t \t
        %d \n",
    procs[i].pid, procs[i].arr, procs[i].burst,
    procs[i].comp, procs[i].wait, procs[i].turn);
}
printf("\n Average Turnaround Time: %.2f \n",
    avg_turnaround_time);
printf("Average waiting Time: %.2f \n", avg_turnar-
    _ound_time);
printf("Average waiting Time, %.2f \n",
                    avg_waiting_time);
}
```

```
int main() {
    int n;
    proc procs[MAX_PROCESS];
    printf("Enter the number of processes:");
    scanf("%d", &n);
    for(int i=0; i<n; i++) {
        procs[i].pid = i+1;
        printf("Enter arrival time for process %d", i+1);
        scanf("%d", &procs[i].arr);
        printf("Enter burst time for %d", i+1);
        scanf("%d", &procs[i].burst);
        procs[i].processed = 0;
    }
    sjf_np(procs, n);
    return 0;
}
```

Output:

processID	Arrival time	Burst time	comp time	TAT	turn around time
1	2	1	7	4	5
2	1	5	16	10	15
3	4	2	8	3	4
4	0	6	6	0	6
5	2	3	11	6	9

Round Robin

```
#include<stdio.h>
#include<stdlib.h>
Struct Queue {
    int pid;
    Struct queue* next;
};
Struct queue* rq=NULL;
Struct queue* create(int p) {
    Struct queue* nn = malloc(Sizeof(Struct queue));
    nn->pid = p;
    nn->next=NULL;
    return nn;
}

void enqueue(int p) {
    Struct queue *nn = create(p);
    if (rq==NULL)
        rq=nn;
    else {
        struct queue* temp=rq;
        while(temp!=NULL)
            temp= temp->next;
        temp->next=nn;
    }
}

int dqueue() {
    int x=0;
    if(rq==NULL)
        return x;
    else {
        Struct queue* temp=rq;
        x= temp->pid;
        rq=rq->next;
        free(temp);
```

```
    return x;
}

void printq() {
    struct queue *temp = rq;
    while(temp != NULL){
        printf("%d\t", temp->pid);
        temp = temp->next;
    }
    printf("\n");
}

void swap(int *a, int *b) {
    *a = *a + *b;
    *b = *a - *b;
    *a = *a - *b;
}

void sort(int *pid, int *at, int *bt, int n) {
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            if(at[i] < at[j]) {
                swap(&at[i], &at[j]);
                swap(&bt[i], &bt[j]);
                swap(&pid[i], &pid[j]);
            }
        }
    }
}

int main() {
    int n, t, x=1;
    printf("Enter the number of processes:");
    scanf("%d", &n);
    printf("Enter the time quantum");
    scanf("%d", &t);
```

```
int pid[n], at[n], bt1[n], ct[n], tat[n],
    wt[n], bt2[n], rt[n];
for(int i=0; i<n; i++){
    printf("Enter arrival time and burst time");
    scanf("%d%d", &at[i], &bt1[i]);
    pid[i] = i+1;
}
sort(pid, at, bt1, n);
enqueue(pid[0]);
for(int i=0; i<n; i++){
    bt2[i] = bt1[i];
    rt[i] = -1;
}
int count=0;
int ctvar = at[0];
while(count != n){
    int curp = rq->pid;
    int curi = 0;
    for(int i=0; i<n; i++){
        if(pid[i] == curp){
            curi = i;
            break;
        }
    }
    if(rt[curi] == -1){
        rt[curi] = ctvar - at[curi];
    }
    if(bt2[curi] <= t){
        ctvar += ctvar - at[curi];
    }
    if(bt2[curi] <= t){
        ctvar += bt2[curi];
        bt2[curi] = 0;
    }
```

```
else {
    ctvar += t;
    bt[curi] -= t;
}
while(at[x] <= ctvar && x < n) {
    equeue(pid[x]);
    x += 1;
}
if (bt2[curi] > 0)
    equeue(pid[curi]);
if (bt2[curi] == 0) {
    count += 1;
    ct[curi] = ctvar;
}
dequeue();
}
for(int i=0; i<n; i++) {
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt1[i];
}
float avg_tat = 0;
float avg_wt = 0;
for(int i=0; i<n; i++) {
    avg_tat += tat[i];
    avg_wt += wt[i];
}
printf("pid\tat\tbt\tct\ttat\twt\n");
for(int i=0; i<n; i++) {
printf("%d\t%d\t%d\t%d\t%d\t%d\n",
    pid[i], at[i], bt1[i], ct[i], tat[i], wt[i]);
}
printf("\n");
}
printf("\nAverage Turnaround time: %f", avg_tat/n);
printf("\nAverage waiting time: %f", avg_wt/n);
return 0;
```

Output

Enter the number of processes: 5
Enter the time quantum: 2
Enter arrival time and burst time: 0
5
Enter arrival time and burst time: 1
3
Enter arrival time and burst time: 2
1
Enter arrival time and burst time: 3
2
Enter arrival time and burst time:

pid	at	bt	ct	tat	wt	rt
1	0	5	13	13	8	0
2	1	3	12	11	8	1
3	2	1	5	3	2	2
4	3	2	9	6	4	4
5	4	3	14	10	7	5

Average turn around time: 8.6000
Average waiting time: 5.8000

Rm
6/6/24

Priority scheduling

```
#include<stdio.h>
#include<stdlib.h>
void swap(int *a,int *b){
    *a=*a+*b;
    *b=*a-*b;
    *a=*a-*b;
}
void sort(int *pid,int *at,int *bt,int *prior,int n){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if(at[i]<at[j]){
                swap(&at[i],&at[j]);
                swap(&bt[i],&bt[j]);
                swap(&pid[i],&pid[j]);
                swap(&prior[i],&prior[j]);
            }
        }
    }
}
int highest_priority(int *prior,int s,int e){
    int x=prior[s];
    int j=s;
    for(int i=s;i<e;i++){
        if(prior[i]>x){
            x=prior[i];
            j=i;
        }
    }
    return j;
}
```

```
int main(){
    int n,t,x;
  printf("enter the number of processes");
  scanf("%d",&n);
int pid[n], at[n], bt1[n], ct[n], tat[n], wt[n],
      bt2[n], rt[n], prior[n];
  for(int i=0; i<n; i++){
    printf("Enter arrival time, burst time and
            priority i");
    scanf("%d%d%d", &at[i], &bt[i], &prior[i]);
       pid[i]=i+1;
    }
 sort(pid, at, bt1, prio1, n);
   for(int i=0; i<n; i++){
       bt2[i]=bt1[i];
        rt[i]=-1;
   }
   int arvc=0;
   int count=0;
  int ctvar=at[0];
   int cur=0;
   while(count!=n){
     if(rt[cur]==-1){
         rt[cur]=ctvar:at[cur];
      }
    if(arvc==n){
        ctvar+=bt2[cur];
        bt2[cur]=0;
     }
    else{
      ctvar+=1;
     bt2[cur]-=1;
   }
```

```
for(int i=0; at[i]<=ctval; i++){
    arrvd t=1;
    x=i;
}
if(bt2[cur i]==0){
    count++;
    ct[cur i]=ctval;
    prior[cur i]=-1;
}
cur i = highest-priority(prior,0,x+1);
}
for(int i=0; i<n; i++){
    tat[i]=ct[i]-at[i];
    wt[i]=tat[i]-bt1[i];
}
float avg_tat=0;
float avg_wt=0;
for(int i=0; i<n; i++){
    avg_tat+=tat[i];
    avg_wt+=wt[i];
}
printf("pid\tat\tbt\tct\ttat\twt\t\n\n");
for(int i=0; i<n; i++){
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d
        pid[i],at[i],bt[i],ct[i],tat[i],wt[i],rt[
    printf("\n");
}
printf("\n Average turn around time: %f",avg_tat/n
printf("\n Average waiting time: %f",avg_wt/n);
return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter arrival time, burst time and priority: 0
5
10
Enter arrival time, burst time and priority: 1
4
20
Enter arrival time, burst time and priority: 2
2
30
Enter arrival time, burst time and priority: 4
1
40
```

pid	at	bt	ct	tat	wt	rt
1	0	5	12	12	7	0
2	1	4	8	7	3	0
3	2	2	4	2	0	0
4	4	1	5	1	0	0

Average turn around time: 5.500000

Average waiting time: 2.500000

Rev 6/12/24

Multilevel queue scheduling

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

struct process{
    char name[5];
    int AT;
    int BT;
    int CT;
    int TAT;
    int WT;
    int isSystem;
};

void FCFS( struct process p[], int, int *current_time)
{
    for(int i=0; i<n; i++) {
        if (*current_time < p[i].AT)
            *current_time = p[i].AT;
        p[i].CT = *current_time + p[i].BT;
        p[i].TAT = p[i].CT - p[i].AT;
        p[i].WT = p[i].TAT - p[i].BT;
        *current_time = p[i].CT;
    }
}

int main()
{
    int n, sys_count=0, user_count=0;
    printf("Enter the number of processes:\n");
    scanf("%d", &n);
    struct process p[n], system_p[n], user_p[n];
    printf("Enter the arrival time, burst time and
        type (1 for system process, 0 for
        user process) for all the processes:\n");
```

```
for(int i=0; i<n; i++)
{
  printf("\n\n process %d :\n", i+1);
  sprintf(p[i].name, "p%d", i+1);
  printf("Arrival time:");
  scanf("%d", &p[i].AT);
  printf("Burst Time:");
  scanf("%d", &p[i].BT);
  printf("Type (1 for system, 0 for user):");
  scanf("%d", &p[i].isSystem);

  if (p[i].isSystem)
  {
    system_p[sys_count++] = p[i];
  }
  else
  {
    user_p[user_count++] = p[i];
  }
}
for(int i=0; i<sys_count-1; i++)
{ for(int j=0; j<sys_count-i-1; j++)
  { if (system_p[j].AT > system_p[j+1].AT)
    { struct process temp = system_p[j];
      system_p[j] = system_p[j+1];
      system_p[j+1] = temp;
    }
  }

int current_time=0;
int total_wt=0, total_tat=0;
printf("\n\n process\t Name\t Arrival time\t
Burst time\t completion time\t turn
around time\t waiting time\t Type\n");
int i=0, j=0;
```

```
while(i<sys_count || j<user_count){
    if(i<sys_count && (j>=user_count ||
        system_p[i].AT <= current_time)){
        if(current_time < system_p[i].AT)
            current_time = system_p[i].AT;
        system_p[i].CT = current_time + system_p[i].BT;
        system_p[i].TAT = system_p[i].CT -
                          system_p[i].AT;
        system_p[i].WT = system_p[i].TAT -
                         system_p[i].BT;
        current_time = system_p[i].CT;
printf("%d\t%s\t%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\tsystem\n", i+1, system_p[i].name,
    system_p[i].AT, system_p[i].BT, system_p[i].BT,
    system_p[i].CT, system_p[i].TAT, system_p[i].
    total_wt += system_p[i].WT;
    total_tat += system_p[i].TAT;
    i++;
}
else if(j<user_count){
    if(current_time < user_p[j].AT)
        current_time = user_p[j].AT;
user_p[j].CT = current_time + user_p[j].BT;
user_p[j].TAT = user_p[j].CT - user_p[j].AT;
user_p[j].WT = user_p[j].TAT - user_p[j].BT;
current_time = user_p[j].CT;
printf("%d\t%s\t%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\tuser\n", i+1,
    user_p[j].name, user_p[j].AT,
    user_p[j].BT, user_p[j].CT, user_p[j].TAT,
```

```
usr_pr[j].wt);
    total_wt += usr_pr[j].wT;
    total_tat += usr_pr[j].TAT;
    j++;
}

float avg_wt = (float) total_wt/n;
float avg_tat = (float) total_tat/n;
printf("\n Average waiting time =%.0.3f", avg_wt);
printf("\n Average turn around time = %.0.3f\n",
    avg_tat);
return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter the arrival time, burst time
and type (1 for system process, 0 for user
          process) for all the processes:

process 1:
Arrival time: 0
Burst Time: 3
Type (1 for system, 0 for user): 1

process 2:
Arrival time: 2
Burst time: 6
Type (1 for system, 0 for user): 0

process 3:
Arrival time: 4
Burst time: 4
Type (1 for system, 0, for user): 1
```

process :4
Arrival time: 6
Burst time: 5
Type (1 for system, 0 for user): 0

process 5:
Arrival time: 8
Burst time: 2
Type (1 for system, 0 for user): 1

process	Name	AT	BT	CT	TAT	Type
1	P1	0	3	3	0	system
2	P2	2	9	7	1	user
3	P3	4	13	9	5	system
4	P4	6	18	12	7	user
5	P5	8	15	12	10	system

Average waiting Time = 4.600

Average turn Around time = 8.600

6/6/22

Rate Monotonic program

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_TASKS 10
typedef struct {
  int Ti;
  int Ci;
  int deadline;
  int RT;
  int id;
}Task;
void Input (Task tasks[], int *n_tasks)
{
  printf("Enter number of tasks:");
  scanf("%d", &n_tasks);
  for(int i=0; i<*n_tasks;");
    scan
  for(int i=0; i<*n_tasks; i++)
  {
    task[i].id = i+1;
    printf("Enter Ti of task %d", i+1);
    scanf("%d", &tasks[i].Ti);
    printf("Enter execution time of task
                %d", i+1);
    scanf("%d", &tasks[i].Ci);
    tasks[i].deadline = tasks[i].Ti;
    tasks[i].RT = tasks[i].Ci;
  }
```

```
int compare_by_period(const void *a, const void *b)
{
    return ((Task*)a)->Ti - ((Task*)b)->Ti;
}

void RMS(Task tasks[], int n_tasks, int time_frame)
{
    qsort(tasks, n_tasks, sizeof(Task), compare_by_period);
    printf("\n Rate-monotonic Scheduling:\n");
    for(int time=0; time<time_frame; time++)
    {
        int s_task=-1;
        for(int i=0; i<n_tasks; i++)
        {
            if(time%tasks[i].Ti==0)
            {
                tasks[i].RT = tasks[i].Ci;
            }
            if(tasks[i].RT>0 && (s_task==-1 ||
            tasks[i].Ti < tasks[s_task].Ti))
            {
                s_task = i;
            }
        }
        if(s_task!=-1)
        {
            printf("Time %d: Task %d\n", time,
                tasks[s_task].id);
            tasks[s_task].RT--;
        }
        else{
            printf("Time %d: Idle\n", time);
        }
    }
}
```

```
int main()
{
    Task tasks[MAX_TASKS];
    int n_tasks;
    int time_frame;
    Input(tasks, &n_tasks);
    printf("Enter time frame for simulation: ");
    scanf("%d", &time_frame);
    RMS(tasks, n_tasks, time_frame);
    return 0;
}
```

Output:

```
Enter number of tasks: 3
Enter Ti of task 1: 20
Enter execution time of task 1: 3
Enter Ti of task 2: 5
Enter execution time of task 2: 2
Enter Ti of task 3: 10
Enter execution time of task 3: 2
Enter time frame for simulation: 20
Rate-monotonic scheduling:
time 0: TASK 2
time 1: TASK 2
time 2: TASK 3
time 3: TASK 3
time 4: TASK 1
time 5: TASK 2
```

Time 6 : Task 2
Time 7 : Task 1
Time 8 : Task 1
Time 9 : Idle
Time 10 : Task 2
Time 11 : Task 2
Time 12 : Task 3
Time 13 : Task 3
Time 14 : ~~Task 3~~ Idle
Time 15 : ~~Idle~~ Task 2
Time 16 : Task 2
~~Time 17 : Task 2~~
Time 17 : Idle
Time 18 : Idle
Time 19 : Idle

Earliest Deadline First

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_TASKS 16

typedef struct {
    int Ti;
    int Ci;
    int deadline;
    int RT;
    int n_deadline;
    int id;
} Task;

void Input(Task tasks[], int *n_tasks)
{
    printf("Enter number of tasks");
    scanf("%d", n_tasks);

    for(int i=0; i<*n_tasks; i++)
    {
        tasks[i].id = i+1;
        printf("Enter Ti of task %d:", i+1);
        scanf("%d", &tasks[i].Ti);
        printf("Enter execution time of
                task %d:", i+1);
        scanf("%d", &tasks[i].Ci);
        printf("Enter deadline of task %d",
                i+1);
        scanf("%d", &tasks[i].deadline);
        tasks[i].RT = tasks[i].Ci;
        tasks[i].n_deadline = tasks[i].deadline;
    }
}
```

```
void EDF( Task tasks[], int n_tasks, int time_frame)
{
    printf("\n Earliest-Deadline First
                scheduling :\n");
    for ( int time=0; time< time_frame;
                                time++)
    {
        int s_task = -1;
    for (int i=0; i< n_tasks; i++)
        {
            if (time % tasks[i].Ti == 0)
            {
                tasks[i].RT = tasks[i].Ci;
                tasks[i].n_deadline =
                        time + tasks[i].dead;
            }
        }
    for (int i=0; i<n_tasks; i++)
        {
            if(tasks[i].RT > 0 && (s_task == -1 ||
              tasks[i].n_deadline < tasks[s_task].
                                    n_deadline)
            {
                s_task = i;
            }
        }
        if (s_task != -1)
        {
            printf("Time %d: Task %d\n", time,
                                tasks[s_task].id);
            tasks[s_task].RT --;
        }
```

```
        else {
            printf("Time %d: Idle\n", time);
        }
    }
}

int main()
{
    Task tasks[MAX_TASKS];
    int n_tasks;
    int time_frame;
    Input(tasks, &n_tasks);
    printf("Enter time frame for simulation: ");
    scanf("%d", &time_frame);
    EDF(tasks, n_tasks, time_frame);

    return 0;
}
```

Output:

```
Enter number of tasks: 3
Enter Ti of task 1: 4
Enter execution time of task 1: 1
Enter deadline of task 1: 4
Enter Ti of task 2: 5
Enter execution time of task 2: 2
Enter deadline of task 2: 5
Enter Ti of task 3: 6
Enter execution time of task 3: 3
Enter deadline of task 3: 6
Enter time frame for simulation: 12
```

Earliest- Deadline First scheduling

Time 0 : Task 1
Time 1 : Task 2
Time 2 : Task 2
Time 3 : Idle
Time 4 : Task 1
Time 5 : Task 3
Time 6 : Task 3
Time 7 : Task 3
Time 8 : Task 1
Time 9 : Task 2
Time 10 : Task 2
Time 11 : Idle

6/6

proportional program

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

typedef struct{
    char name[5];
    int tickets;
}process;

int main()
{
    int n, total_tickets=0;
    float total_T=0.0;
printf("Enter the number of processes:");
scanf("%d",&n);

process p[n];

srand(time(NULL));

for(int i=0; i<n; i++)
{
printf("\n process %d:\n", i+1);
sprintf(p[i].name, "p%d", i+1);
printf("Tickets:");
scanf("%d", &p[i].tickets);
total_tickets += p[i].tickets;
total_T += p[i].tickets;
}

printf("\n -- proportional share scheduling --\n");
printf("Enter the time period for scheduling:");

int m;
scanf("%d",&m);
```

```
for( int i=0; i<n; i++)
{
 int winning_ticket = rand() % total_tickets + 1;
 int accumulated_tickets = 0;
 int winner_index;
for(int j=0; j<n; j++)
{accumulated_tickets += p[j].tickets;
if (winning_ticket <= accumulated_tickets)
    {
      winner_index = j;
       break;
    }
  }
printf ("Tickets picked : %d, winner : %s\n",
                winning_ticket, p[winner_index]
                                        .name);
  }
for(int i=0; i<n; i++)
   {printf("In the process : %s gets %.0.2f%%
       of processer Time,\n", p[i].name,
       ((p[i].tickets / total_T) * 100));
    }
  return 0;
}
```

Output:

Enter the number of processes : 3

process 1:

Tickets : 10

process 2:

Tickets : 20

process 3:

Tickets : 30

proportional share scheduling

Enter the Time period for scheduling: 5

Tickets picked : 5, winner : P1

Tickets picked : 25, winner : P2

Tickets picked : 35, winner : P3

Tickets picked : 50, winner : P3

Tickets picked : 15, winner : P2

The process : P1 gets 16.67 % of processor Time.

The process : P2 gets 33.33 % of processor Time.

The process : P3 gets 50.00 % of processor Time

9/12/16

Semaphore

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unist.h>
#define MAX_ITEMS 20
#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
  int head = 0;
  int tail = 0;
  int cnt = 0;
  pthread_mutex_t mtx;
  pthread_cond_t cond_full;
  pthread_cond_t cond_empty;
void enqueue(int item) {
        buffer[tail] = item;
        tail = (tail+1) % BUFFER_SIZE;
        cnt --;
    return item;
}
void dequeue() {
      int item = buffer[head];
      head = (head+1) % BUFFER_SIZE;
        cnt--;
        return item;
}
```

```
void * producer (void * param) {
    int prod_cnt = 0;
  while(1) {
    int item = rand() % 100;
    pthread_mutex_lock (&mtx);
    while( cnt == BUFFER_SIZE) {
      pthread_cond_wait (&cond_empty, &mtx);
    }
  if (prod_cnt >= MAX_ITMS) {
    pthread_mutex_unlock(&mtx);
    pthread_cond_signal (&cond_full);
    break;
  }
  enqueue (item);
    prodcnt++;
  printf ("produced : %d\n", item);
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_full);
    sleep (rand() % 2);
  }
  return NULL;
}

void * consumer(void* param) {
    int cons_cnt = 0;
  while(1) {
    pthread_mutex_lock(&mtx);
      while (cnt == 0) {
        pthread_cond_wait(&cond_full, &mtx);
      }
```

```
if (const-cnt >= MAX-ITEMS) {
    pthread_mutex_unlock(&mtx);
    pthread_cond_signal(&cond_empty);
    break;
  }

int item = dequeue();
  const-cnt++;
  printf("consumed : %d\n", item);
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_empty);
    sleep(rand()%2);
  }
 return NULL;
}

int main() {
  pthread_t tid_prod, tid_cons;
  pthread_mutex_init(&mtx, NULL);
  pthread_cond_init(&cond_full, NULL);
   pthread_cond_init(&cond_empty, NULL);
  pthread_create(&tid_prod, NULL, producer,
                                  NULL);
 pthread_create(&tid_cons, NULL, consumer, NULL)
   pthread_join(tid_prod, NULL);
   pthread_join(tid_cons, NULL);
 printf("production & consumption\n");
  pthread_mutex_destroy(&mtx);
   pthread_cond_destroy(&cond_full);
  pthread_cond_destroy(&cond_empty);
  return 0;
}
```

output:

produced : 41
consumed : 41
produced : 34
consumed : 34
produced : 69
produced : 78
produced : 62
produced : 5
produced : 81
consumed : 69
consumed : 81
consumed : 61
produced : 95
consumed : 95
produced : 27
consumed : 27
produced : 91
consumed : 91
produced : 2
produced 92
consumed : 2
produced : 21
produced : 18
produced : 47
produced : 71
produced : 69
produced : 67
consumed : 92

R= 27

Dining philosopher program

```
#include<pthread.h>
#include<semaphore.h>
#include<stdio.h>   #include<stdlib.h>
#include<unistd.h>
#define NUM_PHILOSOPHERS 5
sem_t forks[NUM_PHILOSOPHERS];
sem_t mutex;
void* philosopher_one(void* num){
    int id = *(int*)num;
    printf("p %d is waiting\n", id);
    printf("p %d is waiting", id);
    sem_wait(&mutex);
    printf("p %d is granted to eat\n", id);
    sem_wait(&forks[id]);
    sem_wait(&forks[(id+1)%5]);
    sem_post(&mutex);
    sleep(1);
    sem_post(&forks[id]);
    sem_post(&forks[(id+1)%5]);
    return NULL;
}
void* philosopher_two(void* num){
    int id = *(int*)num;
    printf("p %d is waiting\n", id);
    sem_wait(&forks[id]);
    sem_wait(&forks[(id+1)%5]);
```

```
printf("P.y.d is granted to eac+\n", id);
    sleep(1);
sem_post(&forks[id]);
sem_post(&forks[(id+1) y.5]);
return NULL;
}

int main(){
    int num_philosophers, num_hungry;
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers:");
    scanf("%d", &num_philosophers);
    printf("How many are hungry");
    scanf("%d", &num_hungry);
    int hungry_positions[num_hungry];
    for(int i=0; i<num_hungry; i++){
        printf("Enter philosopher %d position:", i+1);
        scanf("%d", &hungry_positions[i]);
        hungry_positions[i]--;
    }
forks = (sem_t*) malloc(num_philosophers *
                        sizeof(sem_t));
pthread_t philosophers[num_hungry];
    int ids[num_hungry];
for(int i=0; i<num_philosophers; i++){
        sem_init(&forks[i], 0, 1);
}
sem_init(&mutex, 0, 1);
```

```
int choice;
printf("\n1. one can eat at a time\t
        2. two can eat at a time\t
        3. Exit \n");
printf("Enter your choice");
scanf("%d", &choice);

if (choice == 1) {
printf("\n Allow one philosopher to eat at
                any time\n");
for(int i=0; i<num_hungry; i++){
  ids[i] = hungry_positions[i];
  pthread_create(&philosophers[i], NULL,
                 philosopher_one, &ids[i]);
  }
}
else if (choice==2) {
printf("\n Allow two philosophers to eat
          at the same time\n");
for (int i=0; i<num_hungry; i++) {
    ids[i] = hungry_positions[i];
    pthread_create(&philosophers[i], NULL,
                   philosopher_two, &ids[i]);
  }
}
else {
  printf("Exiting program.\n");
    free(forks);
    return 0;
}
for(int i=0; i<num_hungry; i++) {
   pthread_join(philosophers[i], NULL);
}
```

```
for(int i=0; i<num_philosophers; i++) {
        sem_destroy(&forks[i]);
    }
    sem_destroy(&mutex);
    free(forks);
    return 0;
}
```

output:

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 5
Enter philosopher 1 position: 1
Enter philosopher 2 position: 2
Enter philosopher 3 position: 3
Enter philosopher 4 position: 4
Enter philosopher 5 position: 5
1. one can eat a time    2. Two can eat at a time
3. exit
Enter your choice: 1
Allow one philosopher to eat at any time
P0 is waiting
P0 is granted to eat
P3 is waiting
P3 is granted to eat
P2 is waiting
P2 is granted to eat
P4 is waiting
P1 is waiting
P4 is granted to eat
P1 is granted to eat
```

Banker's Algorithm

```
#include<stdio.h>
#include<stdbool.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

void calculateNeed(int need[MAX_PROCESSES][MAX_
                   RESOURCES], int max[MAX_PROCESSES
                   [MAX_RESOURCES],
     int allot[MAX_PROCESSES][MAX_RESOURCES]
          int np, int nr) {

  for(int i=0; i<np; i++)
     for(int j=0; j<nr; j++)
        need[i][j] = max[i][j] - allot[i][j];
  }

bool isSafe(int processes[], int avail[], int max[]
     [MAX_RESOURCES], int allot[][MAX_RESOURCES]
      int np, int nr) {

     int need[MAX_PROCESSES][MAX_RESOURCES];

     calculateNeed(need, max, allot, np, nr);

     bool finish[MAX_PROCESSES] = {0};

     int safeSeq[MAX_PROCESSES];

      int work[MAX_RESOURCES];

     for (int i=0; i<nr; i++)
        work[i] = avail[i];

     int count=0;
```

```
while(count<np){
    bool found = false;
    for(int p=0; p<np; p++){
    if (finish[p][;] >work[i])
        break;
  if (j==nr){
    for(int k=0; k<nr; k++)
        work[k] += allot[p][k];
    safeseq[count++] = p;
    finish = 1;
    found = true;
  }
}
if (found == false){
    printf("system is not in safe state\n");
    return false;
}
printf("system is in safe state\n. safe
        sequence is:");
  for(int i=0; i<np; i++)
    printf("%d", safeseq[i]);
    printf("\n");
      return true;
}
```

```
int main(){
 int np, nr;
 printf("Enter number of processes");
  scanf("%d",&np);
 printf("Enter number of resource types:");
    scanf("%d",&nr);
 int processes[MAX_PROCESSES];
 for(int i=0; i<nr; i++)
     scanf("%d",&avail[i]);
 int max[MAX_PROCESSES][MAX_RESOURCES];
      processes[i] = i;
   int avail[MAX_RESOURCES];
   printf("Enter available resources:");
   for(int i=0; i<nr; i++)
     scanf("%d",&avail[i]);
   int max[MAX_PROCESSES][MAX_RESOURCES]
   printf("Enter maximum resource
                  matrix:\n");
 for(int i=0; i<np; i++){
    printf("process %d:", i);
   for(int j=0; j<nr; j++)
       scanf("%d",&max[i][j]);
 }
int alloc[MAX_PROCESSES][MAX_RESOURCES];
 printf("Enter allocation resource matrix:\n");
 for(int i=0; i<np; i++){
       printf("process %d:", i);
     for(int j=0; j<nr; j++)
         scanf("%d",&alloc[i][j]);
 }
```

issafe(processes, avail, max, allot, np, nr);

return 0;

}

```
Enter the number of process : 5
Enter number of resource : 3
Enter available resource matrix : 3 3 2
Enter maximum resource matrix
process 0 : 7 5 3
process 1 : 3 2 2
process 2 : 9 0 2
process 3 : 2 2 2
process 4 : 4 3 3
Enter allocation resource matrix
process 0 : 0 1 0
process 1 : 2 0 0
process 2 : 3 0 2
process 3 : 2 1 1
process 4 : 0 0 2
System is in safe state.
Safe sequence is : 1 3 4 0 2
```

Deadlock Detection

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

#define NUM_PROCESSES 5
#define NUM_RESOURCES 3

int available[NUM_RESOURCES];
int allocation[NUM_PROCESSES][NUM_RESOURCES];
int request[NUM_PROCESSES][NUM_RESOURCES];

bool deadlockDetection(int *safeSequence) {
    int work[NUM_RESOURCES];
    bool finish[NUM_PROCESSES] = {false};

    for(int i=0; i<NUM_RESOURCES; i++) {
        work[i] = available[i];
    }

    int count=0;
    while(count<NUM_PROCESSES) {
        bool found = false;
        for(int i=0; i<NUM_PROCESSES; i++) {
            if(!finish[i]) {
                bool canProceed = true;
                for(int j=0; j<NUM_RESOURCES; j++) {
                    if(request[i][j]>work[j]) {
                        canProceed = false;
                        break;
                    }
                }
            }
        }
    }
```

```
if (canProceed) {
    for (int j=0; j<NUM_RESOURCES; j++) {
        work[j] += allocation[i][j];
    }
    safeSequence[count++] = i;
    finish[i] = true;
    found = true;
  }
}
if (!found) {
    break;
}

for (int i=0; i<NUM_PROCESSES; i++) {
    if (!finish[i]) {
        printf("Deadlock detected, process P%d
                is in deadlock.\n", i);
        return false;
    }
}
printf("No deadlock detected. The system is in
        a safe state.\n");
Printf("Safe sequence:");
for(int i=0; i<NUM_PROCESSES; i++) {
    printf("%P%d", safeSequence[i]);
}
printf("\n");
return true;
}
```

```
int main(){
  int i, j;
  printf("Enter the available resources
          vector: \n");
  for(i=0; i<NUM_RESOURCES; i++){
      scanf("%d", &available[i]);
  }
printf("Enter the allocation matrix:\n");
  for(i=0; i<NUM_PROCESSES; i++){
      for(j=0; j<NUM_RESOURCES; j++){
          scanf("%d", &allocation[i][j]);
      }
  }
printf("Enter the request matrix:\n");
  for(i=0; i<NUM_PROCESSES; i++){
      for(j=0; j<NUM_RESOURCES; j++){
          scanf("%d", &request[i][j]);
      }
  }
int safesequence[NUM_PROCESSES];
  deadlockDetection(safesequence);
    return 0;
}
```

output:

Enter the available Resources vector:

3 3 2

Enter the Allocation matrix:

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

Enter the request matrix:

7 4 3
1 2 2
6 0 0
0 1 1
0 0 2

No deadlock detected.

The system is in a safe state.

Safe sequence : P1 P3 P4 P0 P2

Rm 31/7/24

Memory Allocation

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_PARTITIONS 10
#define MAX_PROCESSES 10
typedef struct {
    int size;
    int isAllocated;
} partition;
typedef struct {
    int size;
    int isAllocated;
    int partitionIndex;
} process;
void allocate(process processes[], int processCount,
partition partitions[], int partitionCount,
int (*findIndex)(partition[], int, int)) {
int totalFragmentation=0;
printf("\n File NO. \t process size \t
        Block NO. \t Block size \t
        Fragments\n");
printf("-- ---- -- \n");
for(int i=0; i<processCount; i++) {
int index = findIndex(partitions, partiti
        -onCount, processes[i].size);
if (index != -1) {
processes[i].isAllocated = 1;
processes[i].partitionIndex = index;
partitions[index].isAllocated = 1;
```

```
int fragmentation = partitions[index].size -
                    processes[i].size;
totalFragmentation += fragmentation;

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i, processes[i].size, index,
       partitions[index].size, fragmentation);
}
else {
printf("%d\t\t%d\t\t-\t\t-\t\t-\n", i, processes[i].size);
}
}

printf("\nTotal Fragmentation: %d\n", totalFragmentation);
}

int firstFitIndex(Partition partitions[],
                  int partitionCount, int processSize) {
int bestIndex = -1;
for(int i=0; i<partitionCount; i++) {
if(!partitions[i].allocated && partitions[i].size
   >= processSize) {
return i;
}
}
return -1;
}
```

```
int bestFitIndex(partition partitions[],
                 int partitionCount, int processSize){
    int bestIndex = -1;
    for (int i=0; i<partitionCount; i++){
    if( !partitions[i].isAllocated && partitions[i].
        size >= processSize){
        if (bestIndex == -1 || partition[i].size <
            partitions[bestIndex].size){
            bestIndex = i;
        }
    }
    return bestIndex;
}

int worstFitIndex(partition partitions[],
                  int partitionCount, int processSize)
{
    int worstIndex = -1;
    for(int i=0; i<partitionCount; i++){
    if(!partitions[i].isAllocated && partitions[i].
        size >= processSize){
        if (worstIndex == -1 || partitions[i].size >
            partitions[worstIndex].size){
            worstIndex = i;
        }
    }
    return worstIndex;
}
```

```
void reset(Partition partitions[], int partitionCount,
           Process processes[], int processCount) {
    for(int i=0; i<partitionCount; i++)
        partitions[i].isAllocated=0;
    for(int i=0; i<processCount; i++){
        processes[i].isAllocated=0;
        processes[i].partitionIndex=-1;
    }
}

int main(){
int partitionCount, processCount, choice,
    fixedOrVariable, alreadyAllocated;
    Partition partitions[MAX_PARTITIONS];
    Process processes[MAX_PROCESSES];
    printf("Enter the number of partitions:");
    scanf("%d",&partitionCount);
    printf("Enter the size of each partition:\n");
    for(int i=0;i<partitionCount;i++){
        printf("partition %d",i);
        scanf("%d",&partitions[i].size);
        partitions[i].isAllocated=0;
    }
    printf("Enter the number of processes:");
    for(int i=0;i<processCount;i++){
        printf("process %d:",i);
        scanf("%d",&process[i].size);
        process[i].isAllocated=0;
        process[i].partitionIndex=-1;
    }
```

```
printf("choose memory Allocation strategy:\n1.
    First-fit\n2.Best-fit\n3.Worst-Fit\n");
    scanf("%d",&choice);

in (*allocationstrategy)(partition[], int, int)=
            NULL;
    switch(choice){
    case 1: allocationstrategy=firstfitIndex; break;
case 2: allocationstrategy = bestfitIndex; break;
case 3: allocationstrategy = worstfitIndex; break;

    default: printf("Invalid choice\n");
            return 1;
}
printf("\n memory management scheme:\n");
 printf("- - ——— _ _ —— \n");
printf("Fil.NO. \t process size \t Block NO.
            \t Block size \t fragment\n");
 printf("- --- - - ---- - \n");
allocate(processes, processcount, partitions
    partitioncount, allocationstrategy);
    return 0;
}
```

output:

Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of blocks
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of files
File 1: 1
File 2: 4

Memory management scheme - First Fit

File no:	File_size	Block_no:	Block_size	Fragment
1	1	1	5	4
2	4	3	7	3

memory management scheme - worst Fit

File no	File_size	BLOCK_no	Block_size	Fragments
1	1	3	7	6
2	4	1	5	1

memory management scheme - Best Fit

File no	File_size	BLOCK_no	Block_size	Fragmen
1	1	2	2	1
2	4	1	5	1

Ram
3/7/24

Page-replacement Algo

```
#include<stdio.h>
#include<stdbool.h>
void fifoPageReplacement(int pages, int
incomingstream[], int frames){
    int pagefaults =0;
    int temp[Frames];
    int front =0;
for(int m=0; m<frames; m++){
        temp[m] = -1;
    }
printf("\n---FIFO page Replacement
                Algorithm---\n");
printf("\n Incoming \t\t Frame 1 \t\t
        frame 2 \t\t frame 3 \t\n");
for(int m=0; m< pages; m++){
        bool pagefault = true;
    for(int n=0; n<frames; n++){
      if(temp[n]== incomingstream[m]){
          pagefault=false;
          break;
    }  }
if (pagefault){
    temp[front] = incomingStream[m];
    front = (front++1) % frames;
      pagefaults++;
}
printf("%d \t\t", incomingstream[m]);
```

```
for(int n=0; n<frames; n++){
    if (temp[n] != -1)
        printf(" %d \t\t ", temp[n]);
    else
        printf (" - \t\t");
  }
  printf ("\n");
 }
 printf("\n Total page Faults \t %d \n",
        pagefaults);
 }
void lrupageReplacement (int pages, int
    incomingstream[], int frames) {
    int pagefault = 0;
    int temp[ frames];
    int used [frames];
   for (int m=0; m<frames; m++) {
        temp[m] = -1;
        used[m] = 0;
    }
printf("\n --- LRU page Replacement
        Algorithm --\n");
 printf("\n Incoming \t Frame 1 \t
        Frame 2 \t Frame 3 \n");
 for(int m=0; m<pages; m++) {
        bool pagefault = true;
        int leastusedIdx = 0;
 for (int n=0; n<frames; n++) {
        if (temp[n] == incomingstream[m]) {
            pagefault = false;
            used[n] = m+1;
```

```
            break;
        }
        if (used[n] < used[leastUsedIdx]) {
            leastUsedIdx = n;
        }
    }

    if (pageFault) {
        temp[leastUsedIdx] = incomingStream[m];
        used[leastUsedIdx] = m+1;
        pageFaults++;
    }

    printf("%d\t\t", incomingStream[m]);
    for (int n=0; n<frames; n++) {
        if (temp[n] != -1)
            printf("%d\t\t", temp[n]);
        else
            printf("-\t\t");
    }
    printf("\n");
}
printf("\nTotal Page Faults:\t%d\n",
        pageFaults);
}

void optimalPageReplacement(int pages,
        int incomingStream[], int frames) {
    int pageFaults=0;
    int temp[frames];
```

```
for(int m=0; m<frames; m++){
        temp[m] = -1;
}
printf("\n --- Optimal page replacement
        Algorithm --\n");
printf("\nIncoming\t Frame1\t Frame2\t
        Fram3\n");
for(int m=0; m<pages; m++){
    bool pagefault = true;
    int replaceIdx = -1;
     int farthest=m;
    for(int n=0; n<frames; n++){
        if(temp[n] == incomingStream[m]){
            pagefault = false;
            break;
        }
    }
if(pagefault){
    for(int n=0; n<frames; n++){
            int j;
    for(j=m+1; j<pages; j++){
        if(temp[n] == incomingstream[j]){
        if(j > farthest){
                farthest = j;
                replaceIdx = n;
        }
    break;
    }}
}
```

```
        if (j == pages) {
            replaceIdx = n;
            break;
        }
    }
    if (replaceIdx == -1) {
        replaceIdx = 0;
    }
    temp[replaceIdx] = incomingStream[m];
    pageFaults++;
}
printf("%d\t\t", incomingStream[m]);
for(int n=0; n<frames; n++) {
    if (temp[n] != -1)
        printf("%d\t\t", temp[n]);
    else
        printf("-\t\t");
}
printf("\n");
}
printf("\nTotal page faults:\t%d\n",
        pageFaults);
}
int main() {
    int pages;
    printf("Enter number of pages in in-
    -coming stream");
    scanf("%d", &pages);
int incomingStream[pages];
printf("Enter the sequence of pages\n");
for(int i=0; i<pages; i++) {
```

```
scanf("%d", &incomingStream[i]); }
int frames;
printf("Enter number of frames");
scanf("%d", &frames);
fifoPageReplacement(pages, incomingStream,
                    frames);
lruPageReplacement(pages, incomingStream,
                   frames);
optimalPageReplacement(pages, incomingStream,
                       frames);
return 0;
}
```

output:

Enter number of pages in incoming stream: 20

Enter the sequence of pages:

0 9 0 1 8 1 8 7 8 7 1 2 8 2 7 8 2 3 8 3

Enter no. of Frames: 3

FIFO

incoming	Frame 1	Frame 2	Frame 3
0	0	—	—
9	0	9	—
0	0	9	1
1	8	9	1
8	8	9	1
1	8	7	1
8	8	7	1
7	8	7	1
8	8	7	2
7	8	7	2
1	8	7	2
2	8	7	2
8	8	7	2
2	8	7	2
7	8	8	2
8	8	8	2
2	3	8	2
3	3	8	2
8	3	8	2

Where page fault occurs

pagefault = 9