

Q) Write a program to implement First come first serve scheduling.

```

#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[i];
    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 = 0;
for (int i=0; i<n; i++) {
    sum_tat += tat[i];
}
avg_tat = sum_tat / n;
printf("average Tat = %f\n", avg_tat);

```

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

Output:

enter total no. of processes : 4
 enter arrival times : 0
 6
 1
 5
 enter burst times : 2 4 5 3
 0 2 4 5
 2
 3
 completion-time [1] = 2
 completion-time [2] = 4.000
 completion-time [3] = 8.000
 completion-time [4] = 12.000
 average TAT = 3.500
 average waiting time = 0.7500

SJF

#define T

#include <stdio.h>

#include

typedef struct {

int pid;

int arr;

int burst;

int comp;

int turn;

int wait;

int processed; /* time required */

} proc;

void sjf_np(proc proc[], int n) {

int curr_time = 0;

int total_comp_time = 0;

int total_wait_time = 0; /* total waiting time */

int total_turnaround_time = 0; /* CHRT */

while(1) {

int shortest_job = -1;

int shortest_burst = 9999;

for (int i=0; i<n; i++) {

if (proc[i].arr <= curr_time & proc[i].proc

- sed == 0)

{

if (proc[i].burst < shortest_burst)

shortest_burst = proc[i].burst;

shortest_job = i;

shortest_job = i;

curr_time = TAT + 1;



```

if (shortest->job == -1)
    break;
process[shortest->job].comp = curr_time + process[shortest->job].burst;
process[shortest->job].turn = process[shortest->job].comp
    - process[shortest->job].arrive;
process[shortest->job].wait = process[shortest->job].turn
    - process[shortest->job].burst;
if (process[shortest->job].wait < 0)
    process[shortest->job].wait = 0;
double avg_turnaround_time = (double) total_turnaround_time / n;
double avg_waiting_time = (double) total_waiting_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%d\t%d\t%d\t%d\t%.2f\n",
        process[i].pid, process[i].arrive, process[i].burst,
        process[i].comp, process[i].wait, process[i].turn);
}
printf ("Average turnaround time: %.2f \n", avg_turnaround_time);
printf ("Average waiting time: %.2f \n", avg_waiting_time);
printf ("Average waiting time: %.2f \n", avg_waiting_time);

```

```

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

```

Sif-np(proc, n);

if not found then return 0 else return 1

Output: 16 5 10 15
processID ArrivalTime BurstTime CompTime WT TurnaroundTime

processID	ArrivalTime	BurstTime	CompTime	WT	TurnaroundTime
1	0	5	5	0	5
2	5	10	15	5	10
3	10	5	15	5	15
4	15	6	21	6	21
5	21	5	26	5	26

Q1) Assume processes P1, P2, P3, P4, P5
with arrival times 0, 5, 10, 15, 21 respectively.

Round Robin

```
#include<stdio.h>
#include<stack stdlib.h> //opening block
Struct Queue {
    int pid;
    Struct Queue* next;
};

Struct Queue* rq=NULL; //initialization
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->next!=NULL)
            temp=temp->next;
        temp->next=nn;
    }
}

int dequeue() {
    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 = q;
    while (temp != NULL) {
        printf("%d", 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 + 1 - count;  
            break;  
        }  
    }
```

```
if (rt[curi] == -1) {  
    rt[curi] = ctvar - at[curi];  
}  
if (bt2[curi] <= t) {  
    ctvar += rt[curi];  
    rt[curi] = 0;  
    if (bt2[curi] <= t) {  
        ctvar += bt2[curi];  
        bt2[curi] = 0;
```

```

else {
    cvar += t;
    bt[curi] = t;
}

while (at[x] <= cvar && x < n) {
    enqueue(p[i][x]);
    x++;
}

if (~bt[curi] > 0) {
    enqueue(pid[curi]);
}

if (bt[curi] == 0) {
    count++ = 1;
    ct[curi] = cvar;
}

dequeue();
}

for (int i=0; i<n; i++) {
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[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\ttat\tbt\tct\ttat-x\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], bt[i], ct[i], tat[i], wt[i]);
}

printf("\n");

printf("\n Average turnaround time : %f", avg_tat/n);
printf("\n Average waiting time : %f", avg_wt/n);

return 0;
}

```

Entered the number of processes: 5 Output

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

Enter arrival time and burst time: 3

2

Enter arrival time and burst time:

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

Average turn around time: 8.60000

Average waiting time: 5.8000

Ran
6/6/24

PRIORITY SCHEDULING

```
#include <stdio.h>
#include <stdlib.h>
#include <math.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], bt[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, bt, prior, n);
    for (int i=0; i<n; i++) {
        bt2[i] = bt[i];
        rt[i] = -1;
    }
    int arrc = 0;
    int count = 0;
    int ctvar = at[0];
    int curi = 0;
    while (count != n) {
        if (rt[curi] == -1) {
            rt[curi] = ctvar = at[curi];
        }
        if (arrc == n) {
            ctvar += bt2[curi];
            bt2[curi] = 0;
        } else {
            ctvar += 1;
            bt2[curi] -= 1;
        }
        count++;
    }
}

```

```

for(int i=0; at[i]<=ctval; i++) {
    arr[i] = 1;
    x++;
}

if (bt[0][cur[0]] == 0) {
    count += 1;
    if (ct[0][cur[0]] == ctval) {
        prior[0][cur[0]] = -1;
        cur[0] = highest_priority(prior, 0, x + b);
    }
}

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

printf("Average turnaround time: %.2f", avg_tat/n);
printf("Average waiting time: %.2f", 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

P: 4
20

Enter arrival time, burst time and priority: 2

2

36

Enter arrival time, burst time and priority: 4

1

46

pid	at	bt	ct	wt	tat	proc.	at + bt
1	0	5	12	7	7	0	12
2	1	4	8	7	3	0	8
3	2	2	4	2	2	0	4
4	4	1	5	1	1	0	5

Average Turn around Time: 5.50000

Average waiting time: 2.50000

Average waiting time

By
611121



Multilevel queue scheduling

2 AUG 2009

```
#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: ");
```

```
scanf("%d", &n);
```

```
struct process p[n], system_p[1], user_p[n];
```

```
printf("Enter the arrival, time, burst time and  
type (1 for system process 0 for  
user process) for all the processes: ");
```

```

for(int i=0; i<n; i++)
{
    printf("\n\n process %d : \n", i+1);
    printf(" name : %s", p[i].name);
    printf(" arrival time : ");
    scanf("%d", &p[i].AT);
    printf(" burst time : ");
    scanf("%d", &p[i].BT);
    printf(" type : (%d for system, 0 for user) : ", );
    scanf("%d", &p[i].isSystem);
    if(p[i].isSystem)
        system = p[sysCount] = p[i];
    else
        user = p[userCount] = p[i];
}

```

```

for(int i=0; i<sysCount; i++)
{
    for(int j=0; j<sysCount-i-1; j++)
    {
        if(system->AT > p[j].AT)
        {
            struct process temp = system->p[j];
            system->p[j] = system->p[j+1];
            system->p[j+1] = temp;
        }
    }
}

```

~~init current-time = 0;~~

int totalwt = 0, totalltat = 0;

int totalwt, totaltat, arrivalTime, turn

printf("\n\n process \t name \t arrival time \t burst time \t completion time \t waiting time \t turn around time \t waiting time \t type \n");

int i=0, j=0;

if(p[0].AT <= 0) p[0].AT = 1;

if(p[0].BT <= 0) p[0].BT = 1;

```

while (icsys-count < user-count) {
    if (icsys-count + i >= user-count)
        system-pc[i].AT <= (current-time + time);
    if (current-time < system-pc[i].AT)
        current-time = system-pc[i].AT;
    system-pc[i].CT = current-time + system-pc[i].BT;
    system-pc[i].TAT = system-pc[i].CT - system-pc[i].AT;
    system-pc[i].WT = system-pc[i].TAT - system-pc[i].BT;
    current-time = system-pc[i].CT;
    printf("%d %s %d %d %d %d %d %d\n",
           i + 1, system-pc[i].name,
           system-pc[i].AT, system-pc[i].BT, system-pc[i].BT,
           system-pc[i].CT, system-pc[i].TAT, system-pc[i].TAT,
           system-pc[i].total-WT);
    total-TAT += system-pc[i].TAT;
    i++;
}
else if (j < user-count) {
    if (current-time < user-pc[j].AT)
        current-time = user-pc[j].AT;
    user-pc[j].CT = current-time + user-pc[j].BT;
    user-pc[j].TAT = user-pc[j].CT - user-pc[j].AT;
    user-pc[j].WT = user-pc[j].TAT - user-pc[j].BT;
    current-time = user-pc[j].CT;
    printf("%d %s %d %d %d %d %d %d\n",
           j + 1, user-pc[j].name,
           user-pc[j].AT, user-pc[j].BT, user-pc[j].BT,
           user-pc[j].CT, user-pc[j].TAT, user-pc[j].TAT);
}

```

```

        user-pj7.wt),
        total-wt += user-pj7.wt;
        total-tat += user-pj7.TAT;
        j++;
    }

    float avg_wt = (float)total_wt/n;
    float avg_tat = (float)total_tat/n;
    printf("Average waiting time = %f", avg_wt);
    printf("Average turn around time = %f\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	WT	TYPE
1 P1	0	3	3	3	0	SYSTEM
2 P2	2	6	9	7	1	USER
3 P3	4	4	13	9	9	SYSTEM
4 P4	6	5	18	12	7	USER
5 P5	8	2	15	12	10	SYSTEM

Average waiting time = 4.600 or 4.60

Average turnaround time = 8.600

maximum wait time 10

Process 1 waiting time 0
Process 2 waiting time 2
Process 3 waiting time 9
Process 4 waiting time 7
Process 5 waiting time 10

Process 1 turn around time 3
Process 2 turn around time 9
Process 3 turn around time 13
Process 4 turn around time 18
Process 5 turn around time 15

Process 1 completion time 3
Process 2 completion time 9
Process 3 completion time 13
Process 4 completion time 18
Process 5 completion time 15

Karve monotonic program

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

#define MAX_TASKS 10

typedef struct {
    int Ti; // execution time of task i
    int Ci; // completion time of task i
    int deadline; // deadline of task i
    int RT; // ready time of task i
    int id; // task 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++) {
        printf("Enter id of task %d: ", i + 1);
        scanf("%d", &tasks[i].id);
        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].Ci;
        tasks[i].RT = tasks[i].Ci;
    }
}
```

```
int compare_by_per, void (const void*, const void*)
```

```
{ return ((task*)a) -> Tp - ((task*)b) -> Tp; }
```

```
void RMS(task tasks[], int n_tasks, int timeframe)
```

```
{ qsort(tasks, n_tasks, sizeof(task), compare_by_period); }
```

```
printf("In Rate-monotonic Scheduling:\n");
```

```
for (int time=0; time<timeframe; time++)
```

```
{
```

```
int s_task=-1;
```

```
for (int i=0; i<n_tasks; i++)
```

```
{
```

```
if (tasks[i].Ti == 0)
```

```
{
```

```
tasks[i].RT = tasks[i].Gi;
```

```
}
```

```
if (tasks[i].RT > 0 & (s_task == -1))
```

```
tasks[i].Ti < tasks[s_task].Ti)
```

```
{
```

```
s_task = i; RT -= (i - s_task);
```

```
RT = max(RT, 0);
```

```
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);
    RMSI (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 execution time of task 3: 10

Enter Ti of task 3: 2

Enter execution time of task 3: 2

Enter time frame for simulation: 20

first-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 8

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

Time 20 : Idle

Time 21 : Idle

Time 22 : Idle

Time 23 : Idle

Time 24 : Idle

Time 25 : Idle

Time 26 : Idle

Time 27 : Idle

Time 28 : Idle

Time 29 : Idle

Time 30 : Idle

Time 31 : Idle



Earliest Deadline First

```
#include<std.h>
#include<stdlib.h>
#define MAX-Tasks 16

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

void Input(Task tasks[], int *nTasks)
{
    printf("Enter number of tasks");
    scanf("%d", nTasks);
    for (int i = 0; i < *nTasks; 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].RT);
        printf("Enter deadline of task %d", i + 1);
        scanf("%d", &tasks[i].deadline);
        tasks[i].nDeadline = tasks[i].deadline;
    }
}
```



```

void EDF( task tasks[], int nTasks, int time,
          int *time)
{
    printf("In Earliest-Deadline First\n");
    printf("Scheduling:\n");
    for (int time=0; time<time+frame;
         time++)
    {
        int sTask = -1;
        for (int i=0; i<nTasks; i++)
        {
            if (tasks[i].RT == 0)
            {
                tasks[i].RT = tasks[i].C;
                tasks[i].nDeadline = (time + tasks[i].dead);
            }
        }
        for (int i=0; i<nTasks; i++)
        {
            if (tasks[i].RT > 0 & (sTask == -1))
            {
                tasks[i].RT -= 1;
                sTask = i;
            }
        }
        if (sTask != -1)
        {
            printf("Time %d: Task %d\n", time,
                  tasks[sTask].id);
            tasks[sTask].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 T_i of task 1: 4

Enter execution time of task 1: 1

Enter deadline of task 1: 4

Enter T_i of task 2: 5

Enter execution time of task 2: 2

Enter deadline of task 2: 5

Enter T_i 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

proportional programming

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

A 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];

scanf(time(NULL));

for (int i = 0; i < n; i++)

{ printf("In process %d:\n", i + 1);

sprintf(pt[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: ");

float m;

scanf("%f", &m);

```

forc int i=0; i<n; itd
    {
        < int winning-ticket=rand()%total-tickets+1;
        int accumulated-tickets=0;
        int winner-index;
        forc(int j=0; j<n; j++)
            {
                accumulated-tickets+=PE[j].tickets;
                if (winning-ticket <= accumulated-tickets)
                    {
                        winner-index=j;
                        break;
                    }
            }
        printf ("Tickets picked : %d, winner : %s\n",
               winning-ticket, PE[winner-index].name);
    }
    forc(int i=0; i<n; i++)
        {
            printf("In the process %s gets %.02f%.\n",
                   PE[i].name,
                   ((PE[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

Rm
616

Computer Networks

Chennai Institute of Technology

Chennai - 600089

Tel : 044-22241100

E-mail : admin@chait.org.in



Semaphores

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.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++;
}

void dequeue() {
    int item = buffer[head];
    head = (head + 1) % BUFFER_SIZE;
    cnt--;
}
```

```
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-ITEMS) {
            pthread_mutex_unlock(&mtx);
            pthread_cond_signal(&cond-full);
            break;
        }
        enqueue(item);
        prod-cnt++;
        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);
        }
        pthread_mutex_unlock(&mtx);
        cons-cnt++;
        printf("consumed : %d\n", item);
    }
}
```

```

if (const ->cnt >= MAX-1) {
    pthread-mutex-unlock(&mtx);
    pthread-cond-signal(&cond-empty);
    break;
}

int item = deque();
const ->cnt++;
printf("consumed: %d\n", item);
pthread-mutex-unlock(&mtx);
pthread-cond-signal(&cond-empty);
sleep(rand(1-2));
}

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, product,
                   NULL);
    pthread-create(&tid-cons, NULL, consumer, NULL);
    pthread-join(tid-prod, NULL);
    pthread-join(tid-cons, NULL);
    printf("production & consumption impl.");
    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

✓
✓

Dining philosophers program

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

#define NUM_PHILOSOPHERS 5
sem_t forks[5];
sem_t mutex;

void* philosopher_one(void* num) {
    int id = *(int*)num;
    printf("P %d is waiting\n", id);
    sem_wait(&mutex);
    printf("P %d is waiting\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.%d is granted to eat\n", id);
sleep(1);
sem-post(&forks[id]);
sem-post(&forks[(id+1)%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(&mult, 0, 0);

```

int choice;
printf("In 1. one can eat at a time\n"
       "2. two can eat at a time\n"
       "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++) {
        pds[i] = hungry - positions[i];
        pthread_create(&philosophers[i], NULL,
                       philosopher_one, &pds[i]);
        // printf("Created philosopher one, %d\n");
    }
} else if (choice==2) {
    printf("\n Allow two philosophers to eat
at the same time\n");
    for (int i=0; i<num_hungry; i++) {
        pds[i] = hungry - positions[i];
        pthread_create(&philosophers[i], NULL,
                       philosopher_two, &pds[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(&matrix);  
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:

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

~~p8 is granted to eat~~

p4 is waiting

p1 is waiting

~~p4 is granted to eat~~

~~p1 is granted to eat~~

Banke's Algorithm

```
#include <csdio.h>
#include <stdlib.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][j] > work[i])
            break;
        if(j == n) {
            for(int k=0; k < n; 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, n;
    printf("Enter number of processes:");
    scanf("%d", &np);
    printf("Enter number of resource types:");
    scanf("%d", &n);
    int processes[MAX PROCESSES];
    for(int i=0; i<np; i++)
        scanf("%d", &processes[i]);
    int max[processes] = {0};
    for(int i=0; i<np; i++)
    {
        int avail[MAX RESOURCES];
        printf("Enter available resources:");
        for(int j=0; j<n; j++)
            scanf("%d", &avail[j]);
        int 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<n; 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<n; j++)
            scanf("%d", &alloc[i][j]);
    }
}

```

issafe processes avail, max, allo, np, nv

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 2

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

Q/37pm



Deadlock Detection

```
#include <stdio.h>
#include <stdlib.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) {
                    safeSequence[count]=i;
                    count++;
                    for (int j=0; j<NUM_RESOURCES; j++) {
                        work[j] += allocation[safeSequence[count]][j];
                    }
                }
            }
        }
    }
}
```

```

if (can proceed) {
    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 %d\n"
               "is in deadlock.\n", i);
        return false;
    }
}

printf("No deadlock detected. The system is in\n"
       "a safe state.\n");
printf("safe sequence:");
for (int i=0; i<NUM-PROCESSES; i++) {
    printf("/%d", safeSequence[i]);
}
printf("\n");
return true;
}

```

```

int main()
{
    int i, j;
    printf("Enter the available resources\nvector: \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[100];
    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	8	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

Run
3 1 2 4



Memory Allocation.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX-PARTITIONS 10
#define MAX-PROCESSES 10
typedef struct {
    int size;
    int pAllocated;
} partition;
typedef struct {
    int size;
    int pAllocated;
    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, partitionCount,
                               processes[i].size);
        if (index != -1) {
            processes[i].pAllocated = 1;
            processes[i].partitionIndex = index;
            partitions[index].pAllocated = 1;
            totalFragmentation++;
        }
    }
}
```

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

```
totalfragmentation += fragmentation;
```

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

```
} else {
```

```
printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t-\n",  
    i, processes[i].size);
```

```
}
```

```
printf("\nTotal fragmentation : %d",  
    totalfragmentation);
```

```
int firstFitIndex(partition partitions[],  
    int partitionCount, int processSize) {
```

```
int bestIndex = -1;
```

```
for (int i = 0; i < partitionCount; i++) {  
    if (!partition[i].allocated && partition[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 || partitions[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", &processes[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);  
  
if (*allocationstrategy) (partition[0], 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;  
> // Main Function  
printf("In memory management scrumpe\n");  
printf("- - - - -\n");  
printf("File No. Lt. Process size Lt. Block no.  
Lt. Block size Lt. fragment\n");  
printf("- - - - -\n");  
allocate (processes, processcount, partitions,  
partitioncount, allocationstrategy);  
return 0;
```

```
>  
end
```

Output

Enter the number of blocks: 3

Enter the number of files: 8

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

File no.	File-size	Block-no.	Block-size	Fragment
1	1	1	5	4
2	4	3	7	3

Memory management scheme - Worst Fit

File no.	File-size	Block-no.	Block-size	Fragment
1	1	3	7	6
2	4	1	5	1

Memory management scheme - Best Fit

File no.	File-size	Block-no.	Block-size	Fragment
1	1	2	2	1
2	4	1	5	1

↓
Best fit

Page Replacement + FIFO

```
#include<stdio.h>
#include<stdlib.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 ("In Incoming Stream Frame 1 | Frame 2 | Frame 3\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 | ", incomingStream[m]);
    }
}
```

```

for(int n=0; n<frames; n++) {
    if (temp[n] != -1)
        printf("y.dlt.t", temp[n]);
    else
        printf("-.t.t");
}
printf("\n");
printf("\n Total page faults \t y.d\n",
       pagefaults);
}

void lruPageReplacement (int pages, int
incomingsream[], int frames) {
int pagefaults = 0;
int temp[frames];
int used[frames];
for (int m=0; m<frames; m++) {
    temp[m] = -1;
    used[m] = 0;
}
printf("In --- LRU page replacement
Algorithm -- \n");
printf("In Incoming 1st frame \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] == incomingsream[m]) {
            pageFault = false;
            used[n] = m+1;
        }
    }
    if (pageFault)
        pagefaults++;
}
}

```

```

        break;
    }
    if (used[n] < used[least + usedIdx]) {
        least + usedIdx = n;
    }
}
if (pageFault) {
    temp[least + usedIdx] = incomingStream[m];
    used[least + usedIdx] = m + 1;
    pageFaults++;
}
printf("y-dit it", incomingStream[m]);
for (int n=0; n<frames; n++) {
    if (temp[m] != -1)
        printf("y-dit it", temp[n]);
    else
        printf("-dit it");
    printf("\n");
}
printf("in total page faults : %d\n",
       pageFaults);
}

void optimalPageReplacement(int pages,
                           int incomingStreams[], int frames,
                           int pageFaults=0,
                           int temp[frames]);

```

```

for (int m=0; m<frames; m++) {
    temp[m] = -1;
}

printf("In ... optimal page replacement\n"
       "Algorithm--\n");
printf("1Incoming\1t frame1st frame2\1 t\n"
       "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;
                    }
                }
            }
        }
    }
}

```

```

if (j == pages) {
    replaceIdx = n;
    break;
}
if (replaceIdx == -1) {
    replaceIdx = 0;
}
temp[replaceIdx] = incomingStream[m];
pageFaults++;
printf("%d\t", incomingStream[m]);
for (int n = 0; n < frames; n++) {
    if (temp[n] == -1)
        printf("%d\t", temp[n]);
    else
        printf("-\t");
}
printf("\n");
printf("\n Total page faults : %d\n", pageFaults);
}

int main() {
    int pages;
    printf("Enter number of pages in in-");
    printf("coming stream");
    scanf("%d", &pages);
    int incomingStream[pages];
    printf("Enter the sequence of pages\n");
    for (int i = 0; i < pages; i++) {

```

```

scanf("vd", &incomstream);
int frames;
printf("Enter number of frames:");
scanf("vd", &frames);
fifopageReplacement(pages, incomstream,
frames);
ruuPageReplacement(pages, incomstream,
frames);
optimalPageReplacement(pages, incomstream,
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 8 3 8 3

Enter no. of frames : 3

FIFO

incoming	frame 1	frame 2	frame 3
0	0	9	-
9	0	9	-
0	0	9	-
1	8	9	-
8	8	9	-
1	8	9	-
8	8	9	-
7	8	7	-
8	8	7	-
7	8	7	-
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

pagefault = 9



Scanned with OKEN Scanner