

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  
“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT**  
on

**OPERATING SYSTEMS**

Submitted by

SUSHANTH C (1WA23CS001)

in partial fulfillment for the award of the degree of  
**BACHELOR OF ENGINEERING**  
in  
**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**  
(Autonomous Institution under VTU)  
**BENGALURU-560019**  
**Feb-2025 to June-2025**

B. M. S. College of Engineering,  
Bull Temple Road, Bangalore 560019  
(Affiliated To Visvesvaraya Technological University, Belgaum)  
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “OPERATING SYSTEMS – 23CS4PCOPS” carried out by Sushanth C(1WA23CS001), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Dr.Sema Patil  
Assistant Professor  
Department of CSE  
BMSCE, Bengaluru

Dr. Kavitha Sooda  
Professor and Head  
Department of CSE  
BMSCE, Bengaluru

## Index Sheet

Sl. No.	Experiment Title	Page No.
1.	<p>Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.</p> <ul style="list-style-type: none"> <li>→ FCFS</li> <li>→ SJF (Non-preemptive, preemptive)</li> </ul>	1. 1-10
2.	<p>Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.</p> <ul style="list-style-type: none"> <li>→ Priority (pre-emptive &amp; Non-pre-emptive)</li> </ul>	11-19
3.	<p>Write a C program to simulate Real-Time CPU Scheduling algorithms:</p> <ul style="list-style-type: none"> <li>a) Earliest deadline</li> <li>b) Rate monotonic</li> <li>c) Proportional scheduling</li> </ul>	20-24
4.	Write a C program to simulate producer-consumer problem using semaphores	24-34
5.	Write a C program to simulate the concept of Dining Philosophers problem.	35-38
6.	Write a C program to perform deadlock allocation	39-43
7.	Write a C program to simulate deadlock detection	44-47
8.	<p>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</p>	48-51

	priority than user processes. Use FCFS scheduling for the processes in each queue.	
9.	→Round Robin (Experiment with different quantum sizes for RR algorithm)	52-56
10.	1) Write a C program to perform (MMU): a)best fit b)worst fit c)first fit 2) Write a C program to simulate page replacement algorithms a) FIFO b)LRU c)optimal	57-63

## Course Outcomes

C01	Apply the different concepts and functionalities of Operating System
C02	Analyse various Operating system strategies and techniques
C03	Demonstrate the different functionalities of Operating System.
C04	Conduct practical experiments to implement the functionalities of Operating system.

Name SUSHANTH . C Std \_\_\_\_\_ Sec \_\_\_\_\_

Roll No. \_\_\_\_\_ Subject OS (OBSERVATION) School/College \_\_\_\_\_

School/College Tel. No. \_\_\_\_\_ Parents Tel. No. \_\_\_\_\_

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	6/3	First C F S		2/15
2		Priority Preemptive	10	3
3		Priority (Non-preemptive)	10	15
4		SJF Preemptive	10	
5		SJF Non Preemptive	10	10
6		Round Robin	10	
7		Multilevel Queue	10	
8		Rate Monotonic Scheduling		
9		Earliest deadline First	10	
10		Producer - Consumer	10	
11		Dining Philosophers	10	
12		Banker's algorithm	10	
13		Deadlock detection	10	
14		Deadlock Avoidance		
15.		Memory Management Unit scheme (Best Fit) Working	10	10
16.		FIFO - 10	10	

Priority

## Program -1

### Question:

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

→FCFS

→ SJF (pre-emptive & Non-preemptive)

### Code:

=>FCFS:

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

```
void sort(int proc_id[],int at[],int bt[],int n)
{
    int
min=at[0],temp=0;
for(int i=0;i<n;i++)
{
    {
min=at[i];
    for(int j=i;j<n;j++)
    {
        if(at[j]<min)
        {
            temp=at[i];
at[i]=at[j];
at[j]=temp;
temp=bt[j];
bt[j]=bt[i];
bt[i]=temp;
temp=proc_id[i];
proc_id[i]=proc_id[j];
proc_id[j]=temp;
        }
    }
}
}
```

```
void main()
{
    int n,c=0;    printf("Enter number
of processes: ");    scanf("%d",&n);
int
proc_id[n],at[n],bt[n],ct[n],tat[n],wt[n];
double
```

```

avg_tat=0.0,ttat=0.0,avg_wt=0.0,twt=0.
0;    for(int i=0;i<n;i++)
proc_id[i]=i+1;  printf("Enter arrival
times:\n");   for(int i=0;i<n;i++)
scanf("%d",&at[i]);  printf("Enter
burst times:\n");   for(int i=0;i<n;i++)
scanf("%d",&bt[i]);

    sort(proc_id,at,bt,n);
//completion time   for(int
i=0;i<n;i++)
{
    if(c>=at[i])
c+=bt[i];    else
c+=at[i]-ct[i-1]+bt[i];
ct[i]=c;
}
//turnaround time
for(int i=0;i<n;i++)
tat[i]=ct[i]-at[i];
//waiting time   for(int
i=0;i<n;i++)
wt[i]=tat[i]-bt[i];

printf("FCFS scheduling:\n");   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",proc_id[i],at[i],bt[i],ct[i],tat[i],wt[i]);

for(int i=0;i<n;i++)
{
ttat+=tat[i];twt+=wt[i];
}
avg_tat=ttat/(double)n;   avg_wt=twt/(double)n;
printf("\nAverage turnaround time:%lfms\n",avg_tat);
printf("\nAverage waiting time:%lfms\n",avg_wt);
}

```

Result:

Process	Burst Time	Arrival Time	Waiting Time	Turn Around Time
0	5	0	0	5
1	3	1	4	7
2	8	2	6	14
3	6	3	13	19

---

Average Waiting Time: 5.75  
Average Turnaround Time: 11.25

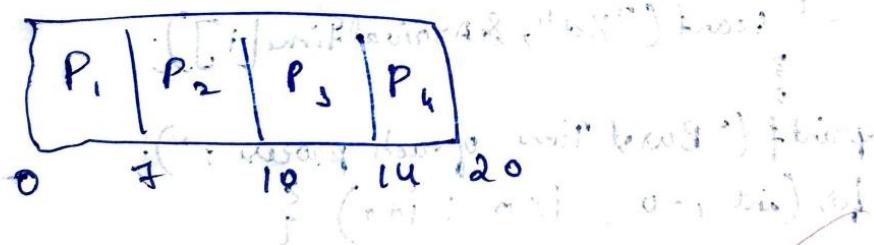
Process returned 0 (0x0) execution time : 0.320 s  
Press any key to continue.

First come first serve (FCFS, MAF, SJF) with

```
#include <stdio.h> } (arrive[i] <= i) {  
int main () { (.) int processes = TAW  
int n; (.) int total = TP  
printf ("no of process: "); (.) int turnaroundTime = TATP  
scanf ("%d", &n); (n < TAW  
int arrivalTime [n]; (n = P)  
int completionTime [n]; : n = MAF  
int TurnAroundTime [n];  
int waitingTime [n]; P/N : P ) following  
for ("Arrival time; i < n; i++) { (TAW  
printf ("Arrival Time of each process: "); (n = P)  
for (int i = 0; i < n; i++) {  
    scanf ("%d", &arrivalTime[i]);  
    }  
    printf ("Burst Time of each process: "); (n = P)  
    for (int i = 0; i < n; i++) {  
        scanf ("%d", &burstTime[i]);  
        }  
        int sum = 0  
        for (int i = 0; i < n; i++) {  
            sum += burstTime[i];  
            completionTime[i] = sum;  
            }  
            for (int i = 0; i < n; i++) {  
                TurnAroundTime[i] = CT[i] - AT[i]  
                }  
                for (int i = 0; i < n; i++) {  
                    waitingTime[i] = TurnAround[i] - burst[i]
```

float 'CT ,TAT ,WT';  
 for (int i=0; i<n; i++) {  
 WT+=Waiting Time (i);  
 CT+=Completion Time (i);  
 TAT+=Turn Around Time (i);  
 }  
 WT=n;  
 CT=n;  
 TAT=n;  
 printf (" CT : %.4f \n TAT : %f \n WT : %.4f " , CT, TAT,  
 WT);

Pgyant Chart



✓ Gantt Chart  
 ✓ Gantt Chart  
 ✓ Gantt Chart  
 ✓ Gantt Chart

Ques. 3. Let's consider the following

Output :  $\text{AT} = \text{Arrival time}$  &  $\text{BT} = \text{Burst time}$

No of process : 4

AT of each process : 00 00  
(d. d1bts) start time

BT of each process : 23 46  
(d. d1bts) burst time

CT : 12.75 0000 } sum of burst

TAT : 12.75 0000 } total time

WT : 7.75 0000 } idle time

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

idle time : 00 00

to : 00 00

from : 00 00

total : 00 00

### SJF(NON PREEMPTIVE) CODE:

```
#include <stdio.h> struct Process  
{  
    int pid, bt, at, wt, tat, rt, ct;  
};  
  
void sortByArrival(struct Process p[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (p[j].at > p[j + 1].at)  
            {  
                struct Process temp = p[j];  
                p[j] = p[j + 1];  
                p[j + 1] = temp;  
            }  
        }  
    }  
  
    void sjfScheduling(struct Process p[], int n) {
```

```

sortByArrival(p, n);

int completed = 0, currentTime = 0;

int totalWT = 0, totalTAT = 0, totalRT = 0;

while (completed < n) {

    int minIndex = -1, minBT = 9999;

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

        if (p[i].at <= currentTime && p[i].bt <

            minBT && p[i].tat == 0)

        {

            minBT = p[i].bt;

            minIndex = i;

        }

    }

    if (minIndex == -1) {

        currentTime++;
        continue;

    }

    p[minIndex].wt = currentTime - p[minIndex].at;

    p[minIndex].tat = p[minIndex].wt + p[minIndex].bt;

    p[minIndex].rt = p[minIndex].wt;

```

```

p[minIndex].ct = currentTime + p[minIndex].bt;

currentTime += p[minIndex].bt;      completed++;

totalWT += p[minIndex].wt;

totalTAT += p[minIndex].tat;

totalRT += p[minIndex].rt;

}

printf("\nPID\tAT\tBT\tCT\tWT\tTAT\tRT\n");

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

    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt, p[i].tat,
p[i].rt);

}

printf("\nAverage WT: %.2f", (float)totalWT / n);

printf("\nAverage TAT: %.2f", (float)totalTAT / n); printf("\nAverage RT:
%.2f\n", (float)totalRT / n);

} int main()

{

    int n;

    printf("Enter number of processes: ");

    scanf("%d", &n); struct Process p[n]; for

```

```

(int i = 0; i < n; i++) {           printf("Enter AT &
BT for P%d: ", i + 1);         scanf("%d %d",
&p[i].at, &p[i].bt);          p[i].pid = i + 1;
p[i].wt = p[i].tat = p[i].rt = p[i].ct = 0;

}

sjfScheduling(p, n);

return 0;
}

```

Output:

PID	AT	BT	CT	WT	TAT	RT
1	0	7	20	13	20	13
2	0	3	3	0	3	0
3	0	4	7	3	7	3
4	0	6	13	7	13	7

```

Enter number of processes: 4
Enter AT & BT for P1: 0 7
Enter AT & BT for P2: 0 3
Enter AT & BT for P3: 0 4
Enter AT & BT for P4: 0 6

PID      AT      BT      CT      WT      TAT      RT
1        0       7       20      13      20       13
2        0       3       3       0       3        0
3        0       4       7       3       7        3
4        0       6       13      7       13       7

Average WT: 5.75
Average TAT: 10.75
Average RT: 5.75

Process returned 0 (0x0)   execution time : 51.361 s
Press any key to continue.
|

```

4. Priority (preemption)  
shortest job first (Non-preemptive)

#include <stdio.h> // for stdio.h  
#include <conio.h> // for clrscr()

typedef struct {

int pid, at, bt, rema;

process;

void swap (process \*a, process \*b);

.. (\*process \*b); to > to. (i) means;

\*b = temp; i.e. 0 < wait minimum.

{Priority - min > priority. (i) means);

void sortbyarrival (process processes[], int n)

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

for (int j=0; j < n-i-1; j++)

if ((process[j].arrival\_time > process[j+1].

at) || (process[j].at = process[j+1].

at & & process[j].priority >

process[j+1].priority))

swap (&process[j], &process[j+1]);

3. --> TQ. (Waiting) means

+ time - time

void prioritypreemptive (process processes[], int n) {

int completed = 0, current\_time = 0, min\_priority,

max\_priority, (Waiting) work =

```

for (int i=0; i < n; i++) { // initialising
    processes[i].remaining_time = processes[i].burst_time;
}

while (completed != n) {
    min_priority = 9999; // max value
    selected = -1; // -1 means no process

    for (int i=0; i < n; i++) {
        if (process[i].at <= at && process[i].remaining_time > 0) {
            if (process[i].priority < min_priority) {
                min_priority = process[i].priority;
                selected = i;
            }
        }
    }

    if (selected == -1) {
        current_time += 1; // time passes
        continue;
    }

    process[selected].remaining_time -= 1; // time consumed
    current_time++;

    if (process[selected].remaining_time == 0) {
        completed++; // process completed
        current_time += process[selected].completion_time;
    }
}

```

$(selected). arrival time$  ;  
 process (selected). waiting - time =  
 process (selected).  
 $TA - process$

$(selected). BT$  ;  
 $process(selected). WT = process(selected).$   
 $TA - process(selected). BT$

$\Sigma$

Output

no of process : 4

BT 1 : 7

AT 1 : 0

BT 2 : 3

AT 2 : 8

BT 3 : 4

AT 3 : 3

BT 4 : 6

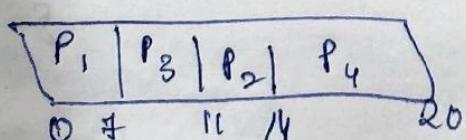
AT 4 : 5

Avg WT = 4.00

Avg TAT : 9.00

P	AT	BT	CT	TAT	WT	RT
P <sub>1</sub>	0	7	7	7	0	0
P <sub>2</sub>	8	3	11	6	3	8
P <sub>3</sub>	3	4	11	8	4	5
P <sub>4</sub>	5	6	20	15	9	9

Avg : 9 Avg : 6



## Preemptive Shortest Job First (SJF)

```
include<stdio.h>

#define MAX 10

typedef struct {
    int pid, at, bt, rt, wt, tat, completed;
} Process;

void sjf_preemptive(Process p[], int n) {    int time = 0,
completed = 0, shortest = -1, min_bt = 9999;

    while (completed < n) {
shortest = -1;      min_bt
= 9999;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].rt > 0 && p[i].rt < min_bt) {
min_bt = p[i].rt;          shortest = i;
            }
        }

        if (shortest == -1) {
time++;      continue;
        }

        p[shortest].rt--;
time++;

        if (p[shortest].rt == 0) {
p[shortest].completed = 1;      completed++;
        }
    }
}
```

```

    p[shortest].tat = time - p[shortest].at;
    p[shortest].wt = p[shortest].tat - p[shortest].bt;
}
}

}

int main() {
Process p[MAX];
int n;

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

for (int i = 0; i < n; i++) {
    p[i].pid = i + 1;
    printf("Enter arrival time and burst time for process %d: ", p[i].pid);
    scanf("%d %d", &p[i].at, &p[i].bt);      p[i].rt = p[i].bt;
    p[i].completed = 0;
}

sjf_preemptive(p, n);

printf("\nPID\tAT\tBT\tWT\tTAT\n");
for (int i = 0; i < n; i++)
    printf("%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat);

return 0;
}

```

```
"C:\Users\ADMIN\Documents\vicky042\SJF preemptive.exe"
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0
8
Enter arrival time and burst time for process 2: 1
4
Enter arrival time and burst time for process 3: 2
9
Enter arrival time and burst time for process 4: 3
5

PID      AT       BT       WT       TAT
1        0        8        9        17
2        1        4        0        4
3        2        9       15       24
4        3        5        2        7

Process returned 0 (0x0)  execution time : 37.778 s
Press any key to continue.
```

SJFC (Preemptive)  $\rightarrow$  (Process number) if

#include <stdio.h>

struct Process {

int id; int BT; int AT; int CT; int ST; void \*a;

} ; initProcess : P(SJFC number) & memory

int compareAT (const void \*a, const void \*b)

{

return ((struct Process \*)a -> AT -> ST (b) AT

}

int compareRT (const void \*a, const void \*b)

{

return ((struct Process \*)a -> AT -> S + (b) AT

}

int compareCT (const void \*a, const void \*b)

{ return ((struct Process \*)a) -> CT ((struct

Process), compareAT); }

while (completedProcesses < numProcesses) {

int index = T[i]; (i) working & T[i] > 0

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

if ((process(i).AT - CT) <= (T[i] & process(i).RT) > 0)

if (process(i).AT - CT <= process(i).RT > 0)

if (process(i).AT - CT <= process(i).RT > 0)

if (miniIndex == -1)

miniIndex = i;

if (miniIndex != -1)

if (process(miniIndex).AT - CT <= process(miniIndex).RT > 0)

```

if (min Inden == -1) {
    (nichtvoll.) > RT;
    <i, gibts> abgelauf.
    CT;
    continue routine;
    & wordt durch
    if (processes[min Inden].RT != 0) {
        RT -= time;
        processes[min Inden].currentTime += time;
        (d * biov * biov time) TA response time
        completedProcesses++;
    }
    TA(i) = TA - RT;
    & ((wordt)) weiter
    pf ("Processes Completed");
    & main()
}

(d * biov time - d * biov time) TA response time
{
    TA(i) += -TA & & ((wordt)) weiter
    & main()
}

(d * biov time, d * biov time) TA response time
int numProcesses;
for (int i=0; i<numProcesses; i++) {
    processes[i].idle();
    (wordt)
    if (pf ("Ende AT")) wordt unterbrochen
    if ("AT & process(i).BT") wordt
    & ((process(i).BT) & & wordt)
    & ((process(i).idle))
    preemptive SJF (processes, numProcesses);
    & ((i))
    & ((i))
}

```

(1 -> abbrechen) &

(i) nicht nötig

Problem :-

			CT	TAT	WT	RT
P <sub>1</sub>	0	4	14	14	9	0
P <sub>2</sub>	1	3	5	4	0	0
P <sub>3</sub>	0	2	26	24	15	15
P <sub>4</sub>	0	1	10	7	2	2

P <sub>4</sub>	P <sub>3</sub>	P <sub>2</sub>	P <sub>1</sub>	P <sub>4</sub>	P <sub>2</sub>	P <sub>1</sub>	P <sub>1</sub>	P <sub>4</sub>
0	1	2	3	4	5	6	7	8 9 10

Ans.

Q<sub>1</sub> Q<sub>2</sub> Q<sub>3</sub>

## PRIORITY SCHEDULING (PREEMPTIVE)

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

struct Process {
    int pid, at, bt, pr, ct, wt, tat, rt, remaining;
};

void findPreemptivePriorityScheduling(struct Process p[], int n) {
    int completed = 0, time = 0, min_idx = -1;    float totalWT = 0,
    totalTAT = 0;

    for (int i = 0; i < n; i++) {
        p[i].remaining = p[i].bt;      p[i].rt
        = -1;
    }

    while (completed != n) {      int
        min_priority = INT_MAX;
        min_idx = -1;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining > 0 && p[i].pr < min_priority) {
                min_priority = p[i].pr;      min_idx = i;
            }
        }

        if (min_idx == -1) {
            time++;      continue;
        }
    }
}
```

```

        if (p[min_idx].rt == -1) {
p[min_idx].rt = time - p[min_idx].at;
    }

    p[min_idx].remaining--;
time++;

    if (p[min_idx].remaining == 0) {
completed++;      p[min_idx].ct =
time;
    p[min_idx].tat = p[min_idx].ct - p[min_idx].at;
p[min_idx].wt = p[min_idx].tat - p[min_idx].bt;
totalWT += p[min_idx].wt;      totalTAT +=
p[min_idx].tat;
    }
}

printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
p[i].pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
}

printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
printf("Average Waiting Time: %.2f\n", totalWT / n);
}

int main() {
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process p[n];

```

```

printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
for (int i = 0; i < n; i++) {      p[i].pid = i + 1;
    printf("Process %d: ", i + 1);
    scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].pr);
}
}

findPreemptivePriorityScheduling(p, n);

return 0;
}

```

```

C:\Users\Admin\Desktop\pric  X  +  ▾

Enter number of processes: 4
Enter Process ID, Arrival Time, Burst Time, Priority: 1 0 10 3
Enter Process ID, Arrival Time, Burst Time, Priority: 2 0 1 1
Enter Process ID, Arrival Time, Burst Time, Priority: 3 3 2 3
Enter Process ID, Arrival Time, Burst Time, Priority: 4 5 1 4

Preemptive Priority Scheduling:
PID      AT      BT      P      CT      TAT      WT
1        0       10      3      11      11       1
2        0       1       1       1       1       0
3        3       2       3      13      10       8
4        5       1       4      14       9       8

Average Waiting Time: 4.25
Average Turnaround Time: 7.75

Process returned 0 (0x0)  execution time : 63.816 s
Press any key to continue.
|
```

Priority & preemptive

first fit

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

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

```
struct Process {
```

```
    int id;
```

```
    int bt;
```

```
    int at;
```

```
    int ct;
```

```
    int tat;
```

```
    int awt;
```

```
}
```

```
int compare ArrivalTime (const void *a, const void *b)
```

```
{  
    return ((struct Process *)a) -> at - ((struct Process *)b) -> at;  
}
```

```
void calculateTimes (struct Process processes[], int n)
```

```
{
```

```
    int time = 0;
```

```
    int completed = 0;
```

```
    while (completed < n) {
```

```
        int shortest = -1;
```

```
        int min_burst = 1000000;
```

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

```
            if (processes[i].at <= min_burst) {
```

~~min\_burst = processes[i].bt;~~~~shortest = i;~~

```
}
```

```
}
```

```

if (shortest == -1) {
    time++;
} else {
    processes[shortest].wt = time + processes[shortest].bt;
    processes[shortest].tat = processes[shortest].bt;
    time = processes[shortest].et;
    completed++;
}
}
}

```

```

void calculate Avg (struct Process processes[], int n) {
    int total_wt = 0, total_tat = 0;
    for (int i=0; i < n; i++) {
        total_wt += processes[i].wt;
        total_tat += processes[i].tat;
    }
    printf ("\n avg wt = %.2f", (float)total_wt/n);
    printf ("\n avg tat = %.2f", (float)total_tat/n);
}

```

```

int main () {
    int n;
    printf ("no of processes:");
    scanf ("%d", &n);
    struct Process processes[n];
    printf ("enter bt and at:\n");
    for (int i=0; i < n; i++) {
        processes[i].id = i+1;
        printf ("bt %.d:", i+1);
        scanf ("%d", &processes[i].bt);
        printf ("at %.d:", i+1);
    }
}

```

Scanf ("y,d"), L processes (T).at;      (1 = burst time) fi  
 processes .ct = 0;                          at init.

}

3. [begin sort (processes, n, size\_of (struct process)) {  
       

      calculate Time (processes, n);      to calculate waiting time  
       calculate Avg (processes, n);      for total times  
       return 0;

}

Output

Process	AT	BT	PB	CT	TAT	WT	RT
P <sub>1</sub>	0	3	5	3	3	0	0
P <sub>2</sub>	2	3	3	11	9	7	7
P <sub>3</sub>	3	5	2	8	5	5	5
P <sub>4</sub>	4	4	4	15	11	7	7
P <sub>5</sub>	6	1	1	9	3	2	2

; (at tot. totet (total)) , "PLX = w proce/" ) fking

; (at tot - totet (total)) , "PLX = test proce/" ) fking

P <sub>1</sub>	P <sub>3</sub>	P <sub>5</sub>	P <sub>2</sub>	P <sub>4</sub>	P <sub>3</sub> (min. tie)
0	3	8	9	11	15

Enter the no of processes (" : bX ") fking

Enter the burst time for each process for each process (" : bX fd ") fking

2(i+1) > i (0 < i < j) ref

i+1 > for [i] processes

; (i+1 : bX fd) fking

; (td. [?] answer & " bX ") fking

; (i+1 : bX to) fking

## PRIORITY SCHEDULING (NON PREEMPTIVE)

```
#include <stdio.h>

struct Process {
    int pid, at, bt, pr, ct, wt, tat, rt;
    int isCompleted; // Flag to check if process is completed
};

void sortByArrival(struct Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].at > p[j].at) {
                struct Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}

void findPriorityScheduling(struct Process p[], int n) {
    sortByArrival(p, n);
    int time = 0, completed = 0;
    float totalWT = 0, totalTAT = 0;

    while (completed < n) {
        int idx = -1, highestPriority = 9999;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].isCompleted == 0) {
                if (p[i].pr < highestPriority) {
                    highestPriority = p[i].pr;
                    idx = i;
                }
            }
        }
    }
}
```

```

    }

    if (idx == -1) {
time++; // CPU idle
    } else {
        p[idx].rt = time - p[idx].at;
time += p[idx].bt;      p[idx].ct =
time;      p[idx].tat = p[idx].ct -
p[idx].at;      p[idx].wt = p[idx].tat -
p[idx].bt;      p[idx].isCompleted =
1;

        totalWT += p[idx].wt;
totalTAT += p[idx].tat;
completed++;
    }
}

printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
p[i].pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
}

printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
printf("Average Waiting Time: %.2f\n", totalWT / n);
}

int main() {
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);  struct Process p[n];
}

```

```

printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
for (int i = 0; i < n; i++) {      p[i].pid = i + 1;
    printf("Process %d: ", i + 1);
    scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].pr);
    p[i].isCompleted = 0;
}

findPriorityScheduling(p, n);

return 0;
}

```

```

Enter number of processes: 5
Enter Arrival Time (AT), Burst Time (BT) & Priority for process 1: 0 3 5
Enter Arrival Time (AT), Burst Time (BT) & Priority for process 2: 2 2 3
Enter Arrival Time (AT), Burst Time (BT) & Priority for process 3: 3 5 2
Enter Arrival Time (AT), Burst Time (BT) & Priority for process 4: 4 4 4
Enter Arrival Time (AT), Burst Time (BT) & Priority for process 5: 6 1 1

Process AT      BT      PT      CT      TAT      WT      RT
1      0        3        5        3        3        0        0
2      2        2        3       11        9        7        7
3      3        5        2        8        5        0        0
4      4        4        4       15       11        7        7
5      6        1        1        9        3        2        2

Average WT: 3.20
Average TAT: 6.20

Process returned 0 (0x0)   execution time : 51.636 s
Press any key to continue.
|

```

temp = process[i]; wait until ready time?

process[i] = process[j];

process[j] = temp;

}

}

3. [o] average, [o] waiting, [o] total time

wait[0] = 0; current time [o] time till

completion[0] = burst[0]; i.o. idle time

turnaround[0] = completion[0]; i.o. latest time

total - turnaround <sup>total burst</sup> = waiting

for (int i=1; i < n; i++) {

wait[i] = wait[i-1] + burst[i];

completion[i] = completion[i-1] + burst[i];

turnaround[i] = completion[i];

total - wait += wait[i];

total - turnaround += turnaround[i];

i + i = [i] average

printf("\n Process \t Burst Time \t Priority\n \t Turnaround Time\n"); o = o + 1;

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

printf("%d \t %d \t %d \t %d \t %d \t %d \n", process[i], burst[i], priority[i], wait[i], turnaround[i]);

} i.o. average = [i] average

printf("\n Average Waiting Time : %.2f ", (float) totalwait / n);

printf("\n [i] turnaround : [i] average + wait

printf("\n Average Turnaround Time : %.2f \n", (float) totalturnaround / n);

\* 3 return 0;

## Priority Scheduling Non Preemptive - part

: (i) arrival = (i) priority

#include <stdio.h> : qsort = (i) arrival

int main () {

int n, i, j;

int burst [20], priorities [20], process [20];

int wait [20], int turnaround [20], time  
completion [20];

int total\_wait = 0, total\_turnaround = 0;

printf ("Enter the number of processes: ");

scanf ("%d", &n);

printf ("Enter burst time and priorities for  
each process: \n");

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

printf ("Process %d: time, i+1);

scanf ("%d", &burst[i]), &priority[i]);

process[i] = i+1;

}

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

for (j=i+1; j<n; j++) {

if (priority[i] < priority[j]) {

temp = priority[i];

priority[i] = priority[j];

priority[j] = temp;

" fix: print process[i] " + time

temp = burst[i];

burst[process[i]] = burst[i];

" / fix: wait (process[i]) > temp " + time

: (n-1) turnaround - total (total)

< 0 reenter

#### EARLIEST DEADLINE CODE:

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

typedef struct {
    int pid;
    int burst_time;
    int deadline;
    int period;
    int remaining_time;
} Process;

// Function to compare processes based on their deadlines
int compare_deadlines(const void *a, const void *b) {
    return ((Process *)a)->deadline - ((Process *)b)->deadline;
}

int main() {
    int num_processes;

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

    Process processes[num_processes];

    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < num_processes; i++) {
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
    }
}
```

```

printf("Enter the deadlines:\n");    for
(int i = 0; i < num_processes; i++) {
scanf("%d", &processes[i].deadline);
}

printf("Enter the time periods:\n");
for (int i = 0; i < num_processes; i++) {
scanf("%d", &processes[i].period);
}

// Sort processes based on deadlines
qsort(processes, num_processes, sizeof(Process), compare_deadlines);

printf("\nEarliest Deadline Scheduling:\n");
printf("PID\tBurst\tDeadline\tPeriod\n");    for
(int i = 0; i < num_processes; i++) {
printf("%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].burst_time, processes[i].deadline,
processes[i].period);
}

int current_time = 0;    int
completed_processes = 0;

printf("\nScheduling occurs for %d ms\n", processes[0].deadline); // Assuming the first process
has the earliest deadline

while (completed_processes < num_processes) {
for (int i = 0; i < num_processes; i++) {        if
(processes[i].remaining_time > 0) {
printf("%dms : Task %d is running.\n", current_time, processes[i].pid);
processes[i].remaining_time--;
current_time++;
}
}
}

```

```

        if (processes[i].remaining_time == 0) {
completed_processes++;
    }
}
}

printf("\nProcess returned %d (0x%X)\texecution time : %.3f s\n", current_time, current_time,
(float)current_time / 1000.0);

return 0;
}

```

```

C:\Users\Admin\Downloads\early.exe
Enter the number of processes: 3
Enter the CPU burst times:
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3

Earliest Deadline Scheduling:
PID      Burst      Deadline      Period
1          2          1              1
2          3          2              2
3          4          3              3

Scheduling occurs for 1 ms
0ms : Task 1 is running.
1ms : Task 2 is running.
2ms : Task 3 is running.
3ms : Task 1 is running.
4ms : Task 2 is running.
5ms : Task 3 is running.
6ms : Task 2 is running.
7ms : Task 3 is running.
8ms : Task 3 is running.

Process returned 9 (0x9)      execution time : 0.009 s

Process returned 0 (0x0)      execution time : 15.281 s
Press any key to continue.

```

```

Earliest Deadline) Firest, bt - (i)q dead.
#include <stdio.h> struct ( tree) : waitability.
type def struct {
    int (pid, BT, RT, wq) * a : bready low
    } process; ((b) : waitability a " ) }

int compare deadline b (context waitability)
return ((process *)) a >= deadline ((process))
deadline;
}

int main () {
    int main () {
        int num - processes; ("bares. test") }

Pf ("Enter no of processes. ");
process process [num - processes]; n tie
Pf ("Enter (P|PT or") ist (rt) }
for i = 0; i < num (n processes) (i). BT );
process (i). pid = i + 1; (a) deadl dead
process (i). RT = process (i). RT
process (i). BT }

}
; (++i: n) i : 0 = i tie) }

Pf (" Enter deadline: "); pi . (i) & deadl
for (b = 0; i < num - processes ; i++) {
scanf ("%d", & process (i). deadline);
}

Pf (" Enter Time: ");
for (int i = 0; i < num ; i++) {
scanf ("%d", & process (i). period);
}

```

qsort ( processes, num\_processes, size\_of (processes) );

pf ("In & Enter .%d") : { S S  
S S S S

processes (i). pid , processes (i). BT

process (i). deadline , process (i). period );

} end of main function

int CT = 0;

int completed\_processes = 0 ;

pf ("In Scheduling occurs for P.%d ms"),

processes (0). deadline );

while ( completed\_processes < num\_processes ) {

for ( int i=0 ; i < num ; processes[i]++ )

{ if (process (i). RT ) {

printf ("%.d ms > task :%.d");

current - Time, process (i). pid );

current - Time++ ;

if (processes (i). RT == 0 ) {

completed\_processes++ ;

};};

Output :-

Enters no of process : 3

2 3 4  
1 2 3

System : 6 ms

(P2D) Burst Deadline and Periodic )

- 1      2      1 : (b1, 243  $\beta$  01") )
  - 2      3      2 : (b2, 243  $\beta$  01") )
  - 3      4 : (b3, 243  $\beta$  01") )
- (b1, (i) waiting, b2, (i) waiting  
• (b3, (i) waiting, b4, (i) waiting)

Scheduling occurs for 6 ms :

0ms : Task1 : (is running, b1, b2, b3, b4)

1ms : Task1 : (is running, b1, b2, b3, b4)  
• (b1, (i) Task1 exec is finished, b2, b3, b4)

2ms : Task2 : (is running, b2, b3, b4)

3ms : Task2 : (is running, b3, b4)

4ms : Task2 : (is running, b4)

5ms : Task3 : (is running, b1, b2, b3, b4)

6ms : Task3 : (is running, b1, b2, b3, b4)

• (b1, (i) deadline, b2, b3, b4)

## RATE-MONOTONIC

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

typedef struct {    int
id, burst, period;
} Task;

int gcd(int a, int b) {    return (b ==
0) ? a : gcd(b, a % b);
}

int lcm(int a, int b) {
return (a * b) / gcd(a, b);
}

int findLCM(Task tasks[], int n) {    int
result = tasks[0].period;    for (int i = 1;
i < n; i++)        result = lcm(result,
tasks[i].period);    return result;
}

void rateMonotonic(Task tasks[], int n) {
float utilization = 0;
printf("\nRate Monotonic Scheduling:\nPID\tBurst\tPeriod\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\n", tasks[i].id, tasks[i].burst, tasks[i].period);
    utilization += (float)tasks[i].burst / tasks[i].period;
}
float bound = n * (pow(2, (1.0 / n)) - 1);
```

```

    printf("\nUtilization: %.6f, Bound: %.6f\n", utilization, bound);
if (utilization <= bound)      printf("Tasks are Schedulable\n");
else
    printf("Tasks are NOT Schedulable\n");
}

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

Task tasks[n];
printf("Enter the CPU burst times: ");
for (int i = 0; i < n; i++)
scanf("%d", &tasks[i].burst);

printf("Enter the time periods: ");
for (int i = 0; i < n; i++) {
scanf("%d", &tasks[i].period);
tasks[i].id = i + 1;
}

rateMonotonic(tasks, n);
return 0;
}

```

```
C:\Users\Admin\Documents\r  X + | v

Enter the number of processes: 3
Enter the CPU burst times:
3 6 8
Enter the time periods:
3 4 5
LCM=60
Rate
Monotone Scheduling:
PID    Burst    Period
1      3        3
2      6        4
3      8        5
4.100000 <= 0.779763 =>false
Process returned 0 (0x0)
Press any key to continue.
3
execution time : 35.633 s

Process returned 0 (0x0)  execution time : 35.649 s
Press any key to continue.
|
```

(initial Rate Monotonic scheduling) nicht besser hin  
 #include <stdio.h>, <conio.h> //d - rinner bis  
 #define MAX\_processes 10 // (i) d - rinner  
 type def struct {  
 int burst\_time[0] = 0; // tis  
 int id; // (n > qmax) abtis  
 int BT; // (o - latencies tis  
 } Task;  
 int gcd (int a, int b) {  
 return (b == 0) ? a : gcd(b, a % b);  
 }  
 int lcm (int a, int b) {  
 return (a + b) / gcd(a, b);  
 }  
 int find (Task tasks[], int n) {  
 int result = task(0).period;  
 for (int i = 1; i < n; i++)  
 result = lcm(result, task(i).period);  
 return result;
 }  
 int find\_LCM (Task tasks[], int n) {  
 int result = tasks(0).period;  
 for (int i = 1; i < n; i++)  
 result = lcm(result, task(i));
 }  
 void rate\_Monotonic (Task tasks[], int n) {  
 float utilization = 0.0; // 0.050000  
 pf ("In Rate Monotonic Scheduling: \n");
 Burst period \n);
 for (int i = 0; i < n; i++) {
 pf ("y. J task (%d)\n", i);
 for (int j = 0; j < tasks(i).q; j++) {
 if (tasks(i).q == j + 1)
 utilization += tasks(i).BT;
 }
 }
 }

task p(i).bd, tasks(p(i).bd period);

utilization = (float) tasks(i).bd / p(i).bd;

}    } build job info

float bound = n \* (pawfiz, fij, off, p) tis

pf ("In utilization : %d");

utilization bound, jobs info

pf ("Tasks are scheduled by ");

else

pf ("Task . period");

}

pf ("Tasks.period");

}    } p() from tri

int main () {

int n; [2023-03-01] 2023-03-01

pf ("Enter the no of process : ");

if ((i <= 0) || (n > i)) pf

Task tasks(n); i + i = big. (i) error

pf ("Enter the P/U Bucket Time : ");

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

tasks(i).id = (i + 1) build info

} (i + i = error - max > i = 0 - i) ref

date monotonic (tasks, n) by " ) fresh

return 0;

}

; ("c: will exit")

} (+i; max > i = 0; i tri) ref

; (bavay. (i) error & ("bV") fresh

```
printf("!!! Deadline miss for p%d at  
time %d\n", p[currentProcess].pid,  
time + 1);
```

y

else

```
printf("%d is IDLE\n", time);
```

y

output.

Enter the number of processes : 3

Enter the CPU burst time;

3 6 8

Enter the time periods:

3 4 5

LCM = 60

Rate Monotone Scheduling :

PID	Burst Period	
1	3	3
2	6	4
3	8	5

$$4 \cdot 1000000 \cdot 3 <= 0.779763 \Rightarrow \text{false}.$$

179

## PRODUCER CONSUMER CODE:

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

int mutex = 1, full = 0, empty, x = 0;
int *buffer, buffer_size; int in = 0,
out = 0;

int wait(int s) {
    return (--s);
}

int signal(int s) {
    return (++s);
}

void producer(int id) {    if ((mutex ==
1) && (empty != 0)) {        mutex =
wait(mutex);        full = signal(full);
empty = wait(empty);        int item =
rand() % 50;        buffer[in] = item;
x++;
printf("Producer %d produced %d\n", id, item);
printf("Buffer:%d\n", item);        in = (in + 1) %
buffer_size;        mutex = signal(mutex);
} else {
    printf("Buffer is full\n");
}
}
```

```

void consumer(int id) {    if
((mutex == 1) && (full != 0)) {
mutex = wait(mutex);      full =
wait(full);      empty =
signal(empty);      int item =
buffer[out];
printf("Consumer %d consumed %d\n", id, item);
x--;
printf("Current buffer len: %d\n", x);
out = (out + 1) % buffer_size;      mutex
= signal(mutex);
} else {
printf("Buffer is empty\n");
}
}

int main() {    int producers, consumers;
printf("Enter the number of Producers:");
scanf("%d", &producers);    printf("Enter
the number of Consumers:");    scanf("%d",
&consumers);    printf("Enter buffer
capacity:");    scanf("%d", &buffer_size);

buffer = (int *)malloc(sizeof(int) * buffer_size);
empty = buffer_size;

for (int i = 1; i <= producers; i++)
printf("Successfully created producer %d\n", i);    for
(int i = 1; i <= consumers; i++)
printf("Successfully created consumer %d\n", i);

srand(time(NULL));

```

```
int iterations = 10;    for (int i =  
0; i < iterations; i++) {  
producer(1);      sleep(1);  
consumer(2);      sleep(1);  
}
```

```
free(buffer);  
return 0;  
}
```

```
Buffer:47
Consumer 2 consumed 47
Current buffer len: 0
Producer 1 produced 9
Buffer:9
Consumer 2 consumed 9
Current buffer len: 0
Producer 1 produced 44
Buffer:44
Consumer 2 consumed 44
Current buffer len: 0
Producer 1 produced 30
Buffer:30
Consumer 2 consumed 30
Current buffer len: 0
Producer 1 produced 1
Buffer:1
Consumer 2 consumed 1
Current buffer len: 0
Producer 1 produced 14
Buffer:14
Consumer 2 consumed 14
Current buffer len: 0
Producer 1 produced 48
Buffer:48
Consumer 2 consumed 48
Current buffer len: 0
Producer 1 produced 2
Buffer:2
Consumer 2 consumed 2
Current buffer len: 0
Producer 1 produced 31
Buffer:31
Consumer 2 consumed 31
Current buffer len: 0
```

Producer consumes system

;(buf) size = buf

```

#include <stdio.h> // for I/O operations
#include <stdlib.h> // for memory allocation
int mutex = 1, full = 0, empty = 0, item;
int *buffer, bufferSize; // buffer size
int in = 0, out = 0;
int wait (int l) {
    return (l); } // Union of
    return (c); } // reservation - reserving, true
    signal (l); } // releasing - releasing, false
void produce (int id) {
    if ((mutex == 1) && (empty != 0)) {
        mutex = wait (mutex);
        full = wait (empty);
        int item = void (); // get
        buffer (in) = item; // put
        in++; // increment
        printf ("produces %d produced by %d\n");
        printf ("Buffer : %d item\n");
        item = (in + 1) % bufferSize;
        mutex = signal (mutex);
    } else {
        printf ("Buffer is full\n");
    }
}
void consume (int in) {
    if (mutex == 1 && full != 0) {
        mutex = wait (mutex);
        full = wait (empty);
        int item = buffer (in);
        buffer (in) = void (); // remove
        in++; // increment
        printf ("consumes %d consumed by %d\n");
    }
}

```

```

    mutex = wait (mutex);
    full = wait (full);
    empty = signal (empty);
    int item = buffer (read);
    print f ("consumes %d consumed %d");
    x--;
    o = two, o = one;
    {
        } (1 time) times ten
    int main () {
        : (3) nested
        int producer - consumer;
        pf ("Enter the no of producer");
        sf ("%d", &producer) + nested;
        pf ("Enter the buffer capacity");
        sf ("%d, & buffer");
        buffer = (int *) malloc (buffer - size);
        : (return) buffer = new
        : (return) times = ten
        : (return) times = ten
        : (return) times = ten
        Output is
        1. produces
        2. consumes
        3. Enter your choice b/n: effed
        Enter your choice b/n: effed
        Produces produce item 1
        Enter your choice b/n: effed
        Buffer is full
        Enter your choice b/n: effed
        consumer consumes item 3
        {
            } (n times) consumed b/n
        : (n times) consumed b/n
    }

```

## DINNING PHILOSOPHER CODE:

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

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

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

sem_t mutex; sem_t
S[N];

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

        state[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);
}

void put_fork(int phnum)
{
    sem_wait(&mutex);

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

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philosopher(void* num)
{

```

```

int* i = (int*) num;

while (1) {
    sleep(1);
    take_fork(*i);
    sleep(1);
    put_fork(*i);
}

int main()
{
    int i;
    pthread_t thread_id[N];

    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);

    return 0;
}

```

```
C:\Users\Admin\Desktop\din> + ^

DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 4
How many are hungry: 4
Enter philosopher 1 position: 1
Enter philosopher 2 position: 2
Enter philosopher 3 position: 3
Enter philosopher 4 position: 4

1. One can eat at a time  2. Two can eat at a time  3. Exit
Enter your choice: 1

Allow one philosopher to eat at any time
P 1 is granted to eat
P 2 is waiting
P 3 is waiting
P 4 is waiting
P 2 is granted to eat
P 1 is waiting
P 3 is waiting
P 4 is waiting
P 3 is granted to eat
P 1 is waiting
P 2 is waiting
P 4 is waiting
P 4 is granted to eat
P 1 is waiting
P 2 is waiting
P 3 is waiting

1. One can eat at a time  2. Two can eat at a time  3. Exit
Enter your choice: 2

Allow two philosophers to eat at same time
combination 1
P 1 and P 3 are granted to eat
P 2 is waiting
P 4 is waiting

combination 2
P 2 and P 4 are granted to eat
P 1 is waiting
P 3 is waiting

1. One can eat at a time  2. Two can eat at a time  3. Exit
Enter your choice: |
```

## Dining Philosophers

```
#include < stdio.h > // header file for stdio.h
#include < stdlib.h > // header file for stdlib.h
int Total_philosophers = 6; // total number of philosophers
int hungry(max); // max number of philosophers hungry at once
return abs(a-b) == 1 || abs(a-b) == 2;
total philosopher - 1; // total number of philosophers
void option1(int count) {
    printf("In Allows one philosopher to eat
at any time");
    for (int i=0; i<count; i++) {
        printf("%d is granted to eat(%n)");
        hungry(i);
        for (int j=0; j<count; j++) {
            if (j != i) {
                printf("%d is waiting");
                hungry(j);
            }
        }
    }
}

void option2(int count) {
    printf("In Allows two philosophers to eat at
the same time");
    int combination = 1;
    for (int i=0; i<count; i++) {
        for (int j=0; j<count; j++) {
            if (!are Neighbours(hungry[i], hungry[j])) {
                combination++;
            }
        }
    }
}
```

adversarial print( )

printf ("Combination %d"), i; // this is to  
 if ( $K_1 = 8 \& K_2 = j$ ) {  
 printf ("P.%d is winning position") total bin  
 hungry (K); : ( ; ) represent bin  
 } c = (d - o) % D; l = (d - o) / D; enter  
 } : ( ; ) represent total

printf ("\n"); : ( ; )

if (combination == 1) {  
 printf ("No combination found where ") rep  
 two non-negative philosophies.  
 : ("concat: \n"); : ( ; ) represent
 }

int main () {  
 int hungry count (initial is 61'); : ( ; )  
 printf ("Enter the total no of philosophies");  
 scanf ("%d", &total philosopher);  
 do {  
 int choice; : (two) tri) to judge bin  
 do {  
 printf ("Adversarial out with n") first  
 pf ("In One concat aboring (n").  
 2. Now can eat a line below tree  
 3. Exit "); : ( ; ) rep  
 pf ("Enter your choice "); : ( ; ) rep  
 sf ("%d", &choice); : ( ; ) rep  
 if (choice > 1) break;
 }
 }
 }

case 1 : option 1 (hungry count) ;  
break ;

straight forward

(i > i) > i < i

(i - i) > i < i

case 2. i, iti, drow\* tri, bora\* tri) waitress tri.

option 2 (hungry count) ;  
break ;

{(++i < i < i = i) tri} ref.

case 2 :

print f ("Empty In");

break

{i < i = i}

default :

print f ("Invalid choice n");

{

}

while (choice != 3) { i++ ; tri = bora \* tri }

return 0 ;

{ (tri) for i < tri } set bits (i < tri), [i] bora

Output : { (++i < i < i = i) tri } ref.

[i][i] bora. [i][i] more = [i][i] bora

driving Philosophers problem

Enters the total no of philosopher : 5

How many philosophers are hungry (i tri) = bora \* tri

Peter philosopher 1 pos 2

Peter philosopher 2 pos 4

Enter philosopher 3 pos 5 = bora \* tri

Allow one philosopher to eat at any time

P2 is allowed. [i] P5 is allowed

P4 is allowed P2 is allowed

P5 is allowed P4 is allowed

P4 is allowed P2 is allowed

P2 is allowed

#### DEADLOCK AVOIDANCE CODE:

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

#define P 5
#define R 3

bool isSafe(int avail[], int max[][R], int allot[][][R]) {
    int need[P][R];  bool finish[P] = {false};  int
    safeSeq[P];  int work[R];

    for (int i = 0; i < P; i++) {      for
        (int j = 0; j < R; j++) {          need[i][j]
        = max[i][j] - allot[i][j];
    }
}

    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]) {          bool
                canAllocate = true;          for
                (int r = 0; r < R; r++) {
                    if (need[p][r] > work[r]) {

```

```

canAllocate = false;
break;
}

}

if (canAllocate) {
for (int r = 0; r < R; r++) {
work[r] += allot[p][r];
}
safeSeq[count++] = p;
finish[p] = true;           found
= true;                   break;
}
}

if (!found) {
printf("System is in an unsafe state\n");
return false;
}

printf("System is in a safe state\nSafe sequence is: ");
for (int i = 0; i < P; i++) {   printf("P%d ",
safeSeq[i]);
}
printf("\n");

return true;
}

int main() {

```

```
int avail[] = {3, 3, 2};
```

```
int max[][][R] = {  
    {7, 5, 3},  
    {3, 2, 2},  
    {9, 0, 2},  
    {2, 2, 2},  
    {4, 3, 3}  
};
```

```
int allot[][][R] = {  
    {0, 1, 0},  
    {2, 0, 0},  
    {3, 0, 2},  
    {2, 1, 1},  
    {0, 0, 2}  
};
```

```
isSafe(avail, max, allot);
```

```
return 0;
```

```
}
```

```
System is in a safe state  
Safe sequence is: P1 P3 P0 P2 P4
```

## Deadlock Avoidance

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, m;
    pf("process & resources : \n");
    sf("%d %d", &n, &m);
    int alloc[n][m], max[n][m], avail[m];
    med[n][m];
    printf("Enter allocation matrix : \n");
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            sf("%d", &alloc[i][j]);
        }
    }
    printf("Enter available matrix \n");
    for (int i=0; i<m; i++) {
        sf("%d", &avail[i]);
    }
    need[n][m];
    for (int i=0; i<n; i++) {
        need[i][j] = max[i][j] - alloc[i][j];
    }
    bool finish[n];
    for (int i=0; i<n; i++) {
        finish[i] = false;
    }
    int safe_seq[n];
    int count=0;
    while (count < n) {
        bool found=false;
        for (int p=0; p<n; p++) {
            if (!finish[p]) {
                bool can_alloc=true;
                for (int j=0; j<m; j++) {
                    if (need[p][j] > avail[j]) {
                        can_alloc=false;
                    }
                }
                if (can_alloc) {
                    safe_seq[count]=p;
                    count++;
                    for (int j=0; j<m; j++) {
                        avail[j] -= need[p][j];
                    }
                    finish[p]=true;
                }
            }
        }
    }
}
```

```

        } marking & labeling
if (can Allocate) {
    for (int k = 0; K < m; K++) (k, avail[k] > 0) available
        avail [K] += alloc [p] [K]; (avail[K] > 0) available
    safe & seq (count++) = p; ; count++
    finish (p) = true; ; max no. of process
    found = true; ; found
}
} ((n, m) & "b.v")
{ (n) lines, (m) columns, (n)(m) safe seq
    ((n)(m) b.v)
}
if (!found) { (n > m : system not allocate sufficient) failing
    printf ("System is not in safe scheduling"); (n > m : i > m) ref.
    return 1; ((i) won't be in "b.v" file)
}
} ((n > m : system allocate sufficient) failing
Output. (n > m : i > m) ref.
Enter (no. of processes) & (resources) ((i), b.v)
5 3 (5) input from
Enter allocation matrix (5 > i > m) ref.
0 1 0 (0 > i > m) ref.
2 0 0 (2 > i > m) ref.
3 0 2 (3 > i > m) ref.
0 0 2 (0 > i > m) ref.
safe & seq
P1 → P3 → P4 → P0 → P2, b.v ref
Enter max & min (5 > i > m) ref.
max 5 min 2 2 2 2
{ (1 2 2) → i, { 0 4 3 } } ref
safe & seq (5 > i > m) ref

```

#### DEADLOCK DETECTION CODE:

```
#include <stdio.h>

int main() {
    int n, m, i, j, k;
    printf("Enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);

    int alloc[n][m], req[n][m], avail[m], finish[n];

    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)      for (j = 0; j
    < m; j++)      scanf("%d",
    &alloc[i][j]);

    printf("Enter request matrix:\n");
    for (i = 0; i < n; i++)      for (j =
    0; j < m; j++)      scanf("%d",
    &req[i][j]);

    printf("Enter available matrix:\n");
    for (i = 0; i < m; i++)
    scanf("%d", &avail[i]);

    for (i = 0; i < n; i++)
    finish[i] = 0;

    int done;    do {
    done = 0;      for (i = 0; i
    < n; i++) {      if
    (finish[i] == 0) {
```

```

int canFinish = 1;
for (j = 0; j < m; j++) {
if (req[i][j] > avail[j]) {
canFinish = 0;
break;
}
}
if (canFinish) {
for (j =
0; j < m; j++)
avail[j]
+= alloc[i][j];
finish[i]
= 1;
done = 1;
printf("Process %d can finish.\n", i);
}
}
}

}

} while (done);

int deadlock = 0;
for (i = 0; i < n; i++)
if (finish[i] == 0)
deadlock = 1;

if (deadlock)
printf("System is in a deadlock state.\n");
else
printf("System is not in a deadlock state.\n");

return 0;
}

```

```
Enter number of processes and resources:  
5 3  
Enter allocation matrix:  
0 1 0  
2 0 0  
3 0 2  
2 1 1  
0 0 2  
Enter request matrix:  
7 5 3  
3 2 2  
9 0 2  
2 2 2  
4 3 3  
Enter available matrix:  
3 3 2  
Process 1 can finish.  
Process 3 can finish.  
Process 4 can finish.  
System is in a deadlock state.  
  
Process returned 0 (0x0)  execution time : 47.372 s  
Press any key to continue.
```

```

Deadlock detector
#include <stdio.h>
#include <stdlib.h>
#define P 5
#define R 3
int (deadlock_detector)(int allocation[R][P],
                         int available[R],
                         int work[R]);
void deadlock_detector(int allocation[R][P],
                       int available[R])
{
    int work[R];
    bool finished[R] = {false}; // initial state
    bool deadlock = false; // deadlock found
    for (int i = 0; i < R; i++) {
        work[i] = available[i]; // initial state
    }
    int count = 0;
    while (count < P) {
        bool found = false; // found a process
        for (int i = 0; i < R; i++) {
            if (!finished[i]) {
                bool can_proceed = true;
                for (int j = 0; j < R; j++) {
                    if (allocation[i][j] > work[j]) {
                        can_proceed = false;
                        break; // no available resources
                    }
                }
                if (can_proceed) {
                    work[i] += allocation[i];
                    finished[i] = true; // mark as finished
                    printf("Process %d finished in %d\n", i);
                    deadlock = false;
                } else {
                    found = true;
                }
            }
        }
        if (found) {
            count++;
        } else {
            deadlock = true;
        }
    }
}

```

```

if (!ford) {
    make;
    for (int i = 0; i < p; i++) {
        if (!finished[i]) {
            if (!deadlock[i]) {
                printf("No deadlock detected. All processes are complete.");
            } else {
                printf("Deadlock detected in process %d.", i);
            }
        }
    }
    int allocation[R][R] = {
        {0, 1, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}
    };
    int request[R][R] = {
        {0, 1, 1}, {0, 0, 1}, {0, 0, 0}, {0, 0, 0}
    };
    int available[R][R] = {
        {1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {0, 0, 0}
    };
    if (!deadlock[0]) {
        deadlock[0] = (allocation[0] > request[0]);
    }
    if (!deadlock[1]) {
        deadlock[1] = (allocation[1] > request[1]);
    }
    if (!deadlock[2]) {
        deadlock[2] = (allocation[2] > request[2]);
    }
    if (!deadlock[3]) {
        deadlock[3] = (allocation[3] > request[3]);
    }
    if (!deadlock[0] & !deadlock[1] & !deadlock[2] & !deadlock[3]) {
        return 0;
    }
}

Output: [P1 P2 P3 P4]
Process P0 is finished. No deadlock.
Process P1 is finished being deleted.
Process P2 is finished being All processes
Process P3 is finished being
Process P4 is finished being

```

## MULTILEVEL QUIENING CODE:

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define TIME_QUANTUM 2

typedef struct {
    int burst_time, arrival_time, queue_type, waiting_time, turnaround_time, response_time,
        remaining_time;
} Process;

void round_robin(Process processes[], int n, int time_quantum, int *time) {
    int done, i;    do {      done = 1;
        for (i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time > time_quantum) {
                    *time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
                } else {
                    *time += processes[i].remaining_time;
                    processes[i].waiting_time = *time - processes[i].arrival_time - processes[i].burst_time;
                    processes[i].turnaround_time = *time - processes[i].arrival_time;
                    processes[i].response_time = processes[i].waiting_time;           processes[i].remaining_time =
                0;
            }
        }
    } while (!done);
}

void fcfs(Process processes[], int n, int *time) {
    for (int i = 0; i < n; i++) {
```

```

        if (*time < processes[i].arrival_time) {
            *time = processes[i].arrival_time;
        }
        processes[i].waiting_time = *time - processes[i].arrival_time;
        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
        processes[i].response_time = processes[i].waiting_time;
        *time += processes[i].burst_time;
    }
}

int main() {
    Process processes[MAX_PROCESSES], system_queue[MAX_PROCESSES],
    user_queue[MAX_PROCESSES];
    int n, sys_count = 0, user_count = 0, time = 0;
    float avg_waiting = 0, avg_turnaround = 0, avg_response = 0, throughput;

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

    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
        &processes[i].queue_type);
        processes[i].remaining_time = processes[i].burst_time;

        if (processes[i].queue_type == 1) {
            system_queue[sys_count++] = processes[i];
        } else {
            user_queue[user_count++] = processes[i];
        }
    }
}

```

```

// Sort user processes by arrival time for FCFS
for (int i = 0; i < user_count - 1; i++) {
    for
(int j = 0; j < user_count - i - 1; j++) {
        if (user_queue[j].arrival_time > user_queue[j + 1].arrival_time) {
Process temp = user_queue[j];
                user_queue[j] = user_queue[j +
1];
                user_queue[j + 1] = temp;
}
}
}

printf("\nQueue 1 is System Process\nQueue 2 is User Process\n");
round_robin(system_queue, sys_count, TIME_QUANTUM, &time); fcfs(user_queue,
user_count, &time);

printf("\nProcess Waiting Time Turn Around Time Response Time\n");

for (int i = 0; i < sys_count; i++) {
    avg_waiting += system_queue[i].waiting_time;
avg_turnaround += system_queue[i].turnaround_time;
avg_response += system_queue[i].response_time;
    printf("%d      %d      %d      %d\n", i + 1, system_queue[i].waiting_time,
system_queue[i].turnaround_time, system_queue[i].response_time);
}

for (int i = 0; i < user_count; i++) {
    avg_waiting
+= user_queue[i].waiting_time;    avg_turnaround +=
user_queue[i].turnaround_time;    avg_response +=
user_queue[i].response_time;
    printf("%d      %d      %d      %d\n", i + 1 + sys_count, user_queue[i].waiting_time,
user_queue[i].turnaround_time, user_queue[i].response_time);
}

```

```
    avg_waiting /= n;
    avg_turnaround /= n;
    avg_response /= n;    throughput
    = (float)n / time;

    printf("\nAverage Waiting Time: %.2f", avg_waiting);
    printf("\nAverage Turn Around Time: %.2f", avg_turnaround);
    printf("\nAverage Response Time: %.2f", avg_response);
    printf("\nThroughput: %.2f", throughput);

    printf("\nProcess returned %d (0x%x) execution time: %.3f s\n", time, time, (float)time);

    return 0;
}
```

```
Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 2
0
1
Enter Burst Time, Arrival Time and Queue of P2: 1
0
2
Enter Burst Time, Arrival Time and Queue of P3: 5
0

1
Enter Burst Time, Arrival Time and Queue of P4: 3
0

2

Queue 1 is System Process
Queue 2 is User Process

Process Waiting Time    Turn Around Time    Response Time
1      0                  2                  0
2      2                  3                  2
3      3                  8                  3
4      8                  11                 8

Average Waiting Time: 3.25
Average Turn Around Time: 6.00
Average Response Time: 3.25
Throughput: 0.36
```

ROUND ROBIN CODE:

```

#include <stdio.h>

#define MAX_PROCESSES 10

typedef struct {    int pid;    int bt;
                    int at;    int queue; } Process; void

sortByArrival(Process p[], int n) {

Process temp;    for (int i = 0; i < n - 1;
i++) {          for (int j = 0; j < n - i -
1; j++) {      if (p[j].at > p[j + 1].at) {
temp = p[j];          p[j] = p[j + 1];
p[j + 1] = temp;
}
}
}

void roundRobin(Process p[], int n, int quantum, int wt[], int tat[], int rt[])
{
    int remaining_bt[MAX_PROCESSES];
    for (int i = 0; i < n; i++)    remaining_bt[i] =
p[i].bt;    int t = 0, completed = 0;    while
(completed < n) {    int executed = 0;
for (int i = 0; i < n; i++) {        if

```

```

(remaining_bt[i] > 0) {           if (rt[i]

== -1) rt[i] = t;               if

(remaining_bt[i] > quantum) {      t

+= quantum;                   remaining_bt[i] -

= quantum;

} else {

t += remaining_bt[i];

tat[i] = t - p[i].at;           wt[i]

= tat[i] - p[i].bt;

remaining_bt[i] = 0;
completed++;

}

executed = 1;
}

if (!executed) t++;

}

}

void fcfs(Process p[], int n, int start_time, int wt[], int tat[], int rt[])
{
    int time = start_time;    for (int i = 0; i < n; i++) {      if (time

```

```

< p[i].at) time = p[i].at;           rt[i] = time - p[i].at;           wt[i] =
rt[i];           tat[i] = wt[i] + p[i].bt;           time += p[i].bt;

}

}

int main() {   int

n, quantum;

Process p[MAX_PROCESSES], sys[MAX_PROCESSES], usr[MAX_PROCESSES];

int sys_count = 0, usr_count = 0;

int wt[MAX_PROCESSES], tat[MAX_PROCESSES], rt[MAX_PROCESSES];

printf("Enter number of processes: ");

scanf("%d", &n); for (int i = 0; i < n;

i++) {

    printf("Enter Burst Time, Arrival Time and Queue (1=System, 2=User) for P%d: ", i + 1);    p[i].pid = i + 1;    scanf("%d %d %d", &p[i].bt,
&p[i].at, &p[i].queue);    if (p[i].queue == 1)

sys[sys_count++] = p[i];    else

usr[usr_count++] = p[i];

    wt[i] = 0;

    tat[i] = 0;    rt[i] = -1;

}

```

```

printf("Enter time quantum for Round Robin scheduling: ");    scanf("%d",
&quantum);    sortByArrival(sys, sys_count);    sortByArrival(usr, usr_count);

roundRobin(sys, sys_count, quantum, wt, tat, rt);    int last_sys_time = (sys_count > 0)
? tat[sys_count - 1] + sys[sys_count - 1].at : 0;    fcfs(usr, usr_count, last_sys_time,
&wt[sys_count], &tat[sys_count], &rt[sys_count]);    printf("\nProcess\tQueue\tWaiting
Time\tTurn Around Time\tResponse Time\n");

for (int i = 0; i < n; i++)    printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid,
p[i].queue, wt[i], tat[i], rt[i]);    float avg_wt = 0, avg_tat = 0, avg_rt = 0; for (int i
= 0; i < n; i++) {
    avg_wt += wt[i];
    avg_tat += tat[i];
    avg_rt += rt[i];
}

printf("\nAverage Waiting Time: %.2f", avg_wt / n);
printf("\nAverage Turn Around Time: %.2f", avg_tat / n);
printf("\nAverage Response Time: %.2f\n", avg_rt / n);    return 0;
}

```

Output:

```
C:\Users\Admin\Desktop\1BN X + ▾  
Enter number of processes: 4  
Enter Burst Time, Arrival Time and Queue (1=System, 2=User) for P1: 2 0 1  
Enter Burst Time, Arrival Time and Queue (1=System, 2=User) for P2: 1 0 2  
Enter Burst Time, Arrival Time and Queue (1=System, 2=User) for P3: 5 0 1  
Enter Burst Time, Arrival Time and Queue (1=System, 2=User) for P4: 3 0 2  
Enter time quantum for Round Robin scheduling: 2  


| Process | Queue | Waiting Time | Turn Around Time | Response Time |
|---------|-------|--------------|------------------|---------------|
| P1      | 1     | 0            | 2                | 0             |
| P2      | 2     | 2            | 7                | 2             |
| P3      | 1     | 7            | 8                | 7             |
| P4      | 2     | 8            | 11               | 8             |

  
Average Waiting Time: 4.25  
Average Turn Around Time: 7.00  
Average Response Time: 4.25  
  
Process returned 0 (0x0) execution time : 19.382 s  
Press any key to continue.  
|
```

$P_1$	$P_2$	$\dots$	$P_{12}$	$P_{14}$	$P_{16}$	$P_{18}$	$P_{20}$
0	2	4	6	8	10	12	14
16	18	20	22	24	26	28	30
32	34	36	38	40	42	44	46

(it is swap of  $P_{18}$ )

(it is "b.v") true

### LAB.3

- C program to simulate multilevel queue

```
#include <stdio.h>
type def struct {
    int pid; ((i))
    int bt; ((i)) waiting
}
void arrival(process p[], int n) {
    process temp;
    for (i=0; i<n; i++) {
        for(j=0; j<n; j-1; j++)
            temp = p(j);
            p(j) = p(j+1);
            p(j+1) = temp;
    }
}
```

O output

two

swap to obs  
TA & TA init

8.78

0.50

C: S TA

A: TA

B: TA

C: TA

D: TA

void roundRobin ( process(), int n, int quantum)

{ int remain\_bt (max. process) > burst

remaining\_bt(i) = p(i).bt[i] - XAM with t

int t=0; completed=0; } to set jobs equal

while (comp < n) { : bi free

: TD free

int executed=0; } done

for (int i=1; i<n; i++) { } loop till

if (remaining\_bt(i) > 0) { } execute

if (remaining\_bt(i) > quantum) { } negtive till

t += quantum; } (i+1) next

remaining\_bt(i) = remaining\_bt(i) - t till

{ else { } busy . (0) done + time till

t += remaining\_bt(i) - i till ref

{ busy\_tat(i) = t + p(i).at; } till

wt(i) = tat(i) - p(i).at; }

completed++; }

0/p { generated = 1; first done } output

Enter the number of processes : 4  
Enter ft BT and quantum (1=syst, 2=usy)  
for P1 : 20 2

{ if (t >= (q)) { } else { } } output

Enter ft BT and quantum (1=syst, 2=usy) for P2

{ if (t >= (q)) { } else { } } output

Enter ft BT and quantum (1=syst, 2=usy) for P3

{ if (t >= (q)) { } else { } } output

P1 Q WT FTBT RT 50 ~

P2 1 1 6 0

P3 1 6 8 6

{ (ft, q, wt, rt) insertion } output : 3.75 Avg: 6.55 Avg: 3.75

void fcfs (process P(), int n, int start,

{ int time, wt(i) } output disk NP ) }

{ int st (C) } (if not busy, then)

for (i=0; i<n; i++) { } till busy

{ if (time < p(i).at) { } > time = p(i).at

{ time t = p(i).bt; }

MEMORY MANAGEMENT UNIT (BEST FIT,WORST FIRST,FIRST FIT) CODE:

```
#include <stdio.h>

#define MAX_BLOCKS 20
#define MAX_PROCESSES 10

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        int maxIndex = -1;
        for (int j = 0; j < m;
        j++) {
            if (blockSize[j] >=
            processSize[i]) {
                if (maxIndex == -1 || blockSize[maxIndex] < blockSize[j]) {
                    maxIndex = j;
                }
            }
        }
        if (maxIndex != -1) {
            allocation[i]
            = maxIndex;
            blockSize[maxIndex] -=
            processSize[i];
        }
    }

    printf("\nWorst Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        if
        (allocation[i] != -1) {
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        } else {
            printf("Process %d not allocated\n", i + 1);
        }
    }
}
```

```

        }
    }
}

void bestFit(int blockSize[], int m, int processSize[], 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 (blockSize[j] >= processSize[i]) {
                if (bestIndex == -1 || blockSize[bestIndex] > blockSize[j]) {
                    bestIndex = j;
                }
            }
        }
        if (bestIndex != -1) {
            allocation[i] = bestIndex;
            blockSize[bestIndex] -= processSize[i];
        }
    }

    printf("\nBest Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        } else {
            printf("Process %d not allocated\n", i + 1);
        }
    }
}

```

```
}
```

```
void firstFit(int blockSize[], int m, int processSize[], 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]  
                >= processSize[i]) {                    allocation[i]  
                    = j;                    blockSize[j] -=  
                    processSize[i];                    break;  
                }  
            }  
    }
```

```
    printf("\nFirst Fit Allocation:\n");  
    for (int i = 0; i < n; i++) {        if  
(allocation[i] != -1) {  
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);  
        } else {  
            printf("Process %d not allocated\n", i + 1);  
        }  
    }
```

```
int main() {  
    int blockSize[MAX_BLOCKS], processSize[MAX_PROCESSES];  
    int m, n;  
  
    printf("Enter the number of memory blocks: ");  
    scanf("%d", &m);
```

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

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

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

firstFit(blockSize, m, processSize, n);
bestFit(blockSize, m, processSize, n);
worstFit(blockSize, m, processSize, n);

return 0;
}
```

**Block 5: 600**

**Enter the number of processes: 4**

**Enter the size of each process:**

**Process 1: 212**

**Process 2: 417**

**Process 3: 112**

**Process 4: 426**

**First Fit Allocation:**

**Process 1 allocated to Block 2**

**Process 2 allocated to Block 5**

**Process 3 allocated to Block 2**

**Process 4 not allocated**

**Best Fit Allocation:**

**Process 1 allocated to Block 4**

**Process 2 not allocated**

**Process 3 allocated to Block 2**

**Process 4 not allocated**

**Worst Fit Allocation:**

**Process 1 not allocated**

**Process 2 not allocated**

**Process 3 allocated to Block 3**

**Process 4 not allocated**

Memory Management (Mind Scheini) abad

#include <stdio.h> ~~book~~ (best fit / worst fit / optimal  
struct Block { ~~book~~ (best fit / worst fit / optimal  
 int block\_no; ~~book~~ (best fit / worst fit / optimal  
 int block\_size; ~~book~~ (best fit / worst fit / optimal  
 int is\_free; ~~book~~ (best fit / worst fit / optimal  
}; ~~book~~ (best fit / worst fit / optimal

struct File { ~~book~~ (best fit / worst fit / optimal  
 int file\_no; ~~book~~ (best fit / worst fit / optimal  
 int file\_size; ~~book~~ (best fit / worst fit / optimal  
}; ~~book~~ (best fit / worst fit / optimal

void belt (struct Block blocks[], int n\_block,  
 struct File file[], int n\_file); ~~book~~ (best fit / worst fit / optimal

{ printf ("\n Memory Management Scheme ");  
 printf (" File no\t File Size\t Block no\t n ");  
 for (int i=0; i<n\_file; i++) { ~~book~~ (best fit / worst fit / optimal  
 { int best\_fit\_block = -1; ~~book~~ (best fit / worst fit / optimal  
 int min\_management = 10000; ~~book~~ (best fit / worst fit / optimal  
 for (int j=0; j<n\_block; j++) { ~~book~~ (best fit / worst fit / optimal  
 if (blocks[j].block\_size == file[i].file\_size) { ~~book~~ (best fit / worst fit / optimal  
 if (blocks[j].is\_free && blocks[j].block\_size > min\_fragment) { ~~book~~ (best fit / worst fit / optimal  
 min\_fragment = blocks[j].block\_size; ~~book~~ (best fit / worst fit / optimal  
 best\_fit\_block = j; ~~book~~ (best fit / worst fit / optimal  
 } ~~book~~ (best fit / worst fit / optimal  
 } ~~book~~ (best fit / worst fit / optimal  
 } ~~book~~ (best fit / worst fit / optimal  
 if (best\_fit\_block != -1) { ~~book~~ (best fit / worst fit / optimal  
 blocks[best\_fit\_block].is\_free = 0; ~~book~~ (best fit / worst fit / optimal  
 printf ("%d\t %d\t %d\t %d\n", file[i].file\_no, file[i].file\_size, best\_fit\_block+1, blocks[best\_fit\_block].block\_size); ~~book~~ (best fit / worst fit / optimal  
 } ~~book~~ (best fit / worst fit / optimal  
 } ~~book~~ (best fit / worst fit / optimal  
 } ~~book~~ (best fit / worst fit / optimal

blocks (best-fit-block) - block-no

block (best-fit-block) block-size > available  
min fragment;

{

else {

pf ("%d\n", n); lf ("%d\n", i);

file(i) - file no;

file(i) - file size;

{

}

int main () {

struct block blocks (n-blocks); file file  
n-blocks & files; file size;

pf ("%d", & n-blocks);

pf ("Enter the no of files\n"); string s;

sf ("%d", & n-files);

struct Block blocks (n-blocks);

struct Block file (n-files);

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

blocks(i).block-no = i; i++;

blocks(i).block-size = 0;

pf ("Enter the size of block\n");

sf ("%d", & blocks(i).block-size);

blocks(i).isfree = 1;

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

blocks(i).block-no = i; i++;

pf ("Enter size of file %d\n");

sf ("%d", & blocks(i).block-size);

blocks(i).isfree = 1;

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

blocks(i).block-size = 0; i++;

blocks(i).isfree = 1; i++;

Output:-

Enter no of blocks : 5

Enter no of files : 4

Enter size of block : 100

Enter size of block : 200

Enter size of block : 400

Output

memory	Fullsize	Block_no	Block_size
1	212	3	300

2	146	4	100
3	426	2	200
4	404	N/A	N/A

FIFO / LRU / optimal : uses memory tree

#include <stdio.h> : (int, int, float) func

int search ( int key, int frame() ) : int tree

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

if ( frame (i) == key ) { found }

: (int, int, float) func

void ( simulate\_FIFO ( int pages(), int n,

## PAGE REPLACEMENT PROGRAM(LRU,OPTIMAL,FIFO)

CODE:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_FRAMES 10
#define MAX_PAGES 50
```

```
bool isPageInFrame(int frames[], int page, int frameCount) {
    for (int i = 0; i < frameCount; i++) {
        if (frames[i] == page)
            return true;
    }
    return false;
}
```

```
int fifo(int pages[], int pageCount, int frameCount) {
    int frames[MAX_FRAMES], front = 0, pageFaults = 0;

    for (int i = 0; i < pageCount; i++) {
        if (!isPageInFrame(frames, pages[i], frameCount)) {
            frames[front] = pages[i];
            front = (front + 1) % frameCount;
            pageFaults++;
        }
    }
    return pageFaults;
}
```

```
int lru(int pages[], int pageCount, int frameCount) {
    int frames[MAX_FRAMES], lastUsed[MAX_FRAMES], pageFaults = 0;
```

```

for (int i = 0; i < pageCount; i++) {
    if (!isPageInFrame(frames, pages[i], frameCount)) {
        int leastRecentlyUsedIndex = 0;           for (int j = 1; j <
frameCount; j++) {
            if (lastUsed[j] < lastUsed[leastRecentlyUsedIndex]) {
                leastRecentlyUsedIndex = j;
            }
        }
        frames[leastRecentlyUsedIndex] = pages[i];
        pageFaults++;
    }
    for (int j = 0; j < frameCount; j++) {
        lastUsed[j]++;
    }
    lastUsed[pages[i] % frameCount] = 0;
}
return pageFaults;
}

```

```

int optimal(int pages[], int pageCount, int frameCount) {
int frames[MAX_FRAMES], pageFaults = 0;

for (int i = 0; i < pageCount; i++) {
    if (!isPageInFrame(frames, pages[i], frameCount)) {
        int farthestIndex = -1, replaceIndex = -1;      for (int j
= 0; j < frameCount; j++) {          int nextUse =
pageCount;

        for (int k = i + 1; k < pageCount; k++) {
if (frames[j] == pages[k]) {
nextUse = k;          break;
}

```

```

        }
        if (nextUse > farthestIndex) {
            farthestIndex = nextUse;           replaceIndex
            = j;
        }
    }
    frames[replaceIndex] = pages[i];
    pageFaults++;
}
return pageFaults;
}

int main() {
    int pages[MAX_PAGES], pageCount, frameCount;

    printf("Enter number of pages: ");
    scanf("%d", &pageCount);
    printf("Enter page sequence: ");
    for
    (int i = 0; i < pageCount; i++) {
        scanf("%d", &pages[i]);
    }
    printf("Enter number of frames: ");
    scanf("%d", &frameCount);

    printf("FIFO Page Faults: %d\n", fifo(pages, pageCount, frameCount));
    printf("LRU Page Faults: %d\n", lru(pages, pageCount, frameCount));
    printf("Optimal Page Faults: %d\n", optimal(pages, pageCount, frameCount));

    return 0;
}

```

**Enter number of pages: 12**

**Enter page sequence: 1 3 0 3 5 6 3 0 3 5 2 3**

**Enter number of frames: 3**



more - no

Block 58136

1	2 1 2	3	800
2	1 4 6	thus 4 pages	0 7 5 6 0 0 0 0
3	4 2 6	(1 bit 2 thus) (0 1 6 0)	5 0 0
4	4 0 4	N/A	N/A
5	1 5 1 5		(C) more tri

```
FIFO / LRU / optimal : visit each n tri  
#include <stdio.h> : (n, 11 b.v) max  
int search ( int key, int frame() ) { int size  
{  
    for ( int i = 0 ; i < size ; i++ ) {  
        if ( frame ( i ) == key ) {  
            return {0, 0, 3} with set 3 } } } }  
void simulate_FIFO ( int pages(), int n,  
                     int transaction ) {  
    int frame ( frameSize ); front = 0, faulted = 0  
    for ( int i = 0 ; i < frameSize ; i++ )  
        frame ( i ) = 1 with page 0 with set 0  
    for ( int i = 0 ; i < frameSize ; i++ )  
        frame ( i ) = 1 with page 1  
    frame ( front ) = pages [ 1 ] }
```

```

if (!search (page (i), frame,
              frameSize ()))
{
    faults++;
    if (faults == 1)
        cout << "1st fault for page ";
    else if (faults == 2)
        cout << "2nd fault for page ";
    else
        cout << "3rd fault for page ";
    cout << endl;
}

printf("FIFO pages/faults : %d\n", faults);
printf("Fault hit : %d\n", faults);

int main()
{
    int n, frameSize;
    printf("Enter the size of the pages");
    scanf("%d", &n);
    int pages[n];
    printf("Enter page strings : ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter the no. of page frames : ");
    if ((n > 0) & (frameSize > 0))
        return 0;
    cout << endl;
    Output();
}

```

93

~~(1) Enter the size > if positive  
 Enter the page string = (i) 30 35 62  
 FIFO pages/faults : 8 : (i) 11 12 (i) 13 (i) 14  
 11 12 13 14 - (tray) miss~~