VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

SUSHMITHA KL(1WA23CS003)

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 **SUSHMITHA KL (1WA23CS003)**, 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 OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

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

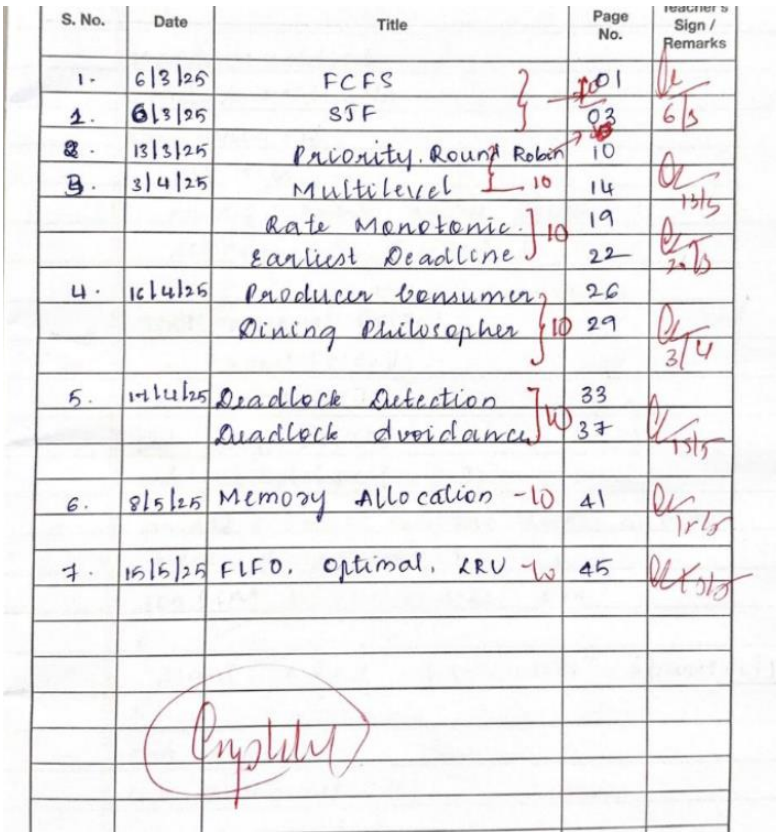
Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	89-100

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.



S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	6/3/25	FCFS	01	
1.	6/3/25	SJF	03	6/3
2.	13/3/25	Priority, Round Robin	10	
3.	3/4/25	Multilevel	14	13/3
		Rate Monotonic	19	
		Earliest Deadline	22	20/3
4.	16/4/25	Producer consumer	26	
		Dining Philosopher	29	3/4
5.	17/4/25	Deadlock detection	33	
		Deadlock avoidance	37	17/3
6.	8/5/25	Memory Allocation	41	1r/5
7.	15/5/25	FIFO, optimal, LRU	45	15/5
		Completed		

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

*FCFS:

```
#include<stdio.h>

void CT(int processes[], int n, int bt[], int at[], int ct[]) {
    ct[0] = at[0] + bt[0];
    for (int i = 1; i < n; i++) {
        ct[i] = (ct[i - 1] > at[i] ? ct[i - 1] : at[i]) + bt[i];
    }
}

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

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

void AVG(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n];
```

```
    CT(processes, n, bt, at, ct);
    TAT(processes, n, bt, at, ct, tat);
    WT(processes, n, bt, at, tat, wt);

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

    printf("\navg bt= %.2f", (float)total_wt / n);
    printf("\navg tat = %.2f", (float)total_tat / n);
}

int main() {
    int n;

    printf("no. of processes: ");
    scanf("%d", &n);

    int processes[n], bt[n], at[n];


        printf("enter bt and tat: \n"); for (int i = 0; i < n; i++) { printf("bt %d: ", i + 1);
        scanf("%d", &bt[i]);
        printf("tat %d: ", i + 1);
        scanf("%d", &at[i]);
        processes[i] = i + 1;
    }
```

```
    AVG(processes, n, bt, at);

    return 0;
}
```

```
no. of processes: 4
enter bt and tat:
bt 1: 7
tat 1: 0
bt 2: 3
tat 2: 0
bt 3: 4
tat 3: 0
bt 4: 6
tat 4: 0

avg bt= 7.75
avg tat = 12.75
```

FCFS

```
#include <stdio.h>
#include <stdlib.h>
int main (){
   int size ;
   printf ("Enter no. of proces :");
   scanf ("%d", &size );
   int arrival [size];
   int burst [size];
   int TAT [size];
   int wait [size];
   int completion [size];
   printf ("Enter arrival time & burst
    time of proces");
   for (int i=0; i< size; i++)
   {
    scanf ("%d %d", &arrival[i], &burst[i]);
   }
   int sum = 0;
   int sumburst = 0;
   int sumcomp = 0;
   for (int i=0; i<size; i++){
      sum = sum + burst[i];
      completion[i] = sum;
      sumburst = sumburst + burst[i];
      sumcomp = + completion[i];
```

```
{
int tatsum = 0;
for (int i = 0; i<size; i++){
  TAT[i] = completion[i] - arrival[i];
  tatsum = tatsum + TAT[i];
}

int sumwait = 0;
for (int i=0; i<size; i++){
  wait[i] = completion[i] - burst[i];
  sumwait = sumwait + wait[i];

float avgtat = tatsum / (float) size;
float avgcomp = sumcomp / (float) size;
float avgwait = sumwait / (float) size;
printf ("avgtat = %f  avgcomp = %f
avgwait = %f", avgtat, avgwait, avgwait);
```

OUTPUT:

```
Enter no. of process: 4
Enter arrival time & burst time of
process 0
7
0
4
0
5
0
3
```

avgtat = 13.250000
avgcomp = 18.25000
avgwait = 8.5000

Gantt chart

P1	P2	P3	P4
0	7	11	16	19

*SJF(Non-preemptive):

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

struct Process {
    int id;
    int  bt;
    int  at;
    int  ct;
    int tat;
    int wt;
};

int compareArrivalTime(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 <= time && processes[i].ct == 0) {
                if (processes[i].bt < min_burst) {
                    min_burst = processes[i].bt;
                    shortest = i;
                }
            }
        }

        if (shortest == -1) {
            time++;
```

```
        } else {
            processes[shortest].ct = time + processes[shortest].bt;
            processes[shortest].tat = processes[shortest].ct - processes[shortest].at;
            processes[shortest].wt = processes[shortest].tat - processes[shortest].bt;

            time = processes[shortest].ct;
            completed++;
        }
    }
}

void calculateAvg(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("\navg wt = %.2f", (float)total_wt / n);
    printf("\navg 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("%d", &processes[i].at);
    processes[i].ct = 0;
  }

  qsort(processes, n, sizeof(struct Process), compareArrivalTime);

  calculateTimes(processes, n);

  calculateAvg(processes, n);

  return 0;
  }
```

```
no. of processes: 4
enter bt and at:
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%
```

`

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

struct Process {
    int id, AT, BT, CT, TAT, WT, 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;
            }
        }
    }
}
```

```c
void calculateSJFNonPreemptive(struct Process p[], int n) {
    int completed = 0;
    int currentTime = 0;
    float totalWT = 0;
    float totalTAT = 0;

    while (completed < n) {
        int shortest = -1, minBT = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (!p[i].completed && p[i].AT <= currentTime && p[i].BT < minBT) {
                minBT = p[i].BT;
                shortest = i;
            }
        }

        if (shortest == -1) {
            currentTime++;
        } else {
            p[shortest].CT = currentTime + p[shortest].BT;
            p[shortest].TAT = p[shortest].CT - p[shortest].AT;
            p[shortest].WT = p[shortest].TAT - p[shortest].BT;
            p[shortest].completed = 1;
            totalWT += p[shortest].WT;
            totalTAT += p[shortest].TAT;
            currentTime = p[shortest].CT;
            completed++;
        }
    }

    printf("\nProcess\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\n", p[i].id, p[i].WT, p[i].TAT);
    }
    printf("\nAverage WT: %.2f", totalWT/n);
    printf("\nAverage TAT: %.2f\n", totalTAT/n);
}

int main() {
    int n;
    printf("Enter no of process: ");
    scanf("%d", &n);

    struct Process p[n];
    for (int i = 0; i < n; i++) {
        p[i].id = i+1;
        printf("Enter Arrival (AT) and Burst Time (BT) for process %d\n", i+1);
        scanf("%d %d", &p[i].AT, &p[i].BT);
        p[i].completed = 0;
    }
```

*SJF (pre-emptive):

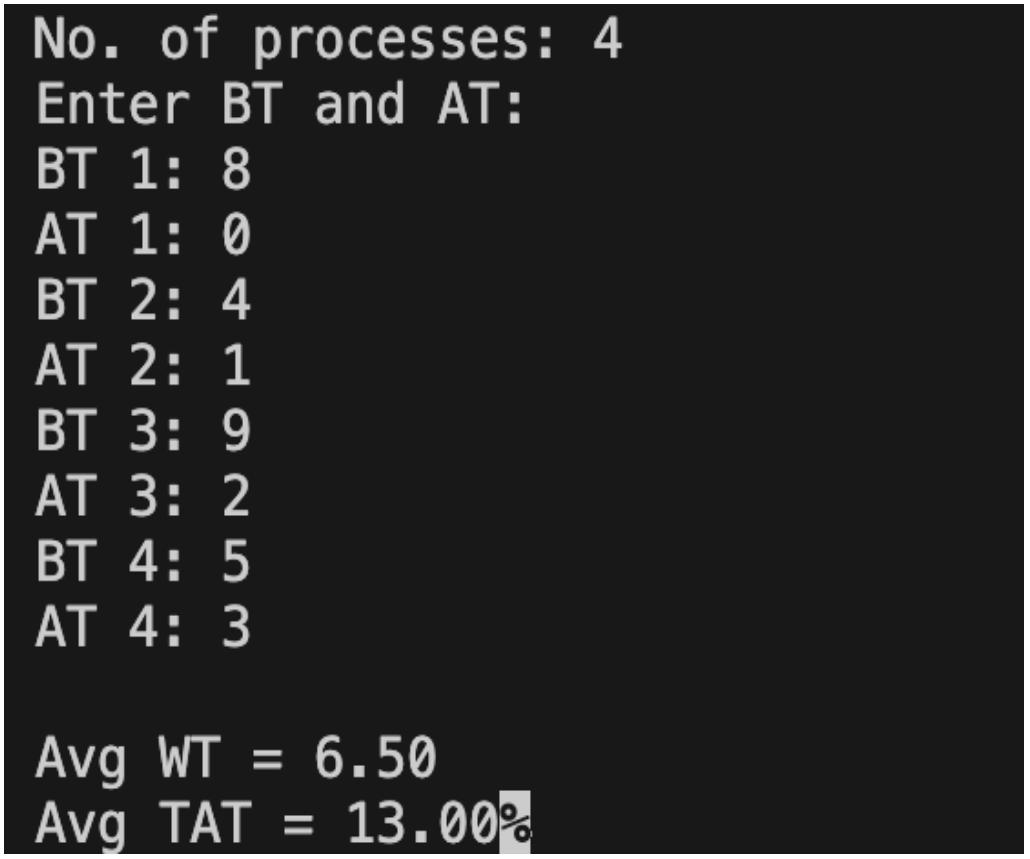
```
#include <stdio.h>
#include <limits.h>

struct Process {
    int id;
    int  bt;
    int  at;
    int   rt;
    int  ct;
    int tat;
    int wt;
};

void calculateTimes(struct Process processes[], int n) {
    int completed = 0, time = 0, shortest = -1;
    int min_burst = INT_MAX;

    while (completed < n) {
        shortest = -1;
        min_burst = INT_MAX;


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

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

`

```
        processes[shortest].rt--;
        time++;

        if (processes[shortest].rt == 0) {
            completed++;
            processes[shortest].ct = time;
            processes[shortest].tat = processes[shortest].ct - processes[shortest].at;
            processes[shortest].wt = processes[shortest].tat - processes[shortest].bt;
        }
    }
}

void calculateAvg(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("\nAvg WT = %.2f", (float)total_wt / n);
    printf("\nAvg 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("%d", &processes[i].at);
        processes[i].rt = processes[i].bt;
    }

    calculateTimes(processes, n);

    calculateAvg(processes, n);

    return 0;
}
```



`

```
calculateSJFNonPreemptive(p,n);
return 0;
}
```

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

struct Process{
  int id, AT, BT, CT, TAT, WT, RT,
      remaining_BT, completed;
};

void calculateSJFPreemptive(struct
      Process p[], int n){
 int completed=0, currentTime=0;
 float totalWT=0, totalTAT=0;

 for(int i=0; i<n; i++){
  p[i].remaining_BT = p[i].BT;
 }

 while(completed<n){
  int shortest=-1, minBT = INT_MAX;
  for(int i=0; i<n; i++){
   if(!p[i].completed && p[i].AT<=
     currentTime && p[i].remaining_BT<
                    minBT){
    minBT = p[i].remaining_BT;
    shortest=i;
   }
  }

  if(shortest==-1){
   currentTime++;
  }else{
   if(p[shortest].remaining_BT ==
                p[shortest].BT)
    p[shortest].RT = currentTime -
                p[shortest].AT;

   p[shortest].remaining_BT--;
   currentTime++;

   if(p[shortest].remaining_BT==0){
    p[shortest].CT = currentTime;
    p[shortest].TAT = p[shortest].CT -
                p[shortest].AT;
    p[shortest].WT = p[shortest].TAT -
                p[shortest].BT;
    p[shortest].completed=1;
    totalWT += p[shortest].WT;
    totalTAT += p[shortest].TAT;
    completed++;
   }
  }
 }
```

```
printf("\n Process\t WT\t TAT\n");
for(int i=0; i<n; i++){
 printf("%d %d %d", p[i].id, p[i].WT,
                p[i].TAT);
}
printf("\n Average WT: %.2f", totalWT/n);
printf("\n Average TAT: %.2f", totalTAT/n);
}

int main(){
 int n;
 printf("Enter no of process:");
 scanf("%d", &n);

 struct Process p[n];
 for(int i=0; i<n; i++){
  p[i].id = i+1;
  printf("Enter AT and BT for process %d:",
                i+1);
  scanf("%d %d", &p[i].AT, &p[i].BT);
  p[i].completed=0;
 }

 calculateSJFPreemptive(p,n);
 return 0;
}
```

OUTPUT:

```
Enter no of processes: 4
Enter AT & BT : 0 8
Enter AT & BT : 1 4
Enter AT & BT : 2 9
Enter AT & BT : 3 5
```

Gantt chart

P1	P2	P4	P1	P3
0	1	5	10	17	26

```
Avg WT : 6.5
Avg TAT : 13
```

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

→ **Priority (Non-pre-emptive):**

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

struct Process {
    int id, bt, at, priority, ct, tat, wt;
};

int comparePriority(const void *a, const void *b) {
    struct Process *p1 = (struct Process *)a;
    struct Process *p2 = (struct Process *)b;
    if (p1->at == p2->at)
        return p1->priority - p2->priority;
    return p1->at - p2->at;
}

void calculateTimes(struct Process processes[], int n) {
    int time = 0, completed = 0;

    while (completed < n) {
        int highest = -1, highestPriority = 1000000;

        for (int i = 0; i < n; i++) {
            if (processes[i].at <= time && processes[i].ct == 0) {
                if (processes[i].priority < highestPriority) {
                    highestPriority = processes[i].priority;
                    highest = i;
                }
            }
        }

        if (highest == -1) {
```

`

```
            time++;
        } else {
            processes[highest].ct = time + processes[highest].bt;
            processes[highest].tat = processes[highest].ct - processes[highest].at;
            processes[highest].wt = processes[highest].tat - processes[highest].bt;

            time = processes[highest].ct;
            completed++;
        }
    }
}

void calculateAvg(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("\nAvg WT = %.2f", (float)total_wt / n);
    printf("\nAvg TAT = %.2f", (float)total_tat / n);
}

int main() {
    int n;
    printf("No. of processes: ");
    scanf("%d", &n);

    struct Process processes[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("%d", &processes[i].at);
        printf("Priority %d: ", i + 1);
        scanf("%d", &processes[i].priority);
        processes[i].ct = 0;
`
```

```
    }

    qsort(processes, n, sizeof(struct Process), comparePriority);

    calculateTimes(processes, n);
    calculateAvg(processes, n);

    return 0;
}
```

```
No. of processes: 5
BT 1: 3
AT 1: 0
Priority 1: 5
BT 2: 2
AT 2: 2
Priority 2: 3
BT 3: 5
AT 3: 3
Priority 3: 2
BT 4: 4
AT 4: 4
Priority 4: 4
BT 5: 1
AT 5: 6
Priority 5: 1

Avg WT = 3.20
Avg TAT = 6.20
```

`

Priority Non pre-emptive

```
#include<stdio.h>

struct Process
{int pid, arrival_time, burst_time,
completion_time, waiting_time,
turn around_time;
};

void priorityNonPreemptive(struct
        Process p[], int n){
int completed=0, time=0, min
    priority, shortest;
float total_waiting=0, total
                turn around=0;

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

for(int i=0; i<n; i++){
if(p[i].arrival_time<=time &&
p[i].priority<min_priority &&
p[i].completion_time==0)
{
 min_priority=p[i].priority;
 shortest=i;
}}
```

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

p[shortest].completion_time=time+
        p[shortest].burst_time;
p[shortest].turnaround_time=
p[shortest].completion_time-p[shortest].
                arrival_time;
p[shortest].waiting_time=p[shortest].
turnaround_time-p[shortest].burst_time;

total_waiting+=p[shortest].waiting
                time;
total_turnaround+=p[shortest].
                turnaround_time;
time+=p[shortest].burst_time;
completed++;
}

printf("Non Pre-emptive Priority
        Scheduling\n");
printf("PID\t AT\t BT\t P\t CT\t TAT\t
                WT\t");

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].pid, p[i].arrival_time, p[i].burst
time, p[i].priority, p[i].completion
```

```
p[i].turnaround_time, p[i].waiting_time);

printf("Average waiting time: %.2f\n", total_waiting);
printf("Average Turnaround Time: %.2f\n", total_turnaround);
}

int main(){
int n;
printf("Enter no. of processes: ");
scanf("%d", &n);

struct Process p[n];

for(int i=0; i<n; i++){
printf("Enter Process ID, Arrival Time, Burst Time, Priority: ");
scanf("%d %d %d %d", &p[i].pid, &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
p[i].completion_time = 0;
}

priorityNonPreemptive(p, n);
return 0;
}
```

OUTPUT:

```
Enter no of processes: 5
Enter Process ID, Arrival Time, Burst Time, Priority: 1 0 4 2
Enter Process ID, Arrival Time, Burst Time, Priority: 2 1 3 3
Enter Process ID, Arrival Time, Burst Time, Priority: 3 2 1 4
Enter Process ID, Arrival Time, Burst Time, Priority: 4 3 5 5
Enter Process ID, Arrival Time, Burst Time, Priority: 5 4 2 5
```

Non Preemptive Priority Scheduling

PID	AT	BT	P	CT	TAT	WT
1	0	4	2	4	4	0
2	1	3	3	7	6	3
3	2	1	4	8	6	5
4	3	5	5	13	10	5
5	4	2	5	15	11	9

Average Waiting Time: 4.40
Average Turnaround Time: 7.40

→ Priority (pre-emptive):

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

struct Process {
    int id, bt, at, priority, ct, tat, wt, rt;
};

int compareArrival(const void *a, const void *b) {
    struct Process *p1 = (struct Process *)a;
    struct Process *p2 = (struct Process *)b;
    return p1->at - p2->at;
}

void calculateTimes(struct Process processes[], int n) {
    int time = 0, completed = 0, min_priority, shortest;
    for (int i = 0; i < n; i++) processes[i].rt = processes[i].bt;

    while (completed < n) {
        shortest = -1;
        min_priority = 1000000;

        for (int i = 0; i < n; i++) {
            if (processes[i].at <= time && processes[i].rt > 0 && processes[i].priority <
min_priority) {
                min_priority = processes[i].priority;
                shortest = i;
            }
        }
```

`

```
        if (shortest == -1) {
            time++;
        } else {
            processes[shortest].rt--;
            time++;
            if (processes[shortest].rt == 0) {
                processes[shortest].ct = time;
                processes[shortest].tat = processes[shortest].ct - processes[shortest].at;
                processes[shortest].wt = processes[shortest].tat - processes[shortest].bt;
                completed++;
            }
        }
    }
}

void calculateAvg(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("\nAvg WT = %.2f", (float)total_wt / n);
    printf("\nAvg TAT = %.2f", (float)total_tat / n);
}

int main() {
    int n;
    printf("No. of processes: ");
    scanf("%d", &n);

    struct Process processes[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("%d", &processes[i].at);
```
`

```
        printf("Priority %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    qsort(processes, n, sizeof(struct Process), compareArrival);

    calculateTimes(processes, n);
    calculateAvg(processes,n);
    return 0;
}
```

```
No. of processes: 7
BT 1: 8
AT 1: 0
Priority 1: 3
BT 2: 2
AT 2: 1
Priority 2: 4
BT 3: 4
AT 3: 3
Priority 3: 4
BT 4: 1
AT 4: 4
Priority 4: 5
BT 5: 6
AT 5: 5
Priority 5: 2
BT 6: 5
AT 6: 6
Priority 6: 6
BT 7: 1
AT 7: 7
Priority 7: 1

Avg WT = 9.86
Avg TAT = 13.71
```

`

Round Robin:

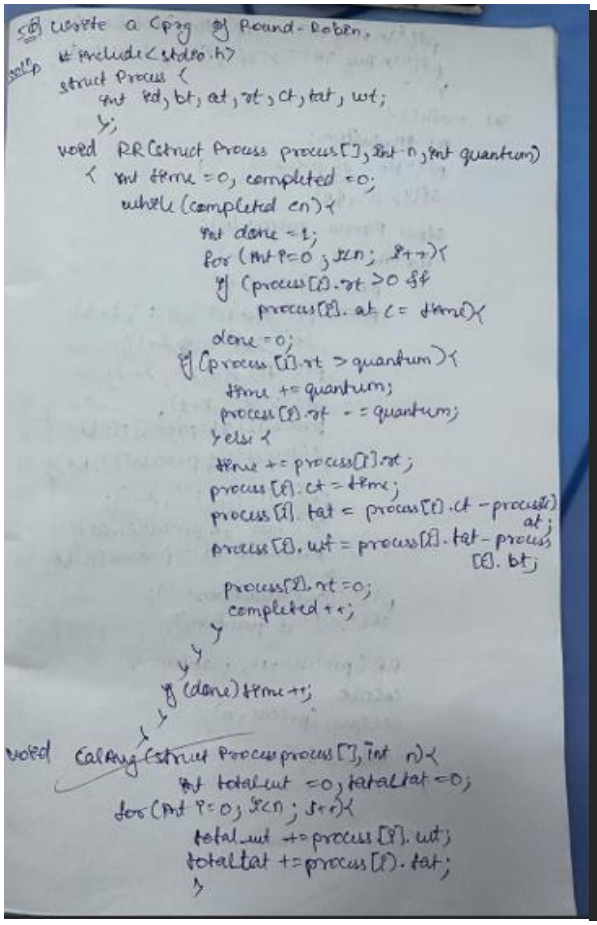
```
#include <stdio.h>

struct Process {
    int id, bt, at, rt, ct, tat, wt;
};

void roundRobin(struct Process processes[], int n, int quantum) {
    int time = 0, completed = 0;
    while (completed < n) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (processes[i].rt > 0 && processes[i].at <= time) {
                done = 0;
                if (processes[i].rt > quantum) {
                    time += quantum;
                    processes[i].rt -= quantum;
                } else {
                    time += processes[i].rt;
                    processes[i].ct = time;
                    processes[i].tat = processes[i].ct - processes[i].at;
                    processes[i].wt = processes[i].tat - processes[i].bt;
                    processes[i].rt = 0;
                    completed++;
                }
            }
        }
        if (done) time++;
    }
}

void calculateAvg(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;
```

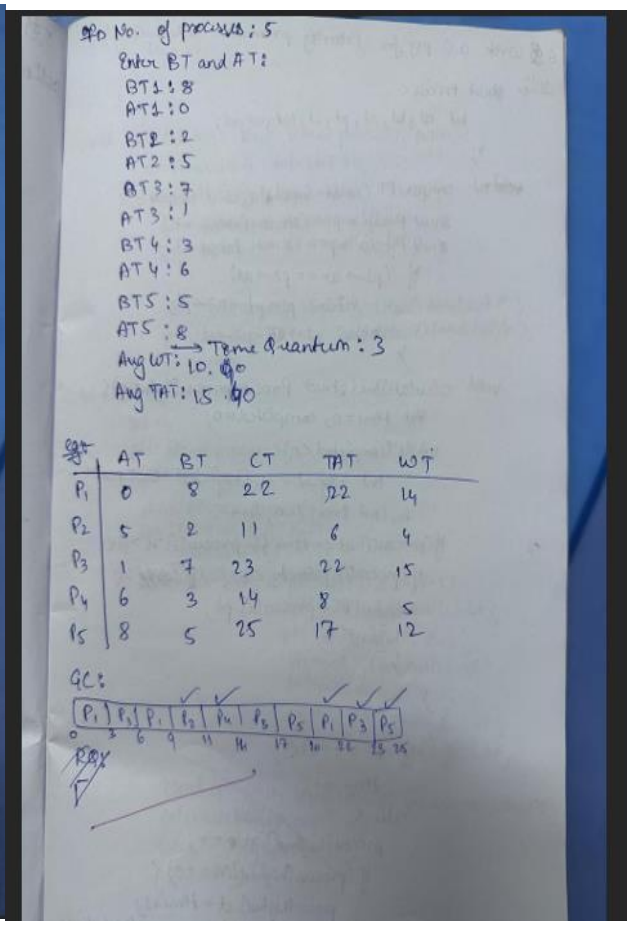
`

```
    }
    printf("\nAvg WT = %.2f", (float)total_wt / n);
    printf("\nAvg TAT = %.2f", (float)total_tat / n);
}

int main() {
    int n, quantum;
    printf("No. of processes: ");
    scanf("%d", &n);

    struct Process processes[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("%d", &processes[i].at);
        processes[i].rt = processes[i].bt;
    }

    printf("Time Quantum: ");
    scanf("%d", &quantum);

    roundRobin(processes, n, quantum);
    calculateAvg(processes, n);

    return 0;
}
```

`

```
No. of processes: 5
BT 1: 8
AT 1: 0
BT 2: 2
AT 2: 5
BT 3: 7
AT 3: 1
BT 4: 3
AT 4: 6
BT 5: 5
AT 5: 8
Time Quantum: 3

Avg WT = 10.40
Avg TAT = 15.40
```





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

→Multilevel scheduling:

```
#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("NO. of process: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter BT,AT and Q (1=sys, 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("QT for RR: ");
    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("\nP\tQ\tWT\tTAT\tRT\n");
    for (int i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t\t%d\t\t\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("\nAVG WT: %.2f", avg_wt / n);
    printf("\nAVG TAT: %.2f", avg_tat / n);
    printf("\nAVG RT: %.2f\n", avg_rt / n);

    return 0;
    }
```

```
NO. of process: 4
Enter BT,AT and Q (1=sys, 2=user) for P1: 2 0 1
Enter BT,AT and Q (1=sys, 2=user) for P2: 1 0 2
Enter BT,AT and Q (1=sys, 2=user) for P3: 5 0 1
Enter BT,AT and Q (1=sys, 2=user) for P4: 3 0 2
QT for RR: 2

P       Q       WT      TAT     RT
P1      1       0               2                       0
P2      2       2               7                       2
P3      1       7               8                       7
P4      2       8               11                      8

AVG WT: 4.25
AVG TAT: 7.00
AVG RT: 4.25
```

`

Multilevel

```
#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-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 no. of processes ");
    scanf("%d", &n);

    for (int i=0; i<n; i++){
        printf("Enter BT, AT 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: ");
    scanf("%d", &quantum);

    sortByArrival(sys, sys_count);
    sortByArrival(usr, usr_count);

    roundRobin(sys, sys_count, wt, tat, rt);
    int last_sys_time = (sys_count > 0) ? tat[sys_count-1] + sys[sys_count-1].bt : 0;
    fcfs(usr, usr_count, last_sys_time, &wt[sys_count], &tat[sys_count], &rt[sys_count]);

    printf("\nProcess\tQueue\tWT\tTAT\tRT\n");
    for (int i=0; i<n; i++)
        printf("P%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("\n Avg WT : %.2f", avg_wt/n);
printf("\n Avg TAT : %.2f", avg_tat/n);
printf("\n Avg RT : %.2f", avg_rt/n);
```

OUTPUT:

Enter no. of processes: 4

Enter BT, AT and Queue (1 = System, 2 = User) for P1 : 2 0 1

P2 : 1 0 2

P3 : 5 0 1

P4 : 3 0 2

Process	Queue	WT	TAT	RT
P1	1	0	2	0
P2	2	2	3	2
P3	1	7	8	7
P4	2	8	11	8

Avg

4.Write a C program to simulate Real-Time CPU Scheduling algorithms:

a) Rate- Monotonic:

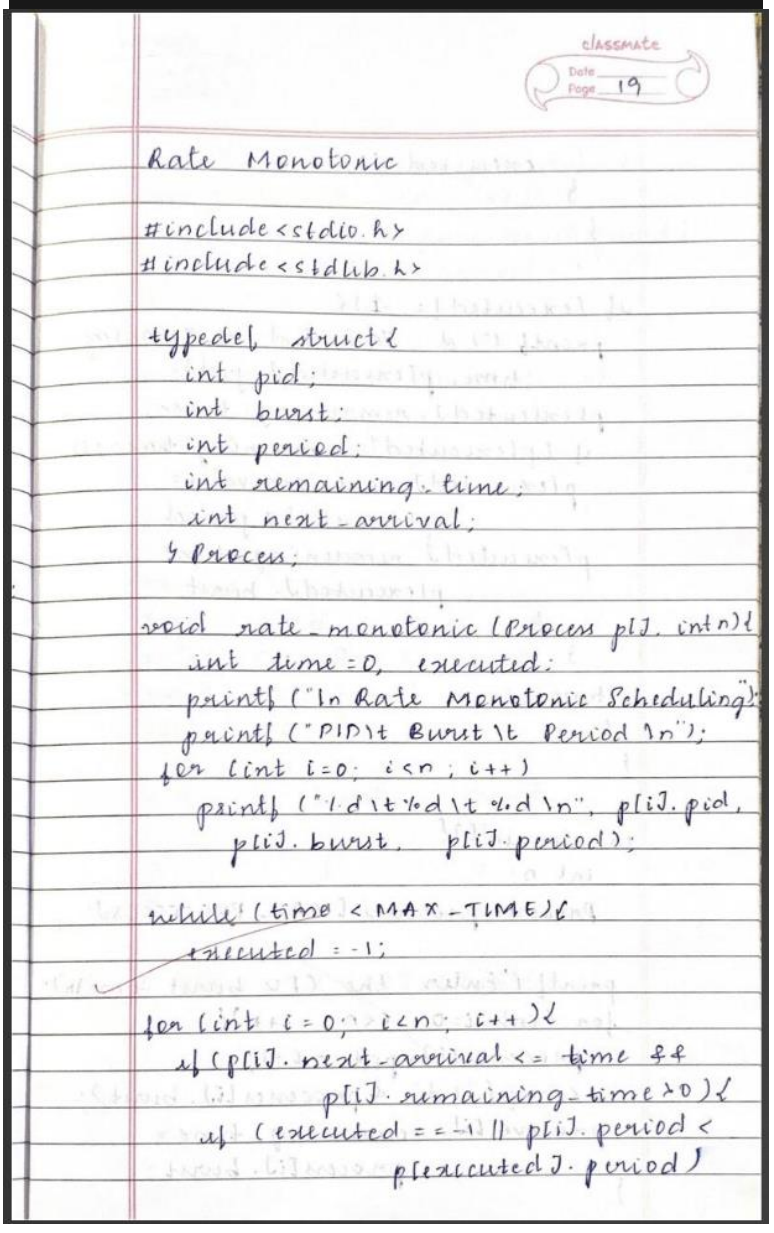
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_PROCESSES 10
#define MAX_TIME 50  // Maximum simulation time

typedef struct {
    int pid;
    int burst;
    int period;
    int remaining_time;
    int next_arrival;
} Process;

void rate_monotonic_scheduling(Process p[], int n) {
    int time = 0, executed;
    printf("\nRate Monotonic Scheduling:\n");
    printf("PID\tBurst\tPeriod\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\n", p[i].pid, p[i].burst, p[i].period);

    while (time < MAX_TIME) {
        executed = -1;

        for (int i = 0; i < n; i++) {
            if (p[i].next_arrival <= time && p[i].remaining_time > 0) {
                if (executed == -1 || p[i].period < p[executed].period)
                    executed = i;
            }
        }

        if (executed != -1) {
```

`

```
            printf("%dms : Task %d is running.\n", time, p[executed].pid);
            p[executed].remaining_time--;

            if (p[executed].remaining_time == 0) {
                p[executed].next_arrival += p[executed].period;
                p[executed].remaining_time = p[executed].burst; // Reset for periodic
execution
            }
        }
        time++;
    }
}

int main() {
    int n;
    Process processes[MAX_PROCESSES];

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

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

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

    rate_monotonic_scheduling(processes, n);

    return 0;
}
```

`

```
Enter the number of processes: 3
Enter the CPU burst times:
3 6 8
Enter the time periods:
3 4 5

Rate Monotonic Scheduling:
PID     Burst   Period
1       3       3
2       6       4
3       8       5
```



Rate Monotonic

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

typedef struct{
    int pid;
    int burst;
    int period;
    int remaining_time;
    int next_arrival;
}Process;

void rate_monotonic(Process p[], int n){
    int time=0, executed;
    printf("\n Rate Monotonic Scheduling:\n");
    printf("PID\t Burst\t Period\n");
    for(int i=0; i<n; i++)
        printf("%d\t%d\t%d\n", p[i].pid,
            p[i].burst, p[i].period);

    while(time < MAX_TIME){
        executed = -1;

    for(int i=0; i<n; i++){
        if(p[i].next_arrival <= time &&
                p[i].remaining_time>0){
            if(executed == -1 || p[i].period <
                    p[executed].period)
```

```
        executed = i;
    }
}

if (executed != -1){
printf ("%d Task %d is running\n",
        time, plexecutedJ.pid);
plexecutedJ.remaining_time--;
if (plexecutedJ.remaining_time == 0){
plexecutedJ.next_arrival +=
        plexecutedJ.period;
plexecutedJ.remaining_time =
        plexecutedJ.burst;
    }
}
time++;
}
}

int main(){
  int n;
  Process processes[MAX_PROCESSES];

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



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

rate_monotonic(processes, n);

return 0;

}
```

OUTPUT:

Enter no. of processes: 3
Enter CPU BT: 3 6 8
Enter time period 3 4 5
LCM = 60

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

4 1000 < = 0.7797 ⇒ false

b)Earliest-deadline:

```
#include    <stdio.h>

int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

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

struct Process {
    int id, burst_time, deadline, period;
};

void earliest_deadline_first(struct Process p[], int n, int time_limit) {
    int time = 0;
    printf("Earliest Deadline Scheduling:\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst_time, p[i].deadline,
p[i].period);
    }

    printf("\nScheduling occurs for %d ms\n", time_limit);
    while (time < time_limit) {
```

`

```
        int earliest = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].burst_time > 0) {
                if (earliest == -1 || p[i].deadline < p[earliest].deadline) {
                    earliest = i;

                }
            }
        }

        if (earliest == -1) break;

        printf("%dms: Task %d is running.\n", time, p[earliest].id);
        p[earliest].burst_time--;
        time++;
    }
}

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

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

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

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

```
    int hyperperiod = processes[0].period;
    for (int i = 1; i < n; i++) {
        hyperperiod = lcm(hyperperiod, processes[i].period);
    }

    printf("\nSystem will execute for hyperperiod (LCM of periods): %d ms\n",
 hyperperiod);

    earliest_deadline_first(processes, n, hyperperiod);

    return 0;
}
```

```
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

System will execute for hyperperiod (LCM of periods): 6 ms
Earliest Deadline Scheduling:
PID     Burst   Deadline        Period
1       2               1               1
2       3               2               2
3       4               3               3

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

`

Earliest deadline

```
#include<stdio.h>

int gcd(int a, int b){
    while(b!=0){
    int temp = b;
     b=a%b;
     a= temp;
    }
     return a;
}
int lcm(int a, int b){
    return (a*b/gcd(a,b));
}
  struct Process{
     int id, burst_time, deadline,
                          period;
};

void earliest_deadline_first(struct
Process p[], int n, int time_limit){
     int time=0;
  printf("Earliest deadline\n");
  printf("PID\t BT\t DL\t P\n");
   for(int i=0; i<n; i++){
     printf("%d\t %d\t %d\t %d\t",p[i].
pid, p[i].BT, p[i].dt, p[i].period);
}
```

```
printf("In Scheduling occurs for %d
ms\n", time_limit);
  while(time < time_limit){
    int earliest = -1;
   for(int i=0; i<n; i++){
    if(p[i].burst_time>0){
     if(earliest == -1 || p[i].dt <
                    p[earliest].dt){
      earliest = i;
    }
   }
  }

if(earliest==-1) break;
 printf("%d ms Task %d is running\n",
   time, p[earliest].id);
   p[earliest].burst_time--;
     time++;
  }
}
  int main(){
    int n;
    printf("Enter no. of processes: ");
    scanf("%d", &n);

struct process process[n];
  printf("Enter CPU BT:\n");
   for(int i=0; i<n; i++){
     scanf("%d", &process[i].
```

```
processes[i].id = i+1;
}

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

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

int hyperperiod = processes[0].period;
for(int i=1; i<n; i++){
    hyperperiod = lcm(hyperperiod,
        processes[i].period);
}
printf("\n System will execute for
hyperperiod (LCM of periods): %d ms\n",
    hyperperiod);

earliest_deadline_first(processes, n,
hyperperiod);
    return 0;
}
```

OUTPUT:

Enter the no of processes 3
Enter CPU scheduling BT: 2 3 4
Enter deadline 1 2 3
Enter time period 1 2 3

System will execute for hyperperiod: 6 ms

PID	BT	DL	P
1	2	1	1
2	3	2	2
3	4	3	3

Scheduling occurs for 6 ms

0 ms: Task 1 running
1 ms: ——"——
2 ms: Task 2 running
3 ms: ——"——
4 ms: ——"——
5 ms: Task 3 running

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

producer-consumer:

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

int mutex = 1;
int full = 0;
int empty=3;
int buffer_item = 0;

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

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

void producer(int id) {
   if ((mutex == 1) && (empty != 0)) {
      int value;
      scanf("%d", &value);
      printf("producer %d produced %d\n", id, value);
      mutex = wait(mutex);
      full = signal(full);
      empty = wait(empty);
      buffer_item = value;
      printf("buffer:%d\n", buffer_item);
      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);
        printf("consumer %d consumed %d\n", id + 1, buffer_item);
        printf("current buffer len: 0\n");
        mutex = signal(mutex);
    } else {
        printf("buffer is empty!\n");
    }
}

int main() {
    int np, nc, cap;

    printf("enter no. of producers:");
    scanf("%d", &np);
    printf("enter no. of consumers:");
    scanf("%d", &nc);
    printf("enter buffer capacity:");
    scanf("%d", &cap);

    empty = cap;

    for (int i = 1; i <= np; i++) {
        printf("producer created: %d\n", i);
    }

    for (int i = 1; i <= nc; i++) {
        printf("consumer created: %d\n", i);
    }

    while (1) {
        producer(1);
        consumer(1);
`
```

```
    }
    return 0;
}
```

```
enter no. of producers:1
enter no. of consumers:1
enter buffer capacity:1
producer created: 1
consumer created: 1
16
producer 1 produced 16
buffer:16
consumer 2 consumed 16
current buffer len: 0
32
producer 1 produced 32
buffer:32
consumer 2 consumed 32
current buffer len: 0
4
producer 1 produced 4
buffer:4
consumer 2 consumed 4
current buffer len: 0
0
producer 1 produced 0
buffer:0
consumer 2 consumed 0
current buffer len: 0
```

`

Producer - consumer

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

int mutex = 1;
int full = 0;
int empty = 3;
int buffer_item = 0;

int wait (int s){
  return (--s);
}
int signal (int s){
  return (++s);
}
void producer (int id){
  if ((mutex == 1) && (empty != 0)){
    int value;
    scanf ("%d", &value);
    printf ("producer %d produced %d\n",
                    id, value);
    mutex = wait (mutex);
    full = signal (full);
    empty = wait (empty);
    buffer_item = value;
    printf ("buffer : %d\n", buffer_item);
    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);
    printf ("consumer %d consumed %d\n",
                id+1, buffer_item);
    printf ("current buffer len 0\n");
    mutex = signal (mutex);
  } else {
    printf ("buffer is empty!!\n");
  }
}

int main (){
  int np, nc, cap;
  printf ("Enter no of producers ");
  scanf ("%d", &np);
  printf ("Enter no of consumers ");
  scanf ("%d", &nc);
  printf ("Enter buffer capacity ");
  scanf ("%d", &cap);

  empty = cap;
```

```
for (int i=1; i<=np; i++){
    printf("producer created: %d\n",i);
}
for (int i=1; i<=nc; i++){
    printf("consumer created: %d\n",i);
}
while (1){
    producer (1);
    consumer (1);
}
return 0;
}
```

OUTPUT:

```
Enter no. of producers: 1
Enter no. of consumers: 1
Enter buffer capacity: 1
Succesfully created producer 1
Producer 1 produced 16
Succesfully created consumer 1
Buffer : 16
Consumer 2 consumed 16
Current buffer len: 0
Producer 1 produced 4
Buffer: 4
Consumer 2 consumed 4
Current buffer len: 0
```

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

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

#define MAX 5
sem_t mutex;
sem_t chopstick[MAX];
int totalPhilosophers;
int hungryCount;
int hungryPhilosophers[MAX];
int choice;

void *philosopher(void *arg) {
    int id = *(int *)arg;
    printf("P %d is waiting\n", id + 1);
    sem_wait(&mutex);
    sem_wait(&chopstick[id]);
    sem_wait(&chopstick[(id + 1) % totalPhilosophers]);
    printf("P %d is granted to eat\n", id + 1);
    sleep(1);
    printf("P %d has finished eating\n", id + 1);
    sem_post(&chopstick[id]);
    sem_post(&chopstick[(id + 1) % totalPhilosophers]);
    sem_post(&mutex);
    pthread_exit(NULL);
}

int main() {
    pthread_t thread[MAX];
    int i;
    printf("Enter the total number of philosophers: ");
    scanf("%d", &totalPhilosophers);
```

`

```
    printf("How many are hungry: ");
    scanf("%d", &hungryCount);
    for (i = 0; i < hungryCount; i++) {
        printf("Enter philosopher %d position (1 to %d): ", i + 1, totalPhilosophers);
        scanf("%d", &hungryPhilosophers[i]);
        hungryPhilosophers[i]--;
    }
    for (i = 0; i < totalPhilosophers; i++) {
        sem_init(&chopstick[i], 0, 1);
    }
    sem_init(&mutex, 0, 1);
    while (1) {
        printf("\n1. One can eat at a time\n");
        printf("2. Two can eat at a time\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        if (choice == 3) {
            break;
        }
        if (choice == 1) {
            printf("Allow one philosopher to eat at any time\n");
            for (i = 0; i < hungryCount; i++) {
                for (int j = 0; j < hungryCount; j++) {
                    printf("P %d is waiting\n", hungryPhilosophers[j] + 1);
                }
                int *id = malloc(sizeof(int));
                *id = hungryPhilosophers[i];
                pthread_create(&thread[i], NULL, philosopher, id);
                pthread_join(thread[i], NULL);
            }
        }
    }
    return 0;
}
```

`

```
How many are hungry: 3
Enter philosopher 1 position (1 to 5): 2
Enter philosopher 2 position (1 to 5): 4
Enter philosopher 3 position (1 to 5): 5

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 2 is waiting
P 4 is waiting
P 5 is waiting
P 2 is waiting
P 2 is granted to eat
P 2 has finished eating
P 2 is waiting
P 4 is waiting
P 5 is waiting
P 4 is waiting
P 4 is granted to eat
P 4 has finished eating
P 2 is waiting
P 4 is waiting
P 5 is waiting
P 5 is waiting
P 5 is granted to eat
P 5 has finished eating

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

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

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 2 is waiting
P 4 is waiting
P 5 is waiting
P 2 is waiting
P 2 is granted to eat
```

`

Dining Philosopher

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

#define N 5
sem_t mutex;
sem_t chopstick [MAX];
int totalPhilosophers;
int hungryCount;
int hungryPhilosophers [MAX];
int choice;

void * philosopher (void * arg){
    int id = *(int *) arg;
    printf ("P %d is waiting \n", id + 1);
    sem_wait (&mutex);
    sem_wait (&chopsticks [id]);
    sem_wait (&chopsticks (id + 1) %
                totalPhilosophers]);
    printf ("P %d is granted to eat", id+1);
    sleep (1);
    printf ("P %d has finished eating \n",
                id + 1);
    sem_post (&chopstick [id]);
    sem_post (&chopstick (id + 1) %
                totalPhilosophers]);
    sem_post (&mutex);
    pthread_exit (NULL);
}
```

```
int main(){
  pthread_t thread[MAX];
  int i;
  printf("Enter no. of philosophers ");
  scanf("%d",&totalPhilosophers);
  printf("How many are hungry ");
  scanf("%d",&hungryCount);
  for(i=0; i<hungryCount; i++){
    printf("Enter philosopher %d position
      (1 to %d) ",i+1,totalPhilosophers);
    scanf("%d",&hungryPhilosopher[i]);
    hungryPhilosophers[i]--;
  }

  for(i=0; i<totalPhilosophers; i++){
    sem_init(&chopstick[i],0,1);
  }
  sem_init(&mutex,0,1);
  while(1){
    printf("1. One can eat at a time");
    printf("2. Two can eat at a time");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    if(choice==3){
      break;
    }
    if(choice==3){
      break;
    }
```

```
    if(choice==1){
      printf("Allow one philosopher to
        eat at any time");
      for(i=0; i<hungryCount; i++)
        for(int j=0; j<hungryCount; j++){
          printf("P %d is waiting\n",
            hungryPhilosophers[j]+1);
        }
      int *id = malloc(sizeof(int));
      *id = hungryPhilosophers[i];
      pthread_create(&thread[i],NULL,
          philosopher,id);
      pthread_join(thread[i],NULL);
    }
  }
  return 0;
}
```

OUTPUT:

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
P3 is granted to eat
P3 is waiting

```
P4 is waiting
P4 is granted to eat
P4 is waiting
P0 is waiting
P0 is granted to eat
P0 is waiting
```

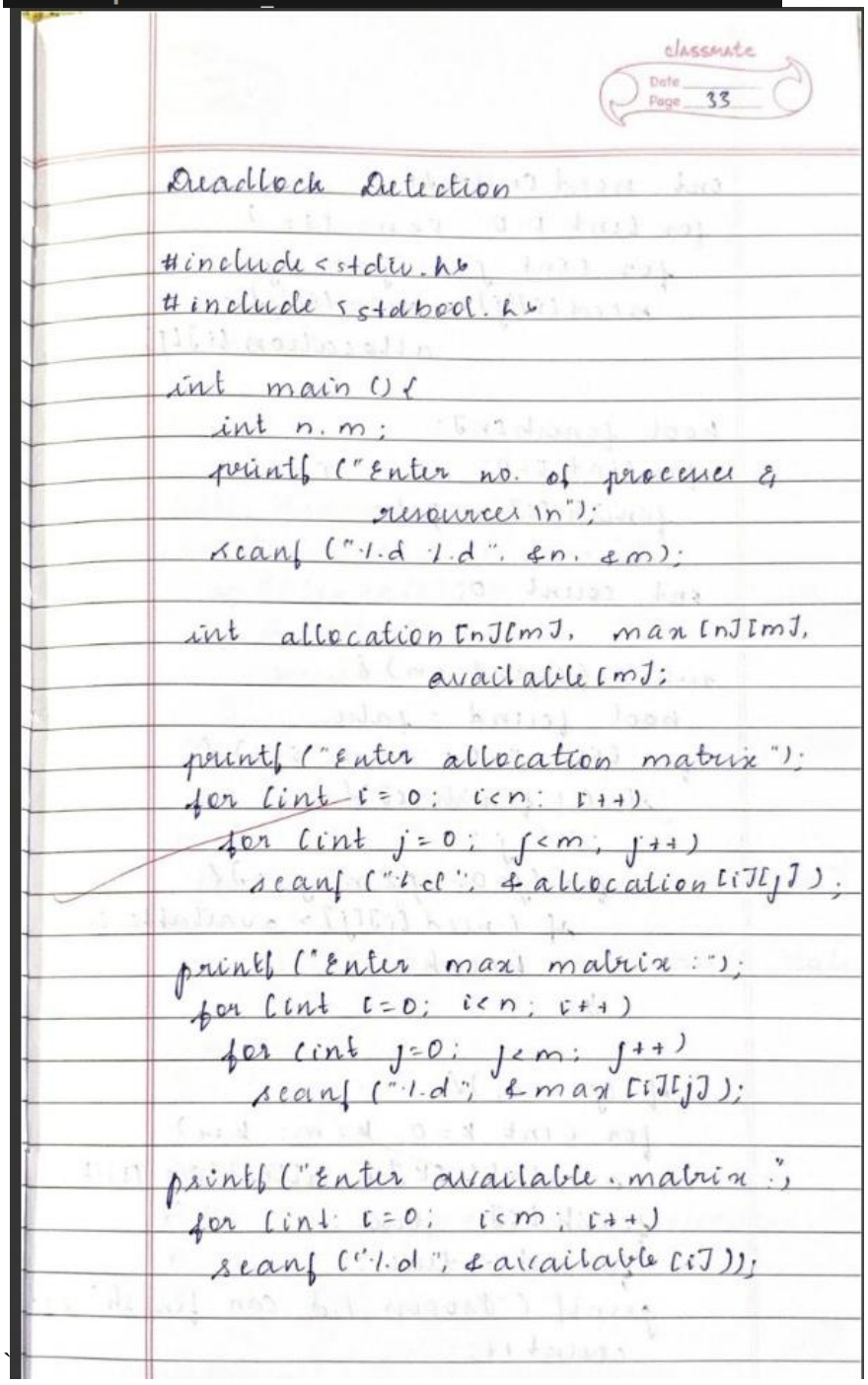
7.Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

```
#include <stdio.h>
#include
<stdbool.h> int
main() {
    int n, m;
    printf("enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);
    int alloc[n][m], max[n][m],
    avail[m]; printf("enter allocation
    matrix:\n"); for (int i = 0; i < n;
    i++)
        for (int j = 0; j < m; j++)
            scanf("%d",
            &alloc[i][j]);
    printf("eter max
    matrix:\n"); for (int i = 0; i
    < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d",
            &max[i][j]);
    printf("enter available
    matrix:\n"); for (int i = 0; i < m;
    i++)
        scanf("%d",
    &avail[i]); int
    need[n][m];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m;
        j++)
            need[i][j] = max[i][j] -
    alloc[i][j]; bool finish[n];
    for (int i = 0; i < n; i++)
        finish[i] = false;
    int
    safeSeq[n];
    int
    work[m];
    for (int i = 0; i < m; i++)
`
```

```
        work[i] = avail[i];
    int count = 0;
    while (count <
    n) {
        bool found = false;
        for (int p = 0; p < n; p++)
            { if (!finish[p]) {

                int j;
                for (j = 0; j < m; j++)
                    if (need[p][j] > work[j])
                        break;
                if (j == m) {
                    for (int k = 0; k < m; k++)
                        work[k] += alloc[p][k];

                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("sys is not in a safe state.\n");
            return 0;
        }
    }
    printf("sys is in safe state.\nsafe sequence is:
    ");
    for (int i = 0; i < n; i++)
        printf("P%d%s", safeSeq[i], (i == n - 1) ? "\n" : " ->
    ");
        return 0;
}
```

`

```
5 3
enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
eter max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
enter available matrix:
3 3 2
sys is in safe state.
safe sequence is: P1 -> P3 -> P4 -> P0 -> P2
```



classmate
Date
Page 33

Deadlock Detection

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

int main (){
   int n, m;
   printf ("Enter no. of processes &
            resources in");
   scanf ("%d %d", &n, &m);

int allocation[n][m], max[n][m],
                available[m];

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

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

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

```
int need[n][m];
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
    need[i][j] = max[i][j] -
                 allocation[i][j]

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

int count = 0;

while (count <n){
  bool found = false;
  for (int i=0; i<n; i++){
    if (!finish[i]){
      int j;
      for (j=0; j<m; j++){
        if (need[i][j] > available[j])
          break;
      }

      if (j == m){
        for (int k=0; k<m; k++)
          available[k] += allocation[i][k];
        finish[i] = true;
        found = true;
        printf ("Process %d can finish", i);
        count++;
      }
    }
  }

  if (!found){
    break;
  }
}

bool deadlock = false;
for (int i=0; i<n; i++){
  if (!finish[i]){
    deadlock = true;
    break;
  }
}

if (deadlock)
  printf ("System in deadlock state");
else
  printf ("System not in deadlock state");

return 0;
}
```

OUTPUT:

Enter no. of processes & resources

5

3

Enter allocation matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter max 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
Process 0 can finish
Process 2 can finish.
System not in deadlock state.

8.Write a C program to simulate deadlock detection

```
#include <stdio.h>
#include
<stdbool.h> int
main() {
    int n, m;
    printf("enter num of processes and number of resources:\n");
    scanf("%d %d", &n, &m);
    int alloc[n][m], request[n][m], avail[m];
    printf("enter alloc matrix (%d x %d):\n", n,
    m); for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d",
            &alloc[i][j]);
    printf("enter req matrix (%d x %d):\n", n,
    m); for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &request[i][j]);

    printf("enter avail resource (%d values):\n",
    m); for (int i = 0; i < m; i++)
        scanf("%d",
    &avail[i]); int work[m];
    for (int i = 0; i < m; i++)
        work[i] = avail[i];
    bool finish[n];
    for (int i = 0; i < n; i++) {
        bool hasAllocation =
        false; for (int j = 0; j <
        m; j++) {
            if (alloc[i][j] != 0) {
                hasAllocation = true;
                break;
            }
        }
        finish[i] = hasAllocation ? false : true;
    }
    while (true) {
        bool progress = false;
```

`

```
    for (int i = 0; i < n;
        i++) { if (!finish[i])
        {
            bool canGrant = true;

            for (int j = 0; j < m; j++) {
                if (request[i][j] > work[j]) {
                    canGrant = false;
                    break;
                }
            }

            if (canGrant) {
                for (int j = 0; j < m; j++)
                    work[j] += alloc[i][j];
                finish[i] =
                true;
                progress =
                true;
            }
        }
    }

    if (!progress)
        break;
}
printf("\nDLD result:\n");
bool deadlock = false;

for (int i = 0; i < n;
    i++) { if (!finish[i])
    {
        printf("process P%d is deadlocked\n", i);
        deadlock = true;
    } else {
        printf("process P%d is not deadlocked\n", i);
    }
}
if (!deadlock)
    printf("\nno deadlock detected in the system.\n");
    return 0;
    }
```

`

```
enter num of processes and number of resources:
5 3
enter alloc matrix (5 x 3):
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
enter req matrix (5 x 3):
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2
enter avail resource (3 values):
0 0 0

DLD result:
process P0 is not deadlocked
process P1 is deadlocked
process P2 is deadlocked
process P3 is deadlocked
process P4 is deadlocked
```

`

Deadlock Avoidance

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

#define P 5
#define R 3

int available[R] = {3, 3, 2};

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

int allocation[P][R] = {
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 2},
    {2, 1, 1},
    {0, 0, 2}
};

int need[P][R];
```

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

bool isSafe(){
  int work[R];
  bool finish[P] = {false};
  int safeSeq[P];
  int count = 0;

  for(int i=0; i<R; i++)
    work[i] = available[i];

  while(count<P){
    bool found = false;
    for(int i=0; i<P; i++){
      if(!finish[i]){
        bool canProceed = true;
        for(int j=0; j<R; j++){
          if(need[i][j] > work[j]){
            canProceed = false;
            break;
          }
        }
        if(canProceed){
          for(int j=0; j<R; j++)
            work[j] += allocation[i][j];

          safeSeq[count++] = i;
          finish[i] = true;
          found = true;
        }
      }
    }

    if(!found){
      printf("System not in safe state");
      return false;
    }
  }
  printf("System in safe state Sequence is:");
  for(int i=0; i<P; i++)
    printf("%d", safeSequence[i]);
  printf("\n");

  return true;
}

int main(){
  calculateNeed();
  isSafe();
  return 0;
}
```

OUTPUT:

Enter no of process & resource

5 3

Enter allocation matrix

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Enter max matrix:

7 5 3

3 2 2

9 0 2

2 2 2

4 3 3

Enter available matrix:

3 3 2

System is in safe state

P1 → P3 → P4 → P0 → P2

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

a) Worst-fit

b) Best-fit

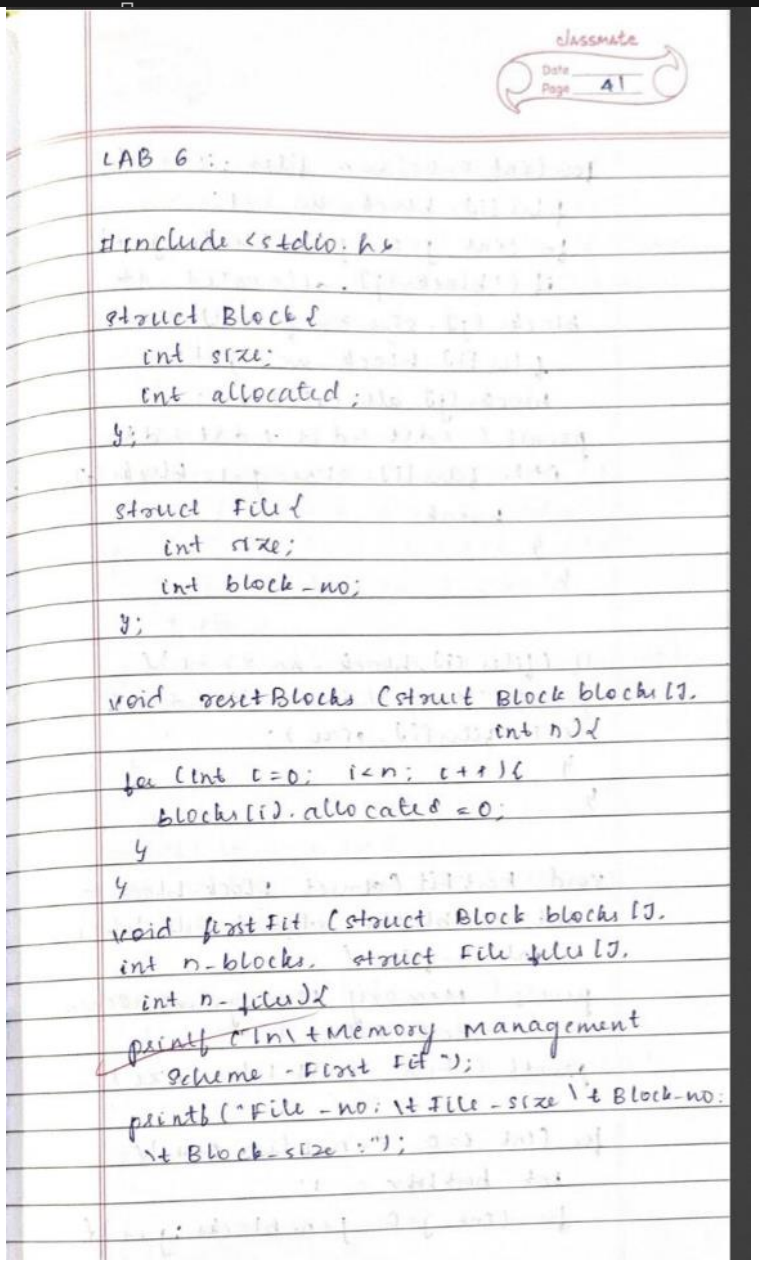
c) First-fit

```
#include <stdio.h>

struct Block {
    int size;
    int
    is_free;
};

struct
    File {
    int
    size;
};

void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\nfirst fit alloc\n");
    printf("file no\tfile size\tblock no\tblock size\tfrag\n");
    for (int i = 0; i < n_files; i++) {
        int allocated = 0;
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].size >= files[i].size) {
                blocks[j].is_free = 0;
                printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, j + 1,
blocks[j].size, blocks[j].size - files[i].size);
                allocated = 1;
                break;
            }
        }
        if (!allocated) {
            printf("file %d cannot be allocated\n", i + 1);
        }
    }
}

        }
        if (worst_block != -1) {
`
```

```
            blocks[worst_block].is_free = 0;
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, worst_block + 1,
blocks[worst_block].size, max_fragment);
        } else {
            printf("file %d cannot be allocated\n", i + 1);
        }
    }
}

int main() {
    int n_blocks, n_files;

    printf("enter the number of blocks:
    "); scanf("%d", &n_blocks);
    printf("enter the number of files:
    "); scanf("%d", &n_files);

    struct Block
    blocks[n_blocks]; struct File
    files[n_files];

    for (int i = 0; i < n_blocks; i++) {
        printf("enter the size of block %d: ", i +
        1); scanf("%d", &blocks[i].size);
        blocks[i].is_free = 1;
    }

    for (int i = 0; i < n_files; i++) {
        printf("enter the size of file %d: ", i +
        1); scanf("%d", &files[i].size);
    }

    firstFit(blocks, n_blocks, files, n_files);

    for (int i = 0; i < n_blocks; i++) blocks[i].is_free = 1;

    bestFit(blocks, n_blocks, files, n_files);

    for (int i = 0; i < n_blocks; i++) blocks[i].is_free = 1;

    worstFit(blocks, n_blocks, files, n_files);

    return 0;
`
```

```
}
```

```
enter the number of blocks: 5
enter the number of files: 4
enter the size of block 1: 100
enter the size of block 2: 500
enter the size of block 3: 200
enter the size of block 4: 300
enter the size of block 5: 600
enter the size of file 1: 212
enter the size of file 2: 417
enter the size of file 3: 112
enter the size of file 4: 420

first fit alloc
file no file size       block no        block size      frag
1       212             2               500             288
2       417             5               600             183
3       112             3               200             88
file 4 cannot be allocated
est fit alloc
file no file size       block no        block size      frag
1       212             4               300             88
2       417             2               500             83
3       112             3               200             88
4       420             5               600             180

worst fit alloc
file no file size       block no        block size      frag
1       212             5               600             388
2       417             2               500             83
3       112             4               300             188
file 4 cannot be allocated
```



LAB 6 :

```
#include <stdio.h>

struct Block {
    int size;
    int allocated;
};

struct File {
    int size;
    int block_no;
};

void resetBlocks (struct Block blocks[],
                  int n){
  for (int i=0; i<n; i++){
    blocks[i].allocated=0;
  }
}
void firstFit (struct Block blocks[],
int n_blocks, struct File files[],
  int n_files){
printf ("\n\tMemory Management
    Scheme - First Fit");
printf ("File_no:\tFile_size\tBlock_no:
  \tBlock_size:");
```

```
for (int i=0; i<n_files ; i++ ){
  files[i].block_no = -1;
  for (int j=0; j<n_blocks ; j++){
    if (!blocks[j].allocated &&
    blocks[j].size >= files[i].size){
      files[i].block_no = j+1;
      blocks[j].allocated = 1;
      printf("%d\t%d\t%d\t%d\t%d\n",
        i+1, files[i].size, j+1, blocks[j].size);
      break;
    }
  }

  if (files[i].block_no == -1){
    printf("%d\t%d\t\t\t\t\n",
      i+1, files[i].size);
  }
}

void bestFit(struct Block blocks[],
  int n_blocks, struct File files[],
    int n_files){
  printf("Memory Management Scheme -
        Best Fit\n");
  printf("File_no\tFile_size");

  for (int i=0; i<n_files; i++){
    int bestIdx = -1;
    for (int j=0; j<n_blocks ; j++){
      if (!blocks[j].allocated &&
      blocks[j].size >= files[i].size){
        if (bestIdx == -1 || blocks[j].size <
              blocks[bestIdx].size){
          bestIdx = j;
        }
      }
      if (bestIdx != -1){
        blocks[bestIdx].allocated = 1;
        files[i].block_no = bestIdx + 1;
        printf("%d\t%d\t%d\t%d\t%d\n",
          i+1, blocks[bestIdx].size);
      } else {
        printf("%d\t%d\t\t\t\t\n", i+1,
                files[i].size);
      }
    }

    printf("\nWorst fit");
    int wf[MAX]; for (int i=0; i<n;
        n_blocks, i++);

    wf[i] = block[i];
    allocate(wf, n_blocks, process, n
                        process);
    return 0;
  }
```

OUTPUT:

First Fit:

Proch no		
1	212	2
2	417	5
3	112	2
4	426	Not allocated

Best Fit:

1	212	4
2	417	2
3	112	3
4	426	5

Worst Fit:

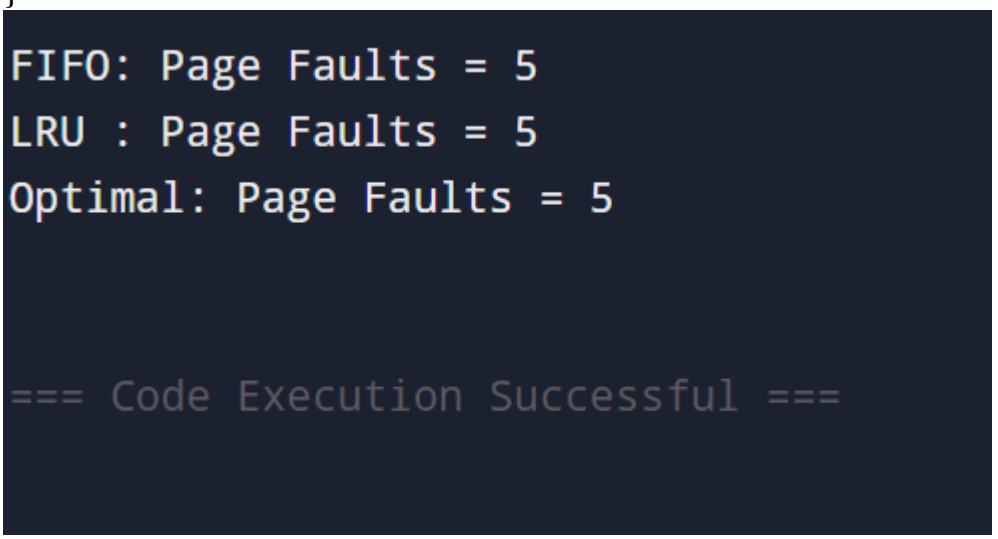
1	212	5
2	417	2
3	112	5
4	426	Not allocated

10. Write a C program to simulate page replacement algorithms of fifo, LRU and optimal

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

#define SIZE 7
#define FRAMES 3

int findLRU(int time[], int n) {
    int min = time[0], pos = 0;
    for (int i = 1; i < n; i++) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}

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

void fifo(int pages[]) {
    int frames[FRAMES], front = 0, count = 0, faults = 0;
    printf("\nFIFO: ");
    for (int i = 0; i < SIZE; i++) {
        bool hit = false;
`
```

```
        for (int j = 0; j < count; j++)
            if (frames[j] == pages[i]) hit = true;

        if (!hit) {
            if (count < FRAMES)
                frames[count++] = pages[i];
            else {
                frames[front] = pages[i];
                front = (front + 1) % FRAMES;
            }
            faults++;
        }
    }
    printf("Page Faults = %d\n", faults);
}

void lru(int pages[]) {
    int frames[FRAMES], time[FRAMES], count = 0, faults = 0;
    printf("LRU : ");
    for (int i = 0; i < SIZE; i++) {
        bool hit = false;
        for (int j = 0; j < count; j++) {
            if (frames[j] == pages[i]) {
                hit = true;
                time[j] = i;
                break;
            }
        }
        if (!hit) {
            if (count < FRAMES) {
                frames[count] = pages[i];
                time[count] = i;
                count++;
            } else {
                int pos = findLRU(time, FRAMES);
                frames[pos] = pages[i];
                time[pos] = i;
            }
            faults++;
        }
    }
    printf("Page Faults = %d\n", faults);
}

void optimal(int pages[]) {
    int frames[FRAMES], count = 0, faults = 0;
    printf("Optimal: ");

`
```

```
    for (int i = 0; i < SIZE; i++) {
        bool hit = false;
        for (int j = 0; j < count; j++) {
            if (frames[j] == pages[i]) {
                hit = true;
                break;
            }
        }
        if (!hit) {
            if (count < FRAMES)
                frames[count++] = pages[i];
            else {
                int pos = findOptimal(pages, frames, SIZE, i + 1);
                frames[pos] = pages[i];
            }
            faults++;
        }
    }
    printf("Page Faults = %d\n", faults);
}

int main() {
    int pages[SIZE] = {1, 2, 3, 2, 1, 4, 5};

    fifo(pages);
    lru(pages);
    optimal(pages);

    return 0;
}
```



`

FIFO. LRU and Optimal.

```
#include <stdio.h>

int search (int key, int frame[], int f){
  for(int i=0; i<f; i++)
    if (frame[i] == key);
      return i;
  return -1;
}

int findLRU (int time[], int f){
  int min = time[0], pos=0;
  for (int i=1; i<f; i++)
    if (time[i]<min) min = time[i],
                     pos=i;
  return pos;
}

int predict (int pages[], int frame[],
  int n, int idx, int f){
  int res=-1, far=idx;
  for(int i=0; i<f; i++){
    int j;
    for(j=idx; j<n; j++)
      if (j>far) far=j, res=i;
    break;
  }
```

```
    if (j == n) return i;
  }
  return (res == -1) ? 0 : res;
}

void simulate (int pages[], int n,
        int f, int type ){
  int frame [f], time [f], count = 0,
                 faults = 0;
  for (int i = 0; i < f; i++) frames[i] = -1;

  for (int i = 0; i < n; i++){
    int idx = search (pages [i], frames);
    if (idx == -1){
      int rep = 0;
      if (type == 1) rep = faults % f;
      elseif (type == 2) rep = (faults < f) ?
                                   faults :
                   findLRU (time, f);
    elseif (type == 3) rep = (faults < f) ?
    faults : predict (pages, frames, n, i+1);
      frame [rep] = pages [i];
      if (type == 2) time [rep] = count;
        faults++;
    } else if (type == 2){
      time [idx] = count;
    }
    count ++;
}
```



```
const char *names [] = { "", "FIFO", "LRU",
                         "Optimal" };
printf ("%s Page Faults : %d \n",
        names [type], faults);
}

int main (){
  int n, f, algo;
  printf ("Enter no of pages : ");
  scanf ("%d", &n);
  int pages [n];
  printf ("Enter page reference string : ");
  for (int i = 0; i < n; i++)
  scanf ("%d", &pages [i]);
  int pages [n];
  printf ("Enter no of frames");
  scanf ("%d", &f);
  printf ("1 FIFO\n 2 LRU \n 3 Optimal \n
        Enter choice : ");
  scanf ("%d", &algo);
  if (algo >= 1 && algo <= 3
     simulate (pages, n, f, algo);
  else
    printf ("Invalid choice ");
  return 0;
}
```

OUTPUT:

Enter no. of pages: 12
Enter page refernce string:
1 3 0 3 5 6 3 0 1 2 1 2
Enter no of frames: 3
1. FIFO
2. LRU
3. optimal
Enter choice: 2

LRU page Faults: 8

Enter choice: 3
optimal Page Faults: 7