

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



**LAB REPORT**  
on  
**OPERATING SYSTEMS**

Submitted by

**YOGARAJ KH (1WA23CS054)**

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 Yogaraj KH(1WA23CS054), 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.

Faculty Incharge Name  
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.	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)	1-10
2.	Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. → Priority (pre-emptive & Non-pre-emptive) →Round Robin (Experiment with different quantum sizes for RR algorithm)	10-15
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.	15-19
4.	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) aRate- Monotonic b) Earliest-deadline First	20-25
5.	Write a C program to simulate producer-consumer problem using semaphores	27-29
6.	Write a C program to simulate the concept of Dining Philosophers problem.	29-32
7.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	32-35
8.	Write a C program to simulate deadlock detection	35-38
9.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	39-42

10.	Write a C program to simulate page replacement algorithms a) FIFO LRU Optimal	43-45
-----	---	-------

## 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.

## 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)

=>FCFS:

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

```
typedef struct {
    int id, at, bt, wt, tat, ct, rem, rt, started;
} Process;

void sortByArrival(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) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}
```

```
void fcfs(Process p[], int n) {
    sortByArrival(p, n);
    int time = 0;

    for (int i = 0; i < n; i++) {
        if (time < p[i].at)
            time = p[i].at;

        p[i].ct = time + p[i].bt;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        time = p[i].ct;
    }
}
```

```

void display(Process p[], int n) {
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

    float totalWT = 0, totalTAT = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%.2f\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);
        totalWT += p[i].wt;
        totalTAT += p[i].tat;
    }

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

void main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }
    printf("First Come First Serve (FCFS)\n");
    fcfs(p, n);
    display(p, n);
}

```

Result:

```

P2 Arrival Time: 0
P2 Burst Time: 3
P3 Arrival Time: 0
P3 Burst Time: 4
P4 Arrival Time: 0
P4 Burst Time: 6

Process AT      BT      CT      TAT      WT      RT
P1      0        7        7        7        0        0
P2      0        3       10       10       7        7
P3      0        4       14       14      10       10
P4      0        6       20       20      14       14

Process returned 0 (0x0)   execution time : 16.832 s
Press any key to continue.

```

2. Write to formulate following scheduling algorithm to find turnaround time a) FCFS

```

Solution: #include <stdio.h>
Struct process {
    int at, bt, ct, tat, wt, rt;
};

void calcFCFS (Struct process proc[], int n) {
    int time = 0;
    for (int i = 0; i < n; i++) {
        if (time < proc[i].at)
            time = proc[i].at;
        proc[i].ct = time + proc[i].bt;
        proc[i].tat = proc[i].ct - proc[i].at;
        proc[i].wt = proc[i].tat - proc[i].bt;
        proc[i].rt = time - proc[i].at;
        time = proc[i].ct;
    }
}

int main() {
    int n;
    printf ("Enter no. of processes:");
    scanf ("%d", &n);
    Struct process proc[n];
    printf ("Enter arrival time and burst time for each process - \n");
    for (int i = 0; i < n; i++) {
        printf ("P%d AT:", i + 1);
        scanf ("%d", &proc[i].at);
        printf ("P%d BT:", i + 1);
        scanf ("%d", &proc[i].bt);
    }
}

```

Code

calcFS(procL, n);

Printf ("process AT BT CT TAT WT RT \n");

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

printf ("P%d AT %d BT %d CT %d TAT %d WT %d RT %d \n",

i+1, proc[i].at, proc[i].bt, proc[i].ct, proc[i].tat,

proc[i].wt, proc[i].rt);

3

return 0;

3

Input:

Enter number of processes: 4

Enter arrival time and burst time for each process:

P1 AT: 0

P1 BT: 7

P2 AT: 0

P2 BT: 3

P3 AT: 0

P3 BT: 7

P4 AT: 0

P4 BT: 6

Process AT BT CT TAT WT RT

P1 0 7 7 7 0 0

P2 0 3 10 10 7 7

P3 0 4 14 14 10 10

P4 0 6 20 20 14 14

Gantt chart

=>SJF(Non-preemptive):

```
#include <stdio.h>
#include <limits.h>
typedef struct {
    int id, arrival, burst, completion, turnaround, waiting;
} Process;
```

```

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

```

```

void sjf_non_preemptive(Process p[], int n) {
    int completed = 0, time = 0, minIdx;
    int isCompleted[n];
    for (int i = 0; i < n; i++) isCompleted[i] = 0;
    while (completed < n) {
        minIdx = -1;
        int minBurst = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (!isCompleted[i] && p[i].arrival <= time && p[i].burst < minBurst) {
                minBurst = p[i].burst;
                minIdx = i;
            }
        }
        if (minIdx == -1) { time++; continue; }
        p[minIdx].completion = time + p[minIdx].burst;
        p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival;
        p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst;
        time = p[minIdx].completion;
        isCompleted[minIdx] = 1;
    }
}

```

```

        completed++;
    }
}

void display(Process p[], int n) {
    printf("\nPID Arrival Burst Completion Turnaround Waiting\n");
    float totalWT = 0, totalTAT = 0;
    for (int i = 0; i < n; i++) {
        printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion,
               p[i].turnaround, p[i].waiting);
        totalWT += p[i].waiting;
        totalTAT += p[i].turnaround;
    }
    printf("\nAverage Waiting Time: %.2f\n", totalWT / n);
    printf("Average Turnaround Time: %.2f\n", totalTAT / n);
}

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }
    printf("\nShortest Job First (Non-Preemptive) Scheduling\n");
    sjf_non_preemptive(p, n);
    display(p, n);
    return 0;
}

```

}

Result:

<pre>#include &lt;iostream.h&gt; #include &lt;conio.h&gt; Select process S int at, bt, ct, tat, wt, rt, pid; char void Calcsrt (Select process proc[], int n) {     int times[2];     int completed[2];     int min_index = 0, min_index1, i;     for (int i = 0; i &lt; n; i++)         if (proc[i].at &lt;= times[0] &amp;&amp; i &lt; completed[0])             completed[0] = i;     if (completed[0] == -1)         min_index = 23;     else         min_index = 0;     for (int i = 0; i &lt; n; i++)         if (proc[i].at &lt;= times[0] &amp;&amp; i &lt; completed[0] &amp;&amp; proc[i].bt &lt; min_bt)             min_bt = proc[i].bt;     min_index = i; } if (min_index == -1)     time++; else     proc[min_index].ct = time + proc[min_index].bt;     proc[min_index].tat = proc[min_index].ct - proc[min_index].at;     proc[min_index].wt = proc[min_index].tat - proc[min_index].bt;     proc[min_index].rt = time - proc[min_index].at;     time = proc[min_index].ct; } int main() {     int n;     printf ("Enter no. of processes : ");     scanf ("%d", &amp;n);     Select process proc[n];     Printf ("Enter arrival time &amp; burst time for each process\n");     for (int i = 0; i &lt; n; i++)         for (int j = 0; j &lt; 2; j++)             scanf ("%d %d", &amp;proc[i].at, &amp;proc[i].bt);     Calcsrt (proc[n]);     printf ("Process ID AT BT CT TAT WT RT\n");     for (int i = 0; i &lt; n; i++)         printf ("%d %d %d %d %d %d %d\n", proc[i].pid, proc[i].at, proc[i].bt, proc[i].ct, proc[i].tat, proc[i].wt, proc[i].rt);     return 0; } </pre>	<pre>is - completed[0]min_index=7; completed[1]= n y int main() {     int n;     printf ("Enter no. of processes : ");     scanf ("%d", &amp;n);     Select process proc[n];     Printf ("Enter arrival time &amp; burst time for each process\n");     for (int i = 0; i &lt; n; i++)         for (int j = 0; j &lt; 2; j++)             scanf ("%d %d", &amp;proc[i].at, &amp;proc[i].bt);     Calcsrt (proc[n]);     printf ("Process ID AT BT CT TAT WT RT\n");     for (int i = 0; i &lt; n; i++)         printf ("%d %d %d %d %d %d %d\n", proc[i].pid, proc[i].at, proc[i].bt, proc[i].ct, proc[i].tat, proc[i].wt, proc[i].rt);     return 0; } </pre> <p>→ Enter number of processes: Enter AT and BT for each: P1 AT: 0 P1 BT: 7 P2 AT: 0 P2 BT: 3 P3 AT: 0 P3 BT: 4 P4 AT: 0 P4 BT: 6</p>
--	--

```
P1 Arrival Time: 0
P1 Burst Time: 7
P2 Arrival Time: 0
P2 Burst Time: 3
P3 Arrival Time: 0
P3 Burst Time: 4
P4 Arrival Time: 0
P4 Burst Time: 6
```

Process	AT	BT	CT	TAT	WT	RT
P1	0	7	20	20	13	13
P2	0	3	3	3	0	0
P3	0	4	7	7	3	3
P4	0	6	13	13	7	7

Process returned 0 (0x0) execution time : 14.656 s  
Press any key to continue.

Output						
process	AT	BT	CT	TAT	WT	RT
P1	0	7	20	20	13	13
P2	0	3	3	3	0	0
P3	0	4	7	7	3	3
P4	0	6	13	13	7	7

AU  
6/3/25

10/10

=>SJF(preemptive):

```

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

typedef struct {
    int id, arrival, burst, remaining, waiting, turnaround, completion, response, started;
} Process;

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

void sjfPreemptive(Process p[], int n) {
    int completed = 0, time = 0, minIndex, minBurst;

    while (completed < n) {
        minIndex = -1, minBurst = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= time && p[i].remaining > 0) {
                if (p[i].remaining < minBurst || (p[i].remaining == minBurst && p[i].arrival < p[minIndex].arrival)) {
                    minBurst = p[i].remaining;
                    minIndex = i;
                }
            }
        }
    }
}

```

```

    }

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

    if (p[minIndex].started == 0) {
        p[minIndex].response = time - p[minIndex].arrival;
        p[minIndex].started = 1;
    }

    p[minIndex].remaining--;
    time++;
    if (p[minIndex].remaining == 0) {
        p[minIndex].completion = time;
        p[minIndex].turnaround = p[minIndex].completion - p[minIndex].arrival;
        p[minIndex].waiting = p[minIndex].turnaround - p[minIndex].burst;
        completed++;
    }
}

}

```

```

void displayResults(Process p[], int n, const char *title) {
    printf("\n--- %s ---\n", title);
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    float totalWT = 0, totalTAT = 0, totalRT = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion,
               p[i].turnaround, p[i].waiting, p[i].response);
        totalWT += p[i].waiting;
        totalTAT += p[i].turnaround;
        totalRT += p[i].response;
    }
}

```

```

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

}

int main() {
    int n, choice;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n], temp[n];
    printf("Enter Arrival Time and Burst Time:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1; // Auto-generate PID
        scanf("%d %d", &p[i].arrival, &p[i].burst);
        p[i].remaining = p[i].burst;
        p[i].waiting = p[i].turnaround = p[i].completion = p[i].response = p[i].started = 0;
    }
    sjfPreemptive(p, n);
    displayResults(p, n, "Shortest Job First (Preemptive)");
    return 0;
}

```

Result:

```

SJF Preemptive:
#include <csedans.h>
#define MAX 200
typedef struct {
    int at, bt, ct, lat, wt;
} process;
3 processes:
int main() {
    int n, completed = 0, time = 0, min_bt, shrt;
    process p[4];
    printf("Enter the no. of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        if (p[i].at == 0) {
            printf("Enter the burst time for process %d: ", i + 1);
            scanf("%d", &p[i].bt);
            p[i].lat = p[i].bt;
        }
    }
    while (completed < n) {
        shorter_bt = -1, min_bt = 10000;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time & p[i].bt > 0 & p[i].lat < min_bt)
                min_bt = p[i].lat, shrt = i;
        }
        if (shrt == -1) {
            continue;
        }
        p[shrt].lat -= 1;
        p[shrt].ct = time + 1;
        p[shrt].wt = p[shrt].lat - p[shrt].at;
        completed++;
        time++;
    }
}

```

Date \_\_\_\_\_  
Page \_\_\_\_\_

```

printf("M A T \backslash B T \backslash C T \backslash T A T \backslash W T \n");
float Total = 0, Twt = 0;
for (int i = 0; i < n; i++) {
    printf("%d %d %d %d %d\n", p[i].at, p[i].bt, p[i].ct, p[i].wt);
    Total += p[i].lat;
    Twt += p[i].wt;
}
printf("\nAvg TAT: %.2f, Avg WT: %.2f\n", totalTAT / n, totalWT / n);
return 0;
}

```

uments\vicky042\SJF preemptive.exe"

processes: 4  
and burst time for process 1: 0  
and burst time for process 2: 1  
and burst time for process 3: 2  
and burst time for process 4: 3

WT	TAT
9	17
0	4
15	24
2	7

(0x0) execution time : 37.778 s  
continue.

Output:	Enter number of processes: 4			
	AT & BT of P1: 0			
	8			
	AT & BT of P2: 1			
	4			
	AT & BT of P3: 2			
	9			
	AT & BT of P4: 3			
	5			
	AT      BT      CT      TAT      WT			
0	8	17	14	9
1	4	5	4	0
2	9	26	24	15
3	5	10	7	2
Avg TAT: 13.00, Avg WT: 6.50				

## Program - 2

Question: Write a C program to simulate the Priority CPU scheduling algorithm to find turnaround time and waiting time.

=>CPU SCHEDULING:

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

```

#define MAX 10

typedef struct {
    int pid, at, bt, pt, remaining_bt, ct, tat, wt, rt, is_completed, st;
} Process;

void nonPreemptivePriority(Process p[], int n) {
    int time = 0, completed = 0;

    while (completed < n) {
        int lowest_priority = 9999, selected = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && !p[i].is_completed && p[i].pt < lowest_priority) {
                lowest_priority = p[i].pt;
                selected = i;
            }
        }
        if (selected == -1) {
            time++;
            continue;
        }
        if (p[selected].rt == -1) {
            p[selected].st = time;
            p[selected].rt = time - p[selected].at;
        }
        time += p[selected].bt;
        p[selected].ct = time;
        p[selected].tat = p[selected].ct - p[selected].at;
        p[selected].wt = p[selected].tat - p[selected].bt;
        p[selected].is_completed = 1;
        completed++;
    }
}

void preemptivePriority(Process p[], int n) {

```

```

int time = 0, completed = 0;
while (completed < n) {
    int lowest_priority = 9999, selected = -1;

    for (int i = 0; i < n; i++) {
        if (p[i].at <= time && p[i].remaining_bt > 0 && p[i].pt < lowest_priority) {
            lowest_priority = p[i].pt;
            selected = i;
        }
    }

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

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

    p[selected].remaining_bt--;
    time++;

    if (p[selected].remaining_bt == 0) {
        p[selected].ct = time;
        p[selected].tat = p[selected].ct - p[selected].at;
        p[selected].wt = p[selected].tat - p[selected].bt;
        completed++;
    }
}
}

```

```

void displayProcesses(Process p[], int n) {
    float avg_tat = 0, avg_wt = 0, avg_rt = 0;
    printf("\nPID\tAT\tBT\tPriority\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].pt, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
        avg_tat += p[i].tat;
        avg_wt += p[i].wt;
        avg_rt += p[i].rt;
    }
    printf("\nAverage TAT: %.2f", avg_tat / n);
    printf("\nAverage WT: %.2f", avg_wt / n);
    printf("\nAverage RT: %.2f\n", avg_rt / n);
}

int main() {
    Process p[MAX];
    int n, choice;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid);
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
        printf("Priority (lower number means higher priority): ");
        scanf("%d", &p[i].pt);
        p[i].remaining_bt = p[i].bt;
        p[i].is_completed = 0;
        p[i].rt = -1;
    }
}

```

```

while (1) {
    printf("\nPriority Scheduling Menu:\n");
    printf("1. Non-Preemptive Priority Scheduling\n");
    printf("2. Preemptive Priority Scheduling\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            nonPreemptivePriority(p, n);
            printf("Non-Preemptive Scheduling Completed!\n");
            displayProcesses(p, n);
            break;
        case 2:
            preemptivePriority(p, n);
            printf("Preemptive Scheduling Completed!\n");
            displayProcesses(p, n);
            break;
        case 3:
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice! Try again.\n");
    }
}
return 0;
}

```

Result:

## → Priority preemptive

```
#include <stdio.h>
#include <limits.h>
Struct Process {
    int pid, at, bt, pr, ct, tat, wt, rt, remaining;
} p[5];
void APPS (Struct Process p[], int n) {
    int completed = 0, time = 0, min_idx = -1;
    float totalwt = totaltat = 0;
    for (int i = 0; i < n; i++) {
        p[i].rt = -1;
    }
    while (completed != n) {
        if (p[min_idx].remaining == p[i].bt) {
            p[min_idx].rt = -1;
        }
        time++;
        if (p[min_idx].at == time) {
            completed++;
            p[min_idx].ct = time;
            p[min_idx].wt = p[min_idx].tat - p[min_idx].at;
            p[min_idx].tat = p[min_idx].bt + p[min_idx].wt;
            totalwt += p[min_idx].wt;
            totaltat += p[min_idx].tat;
        }
    }
}
```

```
int minPriority = INT_MAX;
min_idx = -1;
for (int i = 0; i < n; i++) {
    if ((p[i].at <= time) && (p[i].remaining > 0) && (p[i].pr < minPriority)) {
        minPriority = p[i].pr;
        min_idx = i;
    }
}
if (min_idx == -1) {
    time++;
    continue;
}
if (p[min_idx].at == -1) {
    p[min_idx].at = time;
}
p[min_idx].remaining--;
time++;
if (p[min_idx].remaining == 0) {
    completed++;
    p[min_idx].ct = time;
    p[min_idx].wt = p[min_idx].tat - p[min_idx].at;
    p[min_idx].tat = p[min_idx].bt + p[min_idx].wt;
    totalwt += p[min_idx].wt;
    totaltat += p[min_idx].tat;
}
printf ("PID AT BT PR CT TAT WT RT\n");
for (int i = 0; i < n; i++) {
    printf ("%d %d %d %d %d %d %d %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 ("\nAverage Waiting Time: %.2f\n", totalwt/n);
```

Enter the number of processes: 5  
 Enter Arrival Time, Burst Time, and Priority for each process:  
 Process 1: 0  
 10  
 3  
 Process 2: 0  
 1  
 1  
 Process 3: 0  
 2  
 4  
 Process 4: 0  
 1  
 5  
 Process 5: 0  
 5  
 2  
 PID AT BT PR CT TAT WT RT  
 1 0 10 3 16 16 6 6  
 2 0 1 1 1 1 0 0  
 3 0 2 4 18 18 16 16  
 4 0 1 5 19 19 18 18  
 5 0 5 2 6 6 1 1  
 Average Turnaround Time: 12.00  
 Average Waiting Time: 8.20

Output: Total number of process : 7
Enter AT, BT, Priority:
process 1: 0 8 3
process 2: 1 9 4
process 3: 3 4 4
process 4: 4 1 5
process 5: 5 6 2
process 6: 6 5 6
process 7: 7 1 1

PID	AT	BT	PR	CT	TAT	WT	RT
1	0	8	3	15	15	7	0
2	1	9	4	17	16	14	4
3	3	4	4	81	78	74	4
4	4	5	2	22	16	17	12
5	5	6	2	12	7	1	0
6	6	5	6	27	21	16	16
7	7	1	1	8	4	0	0

Output:	Enter the no. of processes: 5										
	Enter AT, BT, and Priority for each process:										
Process 1: 0											
1 0 3											
process 2: 0 1 1											
process 3: 0 2 4											
process 4: 0 1 5											
process 5: 0 5 2											
	PID AT AT PR CT TAT WT RT										
P1	0 10 3 16 16 6 6										
P2	0 1 1 1 1 0 0										
P3	0 2 4 18 18 16 16										
P4	0 1 5 19 19 18 18										
P5	0 5 2 6 6 1 1										
	Average TAT : 13.00										
	Average WT : 8.20										
	<table border="1"><tr><td>P2</td><td>P5</td><td>P1</td><td>P3</td><td>P4</td></tr><tr><td>1</td><td>6</td><td>16</td><td>18</td><td>19</td></tr></table>	P2	P5	P1	P3	P4	1	6	16	18	19
P2	P5	P1	P3	P4							
1	6	16	18	19							

### Program - 3

Write a C program to simulate a 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.

#### =>MULTI LEVEL SCHEDULING:

```
#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];
        }
    }
    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;
}

```

Result:

Date: 3-29-19

<pre> Multilevel Queue Scheduling #include &lt;conio.h&gt; #define MAX PROCESSES 10 #define TIME QUANTUM 2 typedef struct {     int bt, at, q_type, wt, tat, rt; } process; void RR(process p[10], int n, int t_q, int time); int done, i; do {     done = 1;     for (i = 0; i &lt; n; i++) {         if (p[i].at &gt; t_q) {             time += t_q;             p[i].at -= t_q;         } else {             *time += p[i].at;             p[i].wt = *time - p[i].at;             p[i].tat = *time - p[i].at;             p[i].rt = p[i].wt;             p[i].wt = 0;         }     } } while (!done); void FCFS(process p[10], int n, int time) {     for (i = 0; i &lt; n; i++) {         if (*time &lt; p[i].at) {             *time = p[i].at;             p[i].wt = *time - p[i].at;             p[i].tat = p[i].wt + p[i].bt;             *time += p[i].bt;         }     } } </pre>	<pre> int main() {     process p[MAX PROCESSES], s_q[MAX PROCESSES], u_q[MAX PROCESSES];     int n, sys_count = 0, v_l = 0, time = 0;     float avg_waiting = 0, avg_tat = 0, avg_response = 0,         throughput;     priority ("Enter no. of processes:");     Scanf ("%d", &amp;n);     for (int i = 0; i &lt; n; i++) {         priority ("Enter AT, A[i] &amp; Burst Time:");         Scanf ("%d %d %d", &amp;p[i].at, &amp;p[i].bt, &amp;p[i].q_type);     }     p[0].wt = p[0].bt;     if (p[0].q_type == 1) {         s_q[sys_count] = p[0];         sys_count++;     } else {         u_q[sys_count] = p[0];         sys_count++;     }     for (int i = 0; i &lt; n - 1; i++) {         for (int j = 0; j &lt; u_q.size(); j++) {             if (u_q[j].at &gt; u_q[i].at) {                 process temp = u_q[i];                 u_q[i] = u_q[j];                 u_q[j] = temp;             }         }     }     priority ("in queue 1 is system process in Queue 2 is user");     process();     float (s_q, sys_count, time quantum, *time);     float (u_q, u_l, *time);     priority ("in process WT TAT RT");     for (int i = 0; i &lt; sys_count; i++) {         avg_waiting += s_q[i].wt;     }     avg_waiting /= sys_count;     throughput = sys_count / time; } </pre>
--	--

```

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          7          2
3      7          8          7
4      8         11          8

Average Waiting Time: 4.25
Average Turn Around Time: 7.00
Average Response Time: 4.25
Throughput: 0.36
Process returned 0 (0x0)  execution time: 11.000 s
Process returned 0 (0x0)  execution time : 21.307 s
Press any key to continue.

```

<pre> enter process : 4 enter BT, AT &amp; Queue: P1: 201 P2: 102 P3: 501 P4: 302 </pre>	<pre> process WT TAT RT P1 0 2 0 P3 2 7 2 P2 7 8 7 P4 8 11 8 </pre>
--	---

ANG WT : 4.25  
 Avg TAT : 7.00  
 Avg RT : 4.25  
 Throughput : 0.37

## Program -4

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

- a) Rate- Monotonic
- b) Earliest-deadline First

=> Rate Monotonic

```
#include <stdio.h>
#define MAX PROCESSES 10
```

```

typedef struct {
    int id;
    int burst_time;
    int period;
    int remaining_time;
    int next_deadline;
} Process;

void sort_by_period(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].period > processes[j + 1].period) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

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

```

```

    return result;
}

double utilization_factor(Process processes[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += (double)processes[i].burst_time / processes[i].period;
    }
    return sum;
}

double rms_threshold(int n) {
    return n * (pow(2.0, 1.0 / n) - 1);
}

void rate_monotonic_scheduling(Process processes[], int n) {
    int lcm_period = calculate_lcm(processes, n);
    printf("LCM=%d\n\n", lcm_period);
    printf("Rate Monotone Scheduling:\n");
    printf("PID  Burst  Period\n");
    for (int i = 0; i < n; i++) {
        printf("%d    %d    %d\n", processes[i].id, processes[i].burst_time, processes[i].period);
    }

    double utilization = utilization_factor(processes, n);
    double threshold = rms_threshold(n);
    printf("\n%.6f <= %.6f => %s\n", utilization, threshold, (utilization <= threshold) ? "true" :
    "false");

    if (utilization > threshold) {
        printf("\nSystem may not be schedulable!\n");
        return;
    }
    int timeline = 0, executed = 0;
}

```

```

while (timeline < lcm_period) {
    int selected = -1;
    for (int i = 0; i < n; i++) {
        if (timeline % processes[i].period == 0) {
            processes[i].remaining_time = processes[i].burst_time;
        }
        if (processes[i].remaining_time > 0) {
            selected = i;
            break;
        }
    }
    if (selected != -1) {
        printf("Time %d: Process %d is running\n", timeline, processes[selected].id);
        processes[selected].remaining_time--;
        executed++;
    } else {
        printf("Time %d: CPU is idle\n", timeline);
    }
    timeline++;
}

```

```

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].id = i + 1;
        scanf("%d", &processes[i].burst_time);
    }
}

```

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

Result :

```

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

System may not be schedulable!

Process returned 0 (0x0) execution time : 18.410 s
Press any key to continue.

```

*Rate monotonic*

```

#include <stdio.h>
#include <math.h>
#define MAX_PROCESS 10

typedef struct {
    int id, bt, period, rt, md;
} process;

void sort(process processes[], int n)
{
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (processes[j].period > processes[i + j].period)
                swap(&processes[i], &processes[i + j]);
}

int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

int lcm(int a, int b)
{
    if (a == 0 || b == 0)
        return a + b;
    else
        return (a * b) / gcd(a, b);
}

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

double wf(process processes[], int n)
{
    double sum = 0;
    for (int i = 0; i < n; i++)
        sum += (double) processes[i].bt / processes[i].period;
    return sum;
}

```

*period*:

```

void sort(process processes[], int n)
{
    int sum_period = lcm(processes, n);
    printf("Lcm = %d\n", sum_period);
    printf("Rate monotonic Scheduling: (%d)\n");
    printf("PnP burst period (%d)\n");
    for (int i = 0; i < n; i++)
        printf("%d.%d %d.%d", processes[i].id,
               processes[i].bt, processes[i].period);
    double utilization = utilization_factor(processes, n);
    double threshold = sum_threshold(n);
    printf("\n%.6f <= %.6f => %s\n", utilization,
          threshold, (utilization < threshold) ? "true" :
          "false");
    if (utilization > threshold)
        printf("In system may not be schedulable\n");
    return;
}

int timeline = 0, executed = 0;
while (timeline < lcm * period)
{
    int selected = -1;
    for (int i = 0; i < n; i++)
        if (timeline - processes[i].period == 0)

```

Output: LCM of the no. of processes = 3  
 Enter the CPU BT : 3 6 8  
 Enter time periods: 3 4 5  
~~LCM = 60~~  
 RMS:  

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

~~4.100000 <= 0.47976 3 => false~~  
~~System may not be Schedulable!~~

=> 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%d\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);
    }

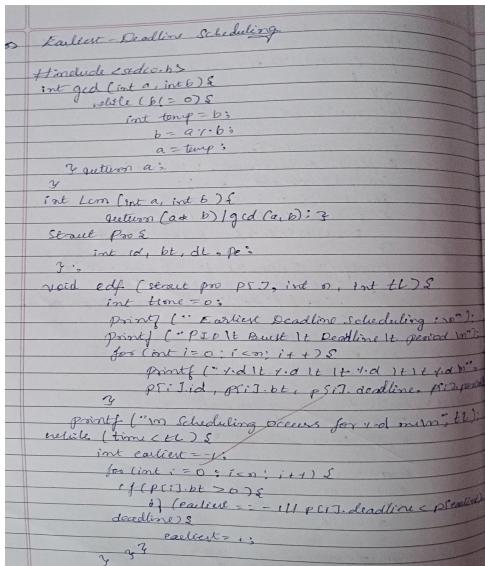
    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].period);
    }

    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;
}

```

## Result:



Handwritten notes for Earliest Deadline Scheduling:

```

    #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 Proc {
        int id, bt, dl, pe;
    };

    void edf (struct Proc ps[], int n, int tl) {
        int time = 0;
        printf ("Earliest Deadline Scheduling : \n");
        printf ("Process ID Burst Time Deadline Period :\n");
        for (int i = 0; i < n; i++) {
            printf ("%d %d %d %d\n", ps[i].id, ps[i].bt, ps[i].dl, ps[i].pe);
        }

        printf ("In scheduling occurs for val min: %d\n", time);
        while (time < tl) {
            int earliest = -1;
            for (int i = 0; i < n; i++) {
                if (ps[i].bt > 0) {
                    if (earliest == -1 || ps[i].deadline < ps[earliest].deadline)
                        earliest = i;
                }
            }
            ps[earliest].bt -= 1;
            if (ps[earliest].bt == 0)
                ps[earliest].dl = -1;
            time++;
        }
    }

```

Handwritten code for Earliest Deadline Scheduling:

```

    #include <stdio.h>
    #include <math.h>
    int gcd (int a, int b) {
        if (a == 0) break;
        printf ("In gcd %d ms > task %d is running in time %d ms\n", b, a);
        ps[earliest].id--;
        time++;
    }
    int lcm (int a, int b) {
        int m1 = a, m2 = b;
        int max = a > b ? a : b;
        for (int i = 1; i < max; i++) {
            if (m1 * i % b == 0) {
                printf ("Lcm of %d & %d is %d\n", a, b, i);
                return i;
            }
        }
    }

    struct Proc {
        int id, bt, dl, pe;
    };

    void edf (struct Proc ps[], int n, int tl) {
        int time = 0;
        printf ("Earliest deadline = %d\n");
        for (int i = 0; i < n; i++) {
            printf ("%d %d %d %d\n", ps[i].id, ps[i].bt, ps[i].dl, ps[i].pe);
        }

        printf ("Enter the time period = %d\n");
        for (int i = 0; i < n; i++) {
            printf ("Process %d has %d ms\n", i, ps[i].pe);
        }

        int hyperperiod = ps[0].pe;
        for (int i = 1; i < n; i++) {
            if (ps[i].pe > hyperperiod) {
                hyperperiod = lcm (hyperperiod, ps[i].pe);
            }
        }

        printf ("earliest edf (%d, %d, %d)\n", n, tl, 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.
(base) thanugeorge@Thanus-MacBook-Pro Desktop %

```

Output : Number of processes : 3  
 Enter CPU BT : 2 3 4  
 Enter deadlines : 1 2 3  
 Enter time period: 1 2 3  
 Earliest deadline scheduling :

PID	Burst	D <sub>d</sub>	P <sub>e</sub>
1	2	1	1
2	3	2	2
3	1	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

6ms : Task 3 is running

## Program 5

Write a C program to simulate producer-consumer problem using semaphores  
 => Producer Consumer

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

```

int mutex = 1, full = 0, empty = 3, x = 0;

void wait(int *s) {
    --(*s);
}

void signal(int *s) {
    ++(*s);
}

void producer() {
    wait(&empty);
    wait(&mutex);
    x++;
    printf("The item produced is %d\n", x);
    signal(&mutex);
    signal(&full);
}

```

```

}

void consumer() {
    wait(&full);
    wait(&mutex);
    printf("Consumed item %d\n", x);
    x--;
    signal(&mutex);
    signal(&empty);
}

int main() {
    int choice;
    do {
        printf("\n1. Produce\n2. Consume\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if ((mutex == 1) && (empty != 0)) {
                    producer();
                } else {
                    printf("The buffer is full\n");
                }
                break;
            case 2:
                if ((mutex == 1) && (full != 0)) {
                    consumer();
                } else {
                    printf("The buffer is empty\n");
                }
                break;
            case 3:
                printf("Exiting.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 3);
    return 0;
}

```

## Result:

Output :

```

1. produce
2. consume
3. Exit

Enter your choice : 1
producer produces item 1
Enter your choice : 1
producer produces item 2
enter your choice : 1
producer produces item 3
Enter your choice : 1
producer produces item 3
Enter your choice : 1
buffers are full
Enter your choice : 2
consumer consumes item 3
Enter your choice : 2
consumer consumes item 2
Enter your choice : 2
consumer consumes item 1
buffer is empty ?

```

```

1. Produce
2. Consume
3. Exit
Enter Choice: 2
Buffer is Empty
Enter Choice: 1
Item produced: 1
Enter Choice: 1
Item produced: 2
Enter Choice: 1
Item produced: 3
Enter Choice: 1
Buffer is Full
Enter Choice: 1
Buffer is Full
Enter Choice: 2
Item Consumed: 3
Enter Choice: 2
Item Consumed: 2
Enter Choice: 2
Item Consumed: 1
Enter Choice: 2
Buffer is Empty
Enter Choice: 2
Buffer is Empty
Enter Choice: 2
Buffer is Empty
Enter Choice: 3

```

```

produce();
else
{
    printf("Buffer is full!\n");
    break;
}
case 2: if ((multi == 1) && (full == 0))
{
    consume();
}
else
{
    printf("Buffer is empty!\n");
    break;
}
default: printf("Invalid choice!\n");
}
}
return 0;
}

```

```

Producer consumer.
#include<cslib.h>
#include<cslib.h>
int multi = 1;
int full = 0, empty = 3, x = 0;
int wait(int s) { return (s); }
int signal(int s) { return (++s); }
void producer() {
    multi = wait(counter);
    full = signal(full);
    empty = wait(empty);
    x++;
    printf("producer produces item %d\n", x);
    multi = signal(multi);
}
void consumer() {
    multi = wait(counter);
    full = signal(full);
    empty = wait(empty);
    x--;
    printf("consumer consumes item %d\n", x);
    multi = signal(counter);
}
int main() {
    int choice;
    include(1);
    printf("1. produce 2. consume 3. Exit\n");
    printf("Enter your choice:");
    scanf("%d", &choice);
    switch(choice) {
        case 1: if ((multi == 1) && (empty == 0))

```

## Program 6

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

```
#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* philosopher(void* num);
void take_fork(int phnum);
void put_fork(int phnum);
void test(int phnum);

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

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

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

    for (i = 0; i < N; i++) {
        // create philosopher processes
        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;
    }

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

// take up chopsticks
void take_fork(int phnum) {
    sem_wait(&mutex);
    // state that hungry
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
    sleep(1);
}

// put down chopsticks
void put_fork(int phnum) {
    sem_wait(&mutex);

    // state that thinking
    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(0);
        put_fork(*i);
    }
}

```

Result:

```

Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 takes fork 5 and 1

```

$\# \text{Combinations} = 120$

property 1: "No combination found where 2 or more philosophers can eat at once";

property 2: "Total philosophers <= 5";

property 3: "Fewer than many are hungry";

property 4: "At most one philosopher can eat at a time";

for first i = 0; i < hungry count; i++ {  
 print ("Total philosophers = " + i + " and hungry count = " + hungry count);  
 for first j = i + 1; j < hungry count; j++ {  
 print (" " + i + " & " + j + " hungry");  
 }  
 print (" " + i + " & " + hungry count);  
}

int choices;  
do {  
 print (" " + choices + " choices can eat at a time");  
 choices++;  
} while (choices < hungry count);  
print (" " + choices);  
switch (choice) {  
 case 1: option 1 (hungry count);  
 break;  
 case 2: option 2 (hungry count);  
 break;  
 case 3: print (" Exiting ... ");  
 break;  
 default: print (" Invalid choice ");  
}  
} while (choice != 0);

Dining - Philosophers

structure <combination>

<combination> = 11 abs (a - b) = = LP - 1;

need option 1 (int count) {  
 print (" " + count + " philosophers can eat at  
 any time ");  
 for (int i = 0; i < count; i++) {  
 print (" " + i + " is guaranteed to eat in ", hungry  
 count);  
 for (int j = i + 1; j < count; j++) {  
 print (" " + i + " & " + j + " are waiting in ", hungry  
 count);  
 }  
 }  
 print (" " + count + " philosophers to eat at  
 same time ");  
 int combinations = 1;  
 for (int i = 0; i < count; i++) {  
 for (int j = i + 1; j < count; j++) {  
 if (i < j) {  
 print (" " + i + " & " + j + " ");  
 print (" " + i + " & " + j + " hungry ");  
 print (" " + i + " & " + j + " combination ");  
 print (" " + i + " & " + j + " are guaranteed to eat  
 in ", hungry count);  
 for (int k = 0; k < count; k++) {  
 print (" " + i + " & " + j + " & " + k + " ");  
 }  
 }  
 }  
 }  
 print (" " + count + " ");  
}

Enter philosopher's problem: 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  
 P 2 is granted to eat  
 P 4 is waiting  
 P 5 is waiting  
 P 4 is granted to eat  
 P 2 is waiting  
 P 5 is waiting granted to eat  
 P 2 is waiting  
 P 4 is waiting

## Program 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, i, j, k;
  printf("Enter number of processes: ");
  scanf("%d", &n);
  printf("Enter number of resources: ");
  scanf("%d", &m);

  int alloc[n][m], max[n][m], avail[m];
  int need[n][m];

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

  printf("Enter max matrix (%d x %d):\n", n, m);
  for (i = 0; i < n; i++) {
    printf("Max for process %d: ", i);
  }
}

```

```

for (j = 0; j < m; j++)
    scanf("%d", &max[i][j]);
}

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

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

bool finish[n];
int safeSeq[n];
int count = 0;

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

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

            if (j == m) {
                for (k = 0; k < m; k++)
                    avail[k] += alloc[i][k];

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

    if (!found) {
        printf("System is not in safe state.\n");
        return 1;
    }
}

```

```

printf("System is in safe state.\n");
printf("Safe sequence is: ");
for (i = 0; i < n; i++) {
    printf("P%d", safeSeq[i]);
    if (i != n - 1)
        printf(" -> ");
}
printf("\n");

return 0;
}

```

## Result:

Banker's Algorithm

```

#include <csairds.h>
#include <stdlib.h>

int condition(int *need, int *work, int *intR, int m)
{
    for (int j = 0; j < m; j++)
        if (need[j] > work[j])
            return 0;
    return 1;
}

int safety(int m, int *allocated, int *need, int *available,
          int *sequence)
{
    int *readed = (int *)malloc((m + sizeof(int)) * sizeof(int));
    for (int i = 0; i < n; i++)
    {
        need[i] = (int++)malloc((n + sizeof(int)));
        for (int j = 0; j < m; j++)
            need[i][j] = max((int*)allocated[i][j], 0);
    }
    *readed = (int++)malloc((n + sizeof(int)));
    for (int i = 0; i < n; i++)
        work[i] = available[i];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
            work[i][j] = work[i] + need[i][j];
    }
    int *finWork = (int++)malloc((n + sizeof(int)));
    for (int i = 0; i < n; i++)
        finWork[i] = 0;
    int safeIndex = 0;
    int changed;
    do
    {
        changed = 0;
        for (int i = 0; i < n; i++)
        {
            if (safeIndex == 0)
                changed++;
            if (condition(need[i], work[i], finWork[i], m))
            {
                sequence[safeIndex] = i;
                changed++;
                for (int j = 0; j < m; j++)
                    finWork[i][j] = work[i][j];
                safeIndex++;
            }
        }
    } while (changed != 0);
}

int main()
{
    int n = 5, m = 3;
    printf("Enter allocation matrix: \n");
    int *allocated = (int++)malloc((n + sizeof(int)) * sizeof(int));
    for (int i = 0; i < n; i++)
    {
        allocated[i] = (int++)malloc((n + sizeof(int)));
        for (int j = 0; j < m; j++)
            allocated[i][j] = 0;
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &allocated[i][j]);
    printf("Enter max matrix: \n");
    int *max = (int++)malloc((n + sizeof(int)) * sizeof(int));
    for (int i = 0; i < n; i++)
    {
        max[i] = (int++)malloc((n + sizeof(int)));
        for (int j = 0; j < m; j++)
            max[i][j] = 0;
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &max[i][j]);
    printf("Enter available matrix: \n");
    int *available = (int++)malloc((n + sizeof(int)) * sizeof(int));
    for (int i = 0; i < n; i++)
        available[i] = 10;
    printf("Safe Sequence: ");
    for (int i = 0; i < n; i++)
        printf("%d ", sequence[i]);
    printf("\n");
}

```

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

```
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 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.  
Safe sequence is: P1 -> P3 -> P4 -> P0 -> P2  
  
Process returned 0 (0x0)  execution time : 47.859 s  
Press any key to continue.
```

## Program 8

Write a C program to simulate deadlock detection

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

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

```
int main() {
```

```

int n, m, i, j, k;

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

int allocation[n][m], request[n][m], available[m];
int work[m];
bool finish[n];

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

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

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

for (i = 0; i < n; i++) {
    bool zero_allocation = true;
    for (j = 0; j < m; j++) {
        if (allocation[i][j] != 0) {
            zero_allocation = false;
            break;
        }
    }
    finish[i] = zero_allocation;
}

bool found_process;
do {
    found_process = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_allocate = true;
            for (j = 0; j < m; j++) {
                if (request[i][j] > work[j]) {

```

```

        can_allocate = false;
        break;
    }
}
if(can_allocate) {
    for (k = 0; k < m; k++)
        work[k] += allocation[i][k];
    finish[i] = true;
    printf("Process %d can finish.\n", i);
    found_process = true;
}
}
}
} while (found_process);

bool deadlock = false;
for (i = 0; i < n; i++) {
    if (!finish[i]) {
        deadlock = true;
        break;
    }
}
if (deadlock)
    printf("System is in a deadlock state.\n");
else
    printf("System is not in a deadlock state.\n");

return 0;
}

```

Result:

```
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.
```

O/P →

process P<sub>0</sub> is finished.

process P<sub>1</sub> is finished

process P<sub>3</sub> is finished

process P<sub>7</sub> is finished

process P<sub>4</sub> is finished.

No deadlock detected, all processes can complete.

### Program 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 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\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    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\n", i + 1, files[i].size, j + 1, blocks[j].size);
                break;
            }
        }
        if (files[i].block_no == -1) {
            printf("%d\t%d\t%d\t_\n", i + 1, files[i].size);
        }
    }
}
```

```

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\n\tMemory Management Scheme – Best Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    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\n", i + 1, files[i].size, bestIdx + 1, blocks[bestIdx].size);
        } else {
            printf("%d\t%d\t_\n", i + 1, files[i].size);
        }
    }
}

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\n\tMemory Management Scheme – Worst Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    for (int i = 0; i < n_files; i++) {
        int worstIdx = -1;
        for (int j = 0; j < n_blocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= files[i].size) {
                if (worstIdx == -1 || blocks[j].size > blocks[worstIdx].size) {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            blocks[worstIdx].allocated = 1;
            files[i].block_no = worstIdx + 1;
            printf("%d\t%d\t%d\t%d\n", i + 1, files[i].size, worstIdx + 1, blocks[worstIdx].size);
        } else {
            printf("%d\t%d\t_\n", i + 1, files[i].size);
        }
    }
}

```

```

int main() {
    int n_blocks, n_files, choice;
    printf("Memory Management Scheme\n");
    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];
    printf("\nEnter the size of the blocks:\n");
    for (int i = 0; i < n_blocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i].size);
        blocks[i].allocated = 0;
    }
    printf("Enter the size of the files:\n");
    for (int i = 0; i < n_files; i++) {
        printf("File %d: ", i + 1);
        scanf("%d", &files[i].size);
    }
    do {
        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        resetBlocks(blocks, n_blocks); // Reset block allocation before each strategy
        switch (choice) {
            case 1:
                firstFit(blocks, n_blocks, files, n_files);
                break;
            case 2:
                bestFit(blocks, n_blocks, files, n_files);
                break;
            case 3:
                worstFit(blocks, n_blocks, files, n_files);
                break;
            case 4:
                printf("\nExiting...\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 4);
    return 0;
}

```

}

## Result:

```
Block 2: 500
Block 3: 455
Block 4: 600
Block 5: 750
Enter the number of files: 3
Enter the size of the file:
File 1: 200
File 2: 300
File 3: 656
1. Best Fit
2. Worst Fit
3. First Fit
4. Exiting
Enter your choice: 1

Memory Management Scheme - Best Fit
File_no :      File_size :      Block_no :      Block_size :
1          200           1            200
2          300           2            300
3          656           5            750

1. Best Fit
2. Worst Fit
3. First Fit
4. Exiting
Enter your choice: 2

Memory Management Scheme - Worst Fit
File_no :      File_size :      Block_no :      Block_size :
1          200           5            750
2          300           4            600
3          656           -            -

1. Best Fit
2. Worst Fit
3. First Fit
4. Exiting
Enter your choice: 3

Memory Management Scheme - First Fit
File_no :      File_size :      Block_no :      Block_size :
1          200           1            200
2          300           2            300
3          656           5            750

1. Best Fit
2. Worst Fit
3. First Fit
4. Exiting
Enter your choice: 4
```

Ulfat: Memory management Scheme  
Enter the number of blocks: 5  
Enter the size of the block:  
Block 1: 200  
Block 2: 300  
Block 3: 455  
Block 4: 600  
Block 5: 750  
1. best fit  
2. worst fit  
3. first fit  
4. exiting  
Memory management unit - Worst fit  
Blocks : File\_no: block\_no: Block\_size:  
1 200 5 750  
2 300 4 600  
3 656 - -  
1. best fit  
2. worst fit  
3. first fit  
4. exiting  
Memory management unit - First fit  
Blocks : File\_no: block\_no: Block\_size:  
1 200 1 200  
2 300 2 300  
3 656 5 750  
1. best fit  
2. worst fit  
3. first fit  
4. exiting  
Enter your choice: 1  
Exiting.....

Contiguous Memory Allocation.

```

#include <sections.h>
struct Block {
    int block_no;
    int b_size, n_free;
};

struct File {
    int file_no, file_size;
};

void insert_B (struct Block blocks[], int n) {
    for (int i=0; i<n; i++) {
        blocks[i].n_free = 1;
    }
}

void printF (struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf ("In memory management scheme - first fit\n");
    printf ("File no: %d File size: %d Block no: %d Block size: %d\n");
    for (int i=0; i<n_files; i++) {
        int allocated = 0;
        for (int j=0; j<n_blocks; j++) {
            if (blocks[j].n_free && blocks[j].block_size == files[i].file_size) {
                printf ("%d %d %d %d\n", files[i].file_no, files[i].file_size, blocks[j].block_no, blocks[j].block_size);
                allocated = 1;
                break;
            }
        }
        if (!allocated)
            printf ("No suitable block found for file %d\n", files[i].file_no);
    }
}

```

```

printf ("n d i t l a t l t l b w"; files[i].file_no,
files[i].file_size);
}

void best_fit (struct Block blocks[], int n_blocks, struct File
files[], int n_files) {
    printf ("In memory management scheme - Best fit\n");
    printf ("File no: %d File size: %d Block no: %d Block size: %d\n");
    for (int i=0; i<n_files; i++) {
        int best = -1;
        for (int j=0; j<n_blocks; j++) {
            if (blocks[j].n_free && blocks[j].block_size == files[i].file_size) {
                if (best == -1 || blocks[j].block_size < blocks[best].block_size)
                    best = j;
            }
        }
        if (best != -1)
            printf ("%d %d %d %d\n", files[i].file_no, files[i].file_size, blocks[best].block_no, blocks[best].block_size);
    }
}

void best_fit_worst_fit (struct Block blocks[], int n_blocks,
struct File files[], int n_files) {
    printf ("In memory management scheme - worst fit\n");
    printf ("File no: %d File size: %d Block no: %d Block size: %d\n");
    for (int i=0; i<n_files; i++) {
        int worst = -1;
        for (int j=0; j<n_blocks; j++) {
            if (blocks[j].n_free && blocks[j].block_size == files[i].file_size) {
                if (worst == -1 || blocks[j].block_size > blocks[worst].block_size)
                    worst = j;
            }
        }
    }
}
```

## Program 9

Write a C program to simulate page replacement algorithms

- a) FIFO
- b) LRU
- c) Optimal

```
#include<stdio.h>
int n,m;
int p[10000],f[10000];
void fifo(){
    for(int i=0;i<m;++i) f[i]=-1;
    int h=0;
    int s=0;
    for(int i=0;i<n;++i){
        int k=0;
        for(int r=0;r<m;++r)
            if(p[i]==f[r]) {++h;k=1;break;}
        if(k==1)continue;
        for(int r=0;r<m;++r)
            if(f[r]==-1){
                f[r]=p[i];k=2;break;}
        if(k==2) continue;
        f[s]=p[i];
    }
}
```

```

    s=(s+1)%m;
}
printf("FIFO :\t\t\t Page-Hit:%d\t\tPage-Fault:%d\n",h,n-h);
}
void lru(){
    for(int i=0;i<m;++i) f[i]=-1;
    int h=0;
    int t[m];
    for(int i=0;i<m;++i) t[i]=0;
    for(int i=0;i<n;++i){
        int k=0;
        for(int r=0;r<m;++r)
            if(f[r]!=-1) ++t[r];
        for(int r=0;r<m;++r)
            if(f[r]==p[i]) {t[r]=1;k=1;break;}
        if(k==1){++h;continue;}
        for(int r=0;r<m;++r)
            if(f[r]==-1){f[r]=p[i];t[r]=1;
                k=2;break;}
        if(k==2) continue;
        int x=0;
        for(int r=1;r<m;++r)
            if(t[r]>t[x]) x=r;
        t[x]=1;f[x]=p[i];
    }
    printf("LRU :\t\t\t Page-Hit:%d\t\tPage-Fault:%d\n",h,n-h);
}
void optimal(){
    for(int i=0;i<m;++i) f[i]=-1;
    int h=0;
    for(int i=0;i<n;++i){
        int k=0;
        for(int r=0;r<m;++r)
            if(f[r]==p[i]) {k=1;
                ++h;break;}
        if(k==1) continue;
        for(int r=0;r<m;++r)

```

```

if(f[r]==-1){
k=2;f[r]=p[i];break;}
if(k==2) continue;
int l=-1,xx=-1,y=1;
for(int j=0;j<m;++j){
    y=1;
    for(k=i+1;k<n;++k)
        if(f[j]==p[k]){
            y=0;
            if(k>l){xx=j;l=k;}break;
        }
    if(y){
        f[j]=p[i];break;}
}
if(y) continue;
f[xx]=p[i];
}
printf("OPTIMAL :\t\t Page-Hit:%d\t\tPage-Fault:%d\n",h,n-h);
}

int main(){
printf("Enter No. of Page String and Page-Frame : ");
scanf("%d%d",&n,&m);
printf("Enter the Page-String : ");
for(int i=0;i<n;++i)
    scanf("%d",&p[i]);
fifo();lru();optimal();
}

```

## RESULT:

```

Enter No. of Page String and Page-Frame : 7 3
Enter the Page-String : 1 3 0 3 5 6 3
FIFO : Page-Hit:1 Page-Fault:6
LRU : Page-Hit:2 Page-Fault:5
OPTIMAL : Page-Hit:2 Page-Fault:5

Process returned 0 (0x0) execution time : 15.202 s
Press any key to continue.
|
```

*FIFO-LRU-Optimal*

```

Data Page
H indicates address
int n, m;
int p[1000], f[1000];
void fifo();
for (int i=0; i<m; ++i) f[i] = -1;
int h=0, a=0;
for (int i=0; i<n; ++i) {
    int k=0;
    for (int r=0; r<m; ++r)
        if (p[i] == f[r]) k += h; l = i; break; }
    if (k == 1) cout << "hit"; else cout << "miss";
    if (f[r] == -1) cout << " ";
    f[r] = p[i]; h = 2; break; }
    if (k == 0) cout << " ";
    f[0] = p[i];
    a = f[0] * m;
}
cout << endl;
cout << "FIFO: " << l << " page-hit " << l << " page-fault: " << d << endl;
cout << endl;
void lru()
{
    for (int i=0; i<m; ++i) f[i] = -1;
    int t=0;
    int tSm;
    for (int i=0; i<m; ++i) tSm = i;
    for (int i=0; i<n; ++i) {
        int k=0;
        for (int r=0; r<m; ++r)
            if (f[r] == -1) k += 1;
        for (int r=0; r<m; ++r)
            if (f[r] == p[i]) k -= 1; f[r] = p[i];
        if (k == 1) cout << "hit"; else cout << "miss";
        f[tSm] = p[i];
        tSm = i;
    }
}
```

```

    if (l == 1) cout << "hit"; else cout << "miss";
    if (f[l] == p[i]) cout << " ";
    f[l] = p[i];
    cout << endl;
    cout << "LRU: " << l << " page-hit " << l << " page-fault: " << d << endl;
    cout << endl;
void optimal()
{
    for (int i=0; i<m; ++i) f[i] = -1;
    int h=0;
    for (int i=0; i<n; ++i) {
        int k=0;
        for (int r=0; r<m; ++r)
            if (f[r] == -1) k += 1;
        if (k == 1) cout << "hit"; else cout << "miss";
        if (f[r] == -1) cout << " ";
        f[r] = p[i];
        cout << endl;
    }
}
```

*4(y)*

```

Data Page
if (k>1) {xx=j+l=k; ? break;
}
if (y) {
    if (y) = p[i]; break;
}
if (y) continue;
if (xx=j+l=k) {
    if (y) = p[i];
}
cout << endl;
cout << "OPTIMAL: " << l << " page-hit " << l << " page-fault: " << d << endl;
cout << endl;
int main()
{
    cout << "Enter no. of page string and page-frame: ";
    Scanf("%d %d", &n, &m);
    cout << "Enter the page-string: ";
    for (int i=0; i<n; ++i)
        Scanf("%d", &p[i]);
    if (l == 1) cout << "hit"; else cout << "miss";
    if (f[l] == p[i]) cout << " ";
    f[l] = p[i];
    cout << endl;
    cout << "FIFO: " << l << " page-hit " << l << " page-fault: " << d << endl;
    cout << endl;
    cout << "LRU: " << l << " page-hit " << l << " page-fault: " << d << endl;
    cout << endl;
    cout << "OPTIMAL: " << l << " page-hit " << l << " page-fault: " << d << endl;
    cout << endl;
}
```

*off*

```

Enter no. of page string and page-frame: 7 3
Enter the page-string: 1 3 0 3 5 6 3
FIFO: page-hit: 1 page-fault: 6
LRU: page-hit: 2 page-fault: 5
OPTIMAL: page-hit: 2 page-fault: 5

    7 8 1
    7 7 7
    0 0
    1

```

*John*