VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VEDIKA S S (1WA23CS037)

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 Vedika S S (1WA23CS037), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Dr Seema Patil
Assistant Professor
Department of CSE
BMSCE, Bengaluru

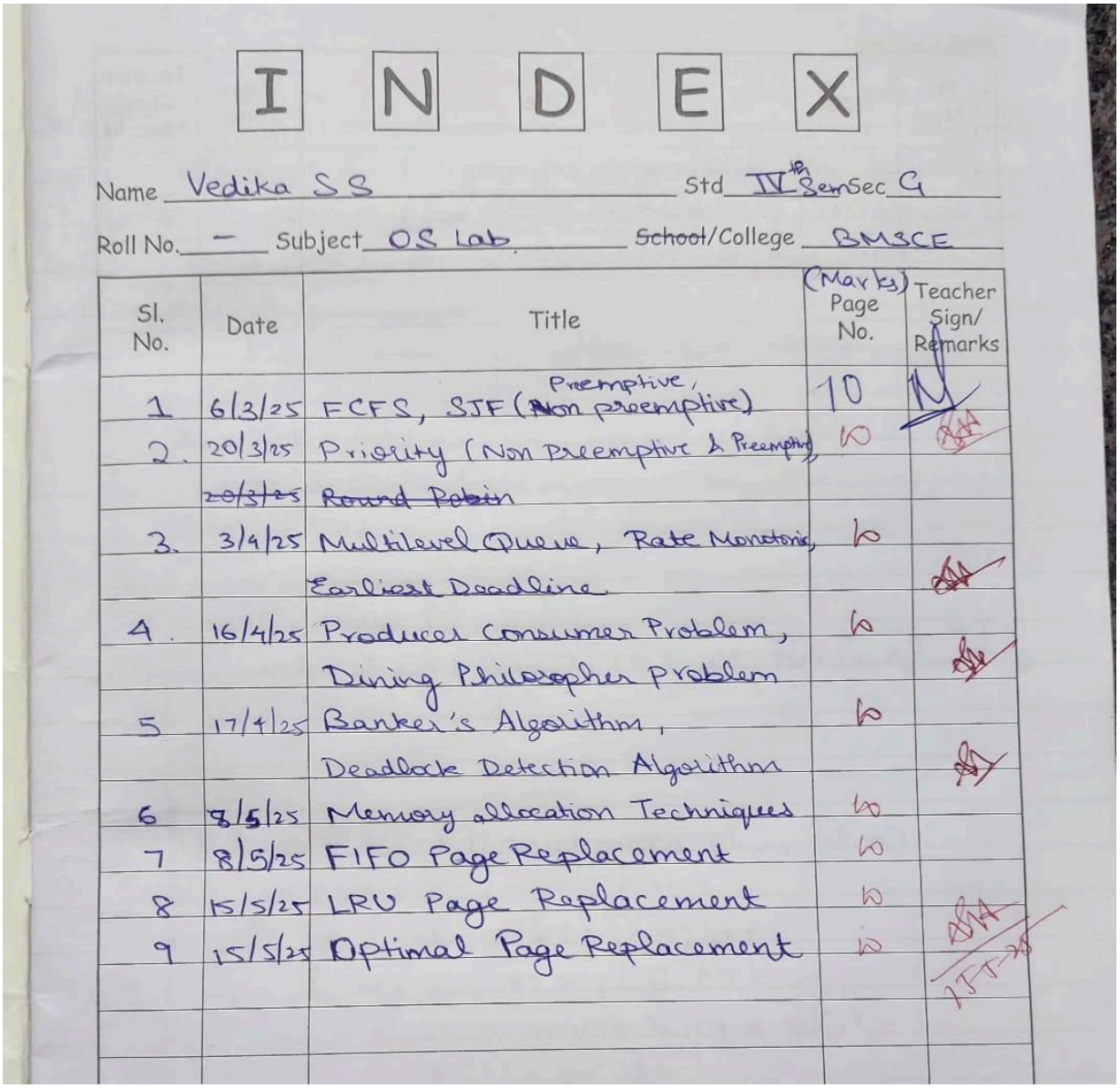
Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1.	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)	1-6
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)	7-11
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 RR and FCFS scheduling for the processes in each queue. Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First	12-24
4.	Write a C program to simulate producer-consumer problem using semaphores Write a C program to simulate the concept of Dining Philosophers problem.	25-33
5.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. Write C program to stimulate deadlock detection	34-40
6.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit Write a C program to simulate page replacement algorithms a) FIFO	41-48
7.	Write a C program to simulate page replacement algorithms b) LRU c) Optimal	49-55

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.



INDEX

Name Vedika S S Std IV Sem Sec G

Roll No. – Subject OS Lab School/College BMSCE

Sl. No.	Date	Title	Page No.	(Marks) Teacher Sign/ Remarks
1	6/3/25	FCFS, SJF (~~Non~~ Preemptive, preemptive)	10	
2.	20/3/25	Priority (Non Preemptive & Preemptive)	10	
	~~20/3/25~~	~~Round Robin~~		
3.	3/4/25	Multilevel Queue, Rate Monotonic, Earliest Deadline	10	
4.	16/4/25	Producer Consumer Problem, Dining Philosopher Problem	10	
5	17/4/25	Banker's Algorithm, Deadlock Detection Algorithm	10	
6	8/5/25	Memory allocation Techniques	10	
7	8/5/25	FIFO Page Replacement	10	
8	15/5/25	LRU Page Replacement	10	
9	15/5/25	Optimal Page Replacement	10	

Program -1

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct process {
   int id, AT, BT, CT, TAT, WT, RT, remaining_BT;
   int completed;
};
void sort_by_AT(struct process p[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i].AT > p[j].AT) {
            struct process temp = p[i];
            p[i] = p[j];
            p[j] = temp;
         }
      }
   }
}
void calculate_FCFS(struct process p[], int n) {
   sort_by_AT(p, n);
   int currentTime = 0;

   for (int i = 0; i < n; i++) {
      if (currentTime < p[i].AT)
         currentTime = p[i].AT;
```

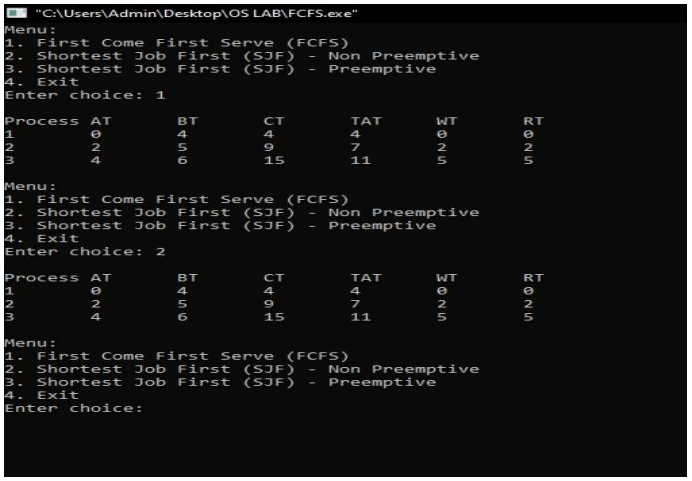
```
        p[i].RT = currentTime - p[i].AT;
        p[i].CT = currentTime + p[i].BT;
        currentTime = p[i].CT;
        p[i].TAT = p[i].CT - p[i].AT;
        p[i].WT = p[i].TAT - p[i].BT;
    }
}
void calculate_SJF_NonPreemptive(struct process p[], int n) {
    int completed = 0, currentTime = 0;
    while (completed < n) {
        int shortest = -1, minBT = 10000;
        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].RT = currentTime - p[shortest].AT;
            p[shortest].CT = currentTime + p[shortest].BT;
            currentTime = p[shortest].CT;
            p[shortest].TAT = p[shortest].CT - p[shortest].AT;
            p[shortest].WT = p[shortest].TAT - p[shortest].BT;
            p[shortest].completed = 1;
            completed++;
        }
    }
}
void calculate_SJF_Preemptive(struct process p[], int n) {
```

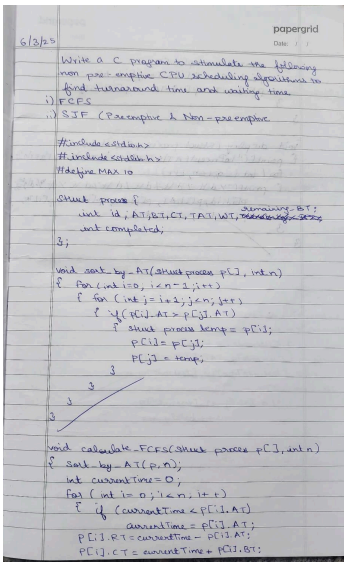
```
    int completed = 0, currentTime = 0;
    for (int i = 0; i < n; i++) {
        p[i].remaining_BT = p[i].BT;
    }
    while (completed < n) {
        int shortest = -1, minBT = 10000;
        for (int i = 0; i < n; i++) {
            if (!p[i].completed && p[i].AT <= currentTime && p[i].remaining_BT < 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;
                completed++;
            }
        }
    }
}
```

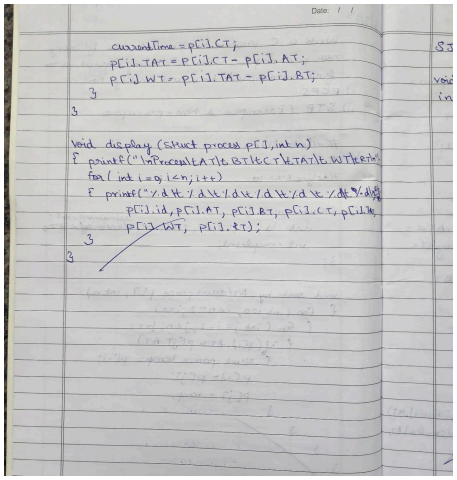
```
void display(struct process p[], int n) {
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
            printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].AT, p[i].BT, p[i].CT, p[i].TAT,
p[i].WT, p[i].RT);
    }
}

int main() {
    int n, choice;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct process p[n];
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("Enter Arrival Time (AT) for process %d: ", i + 1);
        scanf("%d", &p[i].AT);
        printf("Enter Burst Time (BT) for process %d: ", i + 1);
        scanf("%d", &p[i].BT);
        p[i].completed = 0;
    }

    while (1) {
        printf("\nMenu:\n");
        printf("1. First Come First Serve (FCFS)\n");
        printf("2. Shortest Job First (SJF) - Non Preemptive\n");
        printf("3. Shortest Job First (SJF) - Preemptive\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
```

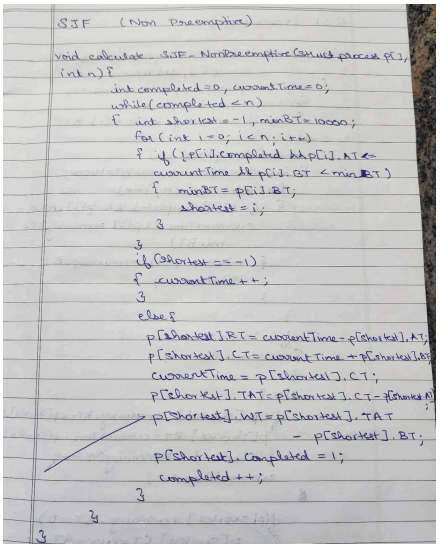
```
        switch (choice) {
            case 1:
                calculate_FCFS(p, n);
                display(p, n);
                break;
            case 2:
                calculate_SJF_NonPreemptive(p, n);
                display(p, n);
                break;
            case 3:
                calculate_SJF_Preemptive(p, n);
                display(p, n);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice. Try again.\n");
        }
    }

    return 0;
}
```

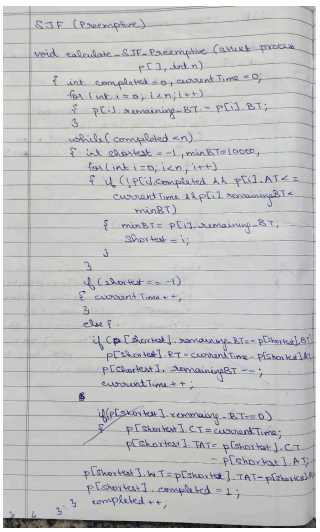
OUTPUT:

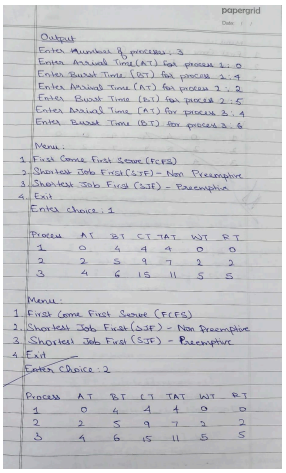

"C:\Users\Admin\Desktop\OS LAB\FCFS.exe"
Menu:
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF) - Non Preemptive
3. Shortest Job First (SJF) - Preemptive
4. Exit
Enter choice: 1
Process AT BT CT TAT WT RT
1 0 4 4 4 0 0
2 2 5 9 7 2 2
3 4 6 15 11 5 5
Menu:
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF) - Non Preemptive
3. Shortest Job First (SJF) - Preemptive
4. Exit
Enter choice: 2
Process AT BT CT TAT WT RT
1 0 4 4 4 0 0
2 2 5 9 7 2 2
3 4 6 15 11 5 5
Menu:
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF) - Non Preemptive
3. Shortest Job First (SJF) - Preemptive
4. Exit
Enter choice:


papergrid
Write a C program to simulate the following non pre-emptive CPU scheduling algorithms to find turnaround time and waiting time
i) FCFS
ii) SJF (Preemptive & Non-preemptive




SJF (Non Preemptive)


SJF (Preemptive)



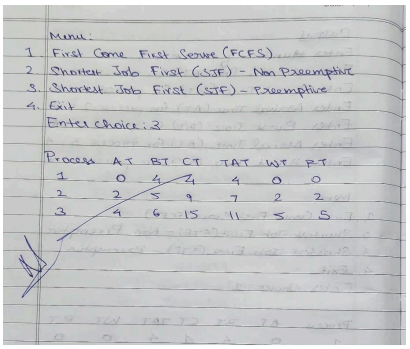
Output
Enter Number of processes : 3
Enter Arrival Time (AT) for process 1 : 0
Enter Burst Time (BT) for process 1 : 4
Enter Arrival Time (AT) for process 2 : 2
Enter Burst Time (BT) for process 2 : 5
Enter Arrival Time (AT) for process 3 : 4
Enter Burst Time (BT) for process 3 : 6

Menu :
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF) - Non Preemptive
3. Shortest Job First (SJF) - Preemptive
4. Exit
Enter choice : 1

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

Menu :
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF) - Non Preemptive
3. Shortest Job First (SJF) - Preemptive
4. Exit
Enter choice : 2

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



Menu :
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF) - Non Preemptive
3. Shortest Job First (SJF) - Preemptive
4. Exit
Enter choice : 3

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

Program -2

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

→ **Priority (pre-emptive & Non-pre-emptive)**

Code:

```
#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 highest_priority = -1, selected = -1;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && !p[i].is_completed && p[i].pt > highest_priority) {
                highest_priority = p[i].pt;
                selected = i;
```

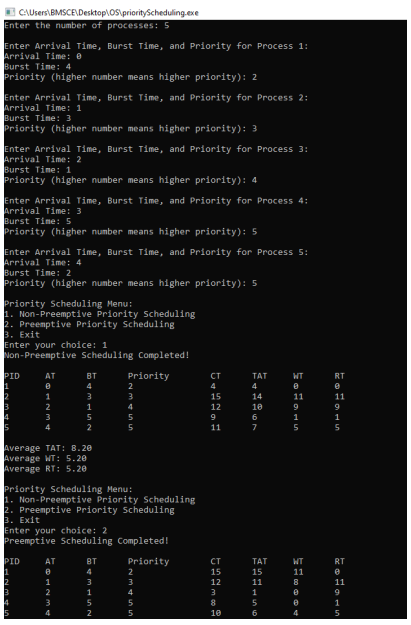
```
            }
        }
        if (selected == -1) {
            time++;
            continue;
        }
        if (p[selected].rt == -1) {
            p[selected].st = time;  // Start time
            p[selected].rt = time - p[selected].at;  // Response Time = Start Time - Arrival Time }
        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 highest_priority = -1, selected = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining_bt > 0 && p[i].pt > highest_priority) {
                highest_priority = p[i].pt;
                selected = i;
            }
        }
        if (selected == -1) {
            time++;
            continue;
```

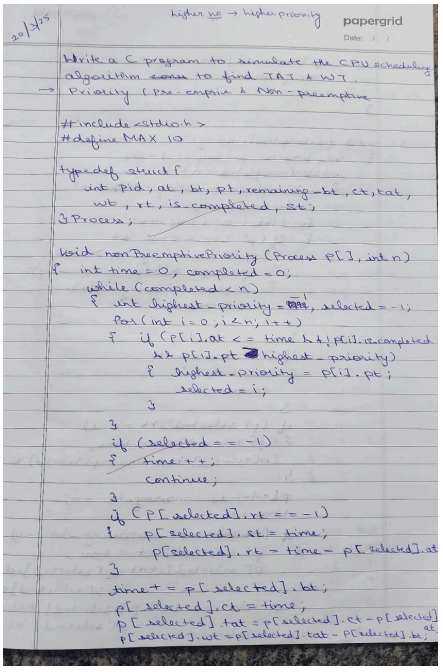
```
        }
        if (p[selected].rt == -1) {
            p[selected].st = time;  // Start time
            p[selected].rt = time - p[selected].at;  // Response Time = Start Time - Arrival Time
        }
        p[selected].remaining_bt--;
        time++;
        if (p[selected].remaining_bt == 0) {
            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\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() {
```

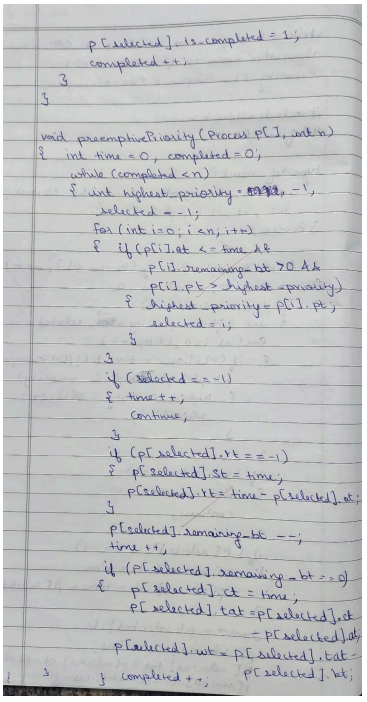
```
Process p[MAX];
int n, choice;
printf("Enter the number of processes: ");
scanf("%d", &n);
for (int i = 0; i < n; i++) {
    p[i].pid = i + 1;
    printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid);
    printf("Arrival Time: ");
    scanf("%d", &p[i].at);
    printf("Burst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

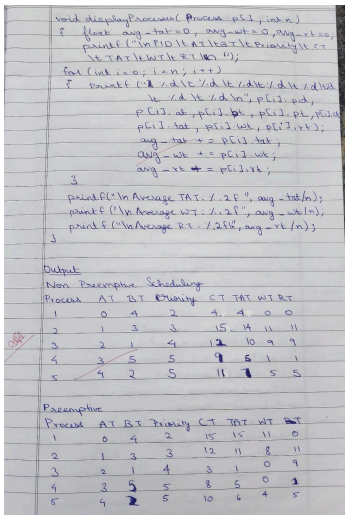
```
            preemptivePriority(p, n);
            printf("Preemptive Scheduling Completed!\n");
            displayProcesses(p, n);
            break;
        case 3:
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice! Try again.\n");
    }
  }

  return 0;
}
```

OUTPUT:









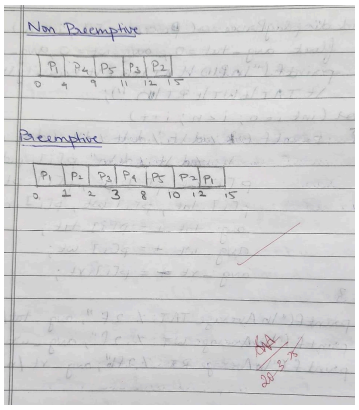
Output

Non Preemptive Scheduling

Process	AT	BT	Priority	CT	TAT	WT	RT
1	0	4	2	4	4	0	0
2	1	3	3	15	14	11	11
3	2	1	4	12	10	9	9
4	3	5	5	9	6	1	1
5	4	2	5	11	7	5	5

Preemptive

Process	AT	BT	Priority	CT	TAT	WT	RT
1	0	4	2	15	15	11	0
2	1	3	3	12	11	8	11
3	2	1	4	3	1	0	9
4	3	5	5	8	5	0	1
5	4	2	5	10	6	4	5



Program -3

Question: 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 RR and FCFS scheduling for the processes in each queue.

Code:

```
#include <stdio.h>

#define MAX_PROCESSES 10

#define TIME_QUANTUM 2


typedef struct {

    int burst_time, arrival_time, queue_type, waiting_time, turnaround_time, response_time, remaining_time;

} Process;


void round_robin(Process processes[], int n, int time_quantum, int *time) {

  int done, i;

  do {

     done = 1;
```

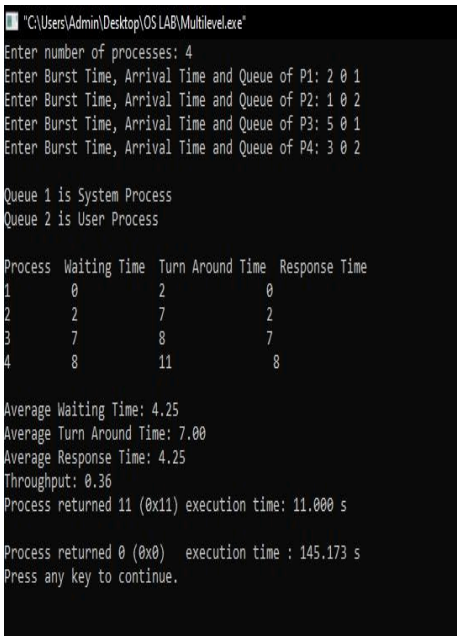
```
        for (i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time > time_quantum) {
                    *time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
           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

void fcfs(Process processes[], int n, int *time) {
    for (int i = 0; i < n; i++) {
        if (*time < processes[i].arrival_time) {
            *time = processes[i].arrival_time;
        }
        processes[i].waiting_time = *time - processes[i].arrival_time;
        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
        processes[i].response_time = processes[i].waiting_time;
        *time += processes[i].burst_time;
```

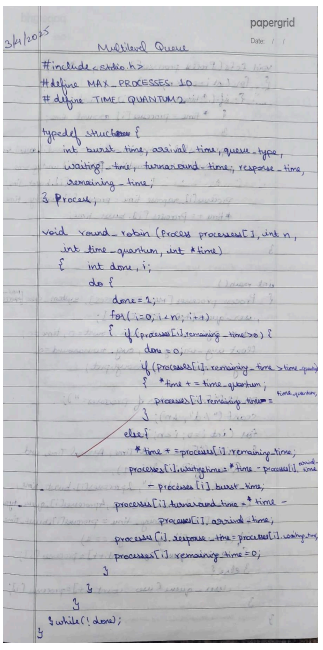
```
    }
}

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

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

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

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

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

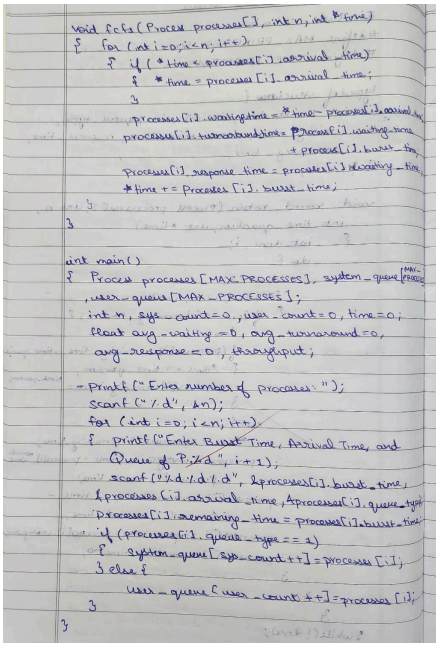
```
            Process temp = user_queue[j];
            user_queue[j] = user_queue[j + 1];
            user_queue[j + 1] = temp;
        }
    }
}

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

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

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

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

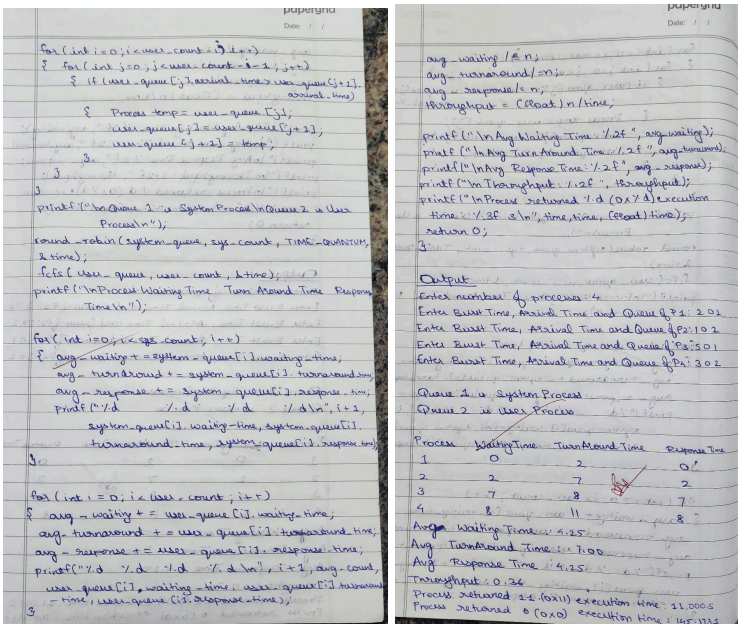
```
    avg_waiting /= n;
    avg_turnaround /= n;
    avg_response /= n;
    throughput = (float)n / time;

    printf("\nAverage Waiting Time: %.2f", avg_waiting);
    printf("\nAverage Turn Around Time: %.2f", avg_turnaround);
    printf("\nAverage Response Time: %.2f", avg_response);
    printf("\nThroughput: %.2f", throughput);
    printf("\nProcess returned %d (0x%d) execution time: %.3f s\n", time, time, (float)time);

    return 0;
}
```

OUTPUT:









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

a) Rate- Monotonic

b) Earliest-deadline First

Code:

a)Rate Monotonic

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#define MAX_PROCESS 10

int num_of_process;
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
    remain_time[MAX_PROCESS], deadline[MAX_PROCESS],
    remain_deadline[MAX_PROCESS];

void get_process_info() {
```

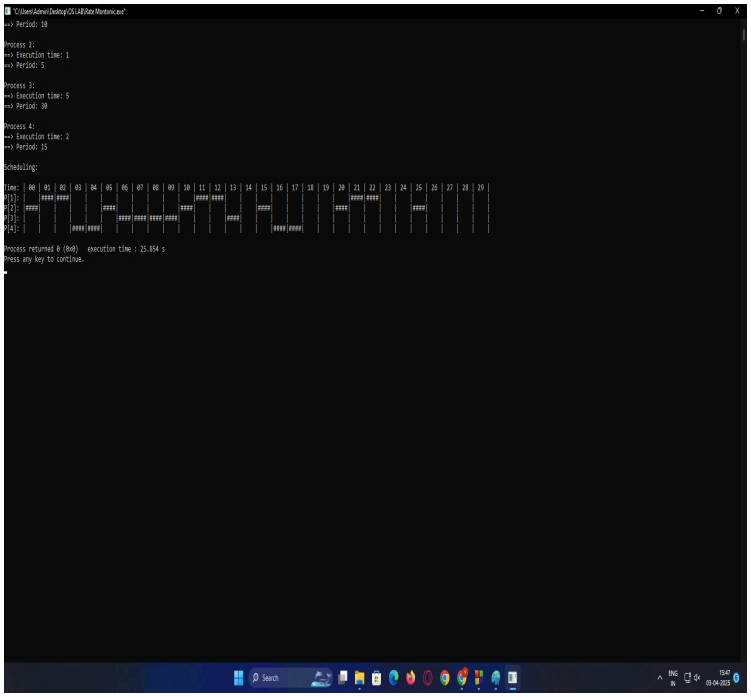
```
    printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
    scanf("%d", &num_of_process);

    if (num_of_process < 1) {
        exit(0);
    }

    for (int i = 0; i < num_of_process; i++) {
        printf("\nProcess %d:\n", i + 1);
        printf("==> Execution time: ");
        scanf("%d", &execution_time[i]);
        remain_time[i] = execution_time[i];
        printf("==> Period: ");
        scanf("%d", &period[i]);
    }
}

int max(int a, int b, int c) {
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else if (c >= a && c >= b)
        max = c;
    return max;
}

void print_schedule(int process_list[], int cycles) {
    printf("\nScheduling:\n\n");
    printf("Time: ");
```

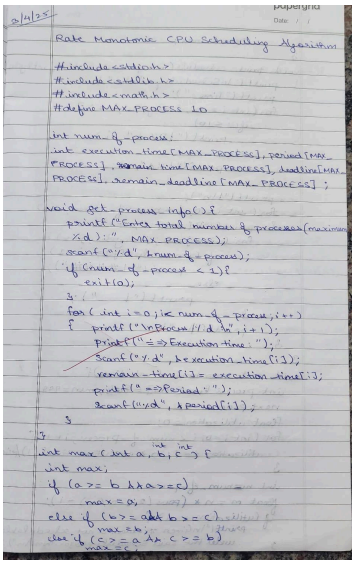
```
    for (int i = 0; i < cycles; i++) {
        if (i < 10)
            printf("| 0%d ", i);
        else
            printf("| %d ", i);
    }
    printf("|\n");

    for (int i = 0; i < num_of_process; i++) {
        printf("P[%d]: ", i + 1);
        for (int j = 0; j < cycles; j++) {
            if (process_list[j] == i + 1)
                printf("|####");
            else
                printf("|    ");
        }
        printf("|\n");
    }
}

void rate_monotonic(int time) {
    int process_list[100] = {0}, min = 999, next_process = 0;
    float utilization = 0;

    for (int i = 0; i < num_of_process; i++) {
        utilization += (1.0 * execution_time[i]) / period[i];
    }

    int n = num_of_process;
    float m = n * (pow(2, 1.0 / n) - 1);
```

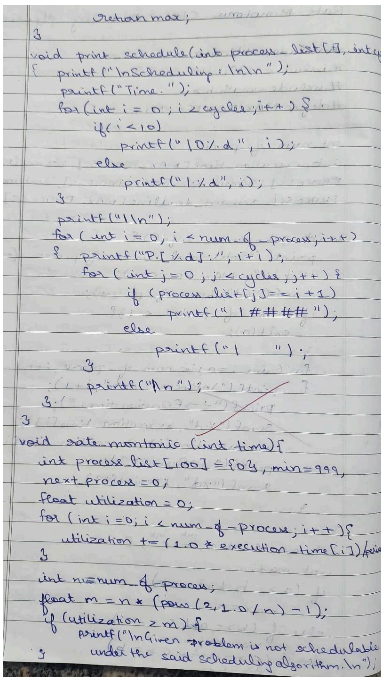
```
    if (utilization > m) {
        printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");
    }

    for (int i = 0; i < time; i++) {
        min = 1000;
        for (int j = 0; j < num_of_process; j++) {
            if (remain_time[j] > 0) {
                if (min > period[j]) {
                    min = period[j];
                    next_process = j;
                }
            }
        }

        if (remain_time[next_process] > 0) {
            process_list[i] = next_process + 1;
            remain_time[next_process] -= 1;
        }

        for (int k = 0; k < num_of_process; k++) {
            if ((i + 1) % period[k] == 0) {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }
    print_schedule(process_list, time);
}

int main() {
```

```
    int observation_time;
    get_process_info();
    observation_time = max(period[0], period[1], period[2]);
    rate_monotonic(observation_time);
    return 0;
}
```

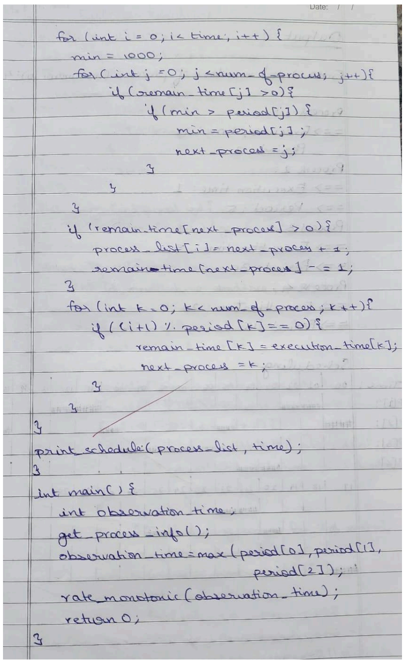
OUTPUT:



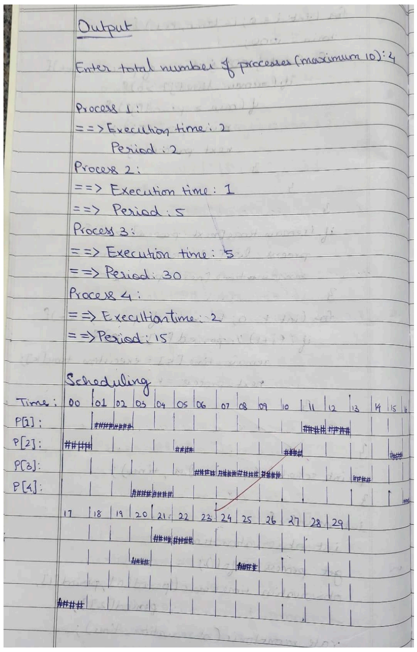




```
return max;
}
void print_schedule(int process_list[], int cycles)
{ printf("\nScheduling:\n\n");
  printf("Time: ");
  for (int i = 0; i < cycles; i++) {
    if(i < 10)
      printf("| 0%d ", i);
    else
      printf("| %d", i);
  }
  printf("|\n");
  for (int i = 0; i < num_of_process; i++)
  { printf("P[%d]: ", i+1);
    for (int j = 0; j < cycles; j++) {
      if (process_list[j] == i+1)
        printf("|####");
      else
        printf("|    ");
    }
    printf("|\n");
  }
}
void rate_monotonic(int time){
  int process_list[100] = {0}, min=999,
  next_process = 0;
  float utilization = 0;
  for (int i=0; i < num_of_process; i++){
    utilization += (1.0 * execution_time[i])/period[i];
  }
  int n = num_of_process;
  float m = n * (pow(2, 1.0/n) - 1);
  if (utilization > m) {
    printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");
  }
```



```
for (int i = 0; i < time; i++) {
  min = 1000;
  for (int j = 0; j < num_of_process; j++){
    if (remain_time[j] > 0){
      if (min > period[j]) {
        min = period[j];
        next_process = j;
      }
    }
  }
  if (remain_time[next_process] > 0){
    process_list[i] = next_process + 1;
    remain_time[next_process] -= 1;
  }
  for (int k = 0; k < num_of_process; k++){
    if ((i+1) % period[k] == 0) {
      remain_time[k] = execution_time[k];
      next_process = k;
    }
  }
}
print_schedule(process_list, time);
}
int main() {
  int observation_time;
  get_process_info();
  observation_time = max(period[0], period[1],
                         period[2]);
  rate_monotonic(observation_time);
  return 0;
}
```



b)Earliest Deadline

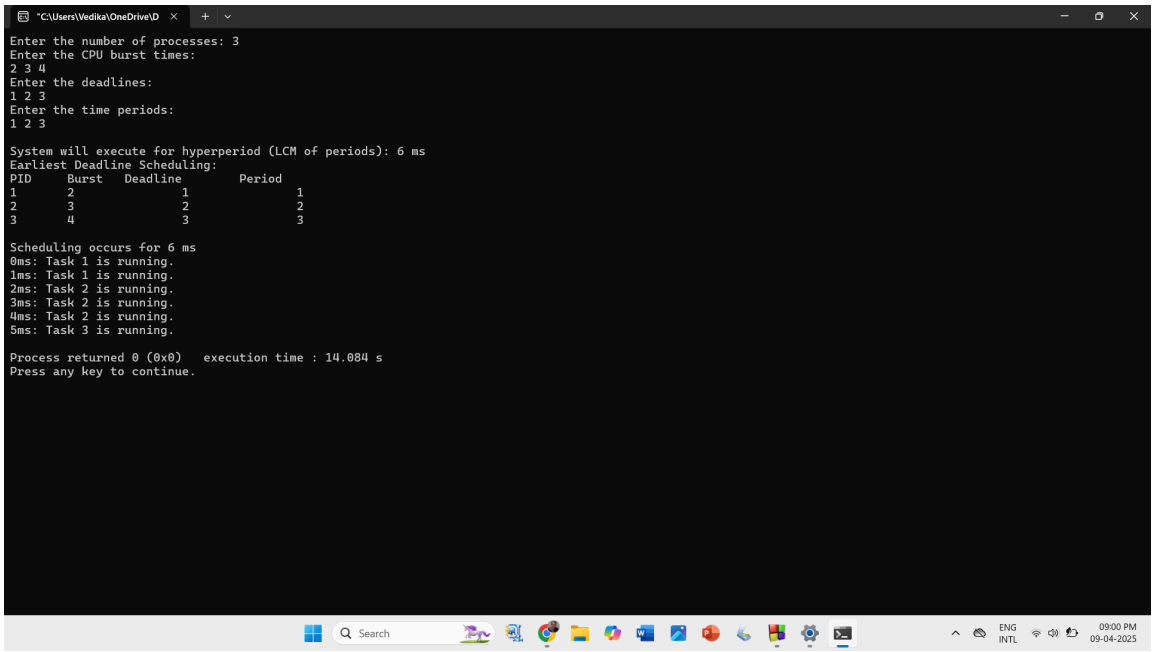
```
#include <stdio.h>
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

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

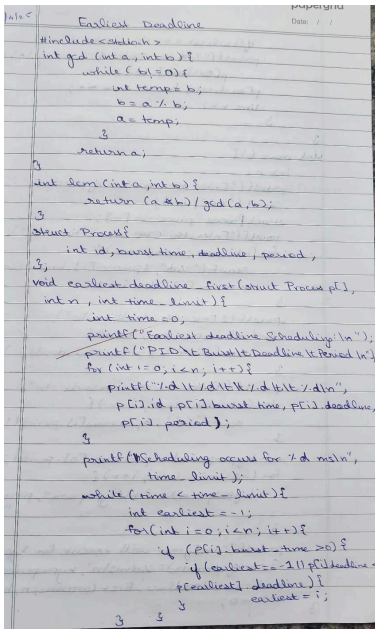
```
struct Process {
    int id, burst_time, deadline, period;
};

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++;
    }
}
```

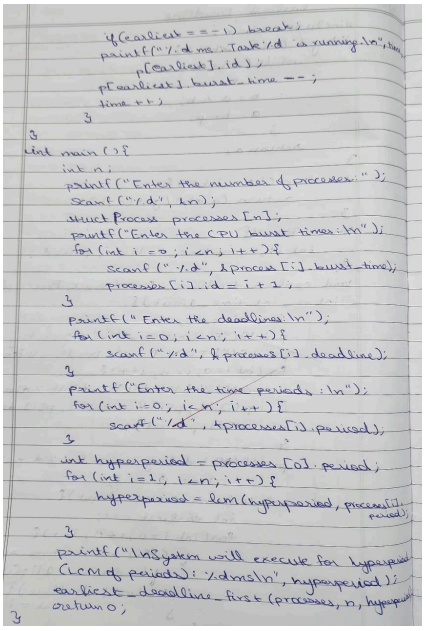
```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    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);
```

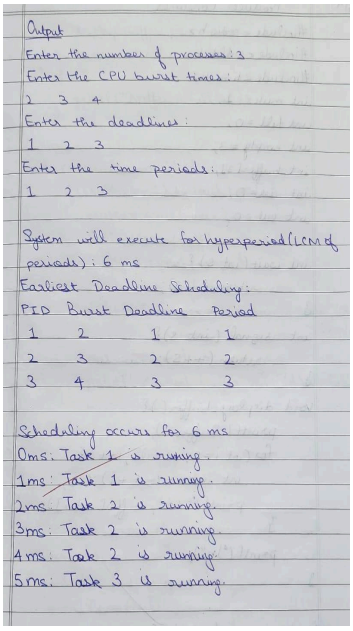
```
    return 0;
}
```

OUTPUT:


"C:\Users\Vedika\OneDrive\D
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.
Process returned 0 (0x0) execution time : 14.084 s
Press any key to continue.
Search
ENG
INTL
09:00 PM
09-04-2025


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\t Burst\t Deadline\t Period\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("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].deadline<
p[earliest].deadline){
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 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): %dms\n",hyperperiod);
earliest_deadline_first(processes,n,hyperperiod);
return 0;
}


Output
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.

Program -4

Question: Write a C program to simulate **producer-consumer problem using semaphores**

Code:

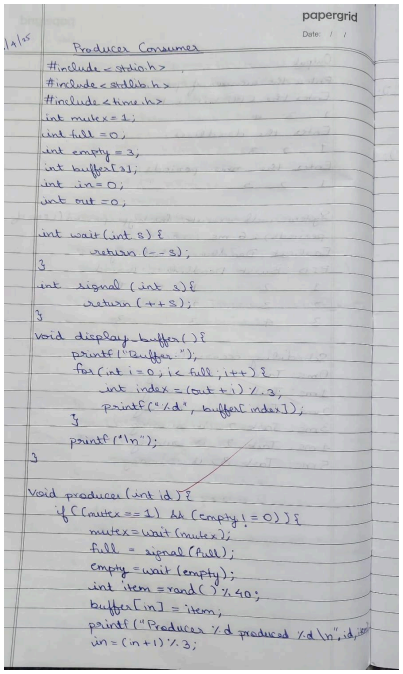
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int mutex = 1;
int full = 0;
int empty = 3;
int buffer[3];
int in = 0;
int out = 0;

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

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

void display_buffer() {
   printf("Buffer:");
   for (int i = 0; i < full; i++) {
      int index = (out + i) % 3;
      printf("%d ", buffer[index]);
   }
   printf("\n");
}
void producer(int id) {
```

```
    if ((mutex == 1) && (empty != 0)) {
        mutex = wait(mutex);
        full = signal(full);
        empty = wait(empty);

        int item = rand() % 40;
        buffer[in] = item;
        printf("Producer %d produced %d\n", id, item);
        in = (in + 1) % 3;

        display_buffer();
        mutex = signal(mutex);
    } else {
        printf("Buffer is full\n");
    }
}

void consumer(int id) {
    if ((mutex == 1) && (full != 0)) {
        mutex = wait(mutex);
        full = wait(full);
        empty = signal(empty);

        int item = buffer[out];
        printf("Consumer %d consumed %d\n", id, item);
        out = (out + 1) % 3;

        printf("Current buffer len: %d\n", full);
        mutex = signal(mutex);
    } else {
        printf("Buffer is empty\n");
```

```
    }
}

int main() {
    int p, c, capacity, choice;

    srand(time(NULL));

    printf("Enter the number of Producers:");
    scanf("%d", &p);
    printf("Enter the number of Consumers:");
    scanf("%d", &c);
    printf("Enter buffer capacity:");
    scanf("%d", &capacity);

    empty = capacity;

    for (int i = 1; i <= p; i++) {
        printf("Successfully created producer %d\n", i);
    }
    for (int i = 1; i <= c; i++) {
        printf("Successfully created consumer %d\n", i);
    }

    while (1) {
        printf("\n1. Produce\n2. Consume\n3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            producer(1);
```

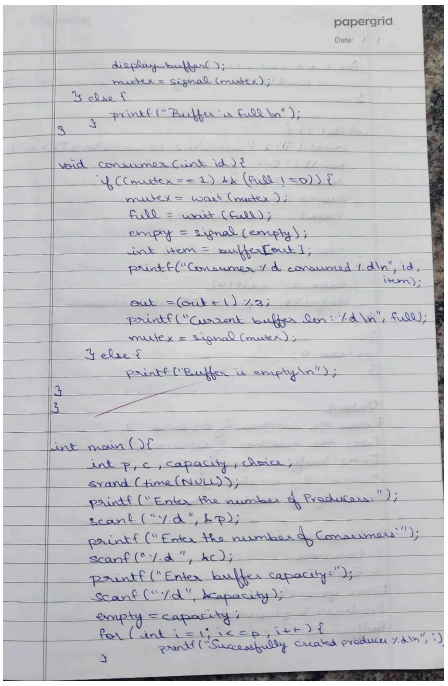
```
            break;
        case 2:
            consumer(2);
            break;
        case 3:
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

OUTPUT:

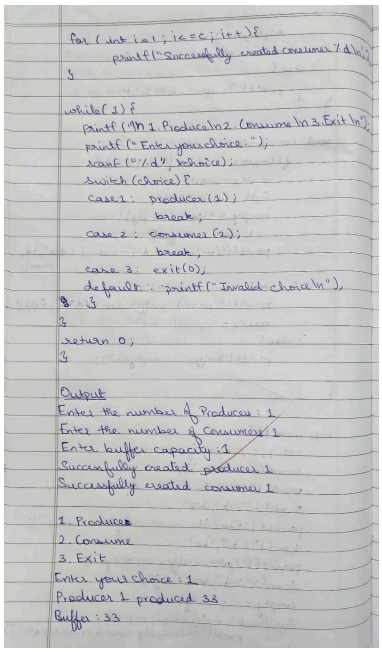


Producer Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int mutex = 1;
int full = 0;
int empty = 3;
int buffer[3];
int in = 0;
int out = 0;

int wait(int s) {
    return (--s);
}
int signal(int s) {
    return (++s);
}
void display_buffer() {
    printf("Buffer: ");
    for(int i = 0; i < full; i++) {
        int index = (out + i) % 3;
        printf("%d", buffer[index]);
    }
    printf("\n");
}

Void producer(int id) {
    if((mutex == 1) && (empty != 0)) {
        mutex = wait(mutex);
        full = signal(full);
        empty = wait(empty);
        int item = rand() % 40;
        buffer[in] = item;
        printf("Producer %d produced %d\n", id,
        in = (in + 1) % 3;
```



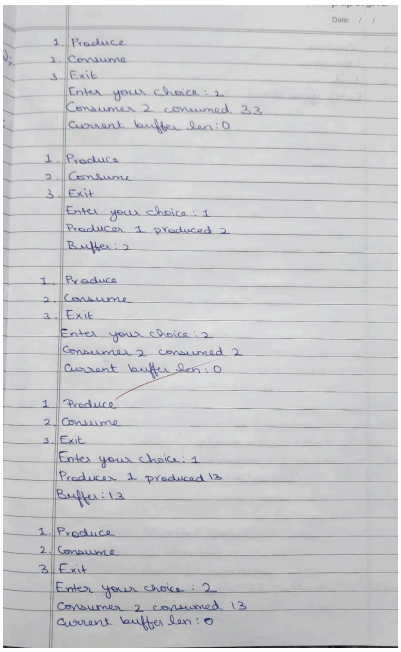
```
        display_buffer();
        mutex = signal(mutex);
    } else {
        printf("Buffer is full\n");
    }
}

void consumer(int id) {
    if((mutex == 1) && (full != 0)) {
        mutex = wait(mutex);
        full = wait(full);
        empty = signal(empty);
        int item = buffer[out];
        printf("Consumer %d consumed %d\n", id,
                                        item);
        out = (out + 1) % 3;
        printf("Current buffer len: %d\n", full);
        mutex = signal(mutex);
    } else {
        printf("Buffer is empty\n");
    }
}

int main() {
    int p, c, capacity, choice;
    srand(time(NULL));
    printf("Enter the number of Producers:");
    scanf("%d", &p);
    printf("Enter the number of Consumers:");
    scanf("%d", &c);
    printf("Enter buffer capacity:");
    scanf("%d", &capacity);
    empty = capacity;
    for(int i = 1; i <= p; i++) {
        printf("Successfully created producer %d\n", i);
    }
```



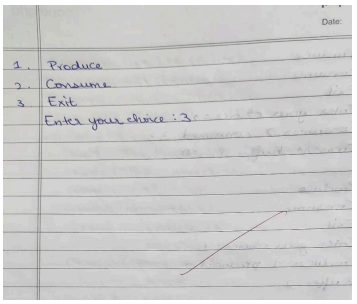
```
    for(int i = 1; i <= c; i++) {
        printf("Successfully created consumer %d\n", i);
    }

    while(1) {
        printf("\n1.Produce\n2.Consume\n3.Exit\n");
        printf("Enter your choice:");
        scanf("%d", &choice);
        switch(choice) {
            case 1: producer(1);
                    break;
            case 2: consumer(2);
                    break;
            case 3: exit(0);
            default: printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Output

```
Enter the number of Producers: 1
Enter the number of Consumers: 1
Enter buffer capacity: 1
Successfully created producer 1
Successfully created consumer 1

1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer 1 produced 33
Buffer: 33
```



```
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer 2 consumed 33
Current buffer len: 0

1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer 1 produced 2
Buffer: 2

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer 2 consumed 2
Current buffer len: 0

1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer 1 produced 13
Buffer: 13

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer 2 consumed 13
Current buffer len: 0
```



```
1. Produce
2. Consume
3. Exit
Enter your choice: 3
```

Question: Write a C program to simulate the concept of **Dining Philosophers** problem.

Code:

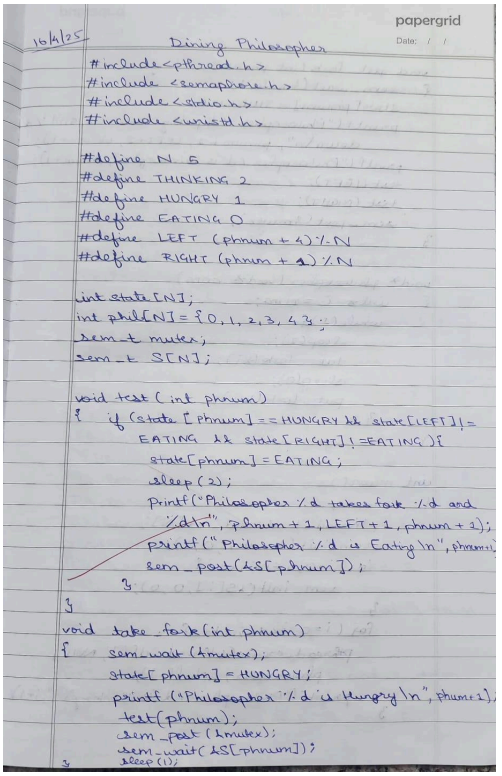
```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define N 5  // Number of philosophers
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];
// Function to test if a philosopher can start eating
void test(int phnum)
{
   if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
{
      state[phnum] = EATING;
      sleep(2); // Simulate time spent eating
      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]);  // Signal that the philosopher can start eating
   }
}
void take_fork(int phnum)
{
```

```
    sem_wait(&mutex);
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);

    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = THINKING;

     printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT + 1, phnum +
1);
    printf("Philosopher %d is thinking\n", phnum + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);  }

void* philosopher(void* num)
{
    int* i = num;
    while (1) {
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
```

```
}

int main()
{
    int i;
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++) {
        sem_init(&S[i], 0, 0);
    }
    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
    for (i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }
    return 0;
}
```

OUTPUT:

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



```
16/4/25                    Dining Philosopher

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

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

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

void test (int phnum)
{   if (state[phnum] == HUNGRY && state[LEFT] !=
        EATING && state[RIGHT] !=EATING ){
        state[phnum] = EATING;
        sleep (2);
        Printf ("Philosopher %d takes fork %d and
          %d\n", phnum + 1, LEFT + 1, phnum + 1);
        printf (" Philosopher %d is Eating\n", phnum+1);
        sem_post(&S[phnum]);
    }
}
void take_fork(int phnum)
{   sem_wait (&mutex);
    state[phnum] = HUNGRY;
    printf ("Philosopher %d is Hungry\n", phnum+1);
    test(phnum);
    sem_post (&mutex);
    sem_wait(&S[phnum]);
    sleep (1);
}
```

```
void put_fork (int phnum)
{   sem_wait(&mutex);
    state[phnum] = THINKING;
    printf ("Philosopher %d putting fork %d and %d
        down\n", phnum + 1, LEFT + 1, phnum + 1);
    printf ("Philosopher %d is thinking\n", phnum + 1);
    test (LEFT);
    test (RIGHT);
    sem_post(&mutex);
}

void* philosopher (void* num)
{   int* i = num;
    while(1){
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

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

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

Output: (infinite loop)
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 is putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 5 is Hungry
Philosopher 4 is putting fork 3 and 4 down
Philosopher 4 is thinking
^[Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
^[Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
^[Philosopher 5 takes fork 4 and 5
```

Continues...

Program -5

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

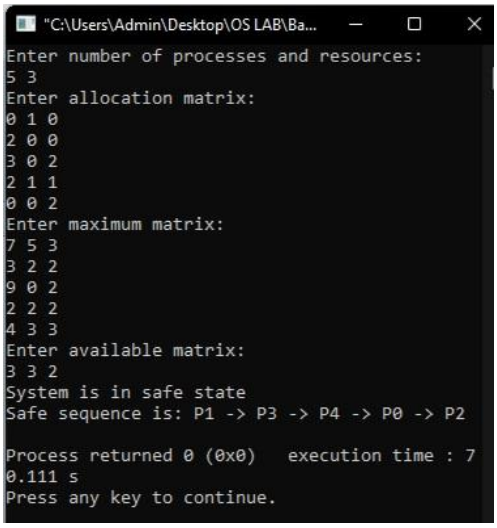
Code:

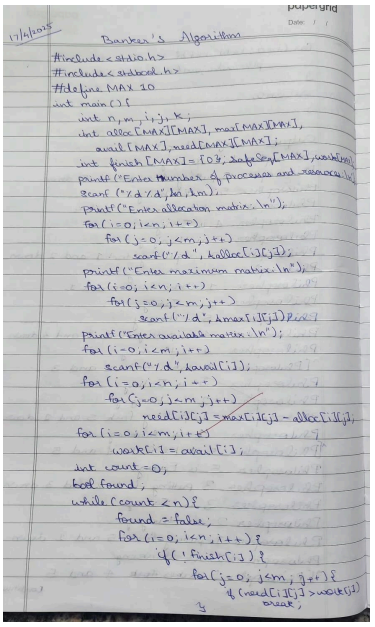
```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10

int main() {
    int n, m, i, j, k;
    int alloc[MAX][MAX], max[MAX][MAX], avail[MAX], need[MAX][MAX];
    int finish[MAX] = {0}, safeSeq[MAX], work[MAX];
    printf("Enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);
    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter maximum matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &max[i][j]);
    printf("Enter available matrix:\n");
    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];
    for (i = 0; i < m; i++)
        work[i] = avail[i];
```

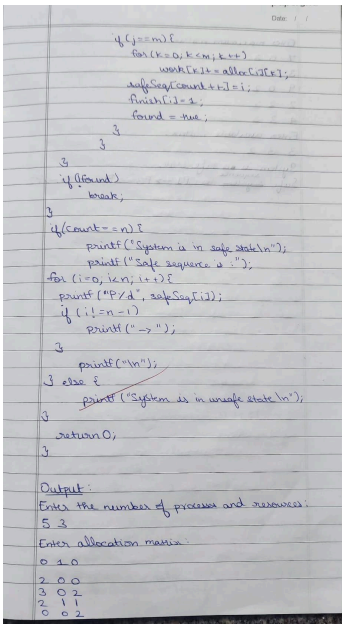
```
int count = 0;
bool found;
while (count < n) {
    found = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            for (j = 0; j < m; j++) {
                if (need[i][j] > work[j])
                    break;
            }
            if (j == m) {
                for (k = 0; k < m; k++)
                    work[k] += alloc[i][k];
                safeSeq[count++] = i;
                finish[i] = 1;
                found = true;
            }
        }
    }
    if (!found)
        break;
}
if (count == n) {
    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(" -> ");
 }
```

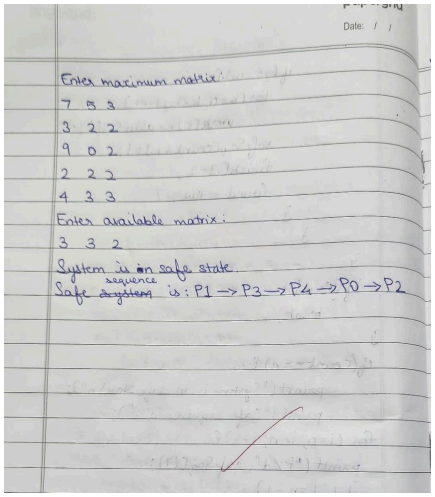
```
        printf("\n");
    } else {
        printf("System is in unsafe state\n");
    }

    return 0;
}
```

OUTPUT:









Question: Write a C program to simulate deadlock detection

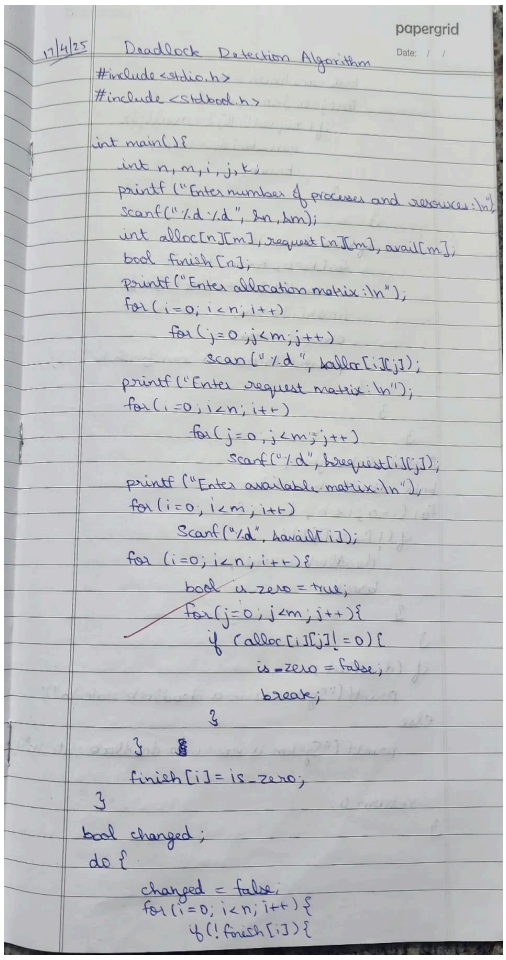
Code:

```
#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 alloc[n][m], request[n][m], avail[m];
    bool finish[n];
    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter request matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &request[i][j]);
    printf("Enter available matrix:\n");
    for (i = 0; i < m; i++)
        scanf("%d", &avail[i]);
    for (i = 0; i < n; i++) {
        bool is_zero = true;
        for (j = 0; j < m; j++) {
            if (alloc[i][j] != 0) {
                is_zero = false;
                break;
            }
```

```
    }
    finish[i] = is_zero;
}
bool changed;
do {
    changed = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_finish = true;
            for (j = 0; j < m; j++) {
                if (request[i][j] > avail[j]) {
                    can_finish = false;
                    break;
                }
            }
            if (can_finish) {
                for (k = 0; k < m; k++)
                    avail[k] += alloc[i][k];
                finish[i] = true;
                changed = true;
                printf("Process %d can finish.\n", i);
            }
        }
    }
} while (changed);
bool deadlock = false;
for (i = 0; i < n; i++) {
    if (!finish[i]) {
```

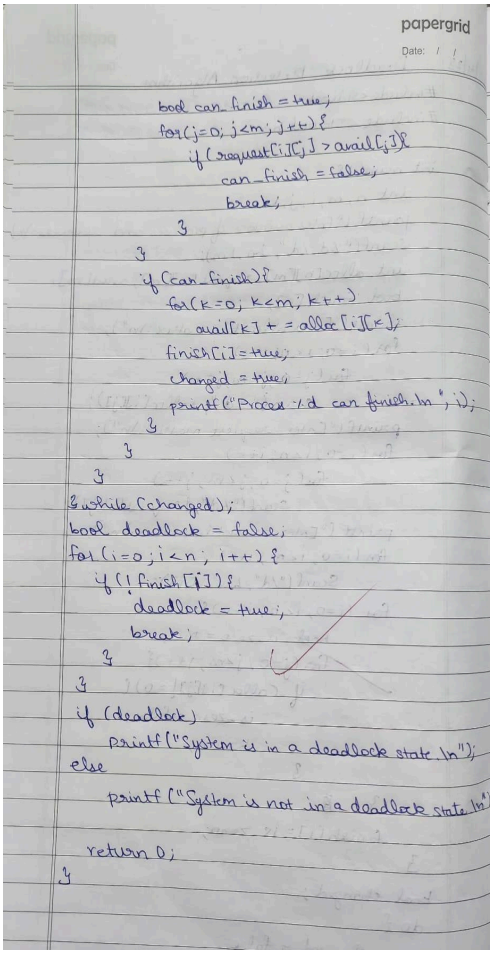
```
            deadlock = true;
            break;
        }
    }
    if (deadlock)
        printf("System is in a deadlock state.\n");
    else
        printf("System is not in a deadlock state.\n");
    return 0;
}
```

OUTPUT:

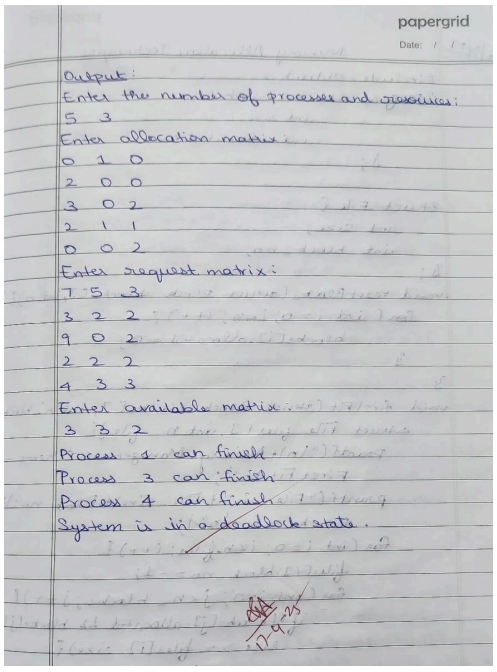


17/4/25 Deadlock Detection Algorithm

```
#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 alloc[n][m], request[n][m], avail[m];
    bool finish[n];
    printf("Enter allocation matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter request matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            scanf("%d", &request[i][j]);
    printf("Enter available matrix:\n");
    for(i=0; i<m; i++)
        scanf("%d", &avail[i]);
    for(i=0; i<n; i++){
        bool is_zero = true;
        for(j=0; j<m; j++){
            if(alloc[i][j] != 0){
                is_zero = false;
                break;
            }
        }
        finish[i] = is_zero;
    }
    bool changed;
    do {
        changed = false;
        for(i=0; i<n; i++){
            if(!finish[i]){
```



```
                bool can_finish = true;
                for(j=0; j<m; j++){
                    if(request[i][j] > avail[j]){
                        can_finish = false;
                        break;
                    }
                }
                if(can_finish){
                    for(k=0; k<m; k++)
                        avail[k] += alloc[i][k];
                    finish[i] = true;
                    changed = true;
                    printf("Process %d can finish.\n", i);
                }
            }
        }
    } while(changed);
    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;
}
```



Output:

```
Enter the 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.
```

Program -6

Question: Write a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit

b)Best-fit

c)First-fit

Code:

```
#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;
```

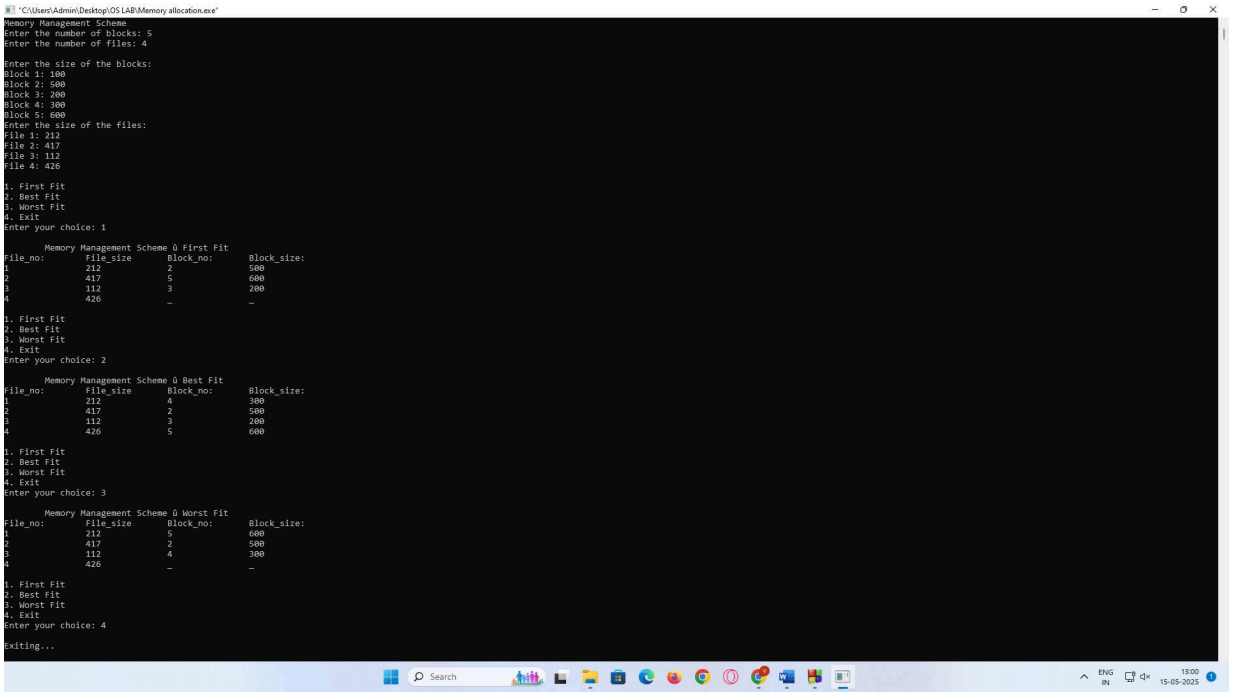
```
            blocks[j].allocated = 1;
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, j + 1, blocks[j].size);
            break;
        }
    }
    if (files[i].block_no == -1) {
        printf("%d\t\t%d\t\t_\t\t_\n", i + 1, files[i].size);
    }
  }
}
void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
  printf("\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\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, bestIdx + 1, blocks[bestIdx].size);
    } else {
      printf("%d\t\t%d\t\t_\t\t_\n", i + 1, files[i].size);
```

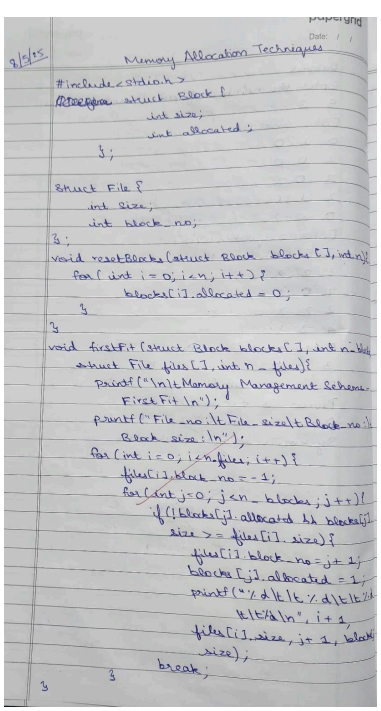
```
        }
    }
}
void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\n\tMemory Management Scheme – Worst Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    for (int i = 0; i < n_files; i++) {
        int worstIdx = -1;
     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

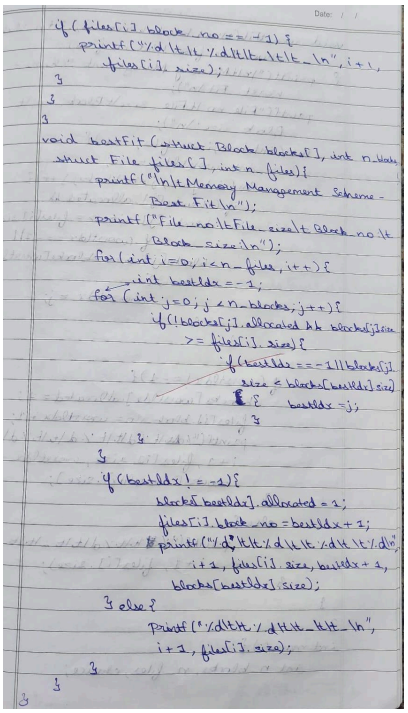
OUTPUT:



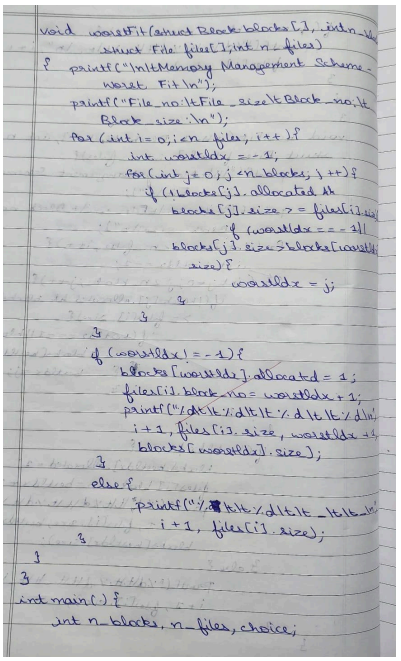


Memory Allocation Techniques

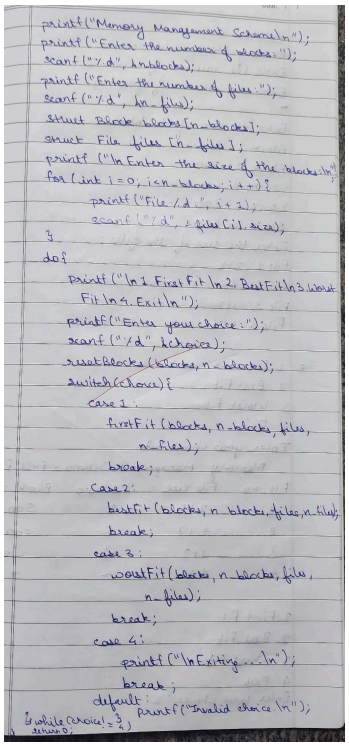
```
#include<stdio.h>
struct Block {
    int size;
    int allocated;
};
struct File {
    int size;
    int block_no;
};
void resetBlocks(struct Block blocks[], int n) {
    for(int i=0; i<n; i++)
        blocks[i].allocated = 0;
}
void firstFit(struct Block blocks[], int n_blocks,
    struct File files[], int n_files){
    printf("\n\tMemory Management Scheme -
        First Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\t
        Block_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\t%d\t\t%d\t\t%
                    d\n", i+1, files[i].size, j+1, block
                    size);
                break;
            }
        }
    }
```



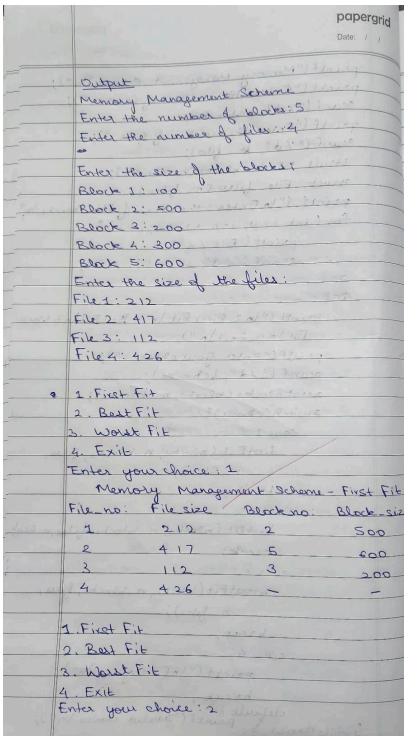
```
    if(files[i].block_no == -1){
        printf("%d\t\t%d\t\t\t\t\t\n", i+1,
            files[i].size);
    }
}
void bestFit(struct Block blocks[], int n_blocks,
    struct File files[], int n_files){
    printf("\n\tMemory Management Scheme -
        Best Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\t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



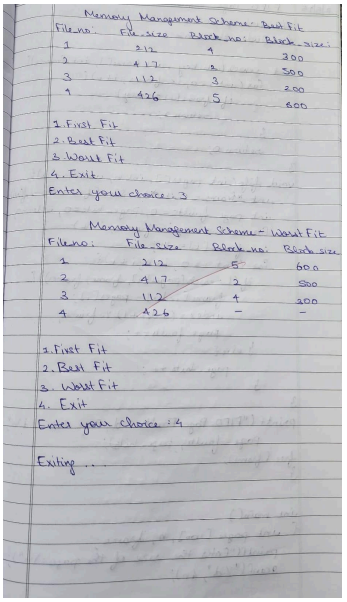
Output

```
Memory Management Scheme
Enter the number of blocks: 5
Enter the number of files: 4

Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter the size of the files:
File 1: 212
File 2: 417
File 3: 112
File 4: 426

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

Memory Management Scheme - First Fit

File_no:	File size	Block no:	Block_size
1	212	2	500
2	417	5	600
3	112	3	200
4	426	–	–

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



Memory Management Scheme - Best Fit

File_no:	File_size	Block_no:	Block_size:
1	212	4	300
2	417	2	500
3	112	3	200
4	426	5	600

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

Memory Management Scheme - Worst Fit

File_no:	File_size	Block_no:	Block size
1	212	5	600
2	417	2	500
3	112	4	300
4	426	–	–

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

Exiting...
```

Question:

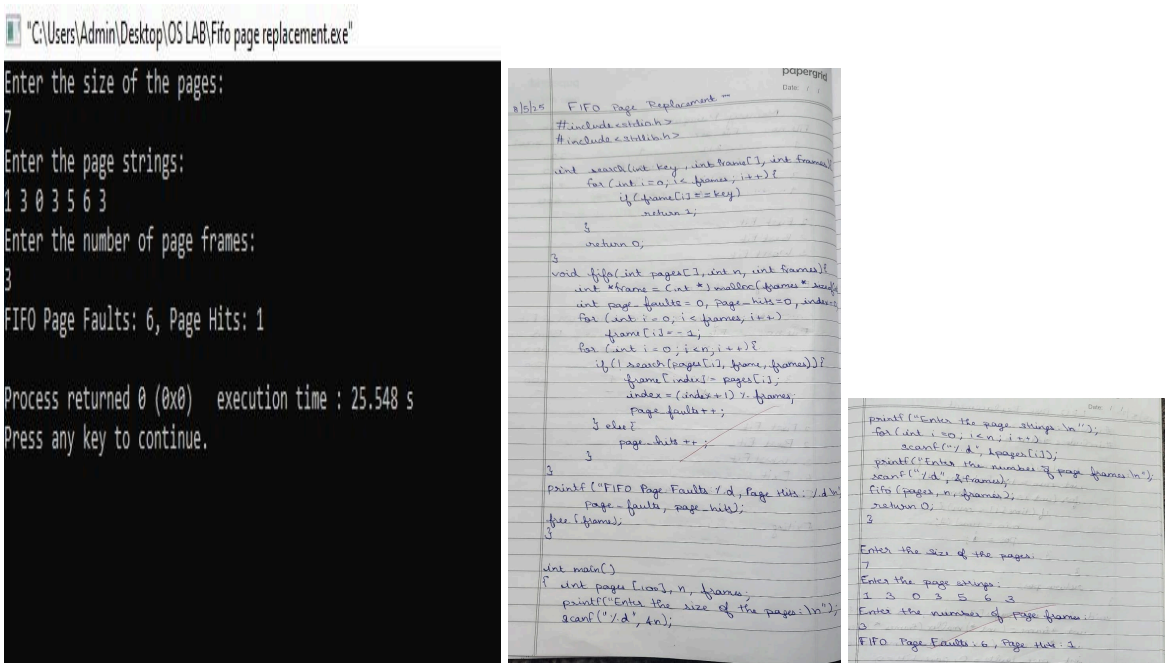
Code:

a)FIFO

```
#include <stdio.h>
#include <stdlib.h>
int search(int key, int frame[], int frames) {
    for (int i = 0; i < frames; i++) {
        if (frame[i] == key)
            return 1;
    }
    return 0;
}
void fifo(int pages[], int n, int frames) {
    int *frame = (int *)malloc(frames * sizeof(int));
    int page_faults = 0, page_hits = 0, index = 0;
    for (int i = 0; i < frames; i++)
        frame[i] = -1;
    for (int i = 0; i < n; i++) {
        if (!search(pages[i], frame, frames)) {
            frame[index] = pages[i];
            index = (index + 1) % frames;
            page_faults++;
        } else {
            page_hits++;
        }
    }
    printf("FIFO Page Faults: %d, Page Hits: %d\n", page_faults, page_hits);
    free(frame);
```

```
}

    int pages[100], n, frames;
    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    printf("Enter the page strings:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter the number of page frames:\n");
    scanf("%d", &frames);
    fifo(pages, n, frames);
    return 0;
}
```

OUTPUT:



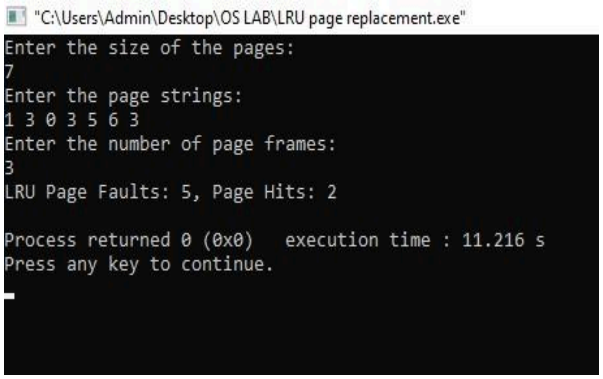
b)LRU

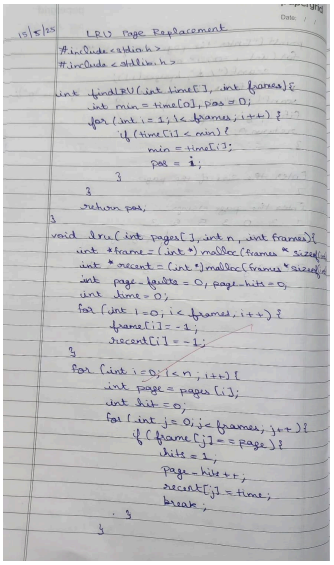
```
#include <stdio.h>
#include <stdlib.h>
int findLRU(int time[], int frames) {
    int min = time[0], pos = 0;
    for (int i = 1; i < frames; i++) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}
void lru(int pages[], int n, int frames) {
    int *frame = (int *)malloc(frames * sizeof(int));
    int *recent = (int *)malloc(frames * sizeof(int));
    int page_faults = 0, page_hits = 0;
    int time = 0;
    for (int i = 0; i < frames; i++) {
        frame[i] = -1;
        recent[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int hit = 0;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == page) {
                hit = 1;
```

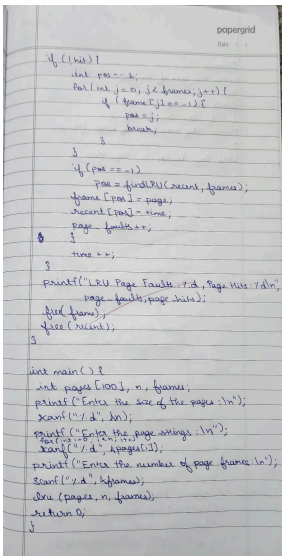
```
            page_hits++;
            recent[j] = time;
            break;
        }
    }
    if (!hit) {
        int pos = -1;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == -1) {
                pos = j;
                break;
            }
        }
        if (pos == -1)
            pos = findLRU(recent, frames);
        frame[pos] = page;
        recent[pos] = time;
        page_faults++;
    }
    time++;
  }
  printf("LRU Page Faults: %d, Page Hits: %d\n", page_faults, page_hits);
  free(frame);
  free(recent);
}
int main() {
  int pages[100], n, frames;
```

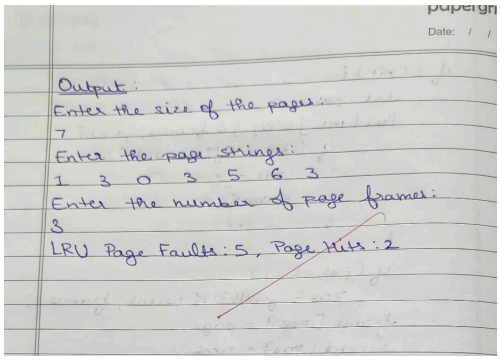
```
    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    printf("Enter the page strings:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter the number of page frames:\n");
    scanf("%d", &frames);
    lru(pages, n, frames);
    return 0;
}
```

OUTPUT:









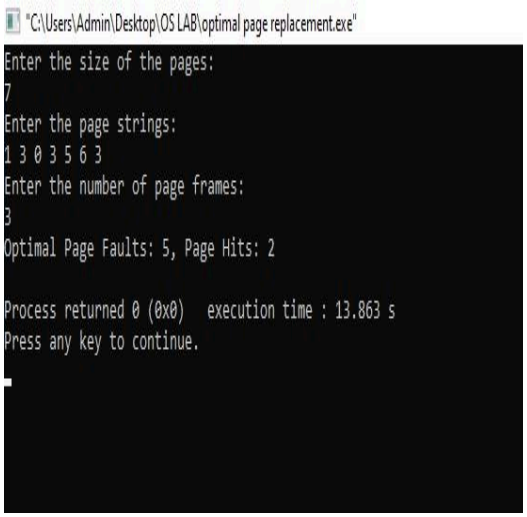
c)Optimal

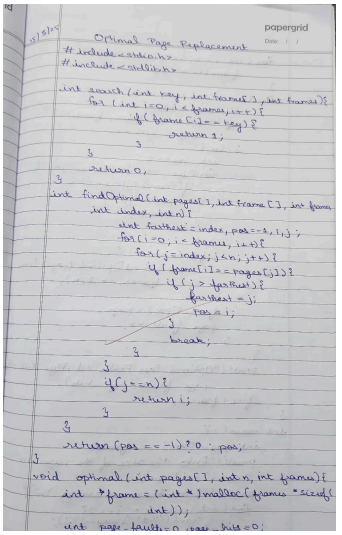
```
#include <stdio.h>
#include <stdlib.h>
int search(int key, int frame[], int frames) {
   for (int i = 0; i < frames; i++) {
      if (frame[i] == key) {
         return 1;
      }
   }
   return 0;
}
int findOptimal(int pages[], int frame[], int frames, int index, int n) {
   int farthest = index, pos = -1, i, j;
   for (i = 0; i < frames; i++) {
      for (j = index; j < n; j++) {
         if (frame[i] == pages[j]) {
            if (j > farthest) {
               farthest = j;
               pos = i;
            }
            break;
         }
      }
      if (j == n) {
         return i;
      }
   }
   return (pos == -1) ? 0 : pos;
```

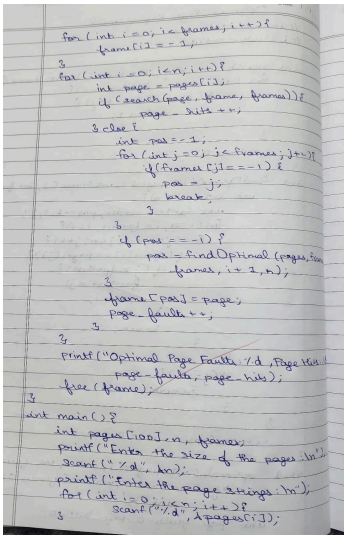
```
}
void optimal(int pages[], int n, int frames) {
    int *frame = (int *)malloc(frames * sizeof(int));
    int page_faults = 0, page_hits = 0;
    for (int i = 0; i < frames; i++) {
        frame[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        int page = pages[i];
        if (search(page, frame, frames)) {
            page_hits++;
        } else {
            int pos = -1;
            for (int j = 0; j < frames; j++) {
                if (frame[j] == -1) {
                    pos = j;
                    break;
                }
            }
            if (pos == -1) {
                pos = findOptimal(pages, frame, frames, i + 1, n);
            }
            frame[pos] = page;
            page_faults++;
        }
    }
    printf("Optimal Page Faults: %d, Page Hits: %d\n", page_faults, page_hits);
    free(frame);
```

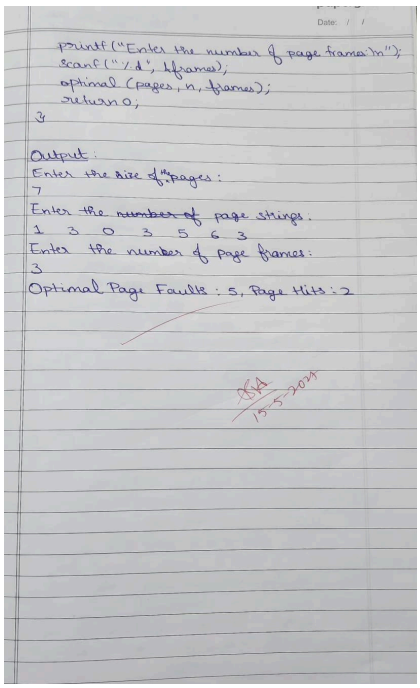
```
}
int main() {
    int pages[100], n, frames;
    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    printf("Enter the page strings:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }
    printf("Enter the number of page frames:\n");
    scanf("%d", &frames);
    optimal(pages, n, frames);
    return 0;
}
```

OUTPUT:


"C:\Users\Admin\Desktop\OS LAB\optimal page replacement.exe"
Enter the size of the pages:
7
Enter the page strings:
1 3 0 3 5 6 3
Enter the number of page frames:
3
Optimal Page Faults: 5, Page Hits: 2
Process returned 0 (0x0) execution time : 13.863 s
Press any key to continue.


papergrid
Optimal Page Replacement




printf("Enter the number of page frames:\n");
scanf("%d", &frames);
optimal(pages, n, frames);
return 0;
Output:
Enter the size of pages:
7
Enter the page strings:
1 3 0 3 5 6 3
Enter the number of page frames:
3
Optimal Page Faults: 5, Page Hits: 2