VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VARSHA A Y (1WA23CS032)

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 Student name (USN), 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

INDEX

NAME: Varsha A.Y STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: ~~FULL STACK WEB DEVELOPMENT~~

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	06/03	FCFS (first come first serve)		
2.	06/03	SJF (Shortest job first)	LAB1	
		SRTF (Shortest remaing time first)		
3.		CPU scheduling	LAB2	
4.		multilevel queue		
5.		Rate monotonic	LAB3	
6.		Deadline		
7.		Produer consumer	LAB4	
8.		Dining philosopher		
9.		Banker's algorithm	LAB5	
10.		Deadlock		
11.		memory mangement	LAB6	
12.		page Replacmt	LAB7	

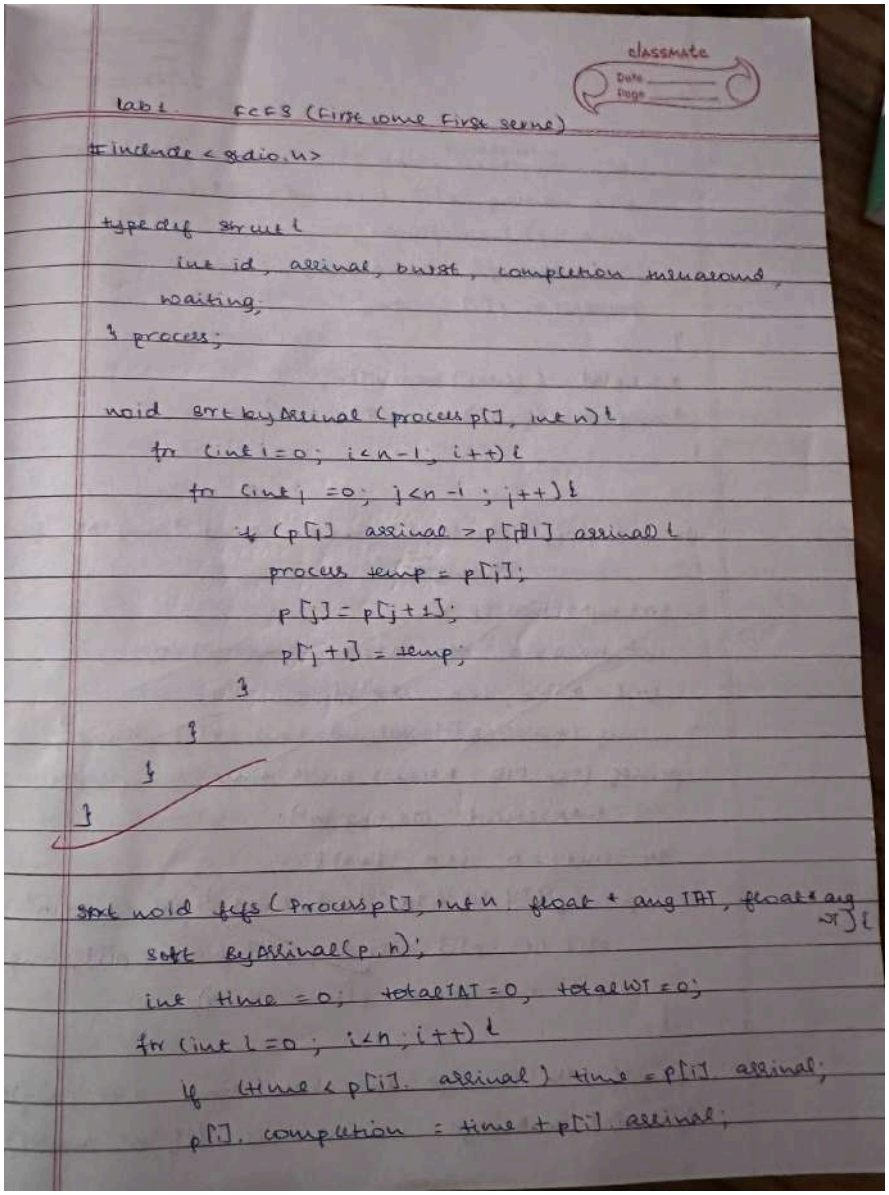
Course Outcomes

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

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)

FCFS:



```
lab 1    FCFS (First come First serve)
#include <stdio.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-1 ; j++) {
            if (p[j] arrival > p[j+1] arrival) {
                process temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
        }
    }
}

sort void fcfs (Process p[], int n, float * avgTAT, float* avg
                                                    WT) {
    sortByArrival(p, n);
    int time = 0; totalTAT = 0, totalWT = 0;
    for (int i =0 ; i<n ; i++) {
        if (time < p[i]. arrival ) time = p[i]. arrival;
        p[i]. completion = time + p[i]. arrival;
```

```
p[i].turnaround = p[i].completion - p[i].arrival;
p[i].waiting = p[i].turnaround - p[i].burst;
time = p[i].completion;
totalTAT += p[i].turnaround;
totalWT += p[i].waiting;
}
*avgTAT = (float) totalTAT/n;
*avgWT = (float) totalWT/n;
}

void display(process p[], int n, float *avgTAT, float *avgWT) {
    sortbyArrival(p, n);
    int time = 0, totalTAT=0, totalWT=0;
    for(int i=0; i<n; i++) {
        if(time < p[i].arrival) time = p[i].arrival;
printf("\n PID Arrival burst time Completion
        Turnaround waiting \n");
    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);
    }
printf("\n Average turnaround time : %2f ", avgTAT);
printf("\n Average waiting time %2f \n", avgWT);
```

```
int main(){
    int n;
    float avgWT, avgTAT;
    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("FCFS");
    fcfs(p, n, &avgTAT
    display(p, n, avgTAT, avgWT);

    return 0;
}
```

output-

output

Enter the number of processes: 4

Enter the Arrival time and Burst time for each processes:

0 7

0 3

0 4

0 6

(FCFS)

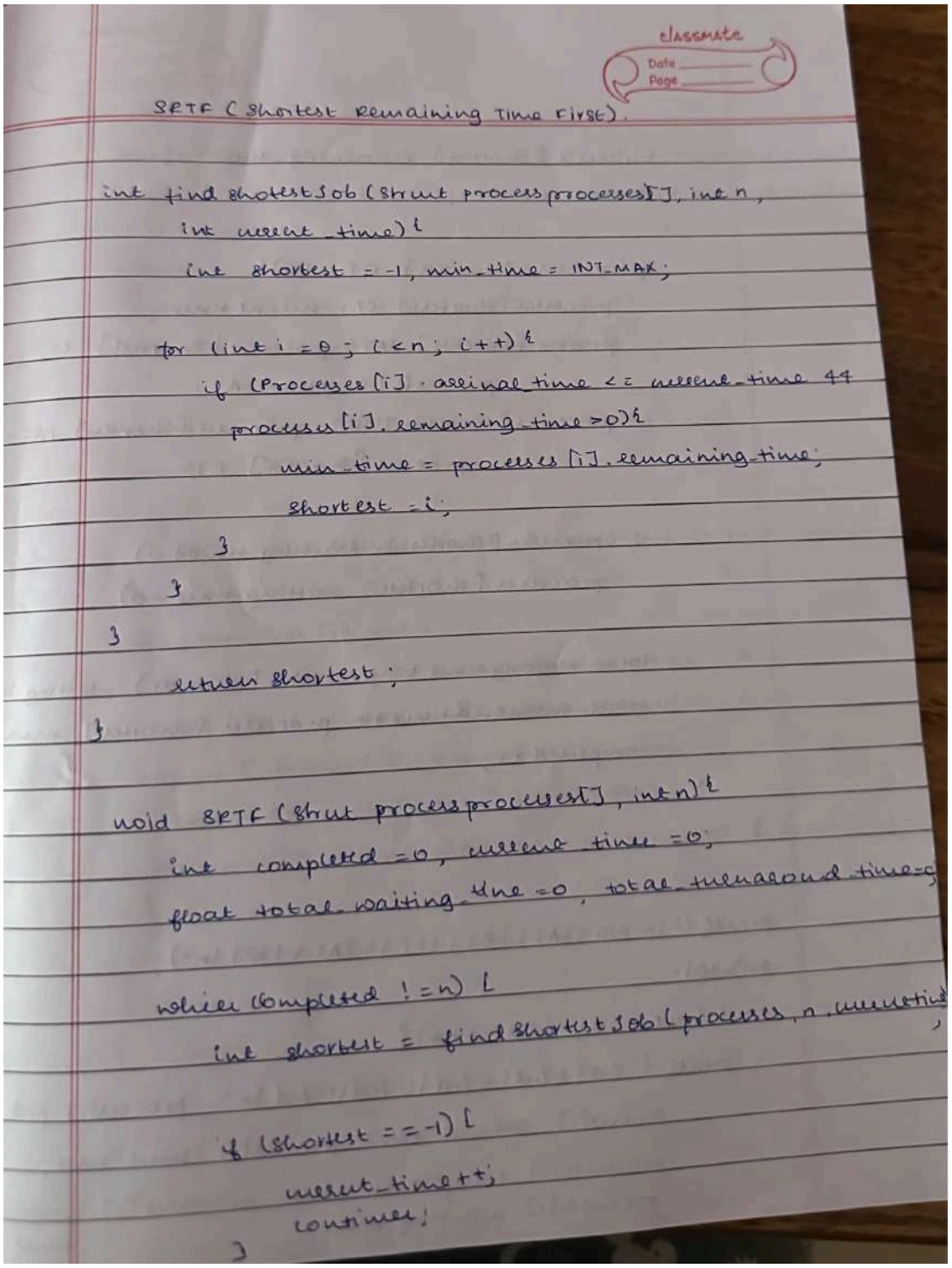
PID	Arrival	Burst	completion	TAT	WT
1	0	7	7	7	0
2	0	3	10	10	7
3	0	4	14	14	10
4	0	6	20	20	14

Average turnaround time : 12.75

Average waiting time : 7.75

6/3/25

SJF PREEMPTIVE:



SRTF (Shortest Remaining Time First).

```
int findShortestJob(struct process processes[], int n,
    int current_time) {
    int shortest = -1, min_time = INT_MAX;

    for (int i = 0; i < n; i++) {
        if (Processes[i].arrival_time <= current_time &&
            processes[i].remaining_time > 0) {
                min_time = processes[i].remaining_time;
                shortest = i;
            }
        }
    }
    return shortest;
}

void SRTF(struct process process[], int n) {
    int completed = 0, current_time = 0;
    float total_waiting_time = 0, total_turnaround_time = 0;

    while(completed != n) {
        int shortest = findShortestJob(processes, n, current_time,

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

```
        processes[shortest].remaining_time--;
        current_time++;

        if (processes[shortest].RT == 0){
            processes[shortest].CT = current_time;
            process[shortest].TAT = processes[shortest].CT -
                                    processes[shortest].AT;
            processes[shortest].WT = processes[shortest].TAT -
                                     process[shortest].BT;

            if (processes[shortest].waiting_time < 0)
                processes[shortest].waiting_time = 0)

            total_waiting_time += processes[shortest].waiting_tim
            total_turnaround_time += processes[shortest].turnaro_tim
            completed++;
        }
    }

    printf("\n PID \t AT \t BT \t CT \t TAT \t WT \n");
    printf("
    for (int i=0; i<n; i++){
        printf("%d \t %d \t %d \t %d \t %d \n", process[i].pid,
            process[i].arrival_time, process[i].burst_time,
            processes[i].completion_time, processes[i].turnaro
            processes[i].waiting_time);
                                                    time
```

```
printf("\naverage turnaroundtime: %.2f", total_turnaround_time/n);
printf("\n average waitingtime : %.2f\n", total_waiting_time/n);

main(){
 int n;
 struct process processes[MAX_processes]

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

 printf(" Enter Arrival time and Burst time for each process\n");
 for (int i=0; i<n; i++) {
 processes[i].pid = i+1;
  printf(" Process %d - Arrival time :", i+1);
  scanf("%d", &processes[i].arrival time);
  printf(" Process %d - Burst time :", i+1);
  scanf("%d", &processes[i].burst time);
  processes[i].remaining_time = processes[i].burst time;
 }

 SRTF (processes, n);
 return 0;
}
```

output:

Enter the number of process: 4

Enter the arrival time and Burst time:

0 8

1 4

2 9

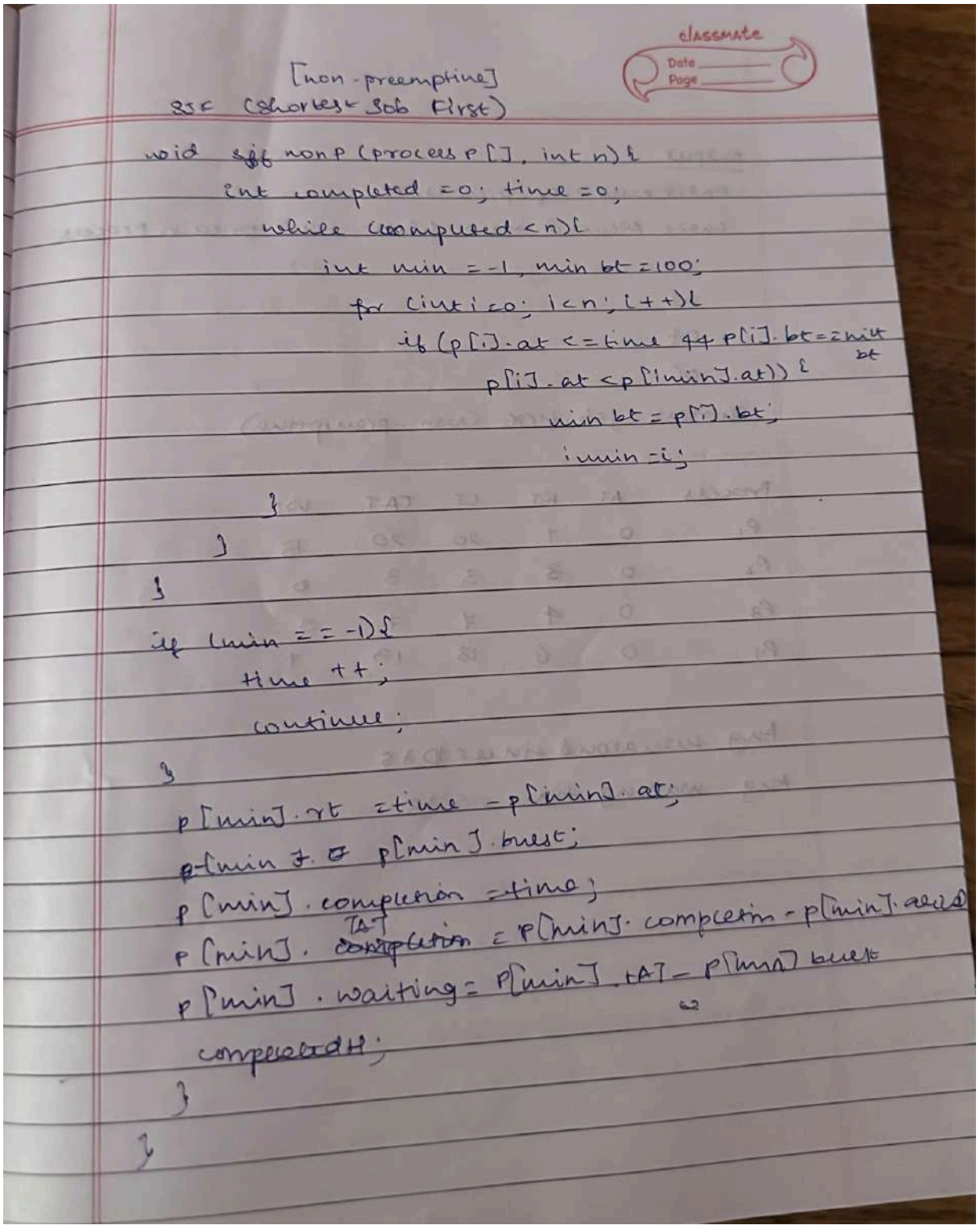
3 5

PID	AT	BT	WT	TAT	CT	RT
1	0	8	9	17	17	0
2	1	4	0	4	5	0
3	2	9	15	24	26	15
4	3	5	2	7	10	2

Average waiting time: 6.50

Average Turnaround time: 13.00

SJF NON-PREEMPTIVE:



```
[non-preemptive]
SJF (Shortest Job First)

void sjf nonP (process p[], int n){
    int completed = 0; time = 0;
    while (completed < n){
        int min = -1, min bt = 100;
        for (int i=0; i<n; i++){
            if (p[i].at <= time && p[i].bt = min bt
                p[i].at < p[imin].at)) {
                min bt = p[i].bt;
                imin = i;
            }
        }
    }
    if (imin == -1){
        time ++;
        continue;
    }
    p[min].rt = time - p[imin].at;
    p[min].bt = p[min].burst;
    p[min].completion = time;
    p[min]. [TAT] = p[min].completion - p[min].arrival
    p[min].waiting = p[min].TAT - p[min].burst
    completed++;
    }
}
```

output

Enter number of process: 4

Enter Arrival time & Burst time for each process

07

03

04

06

Shortest job first (Non-preemptive)

Process	AT	BT	CT	TAT	WT
P_1	0	7	20	20	13
P_2	0	3	3	3	0
P_3	0	4	7	7	3
P_4	0	6	13	13	7

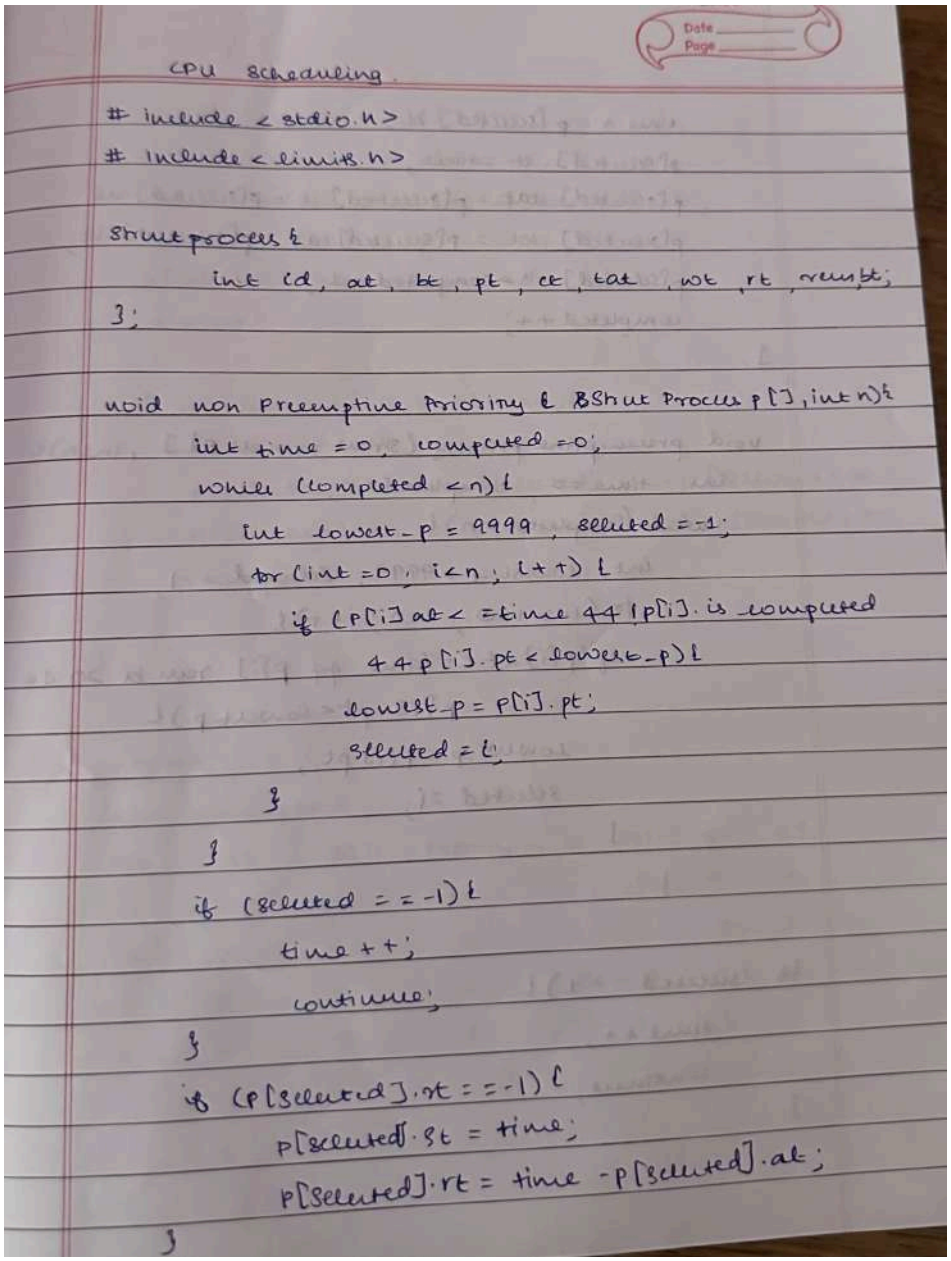
Avg turnaround time: 10.75

Avg waiting time = 5.73

Program-2

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)

PRIORITY PREEMPTIVE AND NON-PREEMPTIVE:



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

struct process {
    int id, at, bt, pt, ct, tat, wt, rt, remBt;
};

void non Preemptive Priority (struct Process p[], int n){
    int time = 0, completed = 0;
    while (completed < n) {
        int lowest_p = 9999, selected = -1;
        for (int i=0; i<n; i++) {
            if (p[i].at <= time && !p[i].is_completed
                && p[i].pt < lowest_p) {
                lowest_p = 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;
        }
```

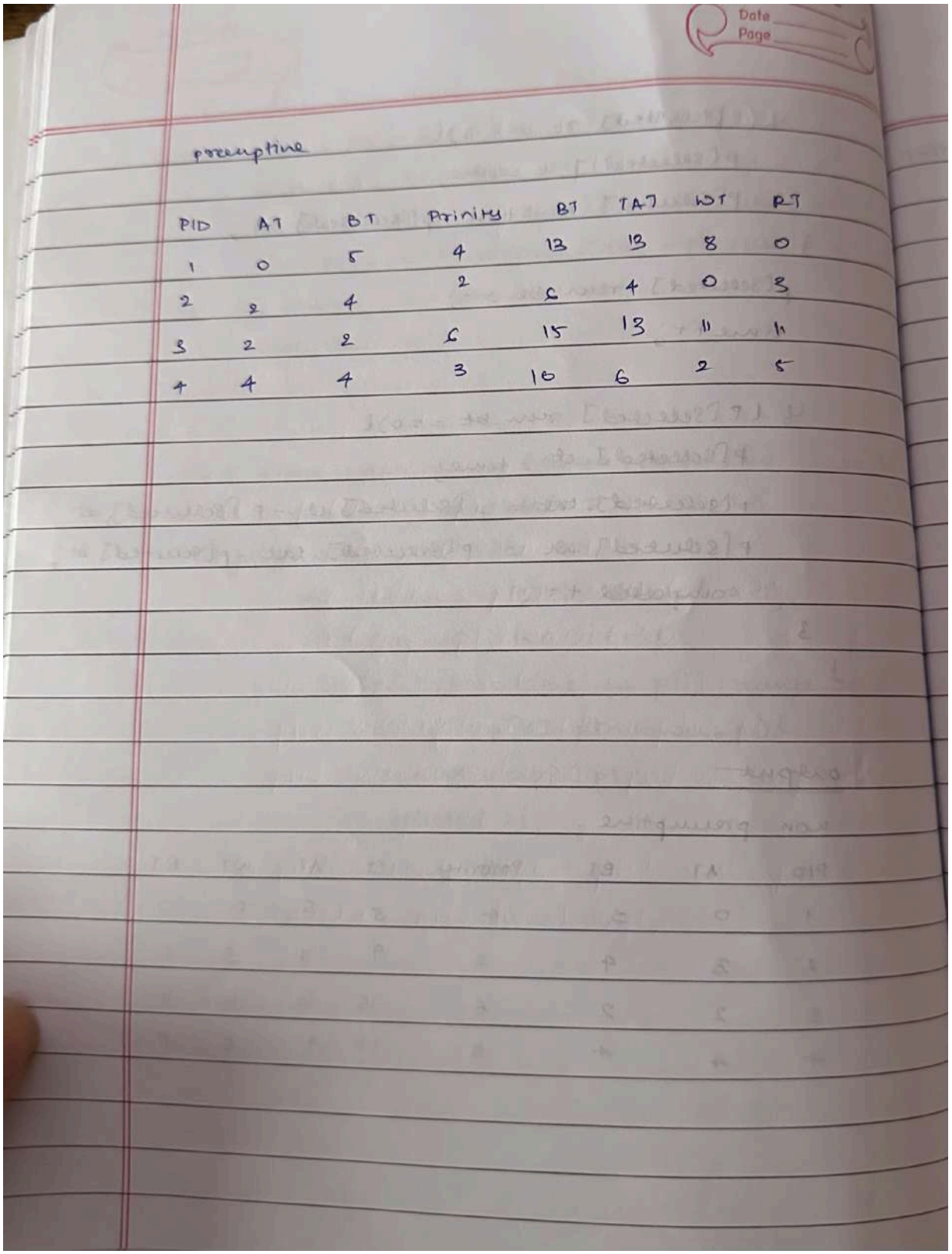
```
        time += p[selected].bt;
        p[selected].ct = time;
        p[selected].tat = p[selected].ct - p[selected].at;
        p[selected].wt = p[selected].tat - p[selected].bt;
        p[selected].is_completed = 1;
        completed++;
    }
}

void preemptive_priority(struct process p[], int n) {
    int time = 0, completed = 0;
    while (completed < n) {
        int lowest_p = 9999, selected = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].rem_bt > 0 &&
                p[i].pt < lowest_p) {
                lowest_p = 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].rem_bt --;
time++;

if(p[selected].rem_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++;
}
}
```

output

non preemptive.

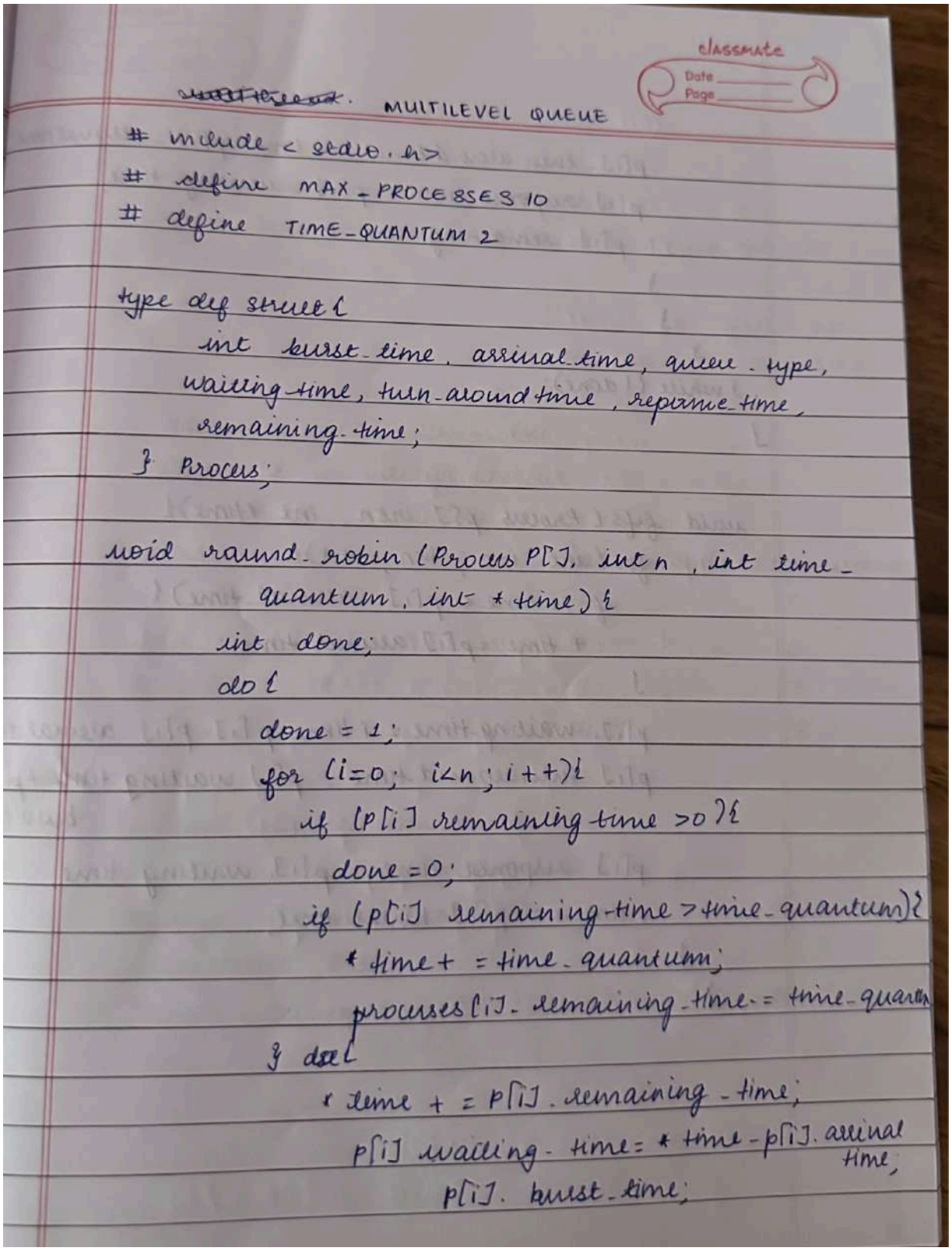
PID	AT	BT	Priority	CT	TAT	WT	RT
1	0	5	4	5	5	0	0
2	2	4	2	9	7	3	3
3	2	2	6	15	13	11	11
4	4	4	3	13	9	5	5



preemptive

PID	AT	BT	Priority	BT	TAT	WT	RT
1	0	5	4	13	13	8	0
2	2	4	2	6	4	0	3
3	2	2	6	15	13	11	11
4	4	4	3	10	6	2	5

ROUND ROBIN:



MULTILEVEL QUEUE

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define TIME_QUANTUM 2

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

void round_robin (Process P[], int n, int time_
        quantum, int * time) {
    int done;
    do {
    done = 1;
    for (i=0; i<n; i++){
      if (p[i] remaining_time >0 ){
        done = 0;
        if (p[i] remaining_time > time_quantum){
          * time + = time_quantum;
          processes[i]. remaining_time - = time_quantum
    } while
        * time + = p[i]. remaining_time;
        p[i] waiting_time = * time - p[i]. arrival
                                          time;
            p[i]. burst_time;
```

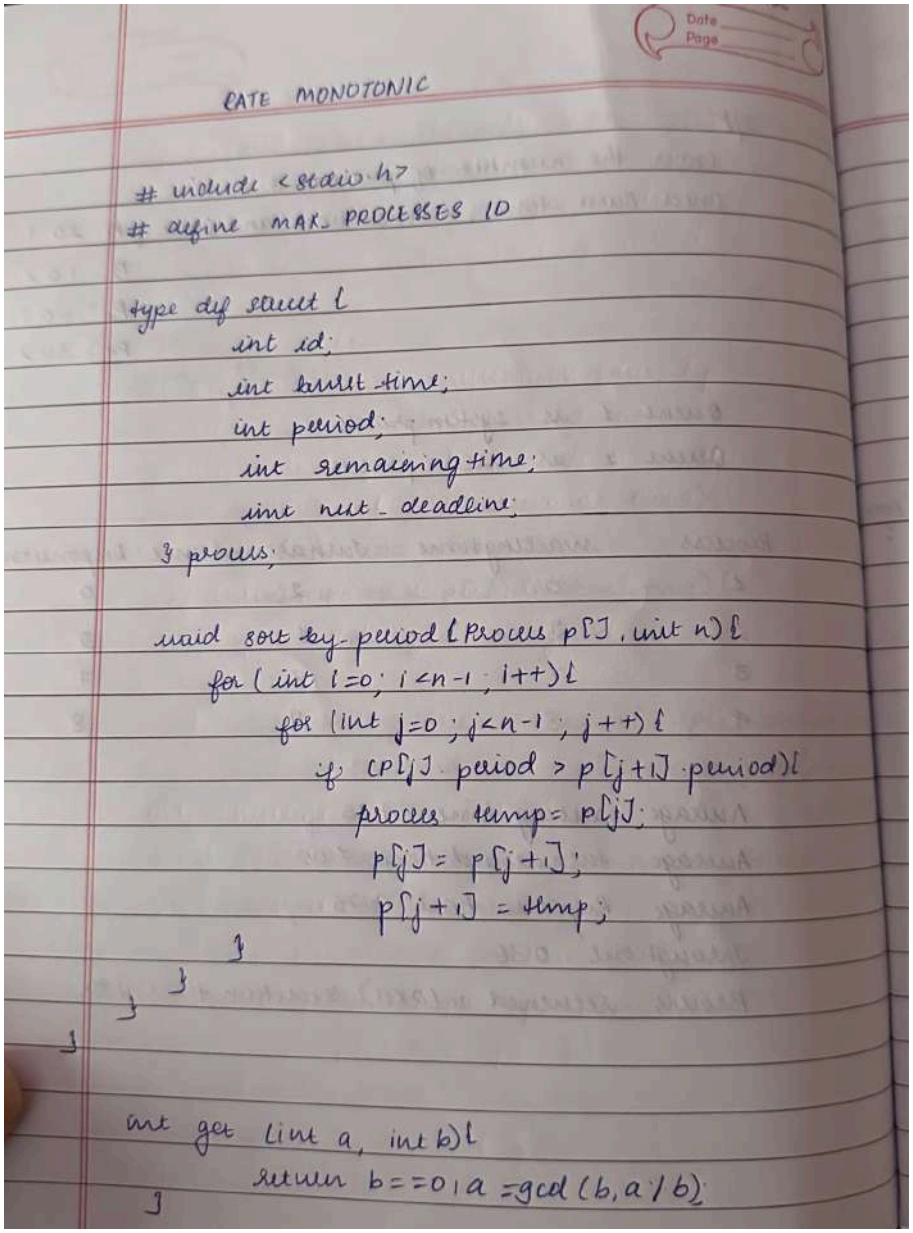
```
        p[i].turn around time = t time - p[i].arrival time
        p[i].response_time = p[i].waiting_time;
        p[i].remaining_time = 0;
      }
    }
  }
} while (!done);
}

void fcfs(Process p[], int n, int *time){
  for(int i=0; i<n; i++){
    if (*time < p[i].arrival_time){
      *time = p[i].arrival_time;
    }
    p[i].waiting_time = *time - p[i] - p[i].arrival time;
    p[i].turnaround time = p[i] waiting time + p[i].
                                              burst time;
    p[i].response_time = p[i].waiting_time;
    *time += p[i] burst time;
  }
}
```

```
        p[i].turn around time = *time - p[i].arrival time
        p[i].response_time = p[i].waiting_time;
        p[i].remaining_time = 0;
      }
    }
  }
} while (!done);
}

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

Program-4

3. Write a C program to simulate Real-Time CPU Scheduling algorithms:
   a) Rate- Monotonic
   b) Earliest-deadline First
   c) Proportional scheduling

a) RATE MONOTONIC:



```
RATE MONOTONIC

# include <stdio.h>
# define MAX_PROCESSES 10

type def struct {
    int id;
    int burst time;
    int period;
    int remaining time;
    int next_deadline;
} process;

void sort by period (Process p[], int n) {
    for (int i=0; i<n-1; i++) {
        for (int j=0; j<n-1; j++) {
            if (p[j] period > p[j+1] period) {
                process temp = p[j];
                p[j] = p[j+1];
                p[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 = process[0].period;
    for( int i=1 ; i<n ; i++){
        result = lcm (result, processes[i].int n]{
        double sum =0;
        for (int i=0; i<n; i++){
            sum += (double) process[i].bursttime/
                    process[i].period;
        }
        return sum;
    }

double rms threshlod (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", lcmperiod);
    printf (" Rate monotone scheduling);
```

```
printf("PID, Burst Period\n");
for(int i=0; i<n; i++){
  printf("%d %d %d\n", process[i].id, process[i].
                burst_time, process[i].period);
}

double utilisation = utilisation_factor(process, n);
double threshold = ens_threshold(n);
printf("\n%d %d %f <= %f ==> %f\n", utilisation,
      threshold, (utilisation <= threshold) ? "true" : "false");

if(utilisation > threshold){
    printf("\n system may not be schedulable\n");
    return;
}

int timeline = 0, count = 0;
while(time < lcm_period){
    int c = 7;
    for(time_i=0; i<c; i++){
        if(time % p[i].period == 0){
            p[i].rem = p[i].b.t;
        }
        if(p[i].rem > 0){
            scheduled = i;
            break;
        }
```

```
if (selected! = = 1) {
    printf (" Time %d Process %d is running;
                timeline p[idJ.id);
    p[id J. rem --;
} else {
    printf (" Time %d : CPU is idle\n", timeline);
}
    timeline + +;
  }
}
```

O/P

Enter number of processes: 3

Enter CPU burst time

3 6 8

Enter time period

3 4 5

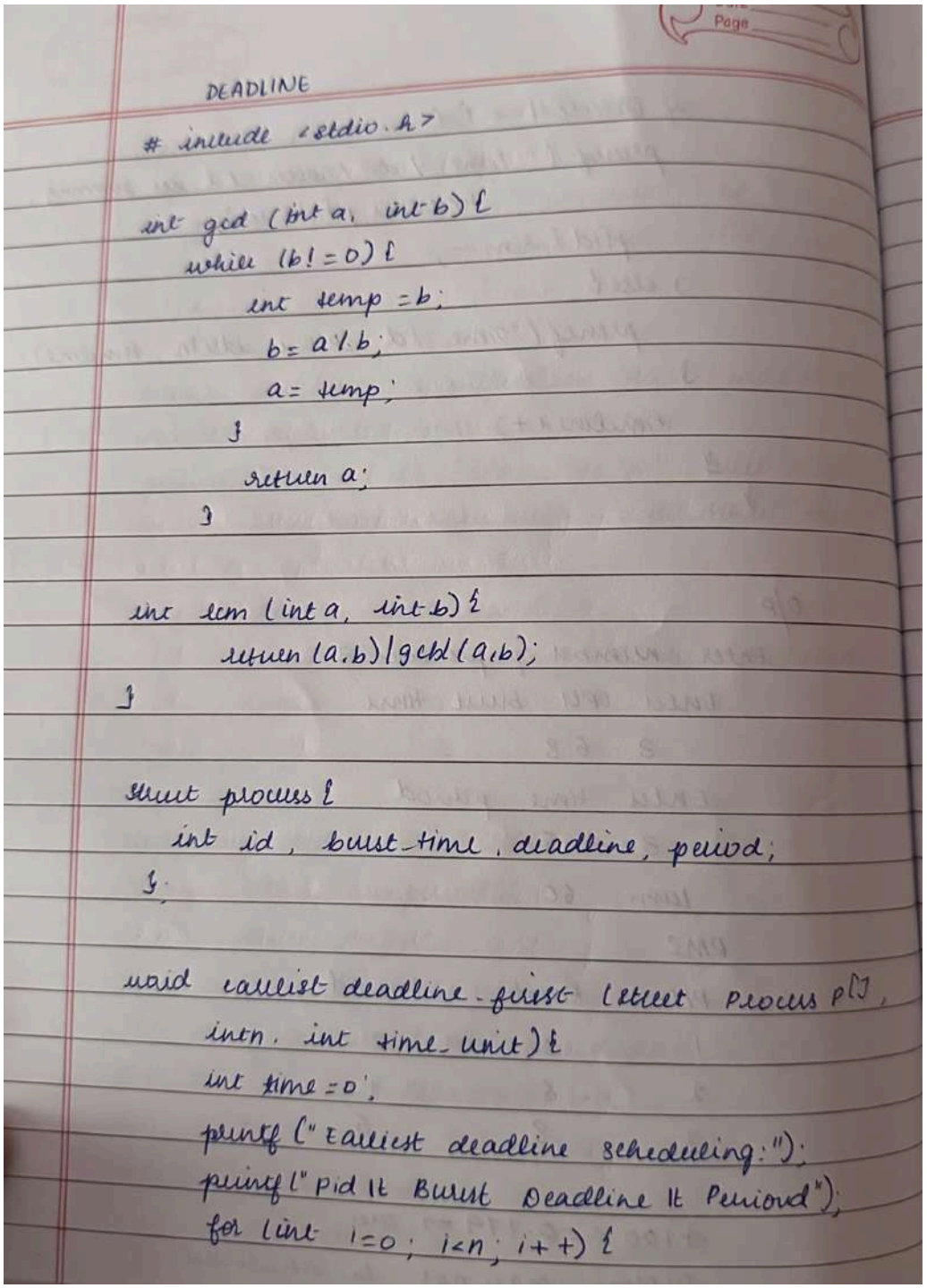
lcm : 60

RMS

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

4.100 <= 0.779 ⇒ can system may not be scheduled!

b) EARLIEST DEADLINE:



```
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_unit) {
  int time = 0;
  printf ("Earliest deadline scheduling:");
  printf ("Pid \t Burst Deadline \t Period");
  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("\n scheduling occurs for %d", 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 time,
        p[earliest].id);
    p[earliest].burst_time--;
    time++;
}
}
```

O/P

Enter the number of process : 3

Enter CPU burst time :

2 3 4

Enter deadlines

1 2 3

Enter the time period :

1 2 3

System will execute for hyperperiod : 6 ms

Earliest Deadline scheduler.

Pid	Burst	deadline	period
1	2	1	1
2	3	2	2
3	4	3	3

Scheduling occurs in for 6ms

0 ms : Task 1 is running

1 ms : Task 1 is running

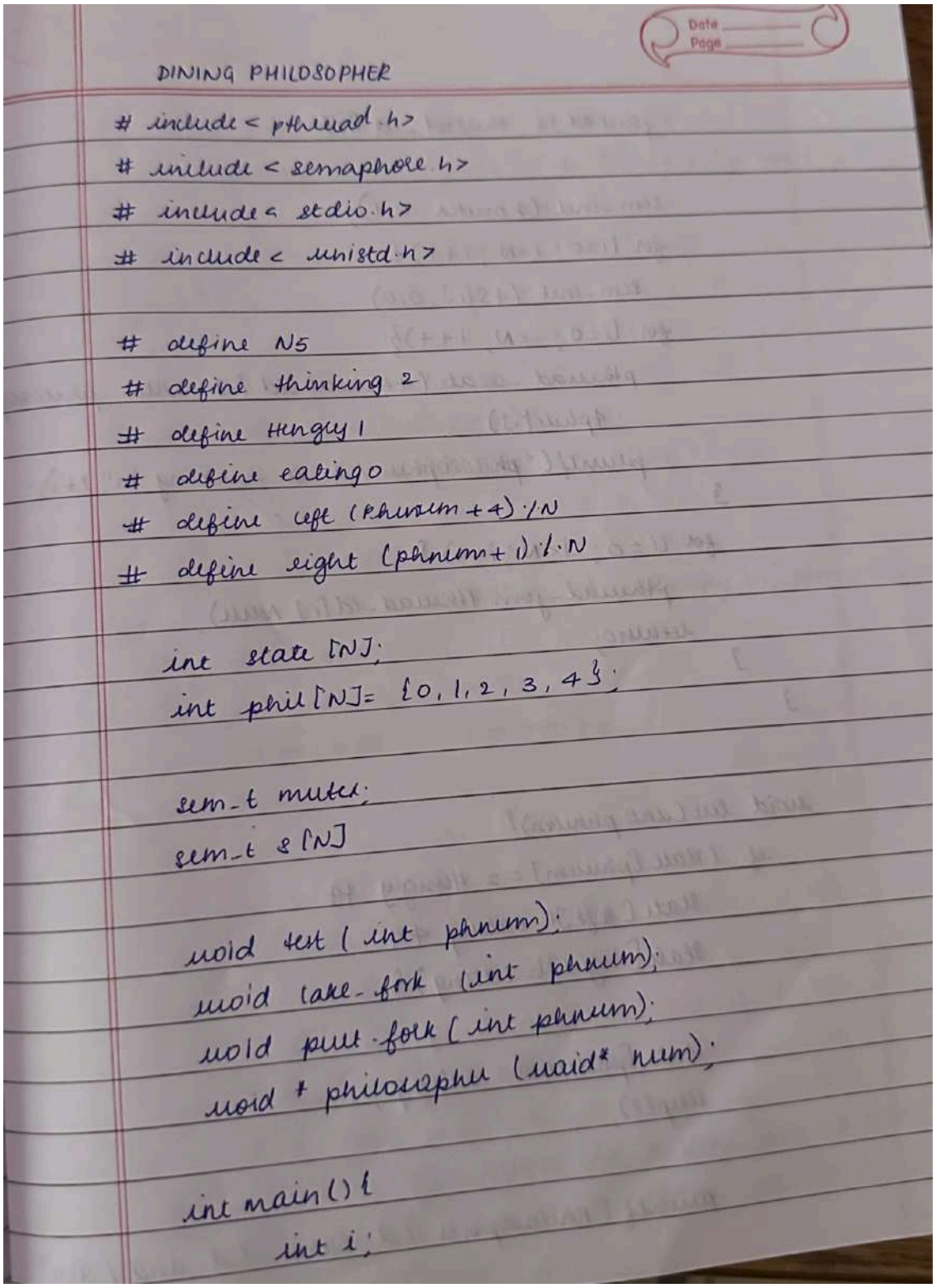
2 ms : Task 2 is running

3 ms : Task 2 is running

4 ms : Task 2 is running

5 ms : Task 3 is running

c) PROPORTIONAL SCHEDULING:



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);
void take_fork (int phnum);
void put_fork (int phnum);
void * philosopher (void* num);

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, philos
    &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[phnum] = Eating;
    sleep(2);

    printf("Philosopher %d takes %d and %d\n",
```

```
        phnum+1, left+1, phnum+1);
        printf("Philosopher %d is Eating\n", phnum+1);

        sem_post(&S[phnum]);
    }
}

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

    state[phnum] = Hungry;
    printf("Philosopher %d is Hungry\n", phnum+1);
    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}

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

    state[phnum] = Thinking;

    printf("Philosopher %d putting fork %d and %d
    down\n", phnum+1, left+1, phnum+1);
```

```
        printf("philosopher %d is thinking\n", phnum+1);

        test(left);
        test(right);

        sem_post(&mutex);
    }

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

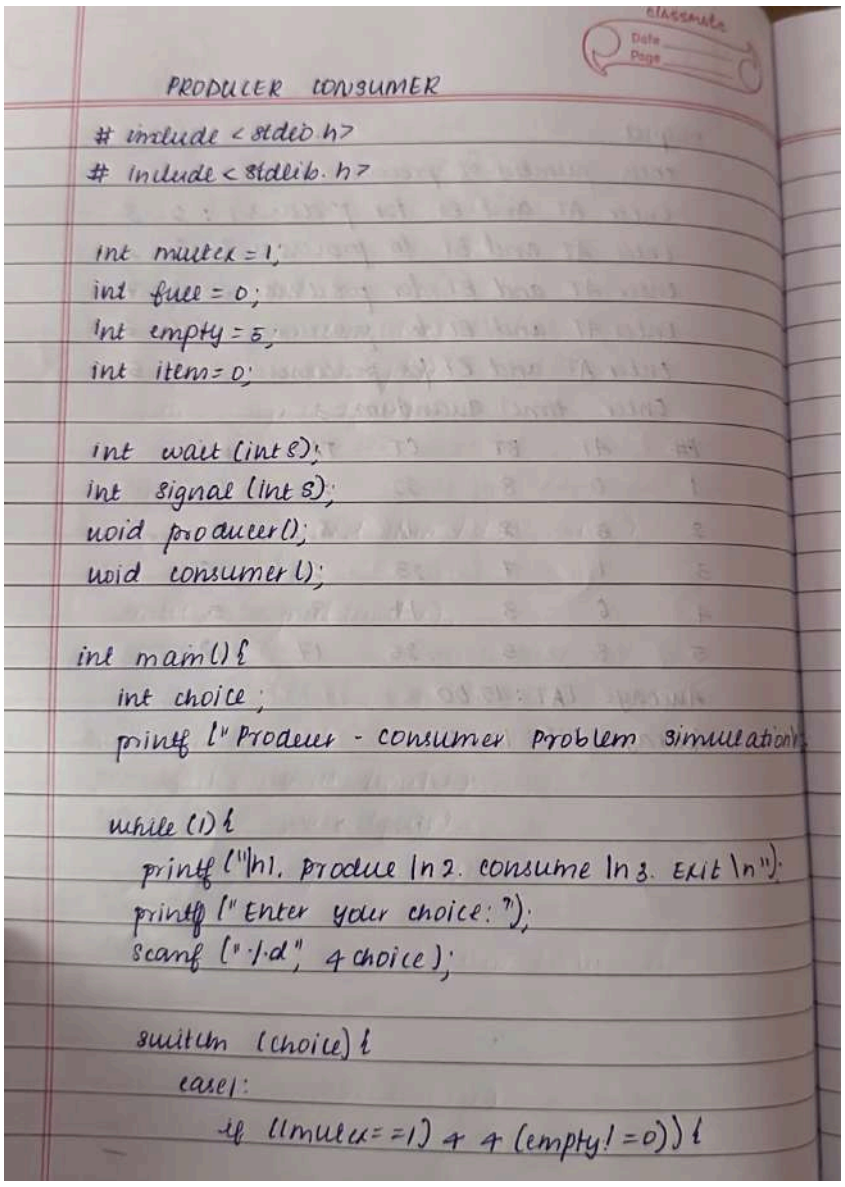
O/P

philosopher 1 is thinking
Philosopher 2 is thinking
philosopher 3 is thinking
Philosopher 4 is thinging
Philosopher 5 is thinking

Philosophu 5 is thungey
— 11 — 4 — 11 —
— 11 — 2 — 11 —
— 11 — 3 — 11 —
Philosophu is 3 takes fork 2 and 3
philosophu 3 is eating
philosophu 1 is thungey

Program-5

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



```
PRODUCER CONSUMER

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

int mutex = 1;
int full = 0;
int empty = 5;
int item = 0;

int wait (int s);
int signal (int s);
void producer();
void consumer();

int main() {
  int choice;
  printf("Producer - consumer Problem simulation\n");

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

    switch (choice) {
      case 1:
        if ((mutex == 1) && (empty != 0)) {
```

```
int signal (int S) {
    return ++S;
}

void producer() {
    mutex = wait (mutex);
    empty = wait (empty);
    full = signal (full);

    item++;
    printf ("Produced item :%d\n", item);

    mutex = signal (mutex);
}

void consumer() {
    mutex = wait (mutex);
    full = wait (full);
    empty = signal (empty);

    printf ("Consumed item :%d\n", item);
    item--;

    mutex = signal (mutex);
}
```

output:

producer-consumer Problem simulation

1. Produce
2. Consume
3. Exit

Enter your choice : 1

produced item: 1

1. Produce
2. Consume
3. Exit

Enter your choice : 2

consumed item 1

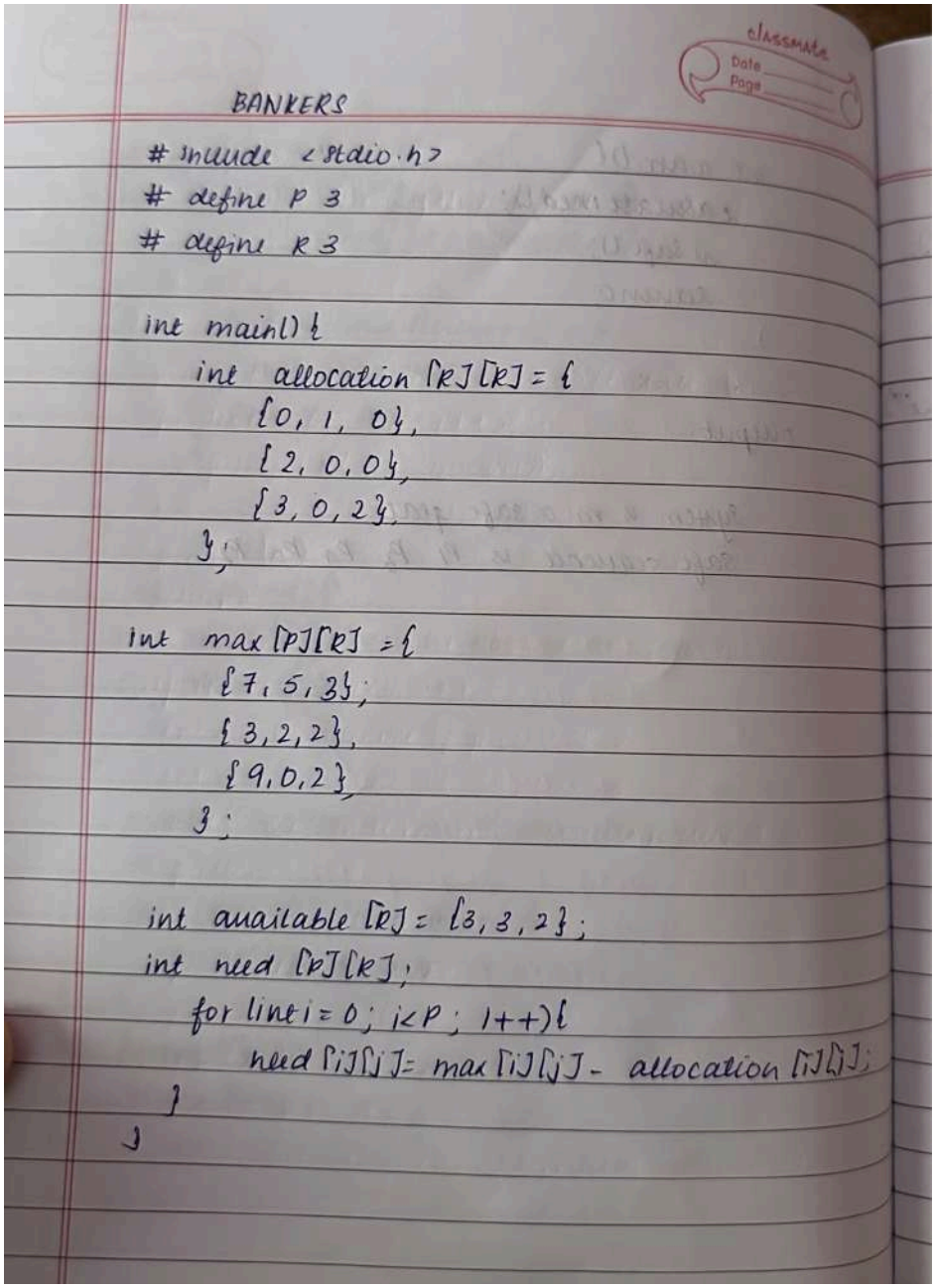
1. Produce
2. Consume
3. Exit

Enter your choice : 2

Buffer is empty or mutex is locked. cannot consume.

Program-7

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



```
BANKERS

#include <stdio.h>
#define P 3
#define R 3

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

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

int available[R] = {3, 3, 2};
int need[P][R];
    for(int i = 0; i<P; i++){
        need[i][j] = max[i][j] - allocation[i][j];
    }
}
```

```
int finish[P] = {0}, safe seq[P], work[R];
for (int i=0; i<R; i++) {
    work[i] = available[i];
}
int count=0;
while (count<P) {
    int found=0;
    for (int p=0; p<P; p++) {
        if (!finished[p]) {
            int canRun=1;
            for (int r=0; r<R; r++) {
                if (need[p][r] > work[r]) {
                    canRun=0;
                    break;
                }
            }
            if (canRun) {
                for (int r=0; r<R; r++) {
                    work[r] += allocation[p][r];
                }
                safe seq[count++] = p;
                finish[p] = 1;
                found=1;
            }
        }
    }
}
```

```
    if (!found) {
        printf ("The system is not in a safe state.\n");
        return 0;
    }
}

printf ("The system is in a safe state. \n safe sequence:");
for (int i=0; i<P; i++) {
    printf ("P%d", safeseq[i]);
}
printf ("\n");
return 0;
}
```

output:

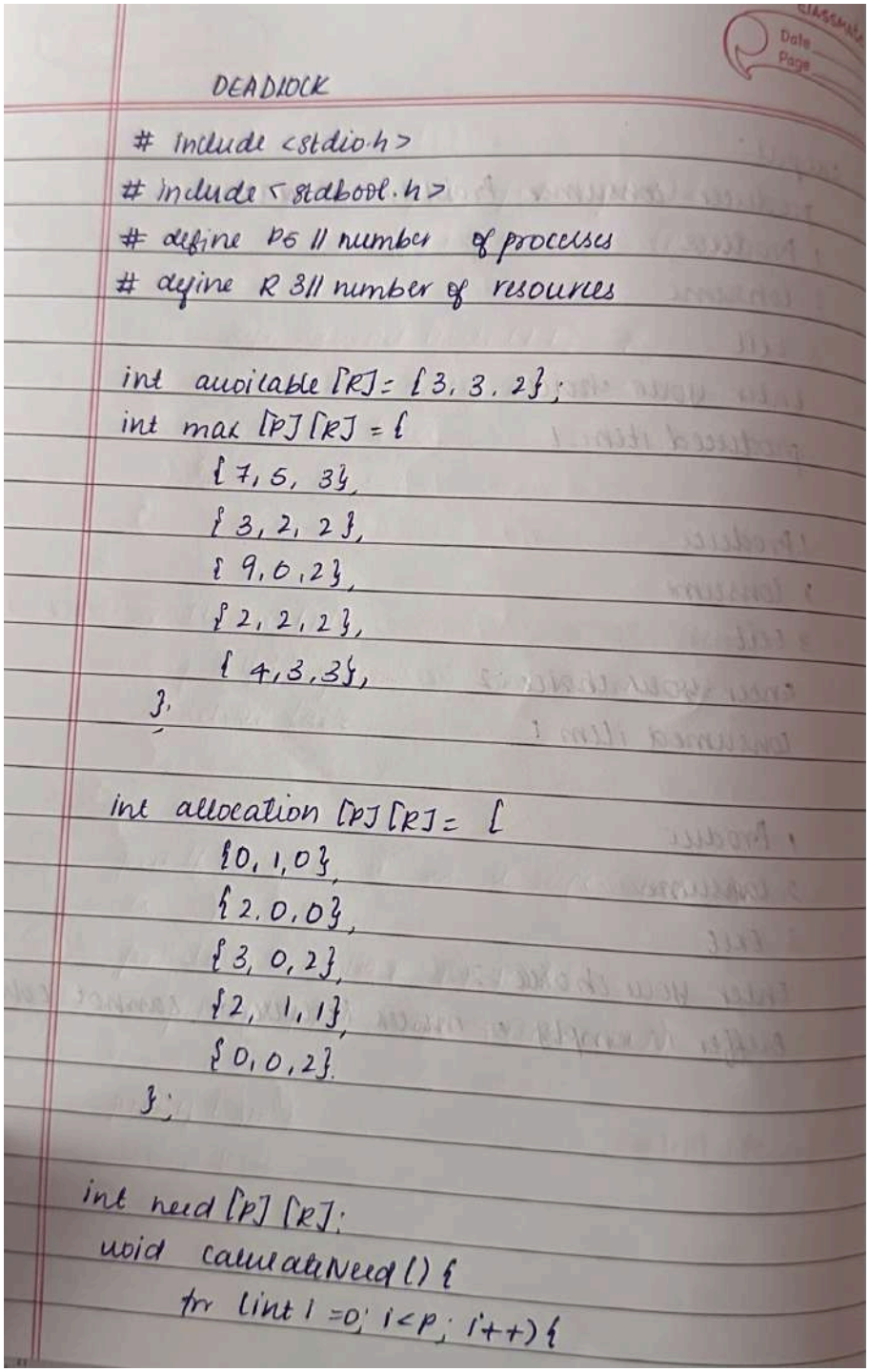
Enter the number of processes: 5
Enter the number of resources: 3
Enter the maximum resources required by each processes:
process P_0: 7 5 3
process P_1: 3 2 2
process P_2: 9 0 2
process P_3: 2 2 2
process P_4: 4 3 3
Enter the resources currently allocated to each process:

Process P_0 : 0 1 0
Process P_1 : 2 0 0
Process P_2 : 3 0 2
Process P_3 : 2 1 1
Process P_4 : 0 0 2

Enter the available resources :
3 3 2
The system is in safe state.
Safe sequence : P_1 P_3 P_4 P_0 P_2

Program-8

8. Write a C program to simulate deadlock detection.



```
DEADLOCK
# include <stdio.h>
# include <stdbool.h>
# define P5 // number of processes
# define R 3// number of resources

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

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

int need [P] [R];
void calculateNeed () {
    for (int i = 0; i < P; i++) {
```

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

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

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

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

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

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

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

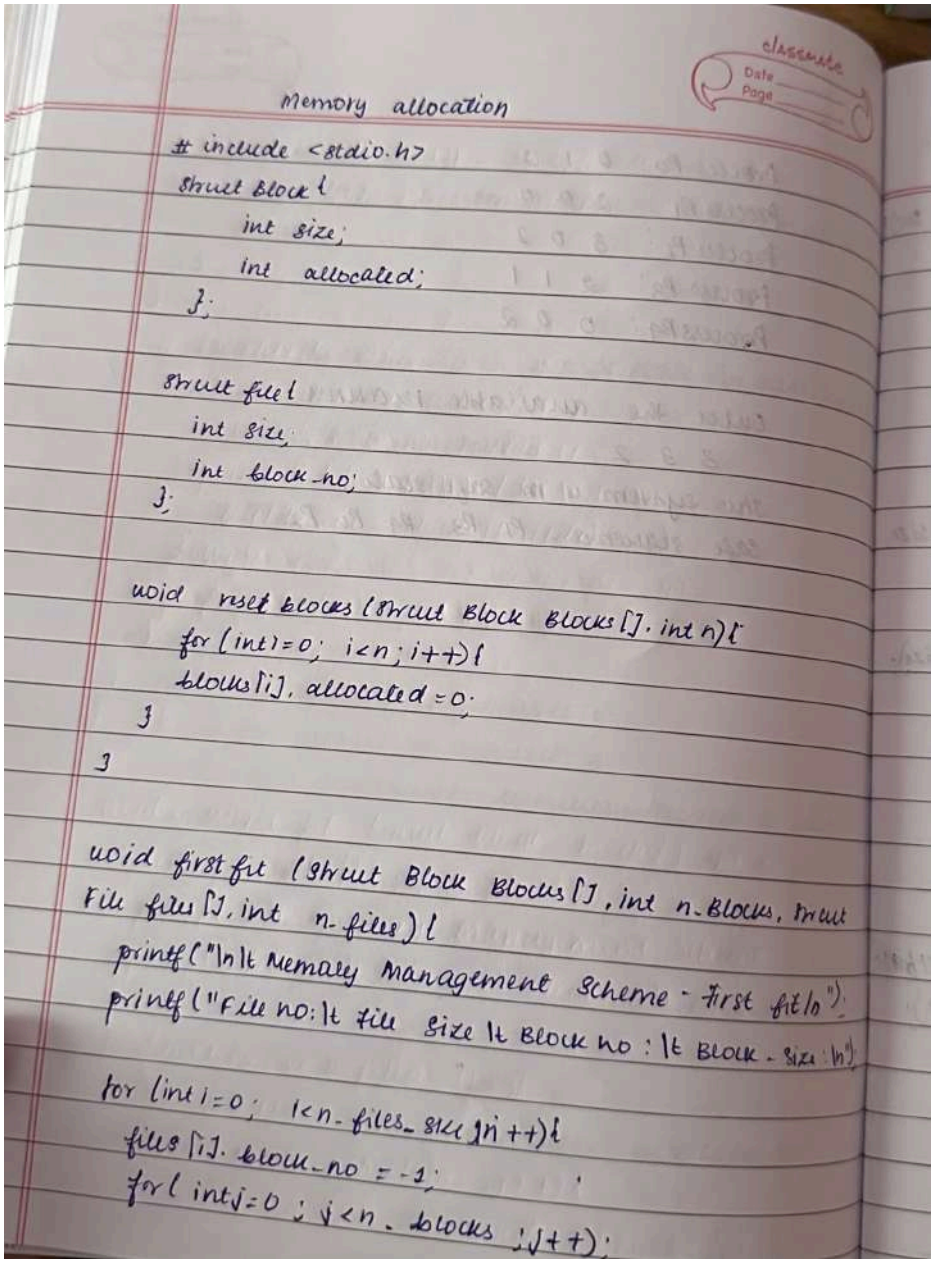
Output:

System is in a safe state
Safe sequence is: P_1 P_3 P_4 P_0 P_2.

Program-9

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

a) Best-fit
b) Worst-fit
c) First-fit



```
Memory allocation

#include <stdio.h>
struct block {
    int size;
    int allocated;
};

struct file {
    int size;
    int block_no;
};

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

void first_fit (struct Block blocks[], int n_blocks, struct
File files[], int n_files) {
  printf("\n\t Memory Management Scheme - First fit\n");
  printf("File no:\t File Size \t Block no : \t Block - Size \n");

  for(int i=0; i<n_files_size;i++){
  files[i].block_no = -1;
  for( int j=0 ; j<n_blocks ;j++);
```

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

void best fit (struct Block blocks[], int n_blocks,
        struct file files[], int n_files) {
printf("\n\t Memory management scheme- Best fit\n");
printf(" File no :\t File_size\t Block_no:\t Block_size:\n");

for (int i=0; i<n_files; i++) {
    int bestIdx = -1;
    for (int j=0; j<n_block; 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 nBlocks,
struct File files[], int n_files) {
    printf("\n\t Memory Management Scheme - Worst Fit");
    printf("\nFile_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 >=
```

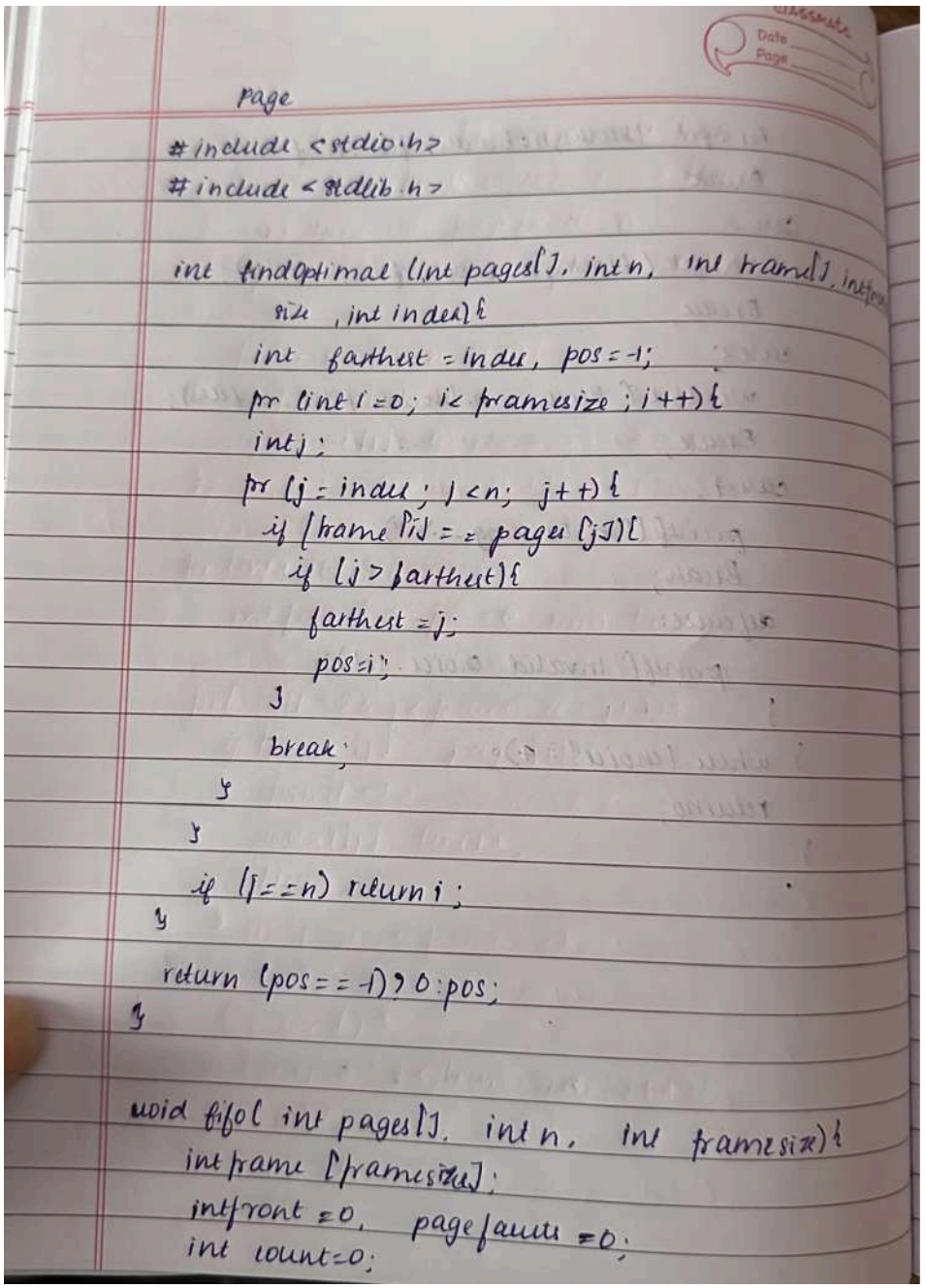
```
file[i].size) {
    if (worstIdx == -1 || block[j].size > block[worstIdx].size)
        worstIdx = j;
    }
  }
}

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

int main() {
    int n_blocks, n_files, choice;
    printf("memory management scheme\n");
    printf("Enter the number of blocks:");
    scanf("%-1.d", &n_blocks);
    printf("Enter the number of files:");
    scanf("%-1.d", &n_files);
```

```
struct block blocks[n_blocks];
struct file files[n_files];

printf("\n Enter the size of the blocks:\n");
for(int i=0; i<n_blocks; i++){
    printf("Block %d: ", i+1);
    scanf("%d", &blocks[i].size);
    block[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("\n 1. First Fit \n 2. Best Fit \n 3. Worst Fit \n4.E
printf("Enter your choice:");
scanf("%d", &choice);

reset Block(blocks, n_blocks);

switch(choice){
    case1:
```

```
        Firstfit (blocks, n_blocks, file, n_files);
        Break;
    case 2:
        Bestfit (block, n_block, files, n_files);
        Break;
    case 3:
        Worstfit ( Blocks, n_blocks, files, n_files);
        Break;
    case 4;
        printf ("\n Existing ..:\n");
        Break;
    default:
        printf(" Invalid choice. \n");
    }
} while (choice != 4);
    return 0;
}
```

Program-10

10. Write a C program to simulate page replacement algorithms
a) FIFO
d) LRU
e) Optimal



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

int findOptimal(int pages[], int n, int frame[], int fra
    size, int index){
    int farthest = index, pos = -1;
    for(int i=0; i< framesize ; i++){
    int j;
    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 fifo( int pages[], int n, int framesize){
    int frame[framesize];
    int front = 0, pagefaults = 0;
    int count=0;
```

```
printf("\n FIFO Page replacement :\n");
for(int i=0; i< framesize; i++) frame[i]=-1;

for (int i=0; i< framesize; i++) frame[i]=-1;
for (int i=0; i<n ; i++){
    int found=0;
    for (int j=0 ; j< framesize ; j++){
        if (frame[j] == pages[i]){
            found=1;
            break;
        }
    }
if (!found) {
    frame [front] = pages[i];
    front = (front+1) % framesize;
    page faults++;
}
printf ("Frames : ");
for (int j=0; j< framesize ; j++){
    if (frame[j] != -1)
        printf ("%d", frame[j]);
    else
        printf ("- ");
}
printf("\n");
}
```

```
void lru ( int pages[], int n, int framesize) {
    int frame[framesize], count[framesize];
    int time = 0, pagefaults = 0;
    printf("\n LRU Page replacement :\n");

    for(int i=0; i<n; i++){
      frame[i] = -1;
      count[i] = 0;
    }

    for(int i = 0; i<n; i++) {
      int found=0, pos = 0, min = time;

      for( int i = 0; j < framesize; j++){
        if (frame[j] == pages[i]) {
          found = 1;
          count[j] = time++;
          break;
        }
      }

      if (!found) {
        for (int j=0; j < framesize; j++){
          if (frame[j] == -1) {
            pos = j;
            break;
          }
```

```
        if(count[j] < min){
            min = count[j];
            pos = j;
        }
    }

    frame[pos] = pages[i];
    count[pos] = time++;
    pagefaults++;
}

printf("Frames:");
for(int j=0; j<framesize; j++){
    if(frame[j] != -1)
        printf("%d", frame[j]);
    else
        printf("-");
}
printf("\n");
}
printf("Total page faults (LRU): %d\n", pagefaults);
}

int main(){
    int n, framesiz, choice;
    printf("Enter number of pages:");
    scanf("%d", &n);
```

```
int pages[n];
printf("enter page reference string:");
for(int i=0; i<n; i++){
    scanf("%d", &pages[i]);
}

do{
    printf("\n -- Pag replacement Algorithms --\n");
    printf("1. FIFO \n2. LRU \n3. optimal \n4. Exit\n");
    printf("Enter your choice ");
    scanf("%d", &choice);
}

switch(choice){
    case1:
        fifo(page, n, framesize);
        break;
    case2;
        lru(page, n framesize);
        break;
    case3;
        optimal(pagu, n, framesize);
        break;
    case4;
        printf("Existing ...\n");
        break;
```

```
default:
    printf("Invalid choice!\n");
  }
} while(choice!=4);
  return 0;
}
```

output:

Enter the size of the pages:

7

Enter the page strings:

1 3 0 3 5 6 3

FIFO Page Faults: 6, Page Hits: 1

Optimal Page Faults: 5, Page Hits: 2

LRU Page faults: 7, Page Hits: 0.