VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

Subramanya J (1BM23CS343)

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 Subramanya J **(1BM23CS343),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. 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.

Megha MJ	**Dr. Kavitha Sooda**
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System
CO2	Analyze various Operating system strategies and techniques
CO3	Demonstrate the different functionalities of Operating System
CO4	Conduct practical experiments to implement the functionalities of Operating system

Program 1

Write - a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. (Any one)

c) FCFS

d) SJF

Code :

==> main_fcfs.c <==

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

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

        arr = create_process_table(pnums);
        get_pid_data(arr, pnums);
        sort_index(arr, pnums, 1);
        calculate_params(arr, pnums);
        display_process_table(arr, pnums);
        free_table(arr, pnums);

        return 0;
}
```

==> main_pr.c <==

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

void calculate_params_priority(int **arr, int pnums) {
  sort_index(arr, pnums, 1);

  int time = 0;
  int completed = 0;
  int *completed_processes = calloc(pnums, sizeof(int));

  while (completed < pnums) {
    int highest_priority = -1;
    int next_process = -1;

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

```
            if (!completed_processes[i] && arr[i][1] <= time) {
                if (highest_priority == -1 || arr[i][7] < highest_priority) {
                    highest_priority = arr[i][7];
                    next_process = i;
                }
            }
        }

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

        completed_processes[next_process] = 1;
        completed++;

        time += arr[next_process][2];
        arr[next_process][3] = time;

        arr[next_process][4] = arr[next_process][3] - arr[next_process][1]; // TAT = CT - AT
        arr[next_process][5] = arr[next_process][4] - arr[next_process][2]; // WT = TAT - BT
        arr[next_process][6] = arr[next_process][5]; // RT = WT (for non-preemptive)
    }

    free(completed_processes);
}

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

	arr = create_process_table(pnums);
	get_pid_data(arr, pnums);
	sort_index(arr, pnums, 1);
	calculate_params_priority(arr, pnums);
	display_process_table(arr, pnums);
	free_table(arr, pnums);

	return 0;
}

==> main_pr_preempt.c <==
```

```c
#include "process.h"
#include <stdio.h>
#include <stdlib.h>

void calculate_params_preemptive_priority(int **arr, int pnums) {
    sort_index(arr, pnums, 1);

    int *remaining_time = malloc(pnums * sizeof(int));
    int *is_started = calloc(pnums, sizeof(int));
    for (int i = 0; i < pnums; i++) {
        remaining_time[i] = arr[i][2];
    }

    int time = 0;
    int completed = 0;

    while (completed < pnums) {
        int highest_priority = -1;
        int next_process = -1;

        for (int i = 0; i < pnums; i++) {
            if (remaining_time[i] > 0 && arr[i][1] <= time) {
                if (highest_priority == -1 || arr[i][7] < highest_priority) {
                    highest_priority = arr[i][7];
                    next_process = i;
                }
            }
        }

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

        if (!is_started[next_process]) {
            arr[next_process][6] = time - arr[next_process][1];
            is_started[next_process] = 1;
        }

        remaining_time[next_process]--;
        time++;

        if (remaining_time[next_process] == 0) {
```

```
            completed++;
            arr[next_process][3] = time; // CT = Current Time
            arr[next_process][4] = arr[next_process][3] - arr[next_process][1]; // TAT = CT - AT
            arr[next_process][5] = arr[next_process][4] - arr[next_process][2]; // WT = TAT - BT
        }
    }

    free(remaining_time);
    free(is_started);
}

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

        arr = create_process_table(pnums);
        get_pid_data(arr, pnums);
        sort_index(arr, pnums, 1);
        calculate_params_preemptive_priority(arr, pnums);
        display_process_table(arr, pnums);
        free_table(arr, pnums);

        return 0;
}

==> main_rr.c <==
#include "process.h"
#include <stdio.h>
#include <stdlib.h>

void calculate_params_rr(int **arr, int pnums, int time_quantum) {
    int *remaining_time = malloc(pnums * sizeof(int));
    int *is_started = calloc(pnums, sizeof(int));
    for (int i = 0; i < pnums; i++) {
        remaining_time[i] = arr[i][2];
    }

    int time = 0;
    int completed = 0;

    while (completed < pnums) {
        int done = 1;
```

```
        for (int i = 0; i < pnums; i++) {
            if (remaining_time[i] > 0 && arr[i][1] <= time) {
                done = 0;

                if (!is_started[i]) {
                    arr[i][6] = time - arr[i][1]; // RT = Start Time - AT
                    is_started[i] = 1;
                }

                if (remaining_time[i] > time_quantum) {
                    time += time_quantum;
                    remaining_time[i] -= time_quantum;
                } else {
                    time += remaining_time[i];
                    arr[i][3] = time; // CT
                    arr[i][4] = arr[i][3] - arr[i][1]; // TAT
                    arr[i][5] = arr[i][4] - arr[i][2]; // WT
                    remaining_time[i] = 0;
                    completed++;
                }
            }
        }

        if (done) {
            time++;
        }
    }

    free(remaining_time);
    free(is_started);
}

int main() {
            int pnums, time_q, **arr;
            printf("Enter the number of processes : ");
            scanf("%d", &pnums);

            printf("Enter the time quantum : ");
            scanf("%d", &time_q);
            arr = create_process_table(pnums);
            get_pid_data(arr, pnums);
            sort_index(arr, pnums, 1);
```

```
        calculate_params_rr(arr, pnums, time_q);
        display_process_table(arr, pnums);
        free_table(arr, pnums);

        return 0;
}

==> main_sjf.c <==
#include "process.h"
#include <stdio.h>
#include <stdlib.h>

void calculate_params_sjf(int **arr, int pnums) {
    sort_index(arr, pnums, 1);

    int time = 0;
    int completed = 0;
    int *completed_processes = calloc(pnums, sizeof(int));

    while (completed < pnums) {
        int shortest_burst = -1;
        int next_process = -1;

        for (int i = 0; i < pnums; i++) {
            if (!completed_processes[i] && arr[i][1] <= time) {
                if (shortest_burst == -1 || arr[i][2] < shortest_burst) {
                    shortest_burst = arr[i][2];
                    next_process = i;
                }
            }
        }

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

        completed_processes[next_process] = 1;
        completed++;

        time += arr[next_process][2];
        arr[next_process][3] = time;
```

```
            arr[next_process][4] = arr[next_process][3] - arr[next_process][1]; // TAT = CT - AT
            arr[next_process][5] = arr[next_process][4] - arr[next_process][2]; // WT = TAT - BT
            arr[next_process][6] = arr[next_process][5]; // RT = WT (for non-preemptive SJF)
    }

    free(completed_processes);
}

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

            arr = create_process_table(pnums);
            get_pid_data(arr, pnums);
            sort_index(arr, pnums, 1);
            calculate_params_sjf(arr, pnums);
            display_process_table(arr, pnums);
            free_table(arr, pnums);

            return 0;
}

==> main_sjf_preempt.c <==
#include "process.h"
#include <stdio.h>
#include <stdlib.h>

void calculate_params_srtf(int **arr, int pnums) {
    int *remaining_time = malloc(pnums * sizeof(int));
    int *is_started = calloc(pnums, sizeof(int));
    for (int i = 0; i < pnums; i++) {
        remaining_time[i] = arr[i][2];
    }

    int time = 0;
    int completed = 0;

    while (completed < pnums) {
        int shortest_remaining = -1;
        int next_process = -1;

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

```
            if (remaining_time[i] > 0 && arr[i][1] <= time) {
                if (shortest_remaining == -1 || remaining_time[i] < shortest_remaining) {
                    shortest_remaining = remaining_time[i];
                    next_process = i;
                }
            }
        }

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

        if (!is_started[next_process]) {
            arr[next_process][6] = time - arr[next_process][1];
            is_started[next_process] = 1;
        }

        remaining_time[next_process]--;
        time++;

        if (remaining_time[next_process] == 0) {
            completed++;
            arr[next_process][3] = time;
            arr[next_process][4] = arr[next_process][3] - arr[next_process][1];
            arr[next_process][5] = arr[next_process][4] - arr[next_process][2];
        }
    }

    free(remaining_time);
    free(is_started);
}

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

    arr = create_process_table(pnums);
    get_pid_data(arr, pnums);
    sort_index(arr, pnums, 1);
    calculate_params_srtf(arr, pnums);
    display_process_table(arr, pnums);
```

```
    free_table(arr, pnums);

    return 0;
}

==> process.c <==
#include <stdio.h>
#include <stdlib.h>

/* When there are pnum processes,
 * Return a 2-D array of dimension
 * pnum x 8
 * */

int **create_process_table(int pnum) {
	int **arr = calloc(pnum, sizeof(int *));
	for(int i = 0; i < pnum; i++) {
		arr[i] = calloc(8, sizeof(int));
	}
	return arr;
}

/* The index to column mapping is :
 * 0 : Process ID
 * 1 : Arrival Time
 * 2 : Burst Time
 * 3 : Completion Time
 * 4 : Turnaround Time
 * 5 : Waiting Time
 * 6 : Response Time
 * 7 : Priority
 * */

void get_pid_data(int **arr, int pnums) {
	int pids = 0;
	for(int i = 0; i < pnums; i++) {
		printf("\nEnter the data for process %d", ++pids);
		arr[i][0] = pids;
		printf("\nArrival Time : ");
		scanf("%d", &arr[i][1]);
		printf("\nBurst Time : ");
		scanf("%d", &arr[i][2]);
		printf("\nPriority :");
```

```
        scanf("%d", &arr[i][7]);
    }
}

void sort_index(int **arr, int pnums, int index) {
    int *ptr;
    for(int i = 0; i < pnums; i++) {
        for(int j = i + 1; j < pnums; j++) {
            if(arr[i][index] > arr[j][index]) {
                ptr = arr[i];
                arr[i] = arr[j];
                arr[j] = ptr;
            }
        }
    }
}

void display_process_table(int **arr, int pnums) {
    printf("\n|PID\t|AT\t|BT\t|CT\t|TAT\t|WT\t|RT\t|Priority\n");
    for(int i = 0; i < pnums; i++) {
        for(int j = 0; j < 8; j++) {
            printf("|%d\t", arr[i][j]);
        }   putchar('\n');
    }   putchar('\n');
}

void calculate_params(int **arr, int pnums) {
    int time = 0;
    for(int i = 0; i < pnums; i++) {
        time = arr[i][1] > time ? arr[i][1] : time;

        /* CT = Time + BT */
        arr[i][3] = time + arr[i][2];

        /* TAT = CT - AT */
        arr[i][4] = arr[i][3] - arr[i][1];

        /* WT = TAT - BT */
        /* FCFS, so RT = WT */
        arr[i][6] = arr[i][5] = arr[i][4] - arr[i][2];

        time = arr[i][3];
    }
```

```
}

void free_table(int **arr, int pnums) {
        for(int i = 0; i < pnums; i++) {
                free(arr[i]);
        }       free(arr);
}
```

Result :

```
First Come First Served :
Enter the number of processes :
Enter the data for process 1
Arrival Time :
Burst Time :
Priority :
Enter the data for process 2
Arrival Time :
Burst Time :
Priority :
Enter the data for process 3
Arrival Time :
Burst Time :
Priority :
Enter the data for process 4
Arrival Time :
Burst Time :
Priority :
|PID    |AT     |BT     |CT     |TAT    |WT     |RT     |Priority
|1      |0      |7      |7      |7      |0      |0      |2
|4      |2      |6      |13     |11     |5      |5      |3
|3      |3      |4      |17     |14     |10     |10     |1
|2      |8      |10     |27     |19     |9      |9      |4
```

```
Shortest Job First :
Enter the number of processes :
Enter the data for process 1
Arrival Time :
Burst Time :
Priority :
Enter the data for process 2
Arrival Time :
Burst Time :
Priority :
Enter the data for process 3
Arrival Time :
Burst Time :
Priority :
Enter the data for process 4
Arrival Time :
Burst Time :
Priority :
|PID    |AT     |BT     |CT     |TAT    |WT     |RT     |Priority
|1      |0      |7      |7      |7      |0      |0      |2
|4      |2      |6      |17     |15     |9      |9      |3
|3      |3      |4      |11     |8      |4      |4      |1
|2      |8      |10     |27     |19     |9      |9      |4
```

Program 2

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

Code :

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

struct Process {
    int PID, AT, BT;
    int CT, TAT, WT;
    int PRI, RT;
};

struct Process *get_processes(int *n) {
    printf("Enter number of processes: ");
    scanf("%d", n);

    struct Process *parr = calloc(*n, sizeof(struct Process));
    for (int i = 0; i < *n; i++) {
        parr[i].PID = i + 1;
        printf("Enter Arrival Time, Burst Time, and Priority (0 for System, 1 for User) for Process %d: ", i + 1);
        scanf("%d %d %d", &parr[i].AT, &parr[i].BT, &parr[i].PRI);
        parr[i].RT = parr[i].BT;
    }
    return parr;
}

int ar_sort(const void *x, const void *y) {
    return ((struct Process *)x)->AT - ((struct Process *)y)->AT;
}

void classify_processes(struct Process *parr, int n, struct Queue **sys, struct Queue **usr) {
    *sys = create_queue(n);
    *usr = create_queue(n);

    for(int i = 0; i < n; i++) {
        if(parr[i].PRI == 0) {
            enqueue_queue(*sys, &parr[i]);
        } else {
            enqueue_queue(*usr, &parr[i]);
        }
    }
}
```

```
void fill_data(struct Process *p, int *time) {
    p->WT = *time - p->AT;
    *time += p->BT;
    p->CT = *time;
    p->TAT = p->CT - p->AT;
    p->RT = 0;
}

void scheduler_fcfs(struct Queue *sys, struct Queue *usr, int n) {
    int time = 0;
    struct Process *current = NULL;

    while(!isempty_queue(sys) || !isempty_queue(usr)) {
        while(!isempty_queue(sys)) {
            struct Process *sys_proc = (struct Process *)frontof_queue(sys);
            if(sys_proc->AT <= time) {
                current = dequeue_queue(sys);
                fill_data(current, &time);
                break;
            } else {
                break;
            }
        }

        if(current == NULL && !isempty_queue(usr)) {
            struct Process *usr_proc = (struct Process *)frontof_queue(usr);
            if(usr_proc->AT <= time) {
                current = dequeue_queue(usr);
                fill_data(current, &time);
            } else {
                time++;
            }
        }

        current = NULL;
    }
}

void display_process_table(struct Process *p_list, int n) {
    float avg_TAT = 0, avg_WT = 0;

    printf("\nPID\tAT\tBT\tPRI\tCT\tTAT\tWT\n");
    printf("------------------------------------------------\n");
    for(int i = 0; i < n; i++) {
        struct Process p = p_list[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
               p.PID, p.AT, p.BT, p.PRI, p.CT, p.TAT, p.WT);
        avg_TAT += p.TAT;
        avg_WT += p.WT;
```

```
    }

    avg_TAT /= n;
    avg_WT /= n;
    printf("\nAverage Turn Around Time: %.2f\n", avg_TAT);
    printf("Average Waiting Time: %.2f\n", avg_WT);
}

int main() {
    int n;
    struct Process *p_list = get_processes(&n);
    struct Queue *sys, *usr;

    qsort(p_list, n, sizeof(struct Process), ar_sort);

    classify_processes(p_list, n, &sys, &usr);
    scheduler_fcfs(sys, usr, n);
    display_process_table(p_list, n);

    free_queue(sys);
    free_queue(usr);
    free(p_list);

    return 0;
}
```

Result :

```
~/notes/ops/lab/02_multi_level_queue (main*) » ./main_multilevel.out          130 ↵ sub@deb
Enter number of processes: 3
Enter Arrival Time, Burst Time, and Priority (0 for System, 1 for User) for Process 1: 0 2 0
Enter Arrival Time, Burst Time, and Priority (0 for System, 1 for User) for Process 2: 4 5 1
Enter Arrival Time, Burst Time, and Priority (0 for System, 1 for User) for Process 3: 0 1 1

PID     AT      BT      PRI     CT      TAT     WT
------------------------------------------------
1       0       2       0       2       2       0
3       0       1       1       3       3       2
2       4       5       1       9       5       0

Average Turn Around Time: 3.33
Average Waiting Time: 0.67
```

Program 3
Write a C program to simulate Real-Time CPU Scheduling algorithms

a) Rate- Monotonic

Code :

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

#define PCAST (struct Process *)

int
proccmp(const void *x, const void *y) {
	return (PCAST x)->ival - (PCAST y)->ival;
}

void
monotonic_scheduler(struct Process *arr, int n) {
	int max_time = arr[n-1].ival * 2;
	int time = 0;
	struct Process *curr = NULL, *prev = NULL;

	while(time < max_time) {

		for(int i = 0; i < n; i++) {
			if(arr[i].next_release == time) {
				arr[i].rtr = true;
				arr[i].next_release += arr[i].ival;
				arr[i].rem_bt = arr[i].bt;
			}
		}

		curr = NULL;
		for(int i = 0; i < n; i++) {
			if(arr[i].rtr == true && arr[i].rem_bt > 0) {
				curr = &arr[i];
				break;
			}
		}

		if(prev != curr) {
			if (prev != NULL && prev->rem_bt > 0)
				printf("PID %d till %d\n", prev->PID, time);
			if (curr != NULL)
				printf("PID %d starts at %d\n", curr->PID, time);
			prev = curr;
		}
```

```
			if(!curr) {time++; continue;}
			curr->rem_bt--;
			if(curr->rem_bt == 0) { curr->rtr = false; }
			time++;

		}
		return;
}

int
main() {
		int n;
		struct Process *arr = get_processes(&n);
		qsort(arr, n, sizeof(struct Process), proccmp);
		putchar('\n');

		monotonic_scheduler(arr, n);

		return 0;
}
```

Result :

```
~/notes/ops/lab/03_real_time_scheduling (main*) » make

Rate Monotonic Scheduler :

Enter the number of processes : Enter the burst time and intervals :
PID 1 : PID 2 : PID 3 : PID 4 : PID 5 :
PID 4 starts at 0
PID 1 starts at 5
PID 5 starts at 25
PID 5 till 30
PID 4 starts at 30
PID 5 starts at 35
PID 2 starts at 40
PID 2 till 60
PID 4 starts at 60
PID 2 starts at 65
PID 3 starts at 75
PID 3 till 80
PID 1 starts at 80
PID 1 till 90
PID 4 starts at 90
PID 1 starts at 95
PID 5 starts at 105
PID 2 starts at 115
PID 2 till 120
PID 4 starts at 120
PID 2 starts at 125
PID 4 starts at 150
PID 3 starts at 155
PID 3 till 160
PID 1 starts at 160
PID 4 starts at 180
PID 5 starts at 185
PID 3 starts at 195
PID 2 starts at 200
PID 2 till 210
```

Program

Write a C program to simulate:

c) Producer-Consumer problem using semaphores.

d) Dining-Philosopher's problem

Code :

==> main_dp.c <==

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define N 5


sem_t chop[N];


void *philosopher(void *num) {

    int id = *(int *)num;

    while (1) {

        printf("Philosopher %d is thinking.\n", id);

        sleep(1);

        sem_wait(&chop[id]);

        sem_wait(&chop[(id + 1) % N]);

        printf("Philosopher %d is eating.\n", id);

        sleep(1);

        sem_post(&chop[id]);

        sem_post(&chop[(id + 1) % N]);

        printf("Philosopher %d finished eating.\n", id);

        sleep(1);

    }

}


int main() {
```

```
    pthread_t phil[N];
    int i, ids[N];
    for (i = 0; i < N; i++) sem_init(&chop[i], 0, 1);
    for (i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&phil[i], NULL, philosopher, &ids[i]);
    }
    for (i = 0; i < N; i++) pthread_join(phil[i], NULL);
    for (i = 0; i < N; i++) sem_destroy(&chop[i]);
    return 0;
}

==> main_pc.c <==
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t empty, full, mutex;

void *producer(void *arg) {
    int item;
    while (1) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
```

```
        printf("Produced: %d at %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
}

void *consumer(void *arg) {
    int item;
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        printf("Consumed: %d from %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&empty);
        sleep(1);
    }
}

int main() {
    pthread_t prod, cons;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    sem_destroy(&empty);
```

```
    sem_destroy(&full);
    sem_destroy(&mutex);
    return 0;
}
```

Result :

```
~/notes/ops/lab/04_semaphores (main*) » ./a.out
Philosopher 1 is thinking.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is eating.
Philosopher 3 is eating.
Philosopher 1 finished eating.
Philosopher 0 is eating.
Philosopher 3 finished eating.
Philosopher 2 is eating.
Philosopher 1 is thinking.
Philosopher 0 finished eating.
Philosopher 4 is eating.
Philosopher 3 is thinking.
Philosopher 2 finished eating.
Philosopher 1 is eating.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is eating.
Philosopher 4 finished eating.
^C
```

```
~/notes/ops/lab/04_semaphores (main*) » ./main_pc
Produced: 83 at 0
Consumed: 83 from 0
Produced: 86 at 1
Consumed: 86 from 1
Produced: 77 at 2
Consumed: 77 from 2
Produced: 15 at 3
Consumed: 15 from 3
Produced: 93 at 4
Consumed: 93 from 4
Produced: 35 at 0
Consumed: 35 from 0
^C
```

Program

Write a C program to simulate:

a) Bankers' algorithm for the purpose of deadlock avoidance.

```
#include <stdio.h>

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

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

    printf("Enter the maximum demand matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allo[i][j];
        }

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

    int finished[n], safeSequence[n], count = 0;
    for (i = 0; i < n; i++) finished[i] = 0;

    while (count < n) {
        int found = 0;
        for (i = 0; i < n; i++) {
            if (!finished[i]) {
```

```
                for (j = 0; j < m; j++)
                    if (need[i][j] > avail[j]) break;
                if (j == m) {
                    for (k = 0; k < m; k++)
                        avail[k] += allo[i][k];
                    safeSequence[count++] = i;
                    finished[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            printf("System is in an unsafe state.\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence: ");
    for (i = 0; i < n; i++) printf("%d ", safeSequence[i]);
    printf("\n");
    return 0;
}
```

Result :

```
~/notes/ops/lab/05_deadlock (main*) » ./main_banker
Enter the number of processes and resources : 2 2
Enter the allo matrix:
2
2
2
2
Enter the maximum demand matrix:
3
3
3
3
Enter the avail resources:
0
0
System is in an unsafe state.
```

Program :

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

d) Worst-fit

e) Best-fit

f) First-fit

Code:

==> main_best.c <==

```
#include <stdio.h>
#include "main_common.c"

void bestFit(int blockSize[], int m, int processSize[], int n) {
   int allocation[n];

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

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

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

        if (bestIdx != -1) {
               allocation[i] = bestIdx;
               blockSize[bestIdx] -= processSize[i];
        }
```

```
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1,
blockSize[allocation[i]]);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    COMMON_MAIN()
    bestFit(blockSize, m, processSize, n);

    return 0;
}



==> main_common.c <==
#define COMMON_MAIN() \
    int m, n; \
    printf("Enter the number of blocks: "); \
    scanf("%d", &m); \
    printf("Enter the number of processes: "); \
    scanf("%d", &n); \
    int blockSize[m], processSize[n]; \
    printf("\nEnter the sizes of the blocks:\n"); \
    for (int i = 0; i < m; i++) { \
        printf("Block %d size: ", i + 1); \
        scanf("%d", &blockSize[i]); \
```

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

==> main_first.c <==
#include <stdio.h>
#include "main_common.c"

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];

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

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

    printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1,
blockSize[allocation[i]]);
```

```
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    COMMON_MAIN()
    firstFit(blockSize, m, processSize, n);

    return 0;
}



==> main_worst.c <==
#include <stdio.h>
#include "main_common.c"

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];

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

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

        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
                    worstIdx = j;
```

```
                }
            }
        }

        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1,
blockSize[allocation[i]]);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    COMMON_MAIN()
    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

Result :

```
gcc main_best.c -o main_best
cat data.txt | ./main_best
Enter the number of blocks: Enter the number of processes:
Enter the sizes of the blocks:
Block 1 size: Block 2 size: Block 3 size: Block 4 size:
Enter the sizes of the processes:
Process 1 size: Process 2 size: Process 3 size: Process 4 size:
Process No.     Process Size    Block No.       Block Size
1               8               1               2
2               20              3               0
3               5               2               5
4               16              4               14
```

```
gcc main_first.c -o main_first
cat data.txt | ./main_first
Enter the number of blocks: Enter the number of processes:
Enter the sizes of the blocks:
Block 1 size: Block 2 size: Block 3 size: Block 4 size:
Enter the sizes of the processes:
Process 1 size: Process 2 size: Process 3 size: Process 4 size:
Process No.     Process Size    Block No.       Block Size
1               8               1               2
2               20              3               0
3               5               2               5
4               16              4               14
```

```
~/notes/ops/lab/06_memory_allocation (main*) » make
gcc main_worst.c -o main_worst
cat data.txt | ./main_worst
Enter the number of blocks: Enter the number of processes:
Enter the sizes of the blocks:
Block 1 size: Block 2 size: Block 3 size: Block 4 size:
Enter the sizes of the processes:
Process 1 size: Process 2 size: Process 3 size: Process 4 size:
Process No.     Process Size    Block No.       Block Size
1               8               4               2
2               20              4               2
3               5               3               15
4               16              Not Allocated
```

Program
Write a C program to simulate page replacement algorithms.

d) FIFO

e) LRU

f) Optimal

Code :

```
==> main_fifo.c <==
#include "page_replacement.h"

int main() {
    int frames, pages, pageFaults = 0, current = 0;
    int referenceString[100], frame[100]; // Max size to prevent runtime issues

    GET_INPUT(frames, pages, referenceString);

    // Initialize frames
    for (int i = 0; i < frames; i++) {
        frame[i] = -1;
    }

    for (int i = 0; i < pages; i++) {
        int found = 0;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == referenceString[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            frame[current] = referenceString[i];
            current = (current + 1) % frames;
            pageFaults++;
        }
```

```
        DISPLAY_FRAMES(referenceString[i], frames, frame);
    }

    PRINT_PAGE_FAULTS(pageFaults);
    return 0;
}

==> main_lru.c <==
#include "page_replacement.h"

int main() {
    int frames, pages, pageFaults = 0;
    int referenceString[100], frame[100]; // Max size to prevent runtime issues

    GET_INPUT(frames, pages, referenceString);

    // Initialize frames
    for (int i = 0; i < frames; i++) {
        frame[i] = -1;
    }

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

        if (!found) {
                int max = -1, index = -1;
```

```
                for (int j = 0; j < frames; j++) {
                    if (frame[j] == -1) { index = j; break; }

                    int next = 0;
                    for (int k = i + 1; k < pages; k++) {
                        if (frame[j] == referenceString[k]) { next = k; break; }
                    }

                    if (next == 0) { index = j; break; }
                    if (next > max) { max = next; index = j; }
                }

                frame[index] = referenceString[i];
                pageFaults++;
            }

            DISPLAY_FRAMES(referenceString[i], frames, frame);
        }

    PRINT_PAGE_FAULTS(pageFaults);
    return 0;
}

==> main_optimal.c <==
#include "page_replacement.h"

int main() {
    int frames, pages, pageFaults = 0;
    int referenceString[100], frame[100];

    GET_INPUT(frames, pages, referenceString);
```

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

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

    if (!found) {
        int max = -1, index = -1;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == -1) { index = j; break; }

            int next = 0;
            for (int k = i + 1; k < pages; k++) {
                if (frame[j] == referenceString[k]) { next = k; break; }
            }

            if (next == 0) { index = j; break; }
            if (next > max) { max = next; index = j; }
        }

        frame[index] = referenceString[i];
        pageFaults++;
    }

    DISPLAY_FRAMES(referenceString[i], frames, frame);
```

```
}

    PRINT_PAGE_FAULTS(pageFaults);
    return 0;
}
```

Result :

```
gcc main_fifo.c -o main_fifo
cat ./data.txt | ./main_fifo
Enter number of frames: Enter number of pages: Enter reference string: 1: 1 -1 -1
2: 1 2 -1
3: 1 2 3
4: 4 2 3
1: 4 1 3
Total Page Faults: 5
gcc main_lru.c -o main_lru
```

```
gcc main_lru.c -o main_lru
cat ./data.txt | ./main_lru
Enter number of frames: Enter number of pages: Enter reference string: 1: 1 -1 -1
2: 1 2 -1
3: 1 3 -1
4: 1 4 -1
1: 1 4 -1
Total Page Faults: 4
gcc main_optimal.c -o main_optimal
```

```
gcc main_optimal.c -o main_optimal
cat ./data.txt | ./main_optimal
Enter number of frames: Enter number of pages: Enter reference string: 1: 1 -1 -1
2: 1 2 -1
3: 1 3 -1
4: 1 4 -1
1: 1 4 -1
Total Page Faults: 4
```

Program :

Write a C program to simulate the following file allocation strategies.

a) Sequential

b) Indexed

c) Linked

Code :