End Term Project
On
Design Principles of Operating System
(CSE 3249)

Submitted by

Name	**: Dinanath Dash**
Reg. No.	**: 2241004161**
Semester	**: 5th**
Branch	**: CSE**
Section	**: 2241026**
Session	**: 2024-2025**
Admission Batch	**: 2022**

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FACULTY OF ENGINEERING & TECHNOLOGY (ITER)

SIKSHA 'O' ANUSANDHAN DEEMED TO BE UNIVERSITY

BHUBANESWAR, ODISHA – 751030

Objective of this Assignment:

- To design a CPU scheduler for simulating a few CPU scheduling policies.
- Implementation of Banker's algorithm to avoid deadlock.

Overview of the Project:

1. One of the main tasks of an operating system is scheduling processes to run on the CPU. The goalof this programming project is to build a program (*use C or Java programming language*) to implement a simulator with different scheduling algorithms discussed in theory. The simulator should select a process to run from the ready queue based on the scheduling algorithm chosen at runtime. Since the assignment intends to simulate a CPU scheduler, it does not require any actual process creation or execution.
2. The goal of this programming project is to build a program (*use C or Java programming language*) to implement banker's algorithms discussed in theory. Create 5 process that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks.

Project Description 1: The C program provides an interface to the user to implement the following scheduling policies as per the choice provided:

1. First Come First Served (FCFS)
2. Round Robin (RR)

Appropriate option needs to be chosen from a switch case-based menu driven program with an option of "Exit from program" in case 5 and accordingly a scheduling policy will print the Gantt chart and the average waiting time, average turnaround time and average response time. The program will take Process ids, its arrival time, and its CPU burst time as input. For implementing RR scheduling, user also needs to specify the time quantum. Assume that the process ids should be unique for all processes. Each process consists of a single CPU burst (no I/O bursts), and processes are listed in order of their arrival time. Further assume that an interrupted process gets placed at the back of the Ready queue, and a newly arrived process gets placed at the back of the Ready queue as well. The output should be displayed in a formatted way for clarity of understanding and visual.

Test Cases:

The program should able to produce correct answer or appropriate error message corresponding to the following test cases:

1. Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and time quantum = 4ms as shown below.

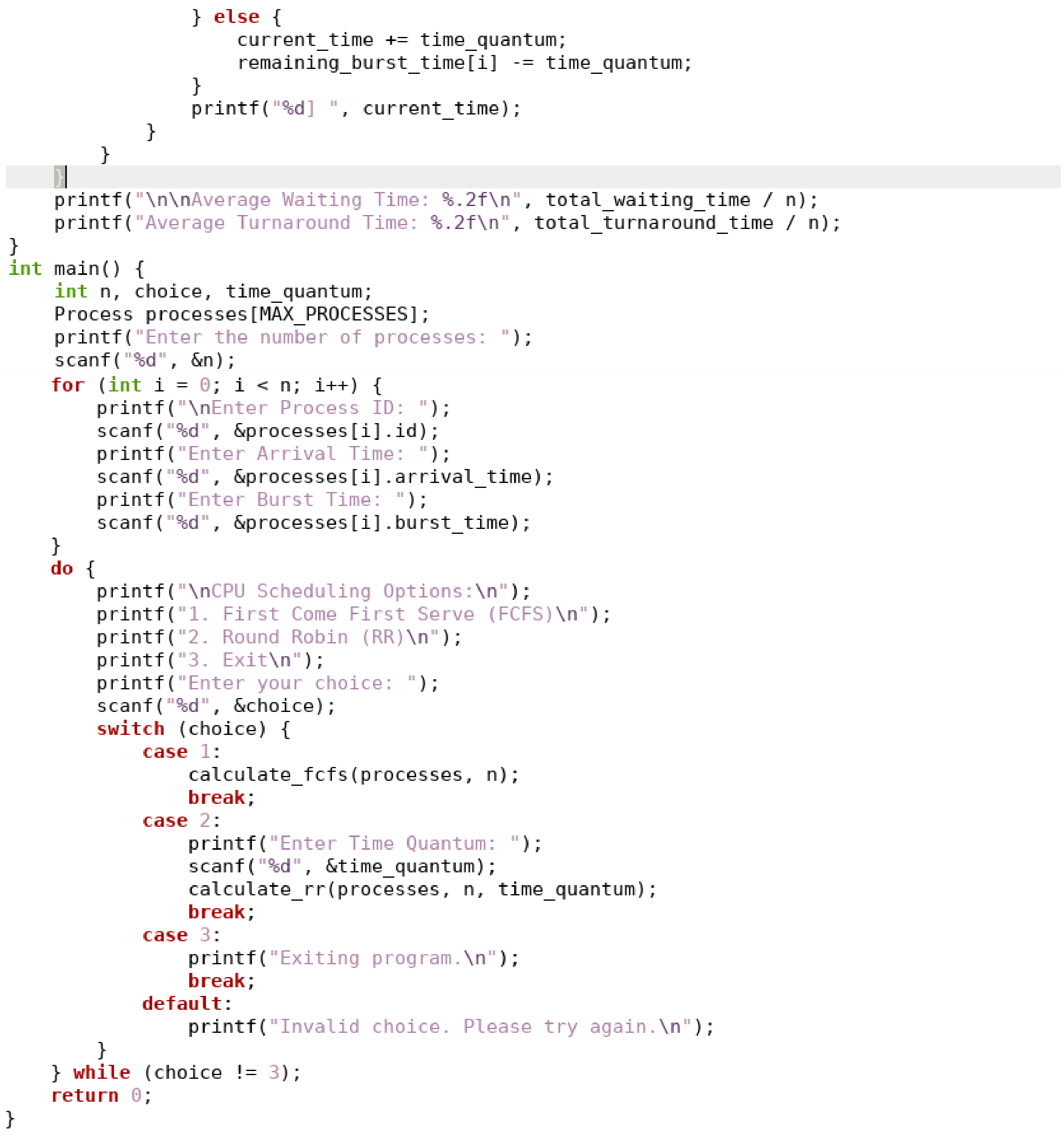
Process	Arrival time	Burst Time
P1	0	10
P2	1	1
P3	2	2
P4	3	1

P5	6	5

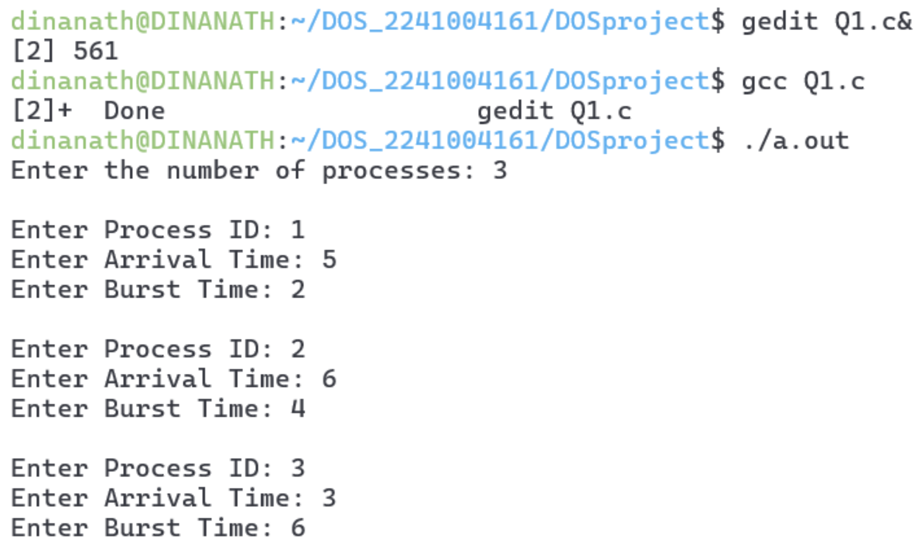
- Input choice 1, and print the Gantt charts that illustrate the execution of these processes using the FCFS scheduling algorithm and then print the average turnaround time, average waiting time and average response time.
- Input choice 2, and print the Gantt charts that illustrate the execution of these processes using the RR scheduling algorithm and then print the average turnaround time, average waiting time and average response time.
- Analyze the results and determine which of the algorithms results in the minimum average waiting time over all processes?

Output-

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESSES 100
typedef struct {
    int id;
    int arrival_time;
    int burst_time;
    int waiting_time;
    int turnaround_time;
    int response_time;
} Process;
void calculate_fcfs(Process processes[], int n) {
    int current_time = 0;
    float total_waiting_time = 0, total_turnaround_time = 0;
    printf("\nGantt Chart (FCFS):\n");
    for (int i = 0; i < n; i++) {
        if (current_time < processes[i].arrival_time) {
            current_time = processes[i].arrival_time;
        }
        processes[i].response_time = current_time - processes[i].arrival_time;
        processes[i].waiting_time = current_time - processes[i].arrival_time;
        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
        printf("P%d [%d - %d] ", processes[i].id, current_time, current_time +
processes[i].burst_time);
        current_time += processes[i].burst_time;
    }
    printf("\n\nAverage Waiting Time: %.2f\n", total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
}
void calculate_rr(Process processes[], int n, int time_quantum) {
    int current_time = 0, completed = 0;
    int remaining_burst_time[MAX_PROCESSES];
    float total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        remaining_burst_time[i] = processes[i].burst_time;
    }
    printf("\nGantt Chart (RR):\n");
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_burst_time[i] > 0 && processes[i].arrival_time <= current_time) {
                printf("P%d [%d - ", processes[i].id, current_time);
                if (remaining_burst_time[i] <= time_quantum) {
                    current_time += remaining_burst_time[i];
                    remaining_burst_time[i] = 0;
                    completed++;
                    processes[i].turnaround_time = current_time - processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
                    total_waiting_time += processes[i].waiting_time;
                    total_turnaround_time += processes[i].turnaround_time;
```



```c
                } else {
                    current_time += time_quantum;
                    remaining_burst_time[i] -= time_quantum;
                }
                printf("%d] ", current_time);
            }
        }
    }
    printf("\n\nAverage Waiting Time: %.2f\n", total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
}
int main() {
    int n, choice, time_quantum;
    Process processes[MAX_PROCESSES];
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("\nEnter Process ID: ");
        scanf("%d", &processes[i].id);
        printf("Enter Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Enter Burst Time: ");
        scanf("%d", &processes[i].burst_time);
    }
    do {
        printf("\nCPU Scheduling Options:\n");
        printf("1. First Come First Serve (FCFS)\n");
        printf("2. Round Robin (RR)\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                calculate_fcfs(processes, n);
                break;
            case 2:
                printf("Enter Time Quantum: ");
                scanf("%d", &time_quantum);
                calculate_rr(processes, n, time_quantum);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);
    return 0;
}
```

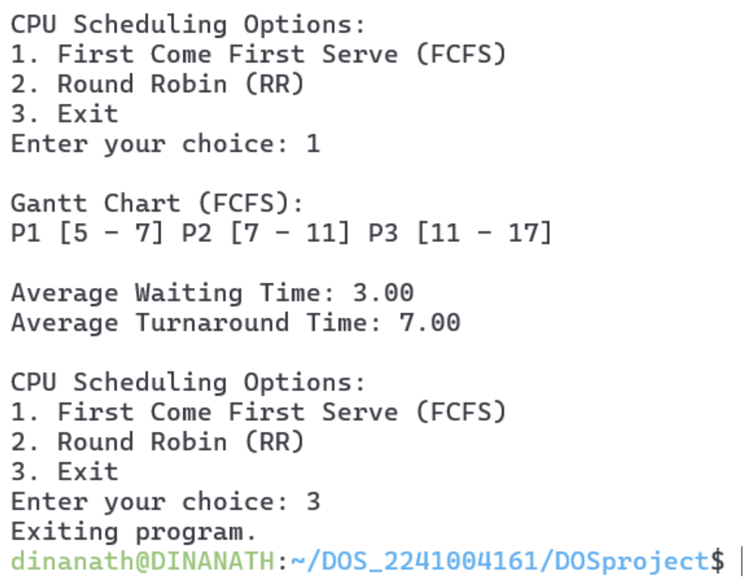


```
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ gedit Q1.c&
[2] 561
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ gcc Q1.c
[2]+  Done                    gedit Q1.c
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ ./a.out
Enter the number of processes: 3

Enter Process ID: 1
Enter Arrival Time: 5
Enter Burst Time: 2

Enter Process ID: 2
Enter Arrival Time: 6
Enter Burst Time: 4

Enter Process ID: 3
Enter Arrival Time: 3
Enter Burst Time: 6
```



```
CPU Scheduling Options:
1. First Come First Serve (FCFS)
2. Round Robin (RR)
3. Exit
Enter your choice: 1

Gantt Chart (FCFS):
P1 [5 - 7] P2 [7 - 11] P3 [11 - 17]

Average Waiting Time: 3.00
Average Turnaround Time: 7.00

CPU Scheduling Options:
1. First Come First Serve (FCFS)
2. Round Robin (RR)
3. Exit
Enter your choice: 3
Exiting program.
dinanath@DINANATH:~/DOS_2241004161/DOSproject$
```

Project Description 2:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Example: Snapshot at the initial stage:

1. Consider the following resource allocation state with 5 processes and 4 resources: There are total existing resources of 6 instances of type R1, 7 instances of type R2, 12 instance of type R3 and 12 instances of type R4.

Process	**Allocation**				**Max**			
	R1	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	0	0	1	2	0	0	1	2
P2	2	0	0	0	2	7	5	0
P3	0	0	3	4	6	6	5	6
P4	2	3	5	4	4	3	5	6
P5	0	3	3	2	0	6	5	2

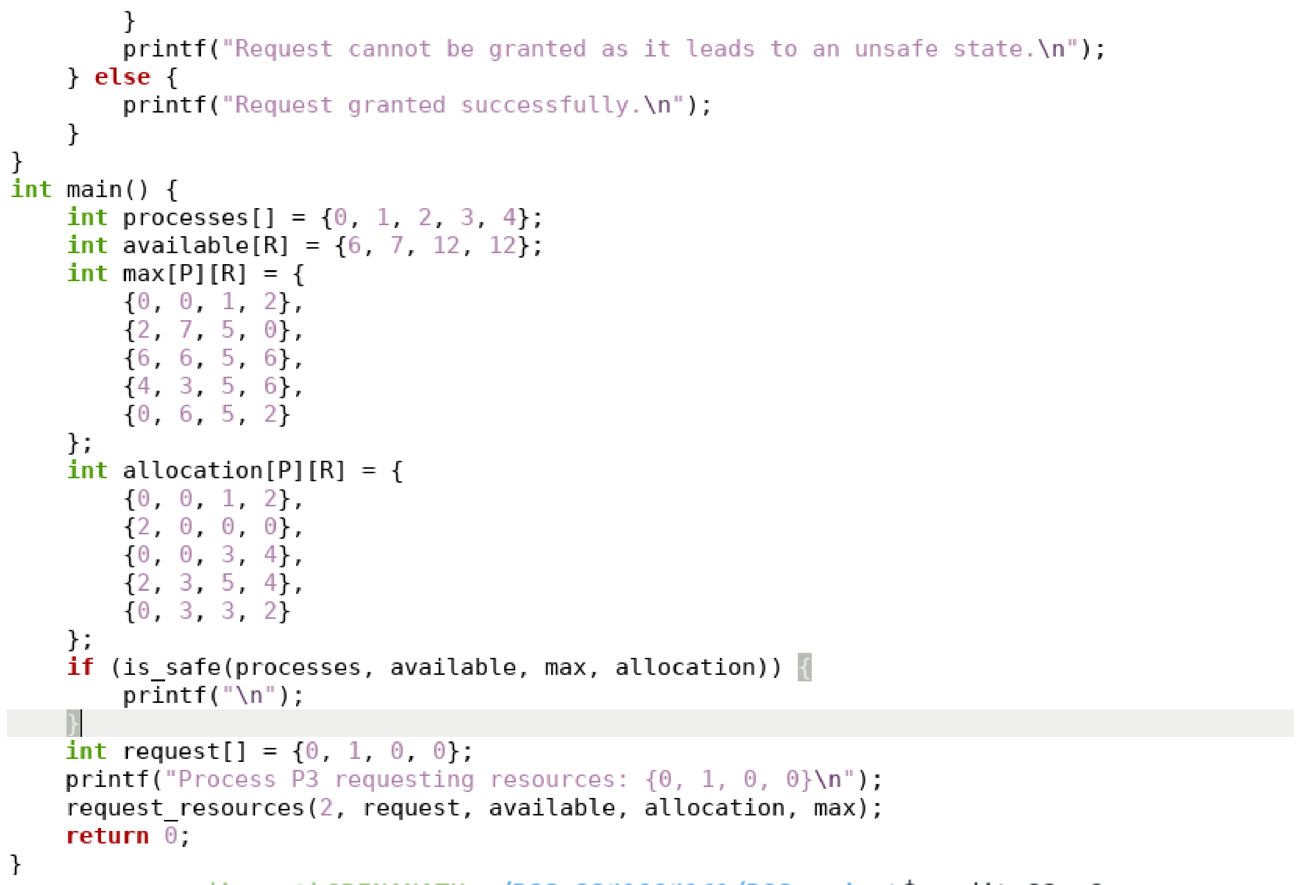
a) Find the content of the need matrix.

b) Is the system in a safe state? If so, give a safe sequence of the process.

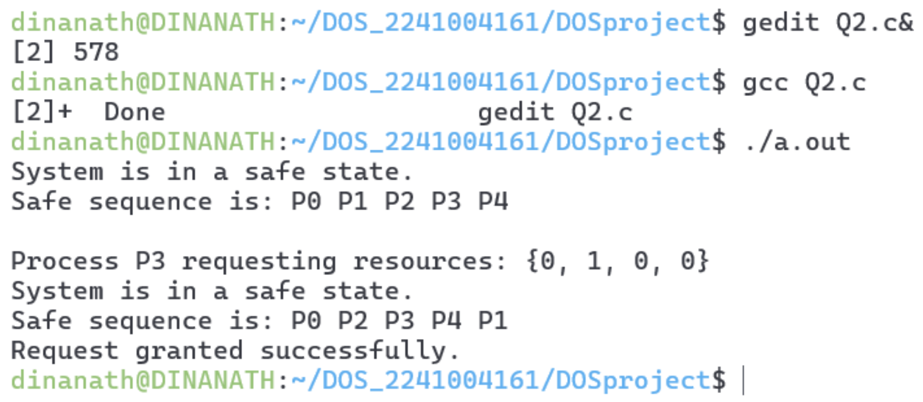
If P3 will request for 1 more instances of type R2, Can the request be granted immediately or not?

Output –

```c
#include <stdio.h>
#include <stdbool.h>
#define P 5
#define R 4
void calculate_need(int need[P][R], int max[P][R], int allocation[P][R]) {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
bool is_safe(int processes[], int available[], int max[P][R], int allocation[P][R]) {
    int need[P][R];
    calculate_need(need, max, allocation);
    bool finish[P] = {false};
    int safe_sequence[P];
    int work[R];
    for (int i = 0; i < R; i++) {
        work[i] = available[i];
    }
    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (!finish[p]) {
                bool can_allocate = true;
                for (int j = 0; j < R; j++) {
                    if (need[p][j] > work[j]) {
                        can_allocate = false;
                        break;
                    }
                }
                if (can_allocate) {
                    for (int j = 0; j < R; j++) {
                        work[j] += allocation[p][j];
                    }
                    safe_sequence[count++] = processes[p];
                    finish[p] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return false;
        }
    }
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++) {
        printf("P%d ", safe_sequence[i]);
    }
    printf("\n");
    return true;
}
void request_resources(int process_id, int request[R], int available[R], int allocation[P]
[R], int max[P][R]) {
    int need[P][R];
    calculate_need(need, max, allocation);
    for (int j = 0; j < R; j++) {
        if (request[j] > need[process_id][j]) {
            printf("Error: Process has exceeded its maximum claim.\n");
            return;
        }
        if (request[j] > available[j]) {
            printf("Error: Resources are not available.\n");
            return;
        }
    }
    for (int j = 0; j < R; j++) {
        available[j] -= request[j];
        allocation[process_id][j] += request[j];
    }
    if (!is_safe((int[]){0, 1, 2, 3, 4}, available, max, allocation)) {
        for (int j = 0; j < R; j++) {
            available[j] += request[j];
            allocation[process_id][j] -= request[j];
```



```c
        }
        printf("Request cannot be granted as it leads to an unsafe state.\n");
    } else {
        printf("Request granted successfully.\n");
    }
}
int main() {
    int processes[] = {0, 1, 2, 3, 4};
    int available[R] = {6, 7, 12, 12};
    int max[P][R] = {
        {0, 0, 1, 2},
        {2, 7, 5, 0},
        {6, 6, 5, 6},
        {4, 3, 5, 6},
        {0, 6, 5, 2}
    };
    int allocation[P][R] = {
        {0, 0, 1, 2},
        {2, 0, 0, 0},
        {0, 0, 3, 4},
        {2, 3, 5, 4},
        {0, 3, 3, 2}
    };
    if (is_safe(processes, available, max, allocation)) {
        printf("\n");
    }
    int request[] = {0, 1, 0, 0};
    printf("Process P3 requesting resources: {0, 1, 0, 0}\n");
    request_resources(2, request, available, allocation, max);
    return 0;
}
```



```
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ gedit Q2.c&
[2] 578
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ gcc Q2.c
[2]+  Done                    gedit Q2.c
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ ./a.out
System is in a safe state.
Safe sequence is: P0 P1 P2 P3 P4

Process P3 requesting resources: {0, 1, 0, 0}
System is in a safe state.
Safe sequence is: P0 P2 P3 P4 P1
Request granted successfully.
dinanath@DINANATH:~/DOS_2241004161/DOSproject$ 
```