End Term Project On

Design Principles of Operating System (CSE 3249)

Submitted by

Name : Koushik Das

Reg. No. : 2341004117(14)

Branch : CSIT

Semester : 5th

Section : 23413B2

Session : 2025-2026

Admission Batch : 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING FACULTY OF ENGINEERING & TECHNOLOGY (ITER) SIKSHA 'O' ANUSANDHAN DEEMED TO BE UNIVERSITY BHUBANESWAR, ODISHA – 751030

End term project
Subject: Design Principles of Operating Systems
Subject code: CSE 3249

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 goal of 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 = 2ms as shown below.

Process	Arrival time	Burst Time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2

   - 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?

```
koushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$ cat > Q1.c
// Objective: To design a CPU scheduler for simulating a few CPU Scheduling policies.
#include <stdio.h>

typedef struct{
    int pid;
    int arrival;
    int burst;
    int remaining;
    int start;
    int completion;
    int waiting;
    int turnaround;
    int response;
    int started;
} Process;


// ==================== FCFS ========================
void fcfs(Process p[], int n){
    int time = 0;
    float awt = 0, att = 0, art = 0;

    printf("\nGantt Chart:\n|");

    for(int i = 0 ; i < n ; i++){
        if(time < p[i].arrival)      // If CPU idle before arrival
            time = p[i].arrival;

        p[i].start = time;
        p[i].response = p[i].start - p[i].arrival;
```

```
        time += p[i].burst;          // Execute process fully

        p[i].completion = time;
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;

        printf("P%d|", p[i].pid);

        awt += p[i].waiting;
        att += p[i].turnaround;
        art += p[i].response;
    }

    printf("\nAverage Waiting Time = %.2f", awt/n);
    printf("\nAverage Turnaround Time = %.2f", att/n);
    printf("\nAverage Response Time = %.2f\n", art/n);
}


// ================= ROUND ROBIN =====================
void roundRobin(Process p[], int n, int tq){
    int time = 0, completed = 0;
    float awt = 0, att = 0, art = 0;

    printf("\nGantt Chart:\n|");

    while(completed < n){
        int executed = 0;     // to avoid infinite loop

        for(int i = 0 ; i < n ; i++){
            if(p[i].arrival <= time && p[i].remaining > 0){


                executed = 1;

                if(!p[i].started){
                    p[i].start = time;
                    p[i].response = p[i].start - p[i].arrival;
                    p[i].started = 1;
                }

                printf("P%d|", p[i].pid);

                if(p[i].remaining > tq){
                    time += tq;
                    p[i].remaining -= tq;
                } else {
                    time += p[i].remaining;
                    p[i].remaining = 0;

                    p[i].completion = time;
                    p[i].turnaround = p[i].completion - p[i].arrival;
                    p[i].waiting = p[i].turnaround - p[i].burst;

                    completed++;
                    awt += p[i].waiting;
                    att += p[i].turnaround;
                    art += p[i].response;
                }
            }
        }

        if(!executed) time++;  // No process ready → time moves 1 unit
    }

    printf("\nAverage Waiting Time = %.2f", awt/n);
    printf("\nAverage Turnaround Time = %.2f", att/n);
    printf("\nAverage Response Time = %.2f\n", art/n);
}



// ====================== MAIN ============================
int main(){
    int n = 5, choice, tq;

    Process p[5] = {
        {1,0,3,3,0,0,0,0,0,0},
        {2,2,6,6,0,0,0,0,0,0},
        {3,4,4,4,0,0,0,0,0,0},
        {4,6,5,5,0,0,0,0,0,0},
        {5,8,2,2,0,0,0,0,0,0}
    };

    while(1){
        printf("\n1. FCFS\n2. Round Robin\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        if(choice == 1){
            fcfs(p, n);
        }
        else if(choice == 2){
            printf("Enter Time Quantum: ");
            scanf("%d", &tq);
```

```
            // Reset remaining & started for RR
            for(int i=0;i<n;i++){
                p[i].remaining = p[i].burst;
                p[i].started = 0;
            }

            roundRobin(p, n, tq);
        }
        else if(choice == 3){
            printf("Program ended succesfully");
            return 0;
        }
        else{
            printf("Invalid choice");
        }
    }
}
koushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$ gcc Q1.c -o Q1
koushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$ ./Q1

1. FCFS
2. Round Robin
3. Exit
Enter choice: 1

Gantt Chart:
|P1|P2|P3|P4|P5|
Average Waiting Time = 4.60
Average Turnaround Time = 8.60
Average Response Time = 4.60

1. FCFS
2. Round Robin
3. Exit
Enter choice: 2
Enter Time Quantum: 2

Gantt Chart:
|P1|P2|P3|P4|P5|P1|P2|P3|P4|P2|P4|
Average Waiting Time = 7.00
Average Turnaround Time = 11.00
Average Response Time = 0.00

1. FCFS
2. Round Robin
3. Exit
Enter choice: 3
Program ended succesfullykoushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$
```

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
P_1	0	0	1	2	0	0	1	2
P_2	2	0	0	0	2	7	5	0
P_3	0	0	3	4	6	6	5	6
P_4	2	3	5	4	4	3	5	6
P_5	0	3	3	2	0	6	5	2

- Find the content of the need matrix.
- Is the system in a safe state? If so, give a safe sequence of the process.
- If P3 requests for 1 more instance of each type R2 and R4, can the request be granted immediately or not?

```
Program ended succesfullykoushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$ cat > Q2.c
// Objective: Implementation of Banker's algorithm to avoid deadlock
#include<stdio.h>
#define P 5 // number of processes
#define R 4 // number of resources
int main(){
    int alloc[P][R] = {{0,0,1,2},{2,0,0,0},{0,0,3,4},{2,3,5,4},{0,3,3,2}};
    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 avail[R] = {0,1,0,4};
    int need[P][R];
    int finish[P] = {0};
    int safeSeq[P];
    // NEED MATRIX
    for(int i=0 ; i<P ; i++){
        for(int j=0 ; j<R ; j++){
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
    printf("Need Matrix:\n");
    for(int i=0 ; i<P ;i++){
        for(int j=0 ; j<R ;j++){
            printf("%d", need[i][j]);
        }
        printf("\n");
    }
    int count = 0;
    // Safety Algorithm
    while(count < P){
        int found = 0;
        for(int i=0 ; i<P ;i++){
            if(!finish[i]){
         if(!finish[i]){
             int j;
             for(j=0 ; j<R ;j++){
                 if(need[i][j] > avail[j]){
                     break;
                 }
             }
             // if all resources can be satisfied
             if(j==R){
                 for(int k=0 ; k<R ; k++){
                     avail[k] += alloc[i][k];
                 }
                 safeSeq[count++] = i;
                 finish[i] = 1;
                 found = 1;
             }
         }
    }
    if(!found){
        break;
    }
}
// Result
if(count == P){
    printf("\nSystem is in SAFE state.\nSafe Sequence: ");
    for(int i=0 ; i<P ; i++){
        printf("P%d", safeSeq[i]+1);
    }
}else{
    printf("\nSystem is NOT in safe state.");
}
```

```
    return 0;
}
koushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$ gcc Q2.c -o Q2
koushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$ ./Q2
Need Matrix:
0000
0750
6622
2002
0320

System is NOT in safe state.koushik_das@LAPTOP-BIHGEI3G:~/DOS_2341004117/DOSEndTermProject$
```