End Term Project

On

Design of Operating Systems (CSE 4049)

Submitted by

Name:	SAYONIKA PAL
Reg. No.:	2141016020
Semester:	5th
Branch:	CSE
Session:	2023-2024
Admission Batch:	2021

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 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 followingscheduling 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 aswell. 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 (inmilliseconds), 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 theseprocesses 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 turnaroundtime, average waiting time and average response time.
- Analyze the results and determine which of the algorithms results in the minimumaverage waiting time over all processes?

Project 1:

CODE:

```
#include <stdio.h>
#include <stdlib.h>
struct Process{
	int process_id;
	int arrival_time;
	int burst_time;
};
struct Process* create_process(int process_id, int arrival_time, int
burst_time){
	struct Process* process = (struct Process*)malloc(sizeof(struct
Process));
	process->process_id = process_id;
	process->arrival_time = arrival_time;
	process->burst_time = burst_time;
	return process;
}
void FCFS(struct Process* processes, int num_processes){
	for(int i=0; i < num_processes;i++){
		for(int	j=i+1; j < num_processes; j++){
			if(processes[j].arrival_time <
processes[i].arrival_time){
				struct Process temp = processes[j];
				processes[j] = processes[i]; processes[i]
				= temp;
			}
		}
		printf("Executing process %d form time %d to %d\n",
processes[i].process_id,
processes[i].arrival_time,processes[i].arrival_time +
processes[i].burst_time);
	}
}
void Round_Robin(struct Process* processes, int num_processes, int
time_quantum) {
	int* remaining_burst_time = (int*)malloc(sizeof(int) *
```

```
num_processes);
    for (int i = 0; i < num_processes; i++) {
        remaining_burst_time[i] = processes[i].burst_time;
    }

    int current_time = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < num_processes; i++) {
        if (remaining_burst_time[i] > 0) {
                done = 0;
                if (remaining_burst_time[i] >= time_quantum) {
                    current_time += time_quantum;
                    remaining_burst_time[i] -= time_quantum;
                    printf("Executing process %d from time %d to
%d\n", processes[i].process_id, current_time - time_quantum,
current_time);
                                }
                else {
                    current_time += remaining_burst_time[i];
                    int temp = remaining_burst_time[i];
                    remaining_burst_time[i] = 0;
                    printf("Executing process %d from time %d to
%d\n", processes[i].process_id, current_time - temp, current_time);
                }

            }

        }

        if (done == 1) {
            break;
        }

    }

    free(remaining_burst_time);
}

int main(){
    struct Process processes[] = {
        {1, 0, 10},
        {2, 1, 1},
        {3, 2, 2},
        {2, 3, 1},
        {3, 6, 5}
    };
```

```
    int num_processes = sizeof(processes) / sizeof(processes[0]);

    printf("\nRound Robin:\n");
    Round_Robin(processes, num_processes, 2);

    printf("\nFCFS:\n");
    FCFS(processes, num_processes);

    return 0;
}
```

OUTPUT:

```
PS D:\SEM 5 PROJECTS\DOS\Project> cd "d:\SEM 5 PROJECTS\DOS\Project\" ; if ($?) { gcc Sheduling.c -o Sheduling } ; if ($?) { .\Sheduling }
```

Round Robin:

```
Executing process 1 from time 0 to 2
Executing process 2 from time 2 to 3
Executing process 3 from time 3 to 5
Executing process 2 from time 5 to 6
Executing process 3 from time 6 to 8
Executing process 1 from time 8 to 10
Executing process 3 from time 10 to 12
Executing process 1 from time 12 to 14
Executing process 3 from time 14 to 15
Executing process 1 from time 15 to 17
Executing process 1 from time 17 to 19
```

FCFS:

```
Executing process 1 form time 0 to 10
Executing process 2 form time 1 to 2
Executing process 3 form time 2 to 4
Executing process 2 form time 3 to 4
Executing process 3 form time 6 to 11
```

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 typeR3 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

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.
c) If P_3 will request for 1 more instances of type R2, Can the request be grantedimmediately or not?

Project 2:

A) Find the content of the need matrix.

Need Matrix (Need[i, j] = Max[i, j] - Allocation[i, j]):

	R1	R2	R3	R4
P1	0	0	0	0
P2	0	7	5	0
P3	6	6	2	2
P4	2	0	0	2
P5	0	3	2	0

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

```
#include <stdio.h>
int processes +5;
int resources = 4;
int available[4] = {6,
7,12,12};
int
max_claim[5][4]
= {
   {0, 0, 1, 2},
   {2, 7, 5, 0},
```

```
  {6, 6, 5, 6},
  {4, 3, 5, 6},
  {0, 6, 5, 2}
};

int allocation[5][4] = {
  {0, 0, 1, 2},
  {2, 0, 0, 0},
  {0, 0, 3, 4},
  {2, 3, 5, 4},
  {0, 3, 3, 2}
};

int need[5][4];

void calculate_need() {
  for (int i = 0; i <
    processes; i++) { for
    (int j = 0; j <
    resources; j++) {
      need[i][j] = max_claim[i][j] - allocation[i][j];
    }

  }

}


int check_safety(int work[], int finish[]) {


int
safe_sequence[proc
```

```
esses]; int count = 0;

    for (int i = 0; i <
      processes; i++) { if
      (finish[i] == 0) {
        int safe = 1;
        for (int j = 0; j <
          resources; j++) { if
          (need[i][j] >
          work[j]) {
            safe
            = 0;
            brea
            k;
          }
        }

        if (safe) {
          safe_sequence[co
          unt++] = i; finish[i]
          = 1;
          for (int j = 0; j <
            resources; j++) {
            work[j] +=
            allocation[i][j];
          }
          i = -1;  // Start again from the beginning
        }

      }

    }
```

```
    // Check if all
    processes are finished
    for (int i = 0; i <
    processes; i++) {
      if (finish[i] == 0) {
        return 0;  // Unsafe state
      }
    }

    // Print the safe
    sequence
    printf("Safe
    Sequence: ");
    for (int i = 0; i < processes;
      i++) { printf("P%d ",
      safe_sequence[i]);
    }
    printf("\n");

    return 1;  // Safe state
  }



int main() {
    int
    work[resour
    ces]; int
    finish[proces
```

```
  ses];

  // Initialize work and
  finish arrays for (int i
  = 0; i < resources;
  i++) {
      work[i] = available[i];
  }

  for (int i = 0; i <
    processes; i++) {
    finish[i] = 0;
  }

  calculate_need();

  // Check if the system is
  in a safe state if
  (check_safety(work,
  finish)) {
    printf("The system is in a safe state.\n");
  } else {
    printf("The system is in an unsafe state.\n");
  }


  return 0;
}
```

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

```
int request_resources(int process_num, int
request[]) {
  // Check if the request is
  within the need matrix for
  (int i = 0; i < resources; i++) {
    if (request[i] > need[process_num][i]) {
      return 0; // Request exceeds maximum claim
    }
  }

  // Check if the request is within
  the available resources for (int i
  = 0; i < resources; i++) {
    if (request[i] > available[i]) {
      return 0; // Not enough available resources
    }
  }

  // Simulate resource
  allocation and check for
  safety for (int i = 0; i <
  resources; i++) {
    available[i] -= request[i];
```

```
  allocation[process_num][i]
  += request[i];
  need[process_num][i] -=
  request[i];
}

int
work[resou
rces]; int
finish[proc
esses];
for (int i = 0; i <
  resources; i++) {
  work[i] =
  available[i];
}

for (int i = 0; i <
  processes; i++) {
  finish[i] = 0;
}

if (check_safety(work, finish)) {
  printf("Request can be granted. The
  system is still in a safe state.\n"); return 1;
} else {
  // Rollback the changes if the
```

```
    request cannot be granted for
    (int i = 0; i < resources; i++) {
      available[i] += request[i];
      allocation[process_nu
      m][i] -= request[i];
      need[process_num][i]
      += request[i];
    }

    printf("Request cannot be granted. The
    system will be in an unsafe state.\n");
    return 0;
  }

}
```