西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab 6 Report

Report Subject: OS Experiment - Lab 6

Student ID: 2018380039

Student Name: Dikshya Kafle

Submission Date: 2021/11/24

Computer Operating System Experiment

Computer Operating System Experiment

Laboratory 6

CPU Scheduling and deadlock

Objective:

- You will build a simulator of a simple operating system to learn more about operating systems in general and process schedulers in specific.
- You will write a banker algorithm program in a high-level language, and use the banker algorithm to simulate the allocation of resources and security checks.

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1. CPU Scheduling

CPU scheduling is the process by which a computer's operating system determines how, in what order, and for how long individual processes in a queue of processes are allowed to access the CPU. Input factors such as the chosen scheduling algorithm, the length of processes, and frequency of processes will have an influence on performance factors such as CPU utilization, average job waiting time, average job response time, and average job turn-around time. Depending on the application, the importance of some factors may weigh more heavily than others. For instance, a system that is designed for heavier human-computer-interaction may require lower average job response time in order to make the system appear more responsive.

In this Lab we will look at the following scheduling algorithms:

- First Come First Served
- Shortest Job First

- Priority scheduling
- Round Robin

You will observe the following output metrics:

- Average Turnaround Time
- Average Waiting Time

You will also vary our random sample of data by altering certain factors which will be discussed later.

The Process Scheduling Simulator will execute a collection of processes that demonstrate a scheduling algorithm of either First-Come/First-Served (FCFS), Shortest Job First (SJF), Priority, or Round Robin (RR) and will then generate visual aids along with statistics to explain the results. The typical scheduling queue is working as follows:

2. Experiments

2.1 Experiment 1: CPU Scheduling Simulator

Understanding the working of scheduling algorithms provides us with a knowledge of how to analyze the scheduling of processes, resource utilization, and performance in real-time applications. Various algorithms perform differently and have their unique set characteristics which are advantageous depending on the scenario and application. A simulator enables us to visualize these characteristics, working, and behaviors of scheduling algorithms. By automating the process of scheduling these tasks provided as input and displaying the output intuitively, which reduces the work on creating the schedule and focuses on analyzing the behaviors of the scheduling algorithms. This Lab focuses on the development of a web application that is machine and platform-independent, with the scope of illustrating multiple scheduling algorithms graphically to help one to draw comparisons and conclusions based on the results.

Implement a process scheduler with N processes executing concurrently. Each process is represented by a process control block (PCB). The process control block can contain the following information: process name, priority number, CPU burst time, used CPU time, process status, etc. The status of each process can be one of three states: ready W (Wait), running R (Run), or completed F (Finish). After the ready process gets the CPU, how long this process can run depends on the scheduling algorithm.

- First-Come/First-Served (FCFS): The process ready queue is arranged in descending order of arrival time. According to processes' arrival time, the processes can obtain their CPU burst with none-preemption.
- Shortest Job First (SJF): according to processes' arrival time and CPU burst, the shortest CPU burst will get their CPU burst.
- Priority scheduling: The process ready queue is arranged in descending order of priority number and arrival time, and the first process of the chain is put in first.
- Round Robin (RR): The process ready queue is arranged in descending order of arrival time. The program that implements processor scheduling by time slice rotation.

```
typedef struct pcb{

    struct pcb *next;               //The next process control block pointer
    char process_name[20];          //Process name

    int process_number;             //Process number

    int process_start_moment;       //Process start time

    int process_burst_time;         //Required running time

    int process_time_slice;         //Time slice

    int process_priority;           //Priority number

}PCB;
```

- Initialize the process control block:

  void init_pcb_table()

- Display process queue:
  void display_process_queue(PCB *queue)

- Create process:
  PCB *create_process()

- First come first serve process scheduling algorithm:
  void FCFS()

- Priority scheduling:

  void PS()

- Round Robin (RR) scheduling algorithm:
  void RR()

…

Requirements:

1) you can start to implement those scheduling algorithms with non-preemption.
2) detailed analysis of the scheduling algorithm, based on a careful analysis, completely understand the role of the main data structures and processes described and shown a flowchart of the main module of the main data structure is given.
3) Follow the prompts, write the function complete, the program became a run.
4) repeatedly run the program, the program execution result of the observation, verify the correctness of the analysis and the results of a given final operation performed by the calculated result and the weighted turnaround time turnaround time.

Input:

Process name:P0
Arrival time:0
CPU burst: 2
…

Output:

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	2	0	2
P1	5	2	7
P2	6	7	13
P3	7	13	20

Average Waiting Time = 5.500000
Average Turnaround Time = 10.500000

Code:

Whole code in zip file attached:

process_schedule.c

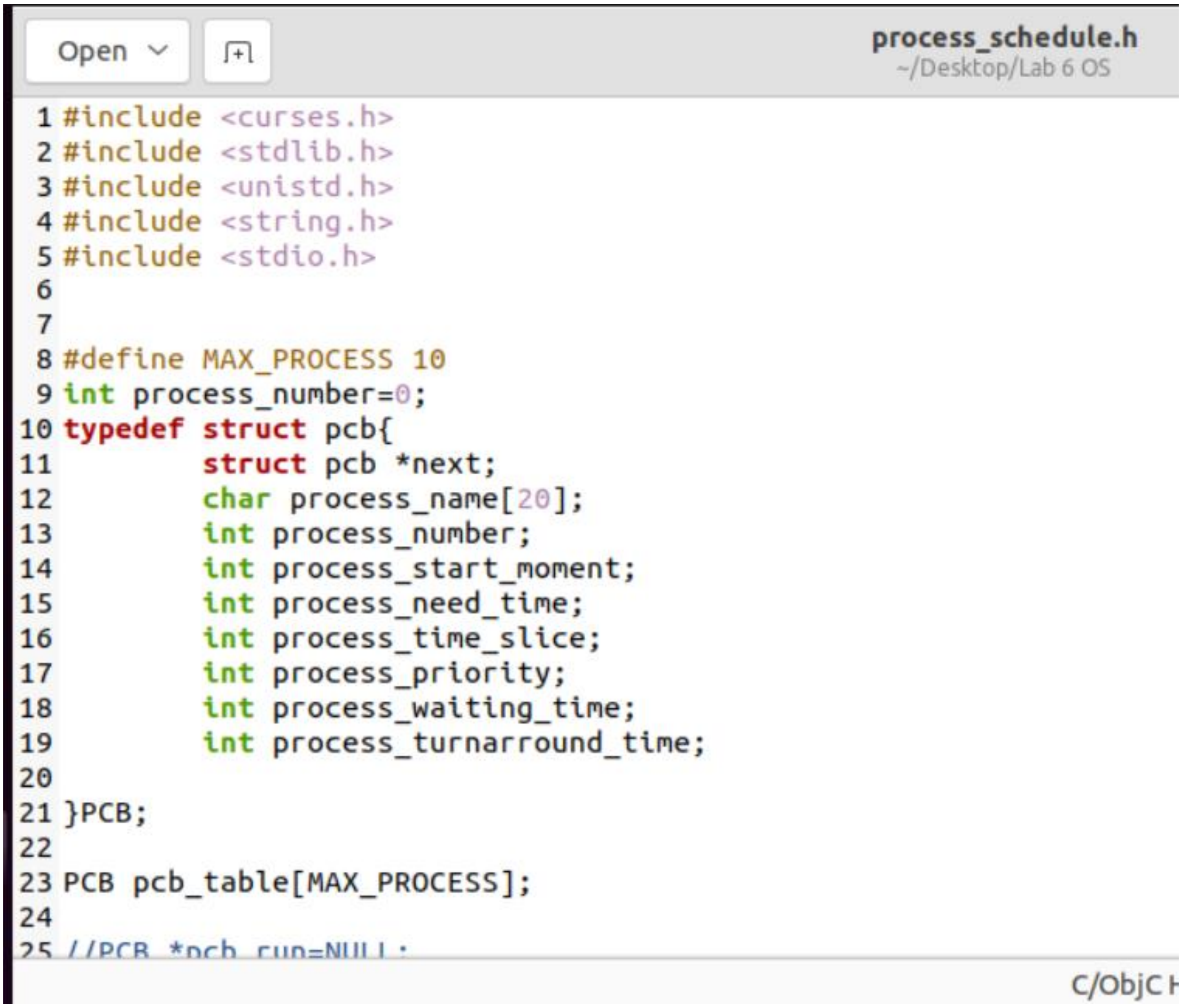
Open

process_schedule.c
~/Desktop/Lab 6 OS

```c
#include "process_schedule.h"

int main(int argc,char *argv[]){
        char select;
        initscr();
        init_pcb_table();
        bool end=false;

        while(!end){
                clear();
                refresh();

                printw("|----------MAIN    MENU--------------|\n");
                printw("|  a: Create a process              |\n");
                printw("|  b: Display processes queue       |\n");
                printw("|  1: First-Come-First-Served (FCFS) |\n");
                printw("|  2: Shortest Job First (SJF)      |\n");
                printw("|  3: Round Robin (RR)              |\n");
                printw("|  4: Priority Scheduling           |\n");
                printw("|  5:exit                           |\n");
                printw("|-----------------------------------|\n");

                printw("select a function(1~8,a~d):");
               refresh();
```

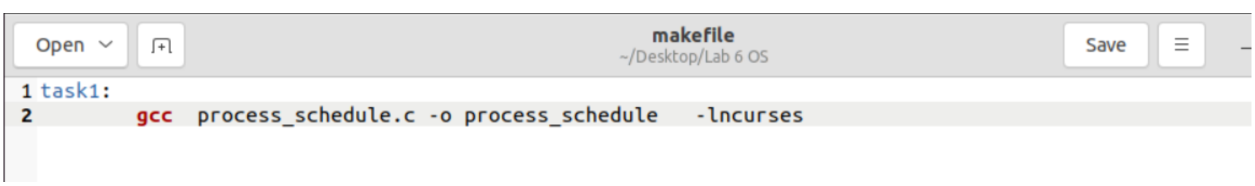
Whole code in zip file attached:

process_schedule.h




```c
#include <curses.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>


#define MAX_PROCESS 10
int process_number=0;
typedef struct pcb{
        struct pcb *next;
        char process_name[20];
        int process_number;
        int process_start_moment;
        int process_need_time;
        int process_time_slice;
        int process_priority;
        int process_waiting_time;
        int process_turnarround_time;

}PCB;

PCB pcb_table[MAX_PROCESS];

//PCB *pcb_run=NULL;
```

makefile:




```makefile
task1:
	gcc  process_schedule.c -o process_schedule   -lncurses
```

Experiment 1: CPU Scheduling Simulator

1.1) The Main Menu:

From the main menu, we assume:

A) Create a process (add a process)

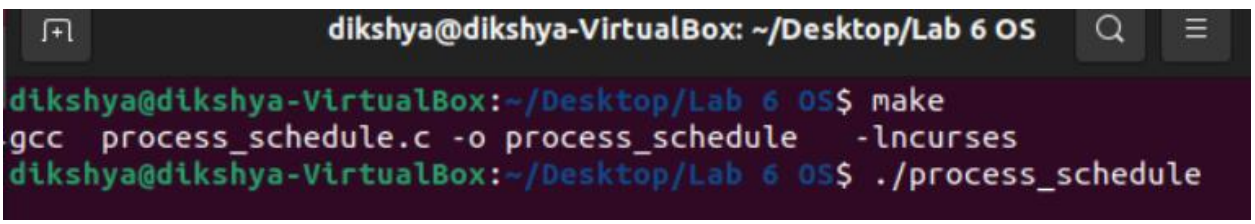
B) Display the list of added processes;

C) First Come First Served(FCFS) scheduling

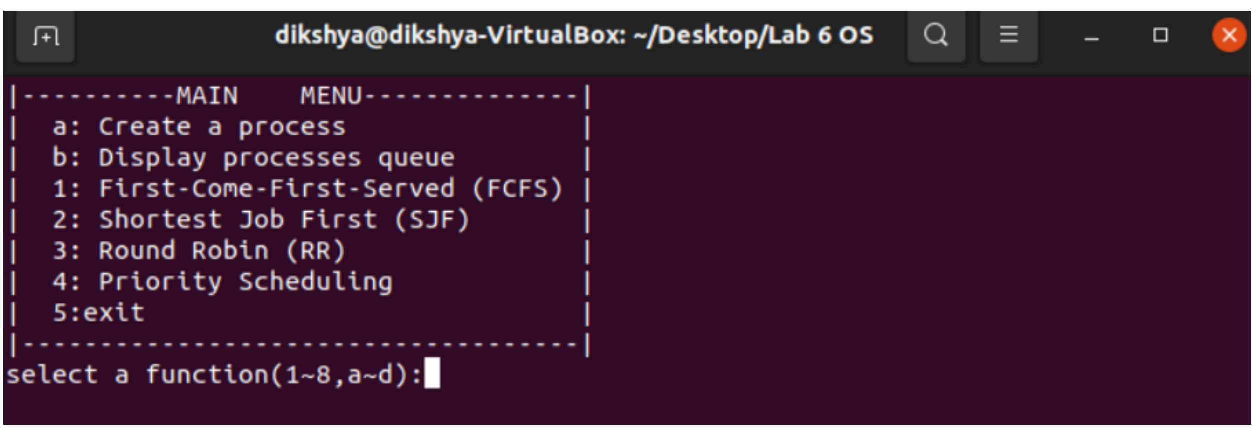
D) Shortest Job First scheduling (SJF)

E) Round Robin (RR) scheduling

F) Priority Scheduling scheduling (low numbers mean higher priority)



We first add a list of processes, then select a scheduling algorithm and analyze the waiting and turnaround time.



1.2) Adding a process:

To add a process, we specify its name, its arrival time, burst time and priority.

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS
please enter the following fields:
Process name: P1
Arrival time: 0
CPU burst: 5
Priority: 0
press any key to continue.
```

1.3) First Come First Served (FCFS):

❖ <u>The processes:</u>

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

 |----------|----------|----------|----------|----------|----------|------------
-| name     | number   | Arrival  | Burst    | Priority |  Waiting | Turnaround
 |----------|----------|----------|----------|----------|----------|------------
-| P1       | 1        | 0        | 5        | 0        | -1       | -1|
 | P2       | 2        | 3        | 9        | 0        | -1       | -1|
 | P3       | 3        | 6        | 6        | 4        | -1       | -1|
 |----------|----------|----------|----------|----------|----------|------------
-|
press any key to continue.
```

❖ <u>The Scheduling:</u>

As expected for this case when scheduled using FCFS, the average waiting time is 3.33 and the average turnaround time is 10.

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

 |----------|----------|----------|----------|----------|----------|-----------
-| name     | number   | Arrival  | Burst    | Priority |  Waiting | Turnaround
 |----------|----------|----------|----------|----------|----------|-----------
-| P1       | 1        | 0        | 5        | 0        | 0        | 5|
 | P2       | 2        | 3        | 9        | 0        | 2        | 11|
 | P3       | 3        | 6        | 6        | 4        | 8        | 14|
 |----------|----------|----------|----------|----------|----------|-----------
-|

Average waiting time = 3.333333
Average turn around time = 10.000000
press any key to continue.
```

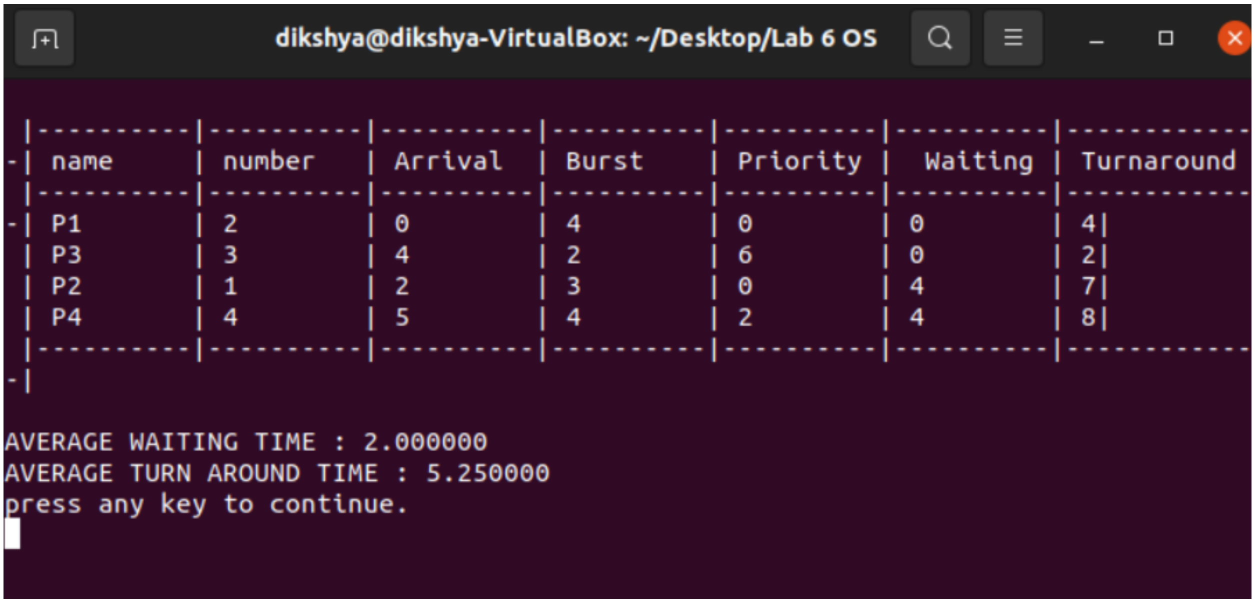
1.4) Shortest Job First (SJF):

❖ The processes:

```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

 |----------|----------|----------|----------|----------|----------|-----------
-| name     | number   | Arrival  | Burst    | Priority |  Waiting | Turnaround
 |----------|----------|----------|----------|----------|----------|-----------
-| P1       | 1        | 2        | 3        | 0        | -1       | -1|
 | P2       | 2        | 0        | 4        | 0        | -1       | -1|
 | P3       | 3        | 4        | 2        | 6        | -1       | -1|
 | P4       | 4        | 5        | 4        | 2        | -1       | -1|
 |----------|----------|----------|----------|----------|----------|-----------
-|
press any key to continue.
```

❖ The Scheduling:

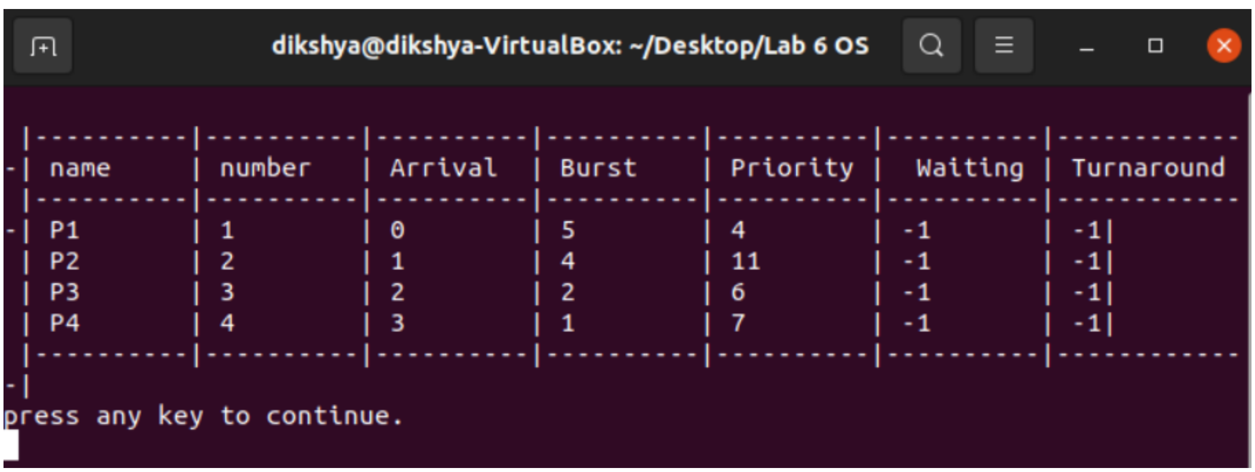
As expected for this case when scheduled SJF, the average waiting time is 2.00 and the average turnaround time is 5.25.



```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

 |----------|----------|----------|----------|----------|----------|------------
-| name     | number   | Arrival  | Burst    | Priority |  Waiting | Turnaround
 |----------|----------|----------|----------|----------|----------|------------
-| P1       | 2        | 0        | 4        | 0        | 0        | 4|
 | P3       | 3        | 4        | 2        | 6        | 0        | 2|
 | P2       | 1        | 2        | 3        | 0        | 4        | 7|
 | P4       | 4        | 5        | 4        | 2        | 4        | 8|
 |----------|----------|----------|----------|----------|----------|------------
-|

AVERAGE WAITING TIME : 2.000000
AVERAGE TURN AROUND TIME : 5.250000
press any key to continue.
```

1.5) Round Robin (RR):

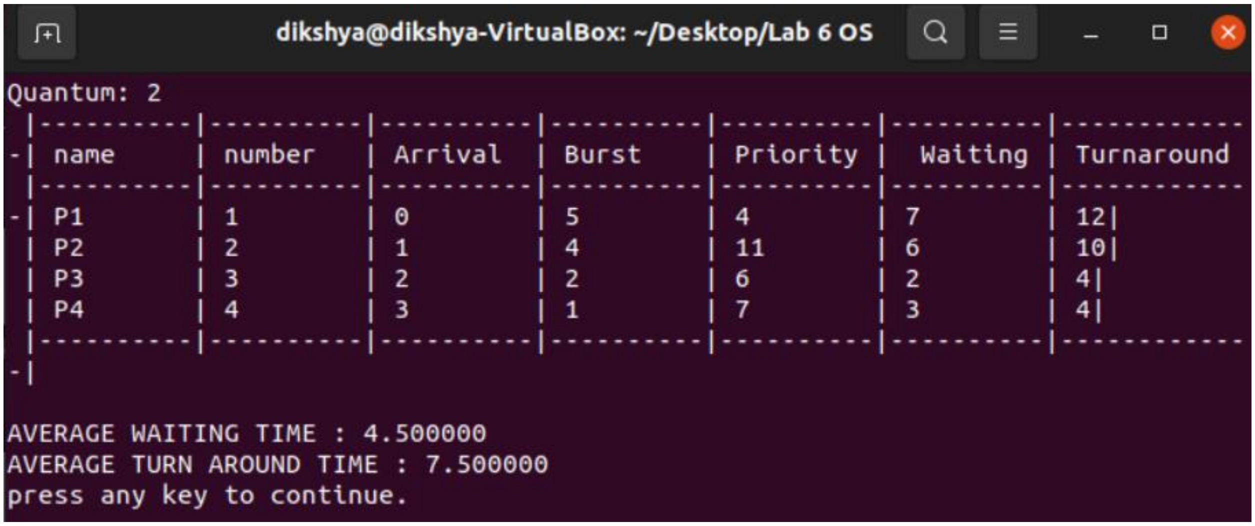
❖ <u>The processes:</u>



```
dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

 |----------|----------|----------|----------|----------|----------|------------
-| name     | number   | Arrival  | Burst    | Priority |  Waiting | Turnaround
 |----------|----------|----------|----------|----------|----------|------------
-| P1       | 1        | 0        | 5        | 4        | -1       | -1|
 | P2       | 2        | 1        | 4        | 11       | -1       | -1|
 | P3       | 3        | 2        | 2        | 6        | -1       | -1|
 | P4       | 4        | 3        | 1        | 7        | -1       | -1|
 |----------|----------|----------|----------|----------|----------|------------
-|
press any key to continue.
```

❖ <u>The Scheduling:</u>

As expected for this case when scheduled using RR where the quantum is 2, the average waiting time is 4.5 and the average turnaround time is 7.5.


dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

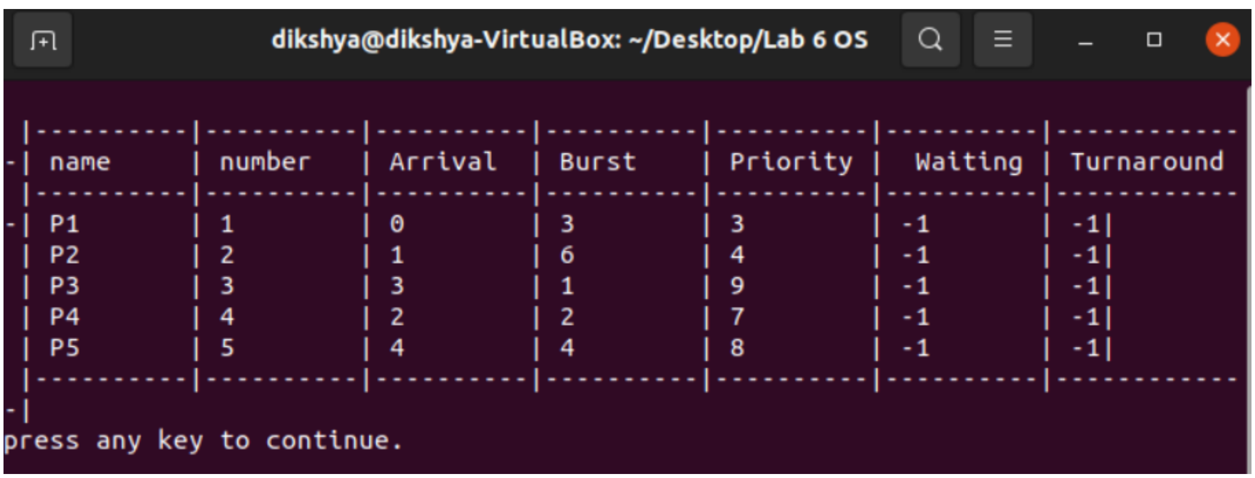
Quantum: 2

name	number	Arrival	Burst	Priority	Waiting	Turnaround
P1	1	0	5	4	7	12
P2	2	1	4	11	6	10
P3	3	2	2	6	2	4
P4	4	3	1	7	3	4

AVERAGE WAITING TIME : 4.500000
AVERAGE TURN AROUND TIME : 7.500000
press any key to continue.

1.6) Priority Scheduling:

❖ <u>The processes:</u>

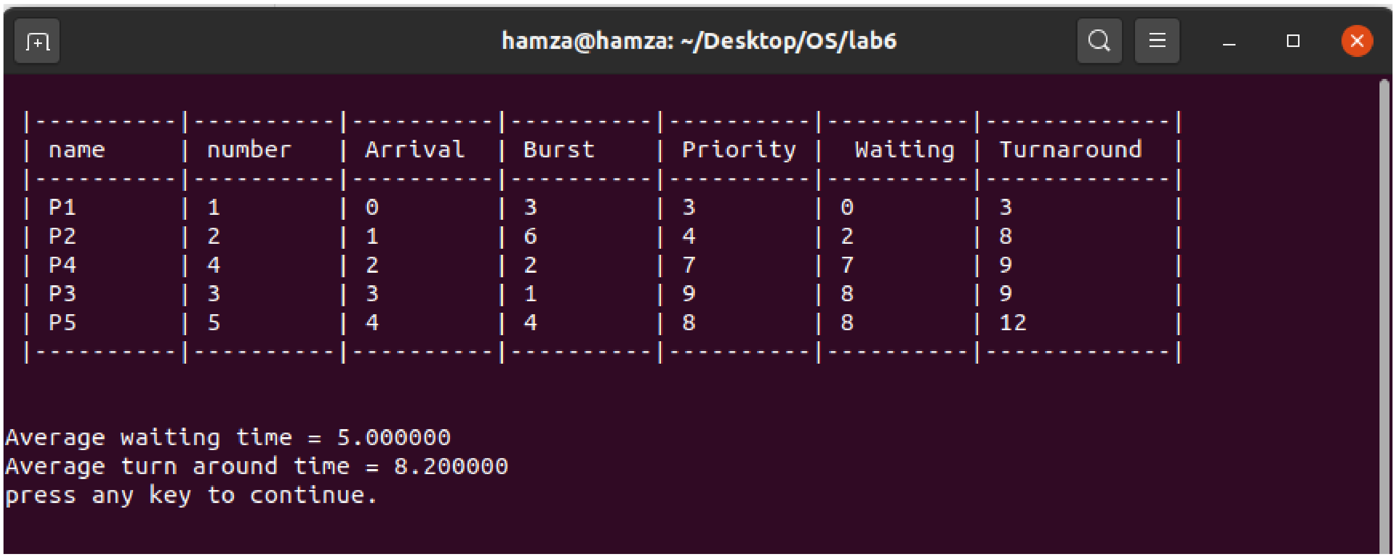

dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

name	number	Arrival	Burst	Priority	Waiting	Turnaround
P1	1	0	3	3	-1	-1
P2	2	1	6	4	-1	-1
P3	3	3	1	9	-1	-1
P4	4	2	2	7	-1	-1
P5	5	4	4	8	-1	-1

press any key to continue.

❖ <u>The Scheduling:</u>

As expected for this case when scheduled using Priority Scheduling (non-preemptive), the average waiting time is 5 and the average turnaround time is 8.2.



```
hamza@hamza: ~/Desktop/OS/lab6
|----------|----------|----------|----------|----------|----------|-------------|
| name     | number   | Arrival  | Burst    | Priority |  Waiting | Turnaround  |
|----------|----------|----------|----------|----------|----------|-------------|
| P1       | 1        | 0        | 3        | 3        | 0        | 3           |
| P2       | 2        | 1        | 6        | 4        | 2        | 8           |
| P4       | 4        | 2        | 2        | 7        | 7        | 9           |
| P3       | 3        | 3        | 1        | 9        | 8        | 9           |
| P5       | 5        | 4        | 4        | 8        | 8        | 12          |
|----------|----------|----------|----------|----------|----------|-------------|

Average waiting time = 5.000000
Average turn around time = 8.200000
press any key to continue.
```

2.2 Experiment 2: Safety Algorithm Implementation

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 a safe-state check to test for possible activities, before deciding whether allocation should be allowed to continue.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:

- **Available**. A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
- **Max.** An n × m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj.
- **Allocation**. An n × m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.
- **Need**. An n × m matrix indicates the remaining resource need of each process. If

Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

Safety Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work* = *Available

   Finish [i] = false for i = 0, 1, …, n- 1

2. Find an *i* such that both:

   (a) ***Finish [i] = false***

   (b) ***Need_i £ Work***

   If no such *i* exists, go to step 4

3. *Work* = *Work* + *Allocation*$_i$
   Finish[i] = *true*
   go to step 2

4. If *Finish [i]* == *true* for all *i*, then the system is in a safe state

Requirements:

1) Implement Safety Algorithm that can determine the system is in a safe state or not if you give the inputs.
2) If system is in the safe sate, please output the execute sequence of processes.

Input:
Enter No. of AVAILABLE Instances for each resource

Enter MAXIMUM instance for a Process & its corresponding resource

Enter instance ALLOCATED for a Process & its corresponding resource

Output:
The system is safe.
The Safe Sequence is P0, P3, P2, P1.

Suppose that, at time T0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Makefile:

```
*makefile
~/Desktop/Lab ...

safety: safety.o
	gcc -o $@ $^

safety.o: safety.c
	gcc -c -g $^ -o $@
```

Code:

safety.c

Please find the whole code attached in the zip file

Open

safety.c
~/Desktop/Lab 6 OS

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>

int allocation[5][3];
int max[5][3];
int need[5][3];
int avaliable[3];

bool finished[5] = {false, false, false, false, false};

bool check_avaliable(int n);

int get_avaliable();
int get_allocation();
int get_max();
int calc_need();
int work(int n);

int main()
{
    get_avaliable();
    get_allocation();
    get_max();
```

safety.c

Please find the whole code attached in the zip file

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 6 OS$ ./safety
Please enter the avaliable amount of three resources, divided by space:
3 3 2
Please enter the allocated resources of process P0, divided by space:
0 1 0
Please enter the allocated resources of process P1, divided by space:
2 0 0
Please enter the allocated resources of process P2, divided by space:
3 0 2
Please enter the allocated resources of process P3, divided by space:
2 1 1
Please enter the allocated resources of process P4, divided by space:
0 0 2
Please enter the maximum requirement of process P0, divided by space:
7 5 3
Please enter the maximum requirement of process P1, divided by space:
3 2 2
Please enter the maximum requirement of process P2, divided by space:
9 0 2
Please enter the maximum requirement of process P3, divided by space:
2 2 2
Please enter the maximum requirement of process P4, divided by space:
4 3 3
The system is safe.
One Safe Sequence is P1, P3, P0, P2, P4.
dikshya@dikshya-VirtualBox:~/Desktop/Lab 6 OS$
```

2.3 Experiment 3: Bankers Algorithm (optional)

For this Lab, you will write a multithreaded program that implements the banker's algorithm. Several customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. This programming assignment combines three separate topics: (1) multithreading, (2) preventing race conditions, and (3) deadlock avoidance.

The Banker

The banker will consider requests from n customers for m resources types. The banker will keep track of the resources using the following data structures:

```
/* these may be any values >= 0 */

   #define NUMBER OF CUSTOMERS 5

   #define NUMBER OF RESOURCES 3

/* the available amount of each resource */

      int available[NUMBER OF RESOURCES];

/*the maximum demand of each customer */

      int maximum[NUMBER OF CUSTOMERS][NUMBER OF RESOURCES];

/* the amount currently allocated to each customer */

      int allocation[NUMBER OF CUSTOMERS][NUMBER OF RESOURCES];

/* the remaining need of each customer */

      int need[NUMBER OF CUSTOMERS][NUMBER OF RESOURCES];
```

The Customers

Create n customer threads that request and release resources from the bank. The customers will continually loop, requesting and then releasing random numbers of resources. The customers' requests for resources will be bounded by their respective values in the need array. The banker will grant a request if it satisfies the safety algorithm outlined in previous experiment. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:

```
   int request resources(int customer num, int request[]);

   int release resources(int customer num, int release[]);
```

These two functions should return 0 if successful (the request has been granted) and –1 if unsuccessful. Multiple threads (customers) will concurrently. access shared data through these two functions. Therefore, access must be controlled through mutex locks to prevent race conditions. The use of Pthreads mutex locks are described in the project entitled “Producer–Consumer Problem”.

Implementation

You should invoke your program by passing the number of resources of each type on the command line. For example, if there were three resource types, with ten instances of the first type, five of the second type, and seven of the third type, you would invoke your program follows:

./banker 10 5 7

The available array would be initialized to these values. You may initialize the maximum array (which holds the maximum demand of each customer) using any method you find convenient.

Code:

Full code on zip file: bankers.c

Open

bankers.c
~/Desktop/Lab 6 OS

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 3

pthread_mutex_t mutex;

int f_num = 0;
int s_num = 0;
int t_num = 0;
int mcount = 0;
int available[NUMBER_OF_RESOURCES];
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

pthread_t tids[NUMBER_OF_CUSTOMERS];

int chk_data(char *num)
{
    int i;
    for (i = 0; num[i]; i++)
    {
        if (num[i] > '9' || num[i] < '0')
        {
            return 0;
        }
    }
```

```
int main(int argc, char *argv[])
{
        if (argc < 4)
        {
                fprintf(stderr, "Error: not enough parameters.");

                return -1;
        }
        else if (argc>4)
        {
                fprintf(stderr, "Error: too many parameters.");

                return -1;
        }

        if (chk_data(argv[1]) == 0 | chk_data(argv[2]) == 0 | chk_data(argv[3]) == 0)
    {
                fprintf(stderr, "Error: input parameter is not a positive integer.");

        return 1;
    }
```

```
        srand((unsigned)time(NULL));

        init();

        if (safe() == 0)
        {
                printf("unsafe at first\n");
                return 0;
        }

        int i;

        for (i = 0; i < NUMBER_OF_CUSTOMERS; i++)
        {
                pthread_t tid;
                tids[i] = tid;
                pthread_attr_t attr;
                pthread_attr_init(&attr);
                pthread_create(&tid, &attr, req, &i);
        }

        for (i = 0; i < NUMBER_OF_CUSTOMERS; i++)
        {
                pthread_join(tids[i], NULL);
        }

        return 0;
}
```

Makefile:

*makefile
~/Desktop/Lab ...

```
bankers: bankers.o
	gcc -o $@ $^ -lpthread

bankers.o: bankers.c
	gcc -c -g $^ -o $@
```

Results:

dikshya@dikshya-VirtualBox: ~/Desktop/Lab 6 OS

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 6 OS$ make
gcc -c -g bankers.c -o bankers.o
gcc -o bankers bankers.o -lpthread
dikshya@dikshya-VirtualBox:~/Desktop/Lab 6 OS$ ./bankers
Error: not enough parameters.dikshya@dikshya-VirtualBox:~/Desktop/Lab 6 OS$
```