\*\*This project is taken from the book Operating system concepts 10th edition\*\*

**Programming Project**

**Scheduling Algorithms**

This project involves implementing several different process scheduling algorithms.

The scheduler will be assigned a predefined set of tasks and will

schedule the tasks based on the selected scheduling algorithm. Each task is

assigned a priority and CPU burst. The following scheduling algorithms will

be implemented:

• First-come, first-served (FCFS), which schedules tasks in the order in which

they request the CPU.

• Shortest-job-first (SJF), which schedules tasks in order of the length of the

tasks’ next CPU burst.

• Priority scheduling, which schedules tasks based on priority.

• Round-robin (RR) scheduling, where each task is run for a time quantum

(or for the remainder of its CPU burst).

• Priority with round-robin, which schedules tasks in order of priority and

uses round-robin scheduling for tasks with equal priority

.

Priorities range from 1 to 10, where a higher numeric value indicates a higher

relative priority. For round-robin scheduling, the length of a time quantum is

10 milliseconds.

**I. Implementation**

The implementation of this project may be completed in either C or Java, and

program files supporting both of these languages are provided in the source

code download for the text. These supporting files read in the schedule of tasks,

insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form[***task name***] [***priority***] [***CPU burst***], with

the following example format:

T1, 4, 20

T2, 2, 25

T3, 3, 25

T4, 3, 15

T5, 10, 10

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It

is assumed that all tasks arrive at the same time, so your scheduler algorithms

do not have to support higher-priority processes preempting processes with

lower priorities. In addition, tasks do not have to be placed into a queue or list

in any particular order.

There are a few different strategies for organizing the list of tasks, as

first presented in Section 5.1.2. One approach is to place all tasks in a single

unordered list, where the strategy for task selection depends on the scheduling

algorithm. For example, SJF scheduling would search the list to find the task

with the shortest next CPU burst. Alternatively, a list could be ordered according

to scheduling criteria (that is, by priority). One other strategy involves

having a separate queue for each unique priority, as shown in Figure 5.7. These

approaches are briefly discussed in Section 5.3.6. It is also worth highlighting

that we are using the terms ***list*** and ***queue*** somewhat interchangeably.

However, a queue has very specific FIFO functionality, whereas a list does not

have such strict insertion and deletion requirements. You are likely to find the

functionality of a general list to be more suitable when completing this project.

**II. C Implementation Details**

The file driver.c reads in the schedule of tasks, inserts each task into a linked

list, and invokes the process scheduler by calling the schedule() function. The

schedule() function executes each task according to the specified scheduling

algorithm. Tasks selected for execution on the CPU are determined by the pick-

NextTask() function and are executed by invoking the run() function defined

in the CPU.c file. A Makefile is used to determine the specific scheduling algorithm

that will be invoked by driver. For example, to build the FCFS scheduler,

we would enter

make fcfs

and would execute the scheduler (using the schedule of tasks schedule.txt)

as follows:

./fcfs schedule.txt

Refer to the README file in the source code download for further details. Before

proceeding, be sure to familiarize yourself with the source code provided as

well as the Makefile.

**IV. Further Challenges**

Two additional challenges are presented for this project:

**1.** Each task provided to the scheduler is assigned a unique task (tid).

If a scheduler is running in an SMP environment where each CPU is

separately running its own scheduler, there is a possible race condition on

the variable that is used to assign task identifiers. Fix this race condition

using an atomic integer.

On Linux and macOS systems, the sync fetch and add() function

can be used to atomically increment an integer value. As an example, the

following code sample atomically increments value by 1:

int value = 0;

\_sync\_fetch\_and\_add(&value,1);

**2.** Calculate the average turnaround time, waiting time, and response time

for each of the scheduling algorithms.