ESE 333: Real Time Operating System

Round-Robin Scheduling
Project 3

1 Introduction

In this assignment you are to implement a round-robin scheduling algorithm with one queue on top of a custom simulation model. The assignment is divided into 3 main sections:

- Reading from input file the processes
- Scheduling them with round-robin algorithm
- modifying your code to account for I/O interrupts

2 Overview

The simulation model comes with a few *structs* that you need to understand:

- computer
- core
- process
- node

computer: a struct that simulates a computer with 4 cores and the time using a single value increment.
core: a struct that simulates a core that takes one process at a time, counts how long a process has been on the core for the current quantum, and a busy signal indicating if the core has a running process or not.
process: a struct simulating a process that has an ID, the time it arrives, and remaining time which indicates how long that process needs to run on a core to finish, and I/O (either 0 or 1 values) which indicates an I/O event has happened and this process has to be taken away from the core – resetting the I/O value (i.e. setting it to 0) then placing it to the back of the linked list queue, and running this process again when its turn has come. In part 3 I/O will be used.

There are a few functions, some will need modifications, and some are there to help you:

- *read_file*: This function handles reading and parsing input of processes from a text file. Your first part of the assignment is to add code in that function to store the input into an upcoming process queue structure. There are two queues in this assignment, one where the scheduling algorithm processes, and the other for "future" processes that will arrive in future. In the first part, you are to add all parsed processes to the "future" queue in the same order of reading. You can assume processes will come in ascending order of arrival time.

- *run_one_step* This function is to not be modified for any reason. This function increments the computer's time by one millisecond and decides on each core, whether to continue to run the current process, or the quantum is used up and should switch to another process.

- *run_one_step_p3:* This function is similar to the one above, but it generates I/O interrupts with a probability at each millisecond, upon which the current process should be switched out, and a new one comes in.

- *remove_proc:* This function is a helper function that removes a process from a core, and either discards it if its service time decreases to 0, or puts it to the tail of the queue otherwise.

- *sched_proc:* This function schedules a process to one of the cores which you do give the ID of the core as a parameter (with the process too).

- *demo:* This function is purely a very basic demonstration of the functions model and how to use. It adds 4 processes, and runs steps then removes a process when it finishes. Your code for part 2 (i.e. scheduling algorithm) is to be inside a while loop that keeps running till all processes are complete, many processes will take multiple quantam to finish.

To summarize, a text file where each line is a process ID name, arrival time, and service time will be parsed and one process struct per process should be created and stored in a "upcoming queue". As simulation runs and time proceeds, processes are to be added to the round-robin queue to be processed when the their arrival times come (i.e. you can schedule a process only upon its arrival time).

3 Reading Input

You are to read input from the sample file and parse it to create a struct of the following parameters from each line:

- Process ID

- service_time

- arrival_time

Therefore, your structure is to store process ID, service time, and arrival time. To refer to the order and how it is parsed, please refer to function **read_file()** which handles parsing file for you. Your task is to create process structs and store information for each process in an "upcoming processes queue".

Note: You can assume that the input file will have ascending order of arrival times (i.e. no need to sort your "upcoming processes queue".

4 Scheduling without I/O interrupts

Some processes may have long service time and occupy a core for a long time, which should not happen in the OS. The OS uses a fixed duration called *quantum*, and a process runs at most a quantum on a core before being switched out. In this assignment, *quantum* is set to 20 (i.e. a process should not stay longer than 20 steps on a core, each step can be one millisecond time). You are to implement a round-robin algorithm that dynamically looks into the processes in the queue and checks each core that has a process to see if the current process has either finished (i.e. service time is zero) or it has stayed on the core as much as *quantum* value (to ease coding for you, there is a value on each core, called *proc_time* that gets incremented with every step, check that value and make sure it does not exceed the quantum value at any point.).

Furthermore, an awaiting process on the "upcoming processes queue" should be added to the round-robin queue when the computer time (i.e. value of time inside *computer struct*) has increased to that process arrival time.

5 Scheduling With I/O Interrupts

For this section, your scheduling algorithm has to be modified to account for I/O interrupts. When a process gets I/O interrupts, it is removed from the cor, placed at the back of the round robin queue, and waits for its turn in the queue to be placed on a core again. For this to function, you need to check at each step whether any of the processes on each of the cores has an I/O trigger (i.e. value of I/O becomes 1 in the process struct). If so, you must take it off the core, enqueue it to the round-robin qeueue and make sure you reset the I/O value to 0. Furthermore, you have to use function *run_one_step_p3()* instead of *run_one_step()*.