CPU Scheduling

1 The Problem

The Simply Irie restaurant thieves are still out there (at least at the time of this writing) and their next target is our own Dining Center. So, one night they break in and steal most of the chairs and tables (no one knows why they didn't take the Nescafe machines instead). The university does not have enough time to add tables in the DC so they have to do with the few tables left for at least two days. That's when you realize that the scheduling policies you learned can be used to reduce some of frustration of scarcity of seats in the DC. Let's say there are N tables left in the Dining Center. Students come in with some random inter-arrival time and each student sits at one table. If all tables are full, they have to wait in line. Note that the line might build up indefinitely. A student that sits at a table will eat for a specific amount of time and then leave. Note that the time it takes each students is pre-determined. So each student knows before joining the line how much they are going to take eating. You want to try out different scheduling algorithms for sitting students at the tables in the first day, and compare them and stick with the best one for the second day. The algorithms are compared by these criteria:

1. Turnaround Time: The time it takes a student from first sitting at a table until they finish and leave DC. The smaller this number the better.

2. Waiting time: The total amount of time a student spent outside of dining area (waiting in line). The smaller this number the better.

Here are the scheduling algorithms:

1. FIFO: First student in, is the first to be served. Seems pretty fair right?

2. Round-Robin: Each student can't take more than a specific time quantum at the table. If they surpass that time slot, they are going to have to get up and wait outside even if they are in the middle of eating (however rude that is).

You need to implement the above algorithms in two separate files: One named round-robin.c or .cpp and the other fifo.c or fifo.cpp, depending on whether you use C, or C++, respectively.

2 The Input

The input of the program is read from a file. The name of the file is going to be taken from user input. The file[1] is structured as follows:

1. The first line of the file is a single integer specifying the time quantum size of the round-robin algorithm.

2. After that, an arbitrary number of lines follow each containing two numbers separated by one space character. The first number is an integer that denotes the relative timestamp of the student arrival. The first timestamp is always zero and timestamps increase afterwards. The value of each timestamp, therefore, indicates how long it has been since the previous arrival. The second number is an integer specifying the amount of time it takes that specific student to eat.

3 The Implementation

You need to solve this problem for number of tables $N = 4$. Each table has a server which is represented by a thread. You should have at least 5 threads:

1. One thread reads information about students from the input file and adds them to the *Queue*. This is the producer.

2. Four threads for scheduling students and serving them accordingly.

The scheduling, itself, is done using threads (the pthread library). You should have one thread for each dining table, as we have one dining area server responsible for each table. You can implement the *Queue* using whatever data structure you see fit (at this point you must have seen enough examples to be able to make a decent choice (we suggest a doubly linked list). Just be aware of two things:

1. The data structure should work with sharing, communication and synchronization mechanisms.

2. The data structure should support infinite (theoretically) queue size.

[1]the file shall be used for both scheduling algorithms although the quantum is not needed in the FIFO algorithm.

4 How to do It

The following are suggested implementation steps:

1. Start with the student producing thread, that is going to give you an idea of what data structure is suitable for the assignment and how you should implement it. There are two general steps here:
   (a) Creating a student, which requires reading arrival time and eating duration of the student from the input file, and adding the info to a proper data structure that represents a student.
   (b) Adding the newly created student to the *Queue*. At this stage you will probably encounter the problem of exceeding the size of the allocated queue data structure as you continually add students. There are two solutions for this problem. The simple one is allocating enough memory in the beginning so that you are sure you won't encounter this issue. The second solution is dynamically allocating new memory regions on the fly. **Mention in your report if you have used this approach and also mention how and where, to get some bonus points.**
2. After you are finished with the producer, write a very simple scheduler that reads the items added to the *Queue* by the producer to make sure the communication between the two works correctly. Add the synchronization mechanisms for the *Queue* as well.
3. Then you can start implementing the scheduling algorithms one by one. At this point you might start to think about the details of representing each student in the *Queue*. We suggest you include these details **at least**:
   (a) A unique ID. This is can be generated automatically using a counter initialized to 0, and incremented before reading each line; i.e. the first student will have $ID = 1$, the second 2, and so on.
   (b) How much of eating the student has left.
4. You might want to start with the FIFO scheduling algorithm since its the easiest one. The scheduling takes place in three general steps:
   (a) Take a student out of the *Queue* according to the scheduling policy. In the case of FIFO, the *Queue* acts like a FIFO queue and you need to take the head of the *Queue* for serving.
   (b) Start the dining process and do busy waiting (a while loop maybe) until time is up. In the FIFO scheduling algorithm which is non-preemptive, the student will sit at the table and eat until s/he is done, then the student will get out of the dining centre.

(c) In the case of Round-Robin, it might as well be the case that the student's time slot is up and s/he needs to be preempted. You need to add the student to the *Queue* again and sit another student at the table. Be aware that any access to the *Queue* must be synchronized using mutex locks or semaphores.

We advise against using the clock() function for timestamps unless you find a way around its limitations. It is better to use the *chrono* library API calls. Also, after reading from the file has finished, your program needs to join with the threads and exit gracefully, deallocating resources.

5 The output

You output will be processed using an interpreter for correction, so make sure you strictly follow the format specified. The program is expected to produce lines of output at different times:

1. Whenever a student arrives and joins the *Queue*, this statement must be printed:

   Arrive `<student ID>`

2. Whenever a student is selected by a scheduling thread to sit at a table, this statement must be printed:

   Sit `<student ID>`

3. Whenever a student is preempted and put in the line, this statement must be printed:

   Preempt `<student ID>`

4. Whenever a student is finished and leaves the Dining Center, this must be printed:

   Leave `<student ID>` Turnaround `<turnaround time measured>` Wait `<waiting time measured>`

Note that the times specified here are in seconds. Just replace whatever is enclosed by `<>` with the required value, no `<>` should be in the output. If your timings are off by a couple of milliseconds, you need to round to the closest integer. e.g. 1.99 s should be rounded to 2 seconds.

6 Grading

This assignment will carry 25%. In addition to **code checking**, your program will be tested against some prepared and well designed test files. You can also provide your own test file.

7 To Submit

You need to submit source files, executables, along with a *Makefile* that will be used to compile and link your shell.