Computer Science 360 – Introduction to Operating Systems
Summer 2021

Assignment #3
Due: Thursday, March 25, 11:55 pm via push to your Gitlab.csc repository

Programming Platform

For this assignment **your code must work on the Virtualbox/Vagrant configuration you provisioned for yourself in assignment #0.** You may already have access to your own Unix system (e.g., Ubuntu, Debian, macOS with MacPorts, etc.) yet I recommend you work as much as possible while with your CSC360 virtual machine. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration. (We cannot give marks for submissions of which it is said "It worked on Visual Studio!")

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. However, sharing of code is strictly forbidden. If you are still unsure about what is permitted or have other questions regarding academic integrity, please direct them as soon as possible to the instructor. (Code-similarity tools will be run on submitted work.) Any fragments of code found on the web and used in your solution **must be properly cited** where it is used (i.e., citation in the form of a comment giving the source of code).

Goals of this assignment

1. Write a C program implementing a simulation of *round-robin CPU scheduling*.

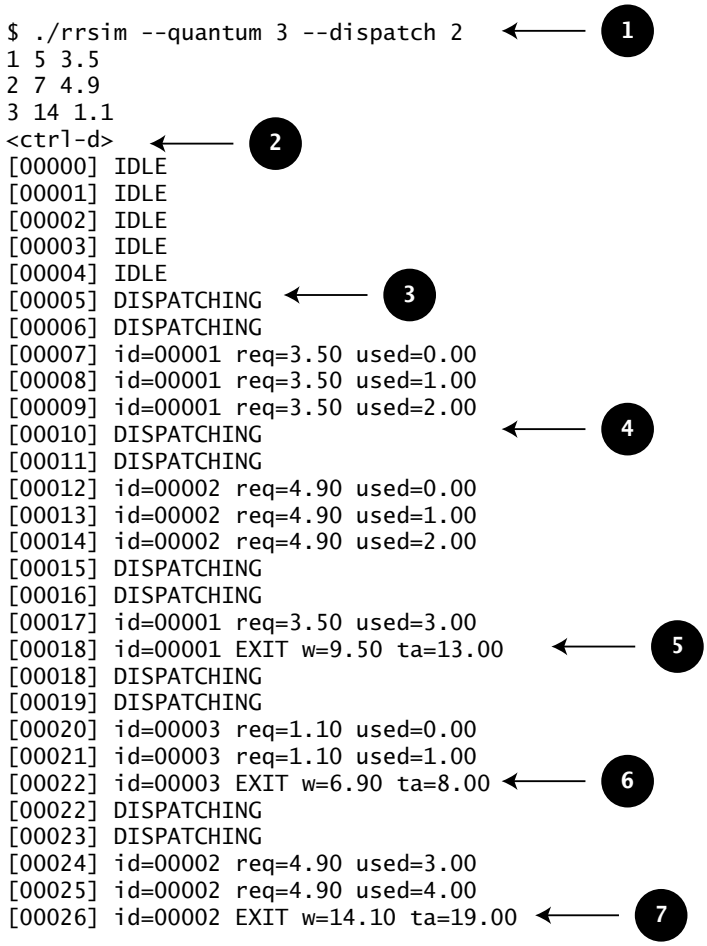
2. Write a script (in a language of your own choice) that produces some analysis the simulator output, and present a visual representation from this analysis.

Task 1: `rrsim.c`

Unlike the previous assignments involving a significant amount of systems programming, in this one your work is to write a C implementation of a round-robin CPU-scheduler simulator.

More specifically your implementation of **`rrsim.c`** will accept from **`stdin`** a stream of incoming tasks – each indicating its arrival time and CPU requirement – and display the actions of the CPU at each *tick* (i.e., was the CPU idle? did a task use a full tick? did a task finish during the tick?). We will work with more abstract *ticks* rather than specific units of time such as milliseconds or microseconds.

Below is a sample output from **`rrsim`**. Numbered arrows denote points having comments below in this assignment handout.



1. For this run of the simulator, the length of the scheduling quantum is three ticks and the cost of a dispatch (i.e., the work of a context switch & of using the ready queue, etc.) is two ticks.

2. The input to **`rrsim`** in this example has been typed in by me at the console, and the <Ctrl-d> is literally the control-key-letter combination (which means

"end-of-file" in Unix). Normally I would pipe the events into **`rrsim`** either from a file or from some event simulator. (Later in this description I briefly show the use of something I call **`simgen`**). Each input line consists of (i) a task number (integer); (ii) an arrival time/tick (integer); and (iii) the number of ticks required to compute the task (float). *You can assume tasks are always provided to the simulator in order of increasing arrival times.*

3. The simulator has been running for five ticks so far, starting at tick 0. Since the first task does not arrive until tick 5, the CPU is idle for the first five ticks. At tick 5 the first scheduling event occurs. Since the cost of each scheduling / dispatch is two ticks for this simulator, we show these two ticks.

4. The first task has now executed for three ticks – that is, the length of the quantum – and since it requires 3.5 ticks of computation it is clear that task 1 is not yet finished. Therefore it must be placed at the end of the ready queue, and the next task scheduled. It just so happens that at tick 7 the task with id 2 arrived, and since this is at the head of the queue, task 2 is scheduled. (Note, again, the cost of the dispatch).

5. Sometime between tick 17 and 18, task 1 was completed. Even though it really only required 0.5 ticks, *we count the whole tick towards the task*. There are two lines printed by the simulator for tick 18 – one to indicate the completion of task 1 (with its waiting time and turnaround reported), and another to indicate a scheduler dispatch (because at time 18 there are tasks on the queue ready to be scheduled).

6. Task 3 – which arrived at tick 14 and requires only 1.1 ticks of computation – starts to run at tick 20. As its required computation fully fits the quantum of 3 for this run of the simulator, it runs to completion (i.e., uses ticks 20 and 21 for computation).

7. Task 2 completes at tick 26. As no more tasks will "arrive" in the simulator, and all simulator tasks are now completed, the line reporting the wait/turnaround statistics for task 2 is also the last line of the simulation.

To simplify your implementation, you must assume a quantum always starts from the beginning of each scheduling event even if the previous task did not use up its whole quantum. Also: The cost of a dispatch occurs for each scheduling event whether or not the same task that used up its quantum is selected immediately for rescheduling (i.e., even if there is only one task in the system). There is no cost for adding a newly-arrived task to the ready queue.

Your output from **`rrsim`** is to match the *formatting* shown in my example run.

You are provided with the following files (on **linux.csc.uvic.ca** within **/home/zastre/csc360/assign3**):

- **rrsim.c**: Your solution to Task 1 must appear in this file. The code provided to you in **rrsim.c** reads tasks from **stdin** and stores them in a linked list (see **linkedlist.[ch]** below). The simulator-loop code itself is to appear in the function named **run_simulation()**.

- **linkedlist.[ch]**: This contains code for operations on a singly-linked list. Each node in the linked list is of type **taskval_t** storing information about a task. The list of events will be in such a linked list, and your ready queue for the simulator must also use a linked list with nodes of **taskval_t** (i.e., ***you are not permitted*** to use an array for the ready queue). Note that moving a task from the incoming event list to the ready queue ***does not*** require creating a brand new node. **Note: You may modify the linked-list implementation (i.e., modify existing routines, add new ones) if it helps with the implementation of your simulation; if you do so make such modifications, you must submit linkedlist.c and linkedlist.h for A#3.**

- **testlist.c**: Contains sample code demonstrating usage of the linked-list functions. Please read and study this to learn how the linked-list functions are to be used. Note routines modifying the list *always* return the head of the resulting list (i.e., **add_front()**, **add_end()**, **remove()**). Operations to be applied on all nodes of a linked list are performed using **apply()**. Please look at the examples in **testlist** using **print_task()** and **increment_count()** – defined in **testlist** – as function-pointer arguments to **apply()** implemented in **linkedlist**.

- **simgen.c**: (see below)

- **makefile**: builds **rrsim**, **linkedlist**, **simgen** and **testlist**.

Because you will want to generate random tasks for your simulator I have provided such a program named **simgen**. It takes two command-line arguments: the number of tasks for the run and the seed for the random number generator. Below is an example of how **simgen** and **rrsim** work together in the **bash** shell (*e.g.*, 1000 tasks with a random-seed value 3141, for which the simulator is executed using a quantum of 50 and a dispatch cost of 0):

```
$ ./simgen 1000 3141 | ./rrsim --quantum 50 --dispatch 0
```

Task 2: Analyzing output from the simulator

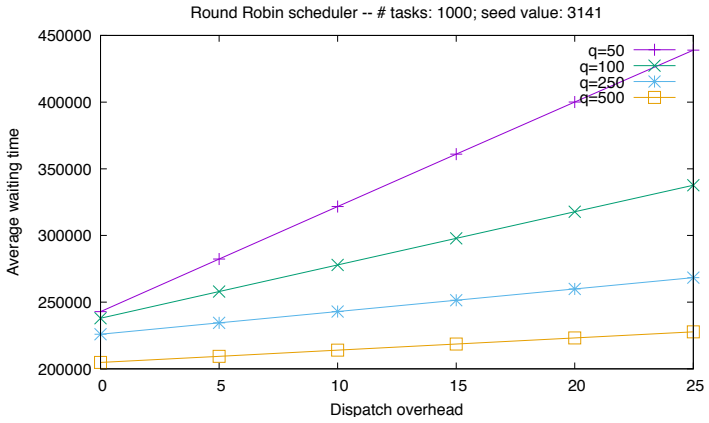
The waiting time and turnaround time of tasks scheduled in a round-robin manner will vary with both the quantum length and the cost of a context switch (i.e., dispatch time). In order to explore this you are to analyze the output of the simulator's run by computing the average waiting time and average turnaround time for all tasks. To make the analysis more interesting you must compute these for different quantum lengths and dispatch costs:

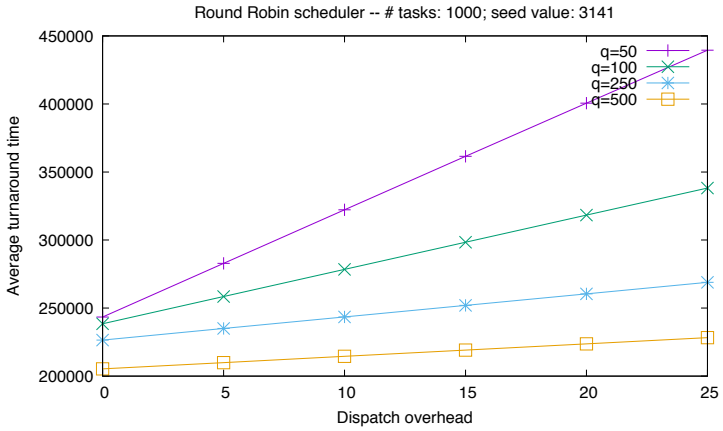
- *quantum lengths*: 50, 100, 250, and 500 ticks
- *dispatch costs*: 0, 5, 10, 15, 20 and 25 ticks

That is, for a particular number of tasks (at least 1000) and a particular seed (*e.g.*, perhaps some permutation of your student number) you must perform 20 different runs of the simulator, and for each run compute the average wait and average turnaround. You are then to plot the result in two separate graphs:

- Plot the average waiting times where the horizontal axis is the dispatch cost and the vertical axis is the average waiting time (both axes in ticks); each plotted line corresponds to a different quantum length.
- Same as above, but instead plot the average turnaround times.

Below are graphs I have generated using data from my own implementation of **`rrsim`**. (Note your graphs will probably vary somewhat depending upon the number of tasks you choose.)





You must write your own scripts to (1) generate the data, (2) compute statistics, and (3) arrange them for plotting. For the two graphs above I used a combination of a **`bash`** script, a Python3 script, and **`gnuplot`** to generate, aggregate and plot the data. However, you should use whatever tools you have at your disposal, but you must at least clearly describe such usage in your submission.

What you *must* submit:

- C source-code file named **`rrsim.c`** containing your solution to Task 1.
- If you have modified the linked-list routines, then you must submit both **`linkedlist.c`** and **`linkedlist.h`**.
- Two PDFs specifically named **`graph_waiting.pdf`** and **`graph_turnaround.pdf`** showing the graphs for Task 2.
- A markdown file named **`README.md`** with a description of the steps you took to complete Task 2.

What you *may* submit:

- Any scripts or programs (or both) used to complete Task 2.

Evaluation

Our grading scheme is relatively simple.

- "A" grade: An exceptional submission demonstrating creativity and initiative. The **`rrsim`** program runs without any problems, the presented graphs meet the description set out for task 2, and **`README.md`** clearly describes steps used to complete task 2.

- "B" grade: A submission completing the requirements of the assignment. The **`rrsim`** program runs without any problems, the presented graphs meet the description set out for task 2, and **`README.md`** clearly describes steps used to complete task 2.

- "C" grade: A submission completing most of the requirements of the assignment. The **`rrsim`** program runs with some problems, or the presented graphs do not meet the description set out for task 2, or **`README.md`** does not describes the steps used to complete task 2, or files are not correctly named.

- "D" grade: A serious attempt at completing requirements for the assignment. The **`rrsim`** program runs with serious problems, and task 2 cannot be evaluated.

- "F" grade: Either no submission is given, or the submission represents very little work.