CSC 343 Operating Systems, Spring 2020

Dr. Dale E. Parson, Assignment 5, Final Exam project, migrating preemptive round-robin CPU scheduling to preemptive short remaining time first (the preemptive variant of shortest job first), AND replacing a condition variable model with a counting semaphore model.
This assignment is due via **make turnitin** from the seme_srtf_final_2020 directory by **11:59 PM on Saturday May 9**. **<u>I will not accept any late assignments after 9 AM Sunday May 10</u>**.

The only time I will answer questions about this assignment is during our final exam period Monday, May 4, 2020, 11:00 a.m. – 1:00 p.m. **in our Zoom room.** I will hand this out and send email a few days ahead, and will only answer questions that clarify what you need to do, e.g., for unclear wording. Unlike previous assignments, I will not spell out or debug code beyond the extensive documentation comments and this handout. My projects are usually learning opportunities in which I will help as requested, but this assignment is a replacement for a regular exam because of our on-line constraints. It will count the same as the other projects, which is 20% of the semester grade each.

Perform the following steps to get my handout. You will need to test on machine mcgonagall as previously explained (**ssh mcgonagall** from acad). I usually edit in one window on acad and test I another on mcgonagall, so I can run **make graphs** on acad after my program compiles on mcgonagall to generate one or more graphical image files for the project state machine(s). **DO NOT FORGET TO ANSWER THE QUESTIONS IN README.txt – they are worth 40% of this assignment. Working code is worth 60% total, divided evenly across srtf.stm and Semaphore.stm per instructions below.**

```
cd  $HOME               # or start out in your login directory
cd  ./OpSys
cp  ~parson/OpSys/seme_srtf_final_2020.problem.zip  seme_srtf_final_2020.problem.zip
unzip seme_srtf_final_2020.problem.zip
cd ./seme_srtf_final_2020
make clean test                 # Works until you hit srtf
make testsrtf                   # How to test srtf.stm after you complete it.
make testSemaphore              # How to test Semaphore.stm after you complete it.
```

<u>PART I: Migrating round robin code to shortest remaining time first, worth 30% of assignment.</u>

Handout file srtf.stm is an identical pair of state machines (processor and thread as usual) to rr.stm, the round robin scheduler. You are starting off with round-robin code because, unlike sjf.stm (shortest job first), srtf.stm is preemptive. Unlike round robin that uses a fixed **quantum** as the max limit for each time spent in the running state, srtf.stm uses an estimate of its upcoming CPU burst time that is an average of its most recent call to sample() for cpu ticks, and its previous estimate. Thread state machine variable **predictedCPUticks** store the estimated ticks. In addition to replacing quantum as the max time spent in the running state per each execution in that state, srtf.stm also uses **predictedCPUticks** to order the readyQ priority min-queue. **The following edits take place in srtf.stm**.

1. Make sure to use the type of Queue used for sjf.stm.
2. Eliminate variable **quantum** from srtf.stm. Variable **predictedCPUticks** now sets the running preemption limit previously set by **quantum**.
3. Variable **predictedCPUticks** also supplies the priority number of enqueuing a thread in the readyQ.
4. Wherever srtf.stm assigns **cpuTicksB4IO = sample(…)**, immediately (on the next line of code) add this line of code:
   predictedCPUticks = int((cpuTicksB4IO+predictedCPUticks)/2);

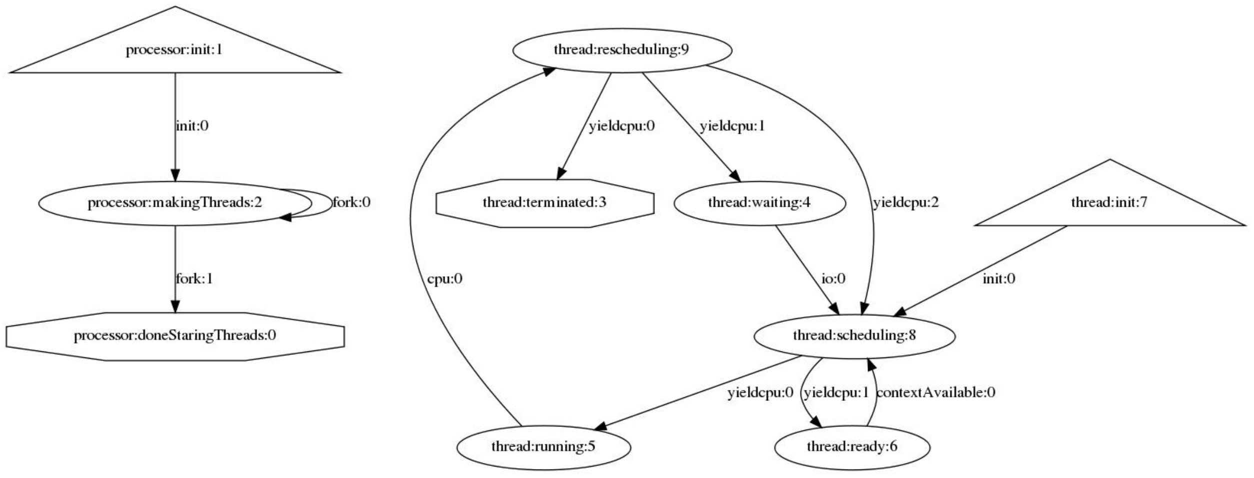
Make sure not to miss any current **cpuTicksB4IO = sample(…)** assignments, which must not change. This new assignment must come on the line after each reinitialization of **cpuTicksB4IO** from sample(…). Variable **predictedCPUticks** is now a running estimate that is the average of the most recent CPU burst and the previous estimate. Dividing by 2 represents an α (alpha) of 0.5 in this formula from slides 14 to 16 of http://faculty.kutztown.edu/parson/secure/osconcepts9th/ch6.ppt

CAN ONLY ESTIMATE THE LENGTH – SHOULD BE SIMILAR TO THE PREVIOUS ONE
THEN PICK PROCESS WITH SHORTEST PREDICTED NEXT CPU BURST.
CAN BE DONE BY USING THE LENGTH OF PREVIOUS CPU BURSTS, USING EXPONENTIAL AVERAGING. COMMONLY, A SET TO ½
PREEMPTIVE VERSION CALLED SHORTEST-REMAINING-TIME-FIRST.

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

5. Make sure to read and address every upper-case **STUDENT** comment in srtf.stm.



The state machine graph of srtf.stm above is identical to the graph of rr.stm, not that of sjf.stm, because of the transition from rescheduling to scheduling, without calling sample(…), when there remain ticks from a previous call to sample() to consume in variable **cpuTicksB4IO**.

Running **make clean testsrtf** tests this part of the project.

Running **make clean test** tests the whole thing.

Make sure to put your name at the top of any source file you modify, and add 1 or 2 lines of comment for each transition that you change.

PART II: Migrating condition variable code for the Producer/Consumer simulation to use 2 counting semaphores and a larger messageBuffer, eliminating the condition variables, worth 30% of assignment.

Handout file Semaphore.stm is an identical pair of state machines (processor and thread as usual) to Condvar.stm, the condition variable solution to assignment 2. Unlike using mutexes and condition variables, it is not necessary to loop repeatedly until a thread's condition is met (available space for a Producer or available data for a Consumer), because the counting semaphore lets into the critical section ONLY the number of Producers that can store data and/or the number of Consumers that can retrieve data for a 4-element messageBuffer. Recall that the atomic spin lock loops repeatedly without blocking; the mutex model blocks in a queue while trying to acquire the mutex (lock) held by another thread, but it still loops until its condition is met; the condition variable model blocks using distinct Producer and Consumer condition variable queues, but each thread still loops until its condition is met. Semaphore.stm, in contrast, lets a Producer into the critical section only when there is already room to produce, and it lets a Consumer into the critical section only when there is already product to consume. It combines condition testing without looping with the signaling mechanism of the condition variable stall states and wait queues.

Follow the detailed, upper-case STUDENT instructions in file Semaphore.stm, which we will go over in class during the May 4 final exam period.

Running **make clean testSemaphore** tests this part of the project.

Running **make clean test** tests the whole thing.

Make sure to put your name at the top of any source file you modify, and add 1 or 2 lines of comment for each transition that you change. You do not have to comment transitions that you delete.

Note the simplification in going from the graph of Condvar.stm:
http://acad.kutztown.edu/~parson/Condvar.jpg
to Semaphore.stm:
http://acad.kutztown.edu/~parson/Semaphore.jpg

After understanding the STUDENT instructions in Semaphore.stm, note the following Java library classes for concrete examples:

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html#Semaphore-int-boolean-
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html#acquire--
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html#release--

using constructor with arguments of (4,true), and acquire() and release() for obtaining and releasing one permit from the semaphore's counter.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html#getAndAccumulate-int-java.util.function.IntBinaryOperator-

for getting the semaphore's atomicBufferPointer into messageBuffer and atomically updating it via incrementing modulo the length of messageBuffer, where messageBuffer itself is an atomic array:

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicIntegerArray.html

Running **make clean test** tests the whole thing.

Make sure to put your name at the top of any source file you modify, and add 1 or 2 lines of comment for each transition that you change. You do not have to comment transitions that you delete.

PART III: Answer all questions in file README.txt[1] in this project directory, worth 40% of the assignment, before making a final run of **make clean test** and then **make turnitin**.

PLEASE NOTE ON README.txt Q5: MEAN_TURNAROUNDTIME is the average time in ticks for all thread state machines from their start out of the init state through their termination in an accept state.

This assignment is due via **make turnitin** from the seme_srtf_final_2020 directory by **11:59 PM on Saturday May 9**. **<u>I will not accept any late assignments after 9 AM Sunday May 10</u>**.

[1] **PLEASE NOTE ON README.txt Q5**: MEAN_TURNAROUNDTIME is the average time in ticks for all thread state machines from their start out of the init state through their termination in an accept state.