CSC 343 – Operating Systems, Fall 2020, Assignment 5, due December 10

This assignment is **DUE By 11:59 PM on Thursday December 10, 2020 via make turnitin on mcgonagall or acad. The standard 10% per day deduction for late assignments applies.**

To get the starting code for the project please follow these steps after logging into acad:

```
cd                          # This goes to your login directory.
mkdir ./OpSys               # should already be there; no error if it says so
cd    ./OpSys
cp    ~parson/OpSys/SjfSrtfFall2020.problem.zip  SjfSrtfFall2020.problem.zip
unzip   SjfSrtfFall2020.problem.zip
cd    ./SjfSrtfFall2020
ssh –l YOURLOGIN   mcgonagall                 # -l is the lower-case letter ell
cd    ./OpSys/SjfSrtfFall2020
```

All of your programming and testing must occur on multiprocessor **mcgonagall**. All other work must occur within your OpSys/SjfSrtfFall2020 directory on mcgonagall. Make sure to read and understand the MEAN versus MEDIAN semester grading plan documented at:
https://faculty.kutztown.edu/parson/fall2020/MeanMedianFall2020.txt .

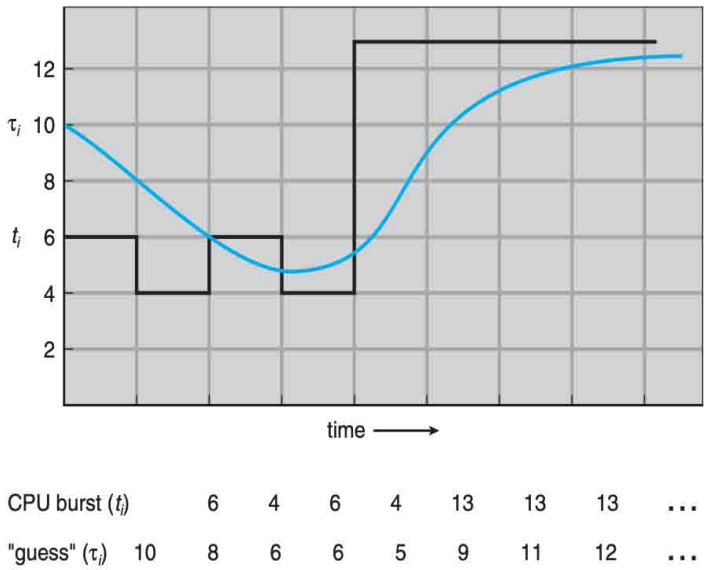
This assignment consists of coding two enhancements to the sjf.stm, shortest-job first CPU (a.k.a. context) scheduler of Assignment 3. State machine sjf.stm is an approximate model that uses perfect foreknowledge of each upcoming CPU burst time by using that actual burst time to schedule threads in the readyQ.

processor.readyQ.enq(thread, **cpuTicksB4IO**);

Such foreknowledge is unrealistic. This assignment forms an estimate of the upcoming CPU burst time from a weighted sum of the most recent CPU burst time (just in the past) and the previous such estimate. Here is the formula from the textbook chapter on CPU Scheduling to estimate each upcoming burst.

- Can only estimate the length – should be similar to the previous one
  - Then pick thread with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  - t_n = actual length of n^{th} CPU burst (our **cpuTicksB4IO** at time of assigning it from sample(…))
  - T_{n+1} = predicted value for next CPU burst (I call this variable **estimate** below)
  - α, where $0 <= \alpha <= 1$ (I call this variable **alpha** below)
  - Define $T_{n+1} = \alpha\ t_n + (1 - \alpha)\ T_n$
- Commonly, α set to ½ (thereby providing equal weight to most recent burst and previous estimate)
- Preemptive version called **shortest-remaining-time-first**

The textbook shows this graph of actual CPU burst times (the Manhattan, squared-off time series) versus the estimate T (the curved time series that lags rapid transitions in actual bursts).



Actual bursts t (cpuTicksB4IO) in black and pre-estimated bursts T (estimate) in blue

Here are the above variables using variable names from the assignment's state machines.

- Can only estimate the length – should be similar to the previous one
  - Then pick thread with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  - **cpuTicksB4IO** = actual length of n^{th} CPU burst as assigned from sample(…)
  - **estimate** = predicted value for next CPU burst
  - **alpha** weight is 0.5 as illustrated by the textbook
  - Define **estimate** = **alpha** * previous **cpuTicksB4IO** + (1 – **alpha**) * previous **estimate**
- Commonly, α set to ½ (thereby providing equal weight to most recent burst and previous estimate)
- Preemptive version called **shortest-remaining-time-first**
- We will initialize **cpuTicksB4IO** and **estimate** to 125 in these state machines, and alpha to 0.5, thereby providing a stable starting point for comparisons.

Automated testing via **make clean test** is similar to previous assignments. I am supply my solutions to fcfs.stm, sjf.stm, and rr.stm schedulers from Assignment 3.

STEP 1 35% of project: Start with the following copy command on the mcgonagall command line:

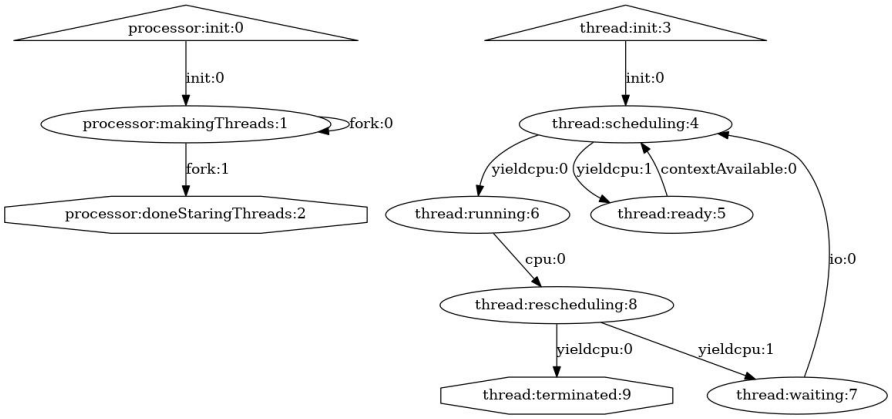
cp sjf.stm sjfEstimate.stm

Edit model **sjfEstimate.stm**, making the following changes.

1. Add variable **estimate** initialized to 125, variable **alpha** initialized to 0.5, and make sure that **cpuTicksB4IO** is initialized to 125. These initializations occur where the variables are declared at the top of the thread state machine, not in transitions.
2. Compute **estimate** as follows immediately **BEFORE** each assignment into **cpuTicksB4IO** from sample(…). This model predicts the next estimate based on the previous CPU burst time and the previous estimate, since this algorithm does not unrealistically predict the future.
   estimate = round((alpha * cpuTicksB4IO) + ((1.0 - alpha) * estimate));

3. Use the **estimate** as the priority argument in each call to processor.readyQ.enq(…), replacing **cpuTicksB4IO** from sjf.stm.
4. Update documentation comments in **sjfEstimate.stm** at the top and wherever you made changes.
5. Running **make testsjfEstimate** should now pass.

Here is the STM graph for **sjfEstimate.stm**, identical to sjf.stm's and fcfs.stm's graph.
https://kuvapcsitrd01.kutztown.edu/~parson/sjfEstimate.jpg



sjfEstimate.jpg

STEP 2 35% of project: Start with the following copy command on the mcgonagall command line:

cp rr.stm sjfPreempt.stm

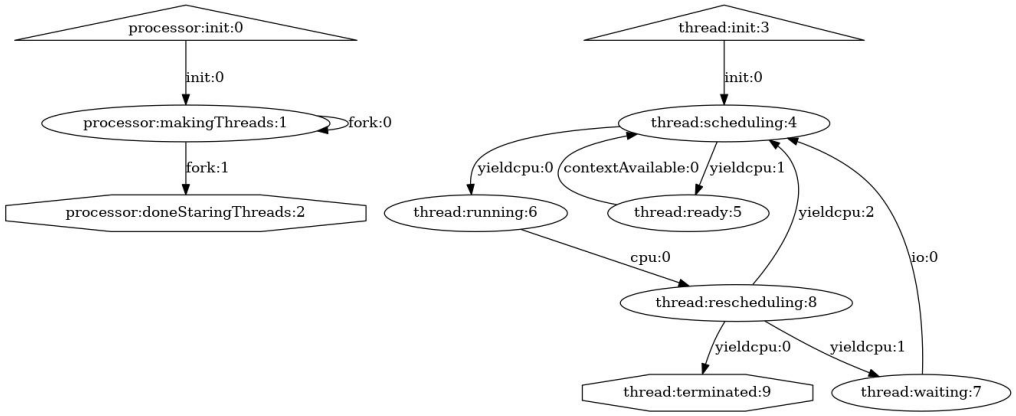
Edit model **sjfPreempt.stm**, making the following changes. We are starting with rr.stm because like rr.stm, **sjfPreempt.stm** is preemptive. It includes rr's added transition from state rescheduling to scheduling when there are remaining **tickstodefer**.

1. Change the name of variable **quantum** to **estimate** everywhere it occurs, initialized to 125. Add variable **alpha** initialized to 0.5, and make sure that **cpuTicksB4IO** is initialized to 125. These initializations occur where the variables are declared at the top of the thread state machine, not in transitions.
2. Change the constructor call processor.readyQ = Queue(…) to make it a priority queue as it is in sjf.stm and sjfEstimate.stm.
3. Compute **estimate** (formerly constant **quantum**) as follows immediately **BEFORE** each assignment into **cpuTicksB4IO** from sample(…). This model predicts the next estimate based on the previous CPU burst time and the previous estimate, since this algorithm does not unrealistically predict the future.

   estimate = round((alpha * cpuTicksB4IO) + ((1.0 - alpha) * estimate));

   This is the same formula as in sjfEstimate.stm.
   Note that the call to cpu(tickstorun) at the end of transition scheduling -> running uses the tickstorun computed from the minimum of the upcoming cpuTicksB4IO and the estimate based on the weighted sum of the previous cpuTicksB4IO and the previous estimate. Using cpu(tickstorun) models preemption.
4. Use the **estimate** as the priority argument in each call to processor.readyQ.enq(…) as in sjfEstimate.stm.
5. Update documentation comments in **sjfPreempt.stm** at the top and wherever you made changes.
6. Running **make testsjfPreempt** should now pass.

Here is the STM graph for **sjfPreempt.stm**, identical to rr.stm's graph.



sjfPreempt.jpg

STEP 3 30% of project: Answer the 3 questions in file README.txt.

After completing the above steps, run **make clean test** one last time to ensure everything works, then run **make turnitin** as before by the assignment deadline.

If you get an error at run time with codeTable index like this:

```
exec(__codeTable__[20],globals,locals)
```

You can run decode.py like this:

$ **python decode.py rr_fifopage.py 20**

```
__codeTable__[20] = compile('pcb.victimQueue = Queue(ispriority=False)','nofile','exec'),
```