CSC 343 – Operating Systems, Spring 2024, Assignment 3, due Thursday April 11

This assignment is due by 11:59 PM on Thursday April 11 via **make turnitin** as explained below.

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/stm3CPUschedSP2024.problem.zip  stm3CPUschedSP2024.problem.zip
unzip   stm3CPUschedSP2024.problem.zip
cd    ./stm3CPUschedSP2024
ssh –l YOURLOGIN   mcgonagall          # -l is the lower-case letter ell
cd    ./ OpSys/stm3CPUschedSP2024
```

This is a completely redesigned project modeling scheduling of threads onto hardware processing units (contexts). All of your programming and testing must occur on multiprocessor **mcgonagall**.

In this assignment **I am supplying a round robin preemptive scheduler** in file **rr.stm** per slides 20-23 in the textbook slides for Chapter 6. You can run **make testrr** to test it. It passes this test as handed out. Figure 1 is the state diagram for **rr.stm**. and **srtf.stm** discussed below. Both are preemptive context schedulers.

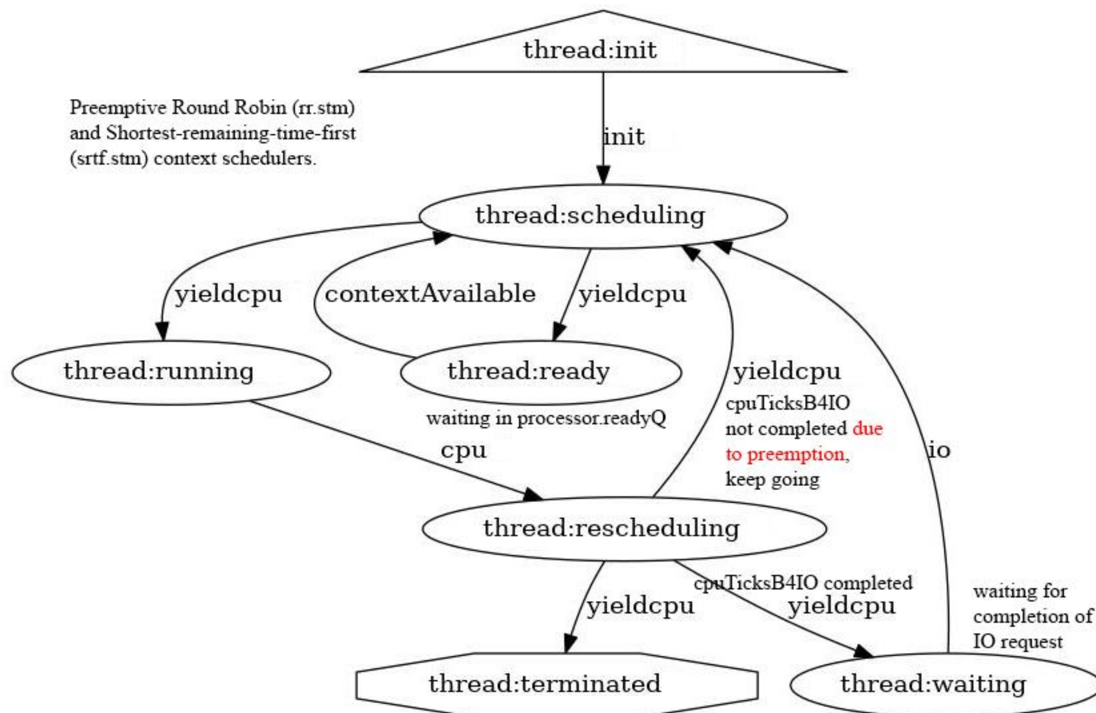

Figure 1: Preemptive context scheduling model for round robin and shortest remaining time first

<u>STUDENT PART 1 (5% of assignment): Filling out initial comments in rr.stm.</u>

Fill in the blanks in the comment block at the top of **rr.stm**. Read through and understand transitions tagged with **STUDENT** comments. You have no changes to make to **rr.stm** beyond the comment block at the top.

Run **make testrr** one more time to make sure it still works after editing.

As before, if you get a run-time error that refers to the codeTable like this:

```
  File "rr.py", line 378, in run
    exec(__codeTable__[15],globals,locals)
```

Use the supplied decode.py utility to find the line of offending source code:

$ **./decode.py rr.py 15**

__codeTable__[15] = compile(**'invalid1 = invalid2'**,'nofile','exec'),

STUDENT PART 2 (20% of assignment): First-come first-served.

FCFS is outlined in slides 10 and 11 in the [textbook slides for Chapter 6].

2a. Enter this command:

cp rr.stm fcfs.stm

2b. FCFS is non-preemptive. Here are the changes to make to **fcfs.stm** after the above copy:

2b1: Find every assignment into **tickstorun** and set it equal to **cpuTicksB4IO**. Since FCFS is non-preemptive, each CPU burst runs to completion without interruption. As a testing step you can set **tickstodefer** to the value 0 (none deferred) and then run **make testfcfs** which should pass.

2b2: Since **tickstodefer** is now 0, remove variable **tickstodefer** and any reference to it, including its presence in transition guard conditions. Keep other tests found in those guard conditions, but remove tests on **tickstodefer**. There is one transition that tests for **tickstodefer > 0** in its guard condition. Since **tickstodefer** is no longer used in this non-preemptive FCFS model, remove that transition entirely. Run **make testfcfs** which should pass. Figure 2 shows the state diagram after this change with the transition removed.

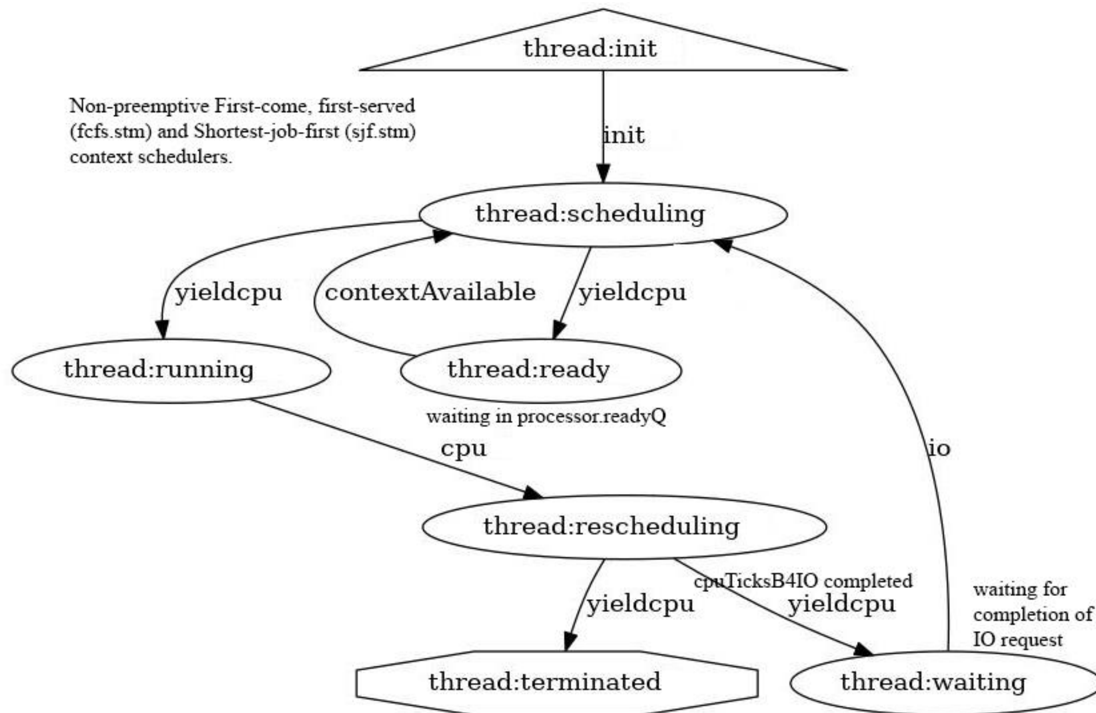

Figure 2: Non-preemptive context scheduling for FCFS, and SJF, and SJFEST in the next steps

2b3. Update comments at the top to reflect the FCFS algorithm and update comments for every transition that you changed or that is tagged with a STUDENT comment. Run **make testfcfs** which should pass.

STUDENT PART 3 (20% of assignment): Shortest job first.

SJF is outlined in slides 12 and 13 in the textbook slides for Chapter 6. It is non-preemptive, matching Figure 2.

3a. Enter this command:

cp fcfs.stm sjf.stm

3b. SJF is also non-preemptive. Here are the changes to make to **sjf.stm** after the above copy:

3b1. Find the constructor call for the **processor.readyQ** in the processor state machine and change the argument to the **Queue** constructor from **False** to **True**. SJF uses a priority min-queue sorted on **cpuTicksB4IO** to schedule threads with minimal CPU burst times at the front of **processor.readyQ**. It "magically" knows each CPU burst time before the bursting thread runs, which is useful as a theoretical, optimal queuing algorithm. Save this change, try running **make testsjf**, and note the ValueError message line.

3b2. Find the **processor.readyQ.enq(thread)** call in sjf.stm for enqueuing a ready thread and add a second argument of **cpuTicksB4IO** to that enq call. This argument sorts threads with smaller CPU burst times to the front of the **processor.readyQ**. Running **make testsjf** now works.

3b3. Update comments at the top to reflect the SJF algorithm and update comments for every transition that you changed or that is tagged with a STUDENT comment. Run **make testsjf** which should pass.

STUDENT PART 4 (20% of assignment): Shortest job first with quantum estimation of CPU burst time.

SJFEST is outlined in slides 14-16 in the [textbook slides for Chapter 6](). It is non-preemptive, matching Figure 2.

4a. Enter this command:

cp sjf.stm sjfEst.stm

4b. SJF is also non-preemptive. Here are the changes to make to **sjfEst.stm** after the above copy:

4b1. This model starts off with the quantum of 125 in the variable declarations of the thread state machine. It now uses **quantum** as the scheduling priority of the processor.readyQ.enq call, replacing **cpuTicksB4IO**. If you run make testsjfEst you will see this diff error:

WARNING, **MEAN_ready** = **419.953125** in sjfEst_crunch.py, = **317.1449275362319** in sjfEst_crunch.ref at 15.0% tolerance.

Using the quantum estimate should reduce average time in the readyQ down from ~420 ticks to ~317 ticks, but first there is another set of code changes to make.

4b2. In the variable initializations of the thread state machine, initialize variable **alpha** to the value 0.5. Alpha is the weight applied to the most recent CPU burst, and (1.0 - alpha) to the previous estimate, summed to provide the next estimate for **readyQ.enq** priority scheduling. Next, in the bottom **waiting -> scheduling** transition**,** update the value in estimate variable **quantum** <u>**BEFORE**</u> assigning a new value into **cpuTicksB4IO**, using this statement:

```
quantum = round((alpha * cpuTicksB4IO) + ((1.0 - alpha) * quantum));
```

Running **make testsjfEst** should now pass with this corrected error:

OK: MEAN_ready at 15.0% tolerance = 317.14

Rounding the result of the quantum estimation update is necessary for scheduling an integer number of ticks.

4b3. Update comments at the top to reflect the SJFEST algorithm and update comments for every transition that you changed or that is tagged with a STUDENT comment. Run **make testsjfEst** which should pass.

Figures 3 and 4 shows estimated CPU burst times using an alpha of 0.5, 0.25, and 0.75 for this model. Note that the higher the alpha value, the closer the estimate tracks actual CPU burst volatility. Lower alpha values smooth the estimate curves. We will go over this in class. You do not need to code alpha values other than 0.5.

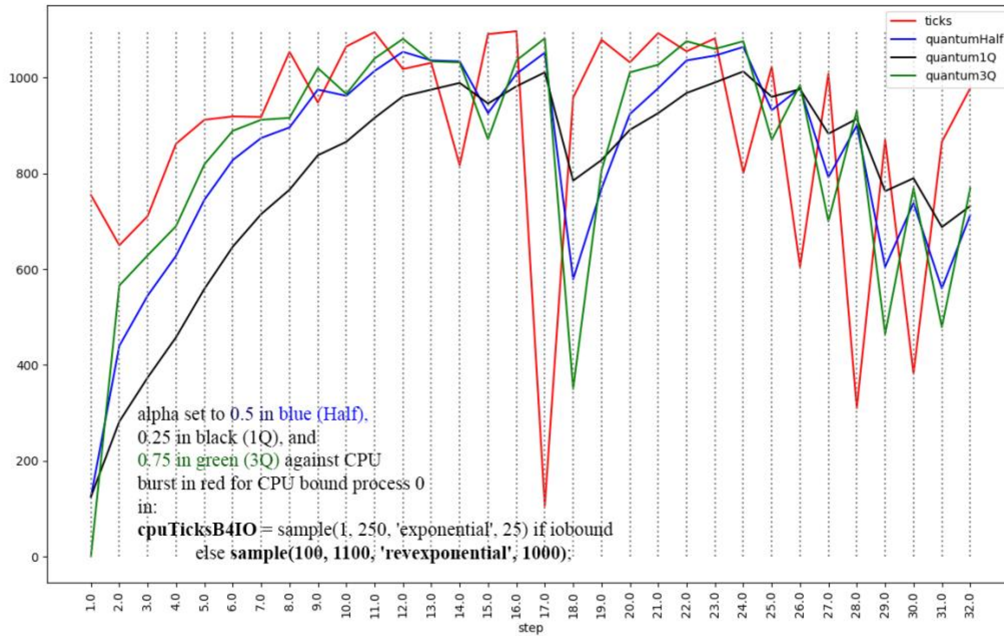

Figure 3: Actual and estimated CPU burst times for CPU bound process 0 in sjfEst.stm

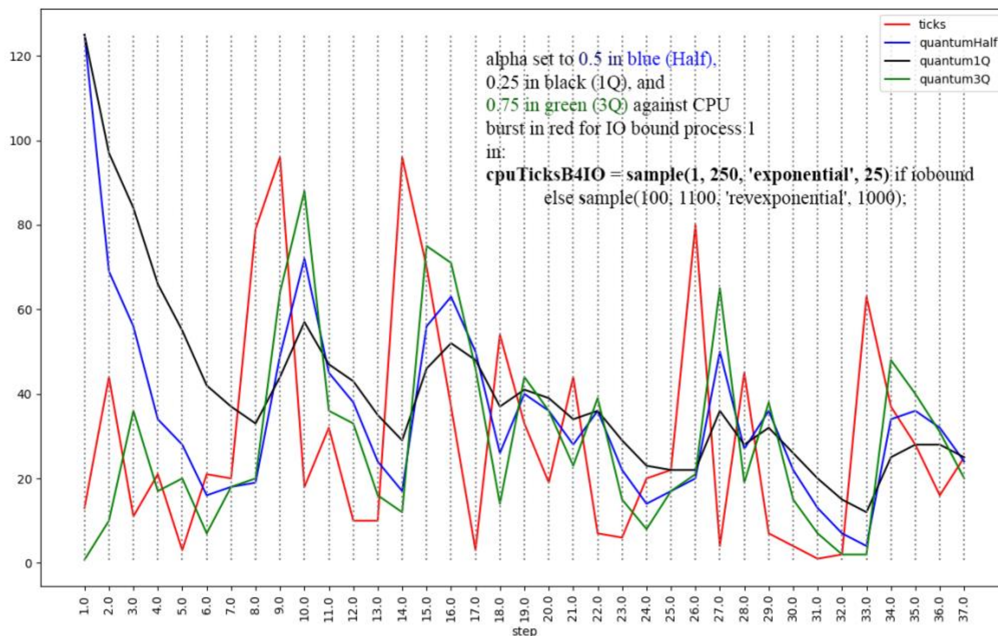

Figure 4: Actual and estimated CPU burst times for IO bound process 1 in sjfEst.stm

Compare these curves to slide 15 in the textbook slides for Chapter 6.

STUDENT PART 5 (20% of assignment): Shortest remaining time first with estimation of CPU burst time.

SRTF is outlined in slides 17-19 in the textbook slides for Chapter 6. It is preemptive, matching Figure 1.

5a. Enter this command:

cp rr.stm srtf.stm

We are starting with RR because, like RR, SRTF is preemptive, matching Figure 1. However, SRTF must add the priority Queue and quantum estimation of SJFEST with the **alpha** value of 0.5, and it must use the **quantum** estimate not only for readyQ.enq thread scheduling, but also for preemption by assigning into variable **tickstodefer**; **rr.stm** supplies the correct assignments into variables **tickstorun** and **tickstodefer** for modeling preemption.

5b. Perform the following steps already taken for SJF and SJFEST into **srtf.stm** as copied from **rr.stm**.

5b1. Find the constructor call for the **processor.readyQ** in the processor state machine and change the argument to the **Queue** constructor from **False** to **True** as in step 3b1. SRTF uses a min priority readyQ.

5b2. Find the **processor.readyQ.enq(thread)** call in srtf.stm for enqueuing a ready thread and add a second argument of **quantum** to that enq call as in sjfEst.stm.

5b3. In the variable initializations of the thread state machine, initialize variable **alpha** to the value 0.5 as in step 4b2. Next, in the bottom **waiting -> scheduling** transition**,** update the value in estimate variable **quantum** BEFORE assigning a new value into cpuTicksB4IO, using this statement:

```
quantum = round((alpha * cpuTicksB4IO) + ((1.0 - alpha) * quantum));
```

After these changes to **srtf.stm**, **make testsrtf** should pass.

5b4. Update comments at the top to reflect the SRTF algorithm and update comments for every transition that you changed or that is tagged with a STUDENT comment. Run **make testsrtf** which should pass. The full **make test** should pass at this point.

STUDENT PART 6 (15% of assignment): Answer the questions in README.txt.

Once you have answered these questions, run **make test** one last time, then **make turnitin** by the project deadline. There is the usual 10% per day late penalty, with a grade of 0% once I go over the solution in the following noon class.