CSC 343 – Operating Systems, Fall 2020, Assignment 3, due November 12

This assignment is **DUE By 11:59 PM on Thursday November 12, 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/ContextSchedFall2020.problem.zip  ContextSchedFall2020.problem.zip
unzip   ContextSchedFall2020.problem.zip
cd    ./ContextSchedFall2020
ssh –l YOURLOGIN   mcgonagall                 # -l is the lower-case letter ell
cd    ./OpSys/ContextSchedFall2020
```

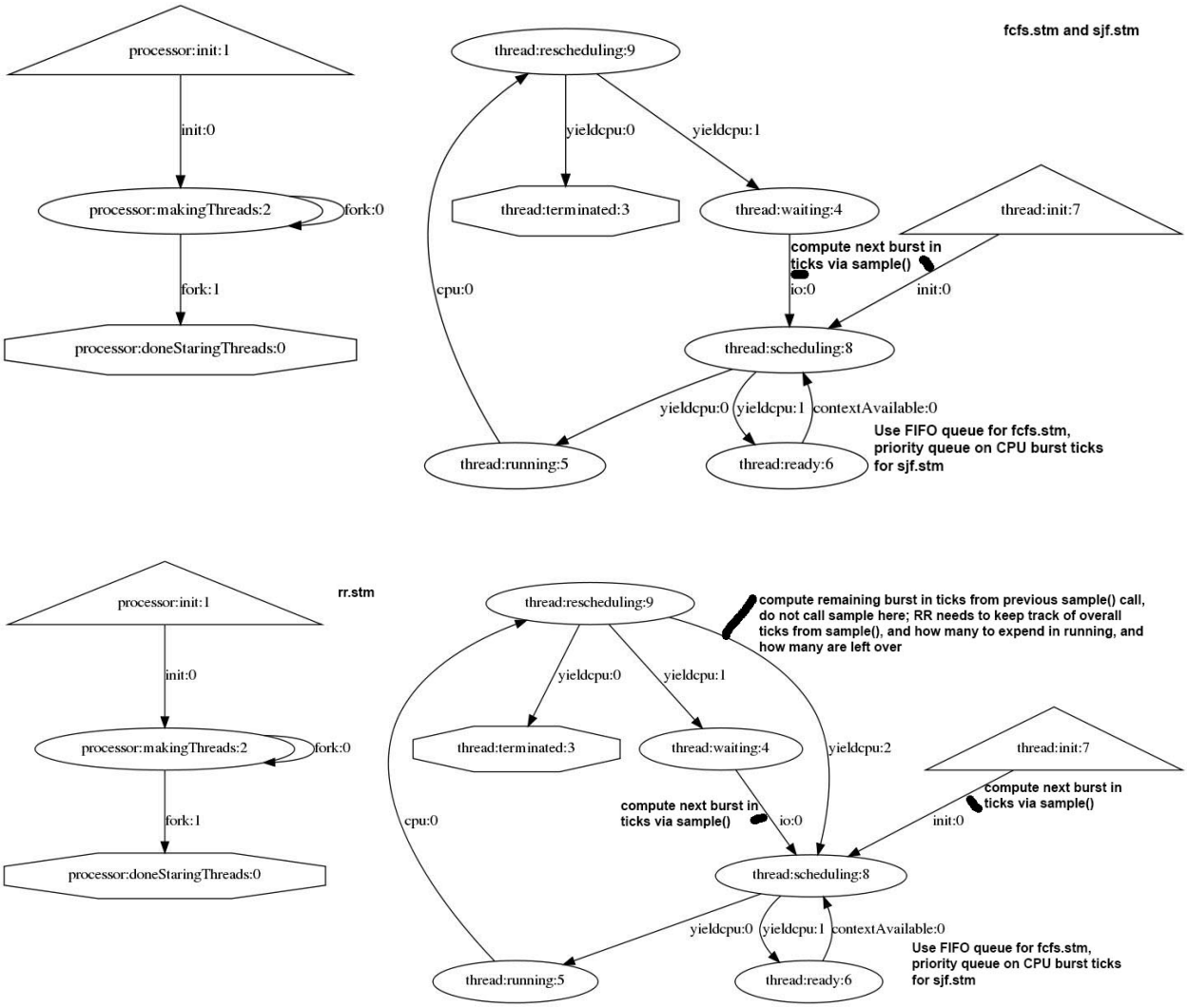
All of your programming and testing must occur on multiprocessor **mcgonagall**. All other work must occur within your OpSys/ContextSchedFall2020 directory on mcgonagall.

In this assignment I am supplying a first-come first-served, non-preemptive scheduler in file fcfs.stm. You can run **make testfcfs** to test it. I have started drafting the file sjf.stm for the shortest-job first scheduler, which schedules a thread into the readyq using the thread's cpu burst time in cpuTicksB4IO, which the shortest time having the earlier priority in the min-queue (readyq). After you add your code you can run **make testsjf** to test it. Finally, you must complete the preemptive round-robin scheduler in rr.stm, which you can test using **make testrr**. When everything runs, make sure your name is added at the top of your source files, and you have added a brief comment for every transition that you change or add. Perform **make clean test** one last time, and then **make turnitin** by the due date deadline. There are notes about the scheduling algorithms in the handout STM files. We will go over them in class.

SJF is worth 50% of the project grade, and RR is worth the other 50%. I will give partial credit for solutions with algorithm bugs, but they must be able to compile.

My state diagrams are here:

http://acad.kutztown.edu/~parson/fcfs.jpg
http://acad.kutztown.edu/~parson/sjf.jpg creates the same graph as fcfs.stm.
http://acad.kutztown.edu/~parson/rr.jpg has a transition going from rescheduling to scheduling, bypassing state **waiting** (for IO completion), when the thread still has ticks left over from the most recent sample() call that it has not yet expended. Since fcfs and sjf are always non-preemptive, threads always expend all sampled()d cpuTicksB4IO as soon they get a CPU; rr, on the other hand, may expend only up to quantum ticks. Any leftover ticks remaining require going from running -> rescheduling -> scheduling to expend some more of those ticks.



Here is what a successful test run looks like. It is FCFS, which is already done and working:

$ make testfcfs

COMPILING fcfs
/bin/bash -c "PYTHONPATH=/home/kutztown.edu/parson/OpSys:.:.. /usr/local/bin/python3.7 /home/kutztown.edu/parson/OpSys/state2codeV17/State2CodeParser.py fcfs.stm fcfs.dot fcfs.py CSC343Compile CSC343Compile"
COMPILING COMPLETED
SIMULATING (TESTING) fcfs
/bin/bash -c "PYTHONPATH=/home/kutztown.edu/parson/OpSys:.:.. STMLOGDIR=/home/kutztown.edu/parson/tmp time /usr/local/bin/python3.7 fcfs.py 2 4 110000 12345 2"
MSG cmd line: ['fcfs.py', '2', '4', '110000', '12345', '2'], usage USAGE: python THISFILE.py NUMCONTEXTS NUMFASTIO SIMTIME SEED|None LOGLEVEL

Scheduler exiting at time 103914 within time limit 110000, simulation has finished.
0.15user 0.02system 0:00.30elapsed 58%CPU (0avgtext+0avgdata 10672maxresident)k

```
0inputs+960outputs (0major+5137minor)pagefaults 0swaps
/bin/bash -c "PYTHONPATH=/home/kutztown.edu/parson/OpSys:.:.. /usr/local/bin/python3.7
crunchlog.py fcfs.log"

DIFFing fcfs_crunch.py fcfs_crunch.ref
OK: MEAN_running at 15.0% tolerance.
OK: MEAN_ready at 15.0% tolerance.
OK: MEAN_waiting at 15.0% tolerance.
OK: MEAN_TURNAROUNDTIME at 15.0% tolerance.
OK: MAX_running at 15.0% tolerance.
OK: MAX_ready at 15.0% tolerance.
OK: MAX_waiting at 15.0% tolerance.
OK: MAX_TURNAROUNDTIME at 15.0% tolerance.
OK: MIN_running at 15.0% tolerance.
OK: MIN_ready at 15.0% tolerance.
OK: MIN_waiting at 15.0% tolerance.
OK: MIN_TURNAROUNDTIME at 15.0% tolerance.

TESTING COMPLETED
```

The **make testsjf** takes similar time to **make testfcfs**; **make testrr** takes marginally longer.

Each test run produces a log file (fcfs.log, sjf.log and rr.log).

Automated testing via **make clean test** is similar to assignment 2. Simulation times in ticks for critical states of the algorithm are checked for consistency with the expected times, to with a 15% allowable margin of difference. These are the measures checked for consistency:

```
cat diffset.py
# ContextSchedFall2020/diffset.py --
# set of simulation properties to test after
# a simulation run. See crunchlog.py

# Map the property to be checked against its (TOLERANCE, RAWTOLERANCE),
# where TOLERANCE is a percatage as a fraction, and RAWTOLERANCE
# is the minimum difference between the simulation value and the
# reference value for the property required to trigger an error.
DIFFMAP = {
   'MEAN_running' :              (.15, 10),
   'MEAN_ready' :                (.15, 10),
   'MEAN_waiting' :              (.15, 10),
   'MEAN_TURNAROUNDTIME' :       (.15, 10),
   'MAX_running' :               (.15, 10),
   'MAX_ready' :                 (.15, 10),
   'MAX_waiting' :               (.15, 10),
   'MAX_TURNAROUNDTIME' :        (.15, 10),
   'MIN_running' :               (.15, 10),
   'MIN_ready' :                 (.15, 10),
   'MIN_waiting' :               (.15, 10),
   'MIN_TURNAROUNDTIME' :        (.15, 10),
}
```

Testing simulated cpu-time analysis for class discussion:

fcfs_crunch.ref:MEAN_running=395.9563953488372
rr_crunch.ref:MEAN_running=101.24087591240875
sjf_crunch.ref:MEAN_running=377.4913294797688

fcfs_crunch.ref:MIN_running=1
rr_crunch.ref:MIN_running=1
sjf_crunch.ref:MIN_running=1

fcfs_crunch.ref:MAX_running=1098
rr_crunch.ref:MAX_running=125
sjf_crunch.ref:MAX_running=1098

fcfs_crunch.ref:MEAN_ready=419.953125
rr_crunch.ref:MEAN_ready=78.75062972292191
sjf_crunch.ref:MEAN_ready=264.5652173913044

fcfs_crunch.ref:MIN_ready=2
rr_crunch.ref:MIN_ready=1
sjf_crunch.ref:MIN_ready=3

fcfs_crunch.ref:MAX_ready=2081
rr_crunch.ref:MAX_ready=282
sjf_crunch.ref:MAX_ready=1878

fcfs_crunch.ref:MEAN_waiting=2481.991017964072
rr_crunch.ref:MEAN_waiting=2586.665671641791
sjf_crunch.ref:MEAN_waiting=2544.7261904761904

fcfs_crunch.ref:MIN_waiting=500
rr_crunch.ref:MIN_waiting=500
sjf_crunch.ref:MIN_waiting=500

fcfs_crunch.ref:MAX_waiting=7284
rr_crunch.ref:MAX_waiting=6938
sjf_crunch.ref:MAX_waiting=6775

fcfs_crunch.ref:MEAN_TURNAROUNDTIME=101895.8
rr_crunch.ref:MEAN_TURNAROUNDTIME=102263.7
sjf_crunch.ref:MEAN_TURNAROUNDTIME=102216.0

fcfs_crunch.ref:MIN_TURNAROUNDTIME=100040
rr_crunch.ref:MIN_TURNAROUNDTIME=100211
sjf_crunch.ref:MIN_TURNAROUNDTIME=100092

fcfs_crunch.ref:MAX_TURNAROUNDTIME=103906
rr_crunch.ref:MAX_TURNAROUNDTIME=105252
sjf_crunch.ref:MAX_TURNAROUNDTIME=104397

fcfs_crunch.ref:MEAN_waiting=2481.99101796
sjf_crunch.ref:MEAN_waiting=2544.72619048
rr_crunch.ref:MEAN_waiting=2586.66567164

fcfs_crunch.ref:MIN_waiting=500

sjf_crunch.ref:MIN_waiting=500
rr_crunch.ref:MIN_waiting=500

fcfs_crunch.ref:MAX_waiting=7284
sjf_crunch.ref:MAX_waiting=6775
rr_crunch.ref:MAX_waiting=6938

fcfs_crunch.ref:MEAN_TURNAROUNDTIME=101895.8
sjf_crunch.ref:MEAN_TURNAROUNDTIME=102216
rr_crunch.ref:MEAN_TURNAROUNDTIME=102263.7

fcfs_crunch.ref:MIN_TURNAROUNDTIME=100040
sjf_crunch.ref:MIN_TURNAROUNDTIME=100092
rr_crunch.ref:MIN_TURNAROUNDTIME=100211

fcfs_crunch.ref:MAX_TURNAROUNDTIME=103906
sjf_crunch.ref:MAX_TURNAROUNDTIME=104397
rr_crunch.ref:MAX_TURNAROUNDTIME=105252

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

```
    exec(__codeTable__[21],globals,locals)
```

You can run decode.py like this:

/usr/bin/python decode.py sjf.py 21

```
__codeTable__[21] = compile('cpu(cpuTicksB4IO)','nofile','exec'),
```

That shows you the line of code that blew up during simulation.