CSCI-1200 Data Structures — Fall 2025
Homework 10 — Priority Queue Scheduling

In this homework we will revisit the autograding jobs scheduler from Homework 4, with several small modifications. You will use the STL `priority queue` data structure to improve the performance of this simulation. Here is an example of the input file, `sample_jobs.txt`:

```
23:00:06.078 csci1100 hw4   waterz      9.267  10
23:00:12.027 csci1100 hw4   jarvim      9.148  10
23:00:13.064 csci2600 a3    mannt      34.255   8
23:00:14.069 csci1100 hw4   guerrh      9.025  10
23:00:22.091 csci1200 hw6   schmio     24.044   4
23:00:23.060 csci1100 hw4   sotoy      10.453  10
23:00:25.071 csci1100 hw4   morrih     11.247  10
23:00:27.030 csci1100 hw4   meltoo      9.813  10
23:00:28.078 csci1200 hw6   leachm     26.428   4
23:00:29.073 csci2600 a3    schroq     44.445   8
```

Each row of the input includes the upload time, course, assignment, student user id, running time (in seconds), and the niceness value. *NOTE:* Unlike Homework 4, each job will simply run on a single processor, and we assume that the expected and actual running times are the same. Additionally, the precision of all timing measurements for this assignment is in milliseconds (1/1000th of a second).

What is the *niceness value*? On Unix/Linux operating systems, each program can specify its priority relative to all other programs. Niceness values are integers ranging from -20 to +20. Typical interactive user programs/jobs will have their niceness / priority set at 0 by default. Background jobs that are less important will have positive niceness values > 0 indicating they can be delayed if the system is busy handling more time-sensitive tasks. The operating system will then be able to prioritize jobs with lower niceness values – that is, jobs with lower niceness values will be allowed both more immediate access to and also a greater proportion of the overall computing resources of the system.

Initial Simulation in the Provided Code

We have modified the provided code from Homework 4. The program takes in two required arguments: the input file above and the number of processors for the simulation. The program also accepts a number of optional arguments: the output visualization file, the output log file, a simulation timestep (in seconds), and a boolean niceness flag. Here is a sample command line showing how to run the program:

```
./simulate.out sample_jobs.txt 4 --vis sample_5.0_vis.txt --log sample_5.0_log.txt --timestep 5.0
```

This is the visualization file for the simulation produced with the above command line:

```
timestamp    | processor 0          | processor 1          | processor 2          | processor 3          | queue
23:00:00.000 |                      |                      |                      |                      |     0
23:00:05.000 |                      |                      |                      |                      |     0
23:00:10.000 | csci1100_hw4_waterz  |                      |                      |                      |     0
23:00:15.000 | csci1100_hw4_waterz  | csci1100_hw4_jarvim  | csci2600_a3_mannt    | csci1100_hw4_guerrh  |     0
23:00:20.000 |                      | csci1100_hw4_jarvim  | csci2600_a3_mannt    | csci1100_hw4_guerrh  |     0
23:00:25.000 | csci1200_hw6_schmio  | csci1100_hw4_sotoy   | csci2600_a3_mannt    |                      |     0
23:00:30.000 | csci1200_hw6_schmio  | csci1100_hw4_sotoy   | csci2600_a3_mannt    | csci1100_hw4_morrih  |     3
23:00:35.000 | csci1200_hw6_schmio  | csci1100_hw4_sotoy   | csci2600_a3_mannt    | csci1100_hw4_morrih  |     3
23:00:40.000 | csci1200_hw6_schmio  | csci1100_hw4_meltoo  | csci2600_a3_mannt    | csci1100_hw4_morrih  |     2
23:00:45.000 | csci1200_hw6_schmio  | csci1100_hw4_meltoo  | csci2600_a3_mannt    | csci1200_hw6_leachm  |     1
23:00:50.000 | csci2600_a3_schroq   |                      |                      | csci1200_hw6_leachm  |     0
23:00:55.000 | csci2600_a3_schroq   |                      |                      | csci1200_hw6_leachm  |     0
23:01:00.000 | csci2600_a3_schroq   |                      |                      | csci1200_hw6_leachm  |     0
23:01:05.000 | csci2600_a3_schroq   |                      |                      | csci1200_hw6_leachm  |     0
23:01:10.000 | csci2600_a3_schroq   |                      |                      | csci1200_hw6_leachm  |     0
23:01:15.000 | csci2600_a3_schroq   |                      |                      |                      |     0
23:01:20.000 | csci2600_a3_schroq   |                      |                      |                      |     0
23:01:25.000 | csci2600_a3_schroq   |                      |                      |                      |     0
23:01:30.000 | csci2600_a3_schroq   |                      |                      |                      |     0
23:01:35.000 |                      |                      |                      |                      |     0
```

And the corresponding log file:

```
csci1100_hw4_waterz   upload: 23:00:06.078  start: 23:00:10.000  finish: 23:00:19.267  nice:  10  processor: 0     wait:    3.922 sec  grade:    9.267 sec
csci1100_hw4_guerrh   upload: 23:00:14.069  start: 23:00:15.000  finish: 23:00:24.025  nice:  10  processor: 3     wait:    0.931 sec  grade:    9.025 sec
csci1100_hw4_jarvim   upload: 23:00:12.027  start: 23:00:15.000  finish: 23:00:24.148  nice:  10  processor: 1     wait:    2.973 sec  grade:    9.148 sec
```

```
csci1100_hw4_sotoy    upload: 23:00:23.060  start: 23:00:25.000  finish: 23:00:35.453  nice:  10  processor: 1     wait:    1.940 sec  grade:   10.453 sec
csci1100_hw4_morrih   upload: 23:00:25.071  start: 23:00:30.000  finish: 23:00:41.247  nice:  10  processor: 3     wait:    4.929 sec  grade:   11.247 sec
csci1200_hw6_schmio   upload: 23:00:22.091  start: 23:00:25.000  finish: 23:00:49.044  nice:   4  processor: 0     wait:    2.909 sec  grade:   24.044 sec
csci2600_a3_mannt     upload: 23:00:13.064  start: 23:00:15.000  finish: 23:00:49.255  nice:   8  processor: 2     wait:    1.936 sec  grade:   34.255 sec
csci1100_hw4_meltoo   upload: 23:00:27.030  start: 23:00:40.000  finish: 23:00:49.813  nice:  10  processor: 1     wait:   12.970 sec  grade:    9.813 sec
csci1200_hw6_leachm   upload: 23:00:28.078  start: 23:00:45.000  finish: 23:01:11.428  nice:   4  processor: 3     wait:   16.922 sec  grade:   26.428 sec
csci2600_a3_schroq    upload: 23:00:29.073  start: 23:00:50.000  finish: 23:01:34.445  nice:   8  processor: 0     wait:   20.927 sec  grade:   44.445 sec
```

Various statistics about the simulation are printed to `std::cout`:

```
timestep:                       5.000 sec
job priority:                        FIFO
time to empty queue:           95.000 sec
maximum waiting time:          20.927 sec
average waiting time:           7.036 sec     10 jobs
nice=4  waiting time:           9.915 sec      2 jobs
nice=8  waiting time:          11.431 sec      2 jobs
nice=10 waiting time:           4.611 sec      6 jobs
idle percentage:               50.493   %
simulation running time:        0.001 sec
```

This first simulation ignored the *niceness value*. When there are multiple autograding jobs waiting for a free processor, the simulation uses a simple first-in, first-out FIFO queue to select the next job to run.

Implementing Niceness with a Priority Queue

Your first task for this assignment is to implement the optional `--nice` command line argument, which indicates that the simulation should use a priority queue to instead choose the job with the lowest niceness value. If there are multiple jobs with the same niceness value, choose the job that has been waiting the longest. So for this command line:

```
./simulate.out sample_jobs.txt 4 --vis sample_5.0_vis.txt --log sample_5.0_log.txt --timestep 5.0 --nice
```

Specifically, the `csci1200_hw6_leach` autograding job is run earlier, since it has `nice = 4.0`. The `csci1100_hw4_meltoo` autograding job is delayed, since it has `nice = 10.0`.

```
csci1100_hw4_waterz   upload: 23:00:06.078  start: 23:00:10.000  finish: 23:00:19.267  nice:  10  processor: 0     wait:    3.922 sec  grade:    9.267 sec
csci1100_hw4_guerrh   upload: 23:00:14.069  start: 23:00:15.000  finish: 23:00:24.025  nice:  10  processor: 3     wait:    0.931 sec  grade:    9.025 sec
csci1100_hw4_jarvim   upload: 23:00:12.027  start: 23:00:15.000  finish: 23:00:24.148  nice:  10  processor: 1     wait:    2.973 sec  grade:    9.148 sec
csci1100_hw4_sotoy    upload: 23:00:23.060  start: 23:00:25.000  finish: 23:00:35.453  nice:  10  processor: 1     wait:    1.940 sec  grade:   10.453 sec
csci1100_hw4_morrih   upload: 23:00:25.071  start: 23:00:30.000  finish: 23:00:41.247  nice:  10  processor: 3     wait:    4.929 sec  grade:   11.247 sec
csci1200_hw6_schmio   upload: 23:00:22.091  start: 23:00:25.000  finish: 23:00:49.044  nice:   4  processor: 0     wait:    2.909 sec  grade:   24.044 sec
csci2600_a3_mannt     upload: 23:00:13.064  start: 23:00:15.000  finish: 23:00:49.255  nice:   8  processor: 2     wait:    1.936 sec  grade:   34.255 sec
csci1100_hw4_meltoo   upload: 23:00:27.030  start: 23:00:50.000  finish: 23:00:59.813  nice:  10  processor: 0     wait:   22.970 sec  grade:    9.813 sec
csci1200_hw6_leachm   upload: 23:00:28.078  start: 23:00:40.000  finish: 23:01:06.428  nice:   4  processor: 1     wait:   11.922 sec  grade:   26.428 sec
csci2600_a3_schroq    upload: 23:00:29.073  start: 23:00:45.000  finish: 23:01:29.445  nice:   8  processor: 3     wait:   15.927 sec  grade:   44.445 sec
```

We can see these changes reflected in the detailed statistics. The average wait time for the jobs with `nice = 4.0` has decreased and the average wait time for the jobs with `nice = 10.0` has increased.

```
timestep:                       5.000 sec
job priority:        niceness, then FIFO
time to empty queue:           90.000 sec
maximum waiting time:          22.970 sec
average waiting time:           7.036 sec     10 jobs
nice=4  waiting time:           7.415 sec      2 jobs
nice=8  waiting time:           8.931 sec      2 jobs
nice=10 waiting time:           6.277 sec      6 jobs
idle percentage:               47.743   %
simulation running time:        0.000 sec
```

Timestep and Idle Processors

The next modification we will explore is the timestep of the simulation. In the provided code, we will only check the availability of processors and start new grading jobs on a fixed timestep. We used 5.0 seconds in the examples above. In our example, the very first autograding job waits nearly four seconds to begin even though all processors are available. So let's decrease the timestep to 0.1 seconds:

```
./simulate.out sample_jobs.txt 4 --vis sample_0.1_vis.txt --log sample_0.1_log.txt --timestep 0.1 --nice
```

We will see in the log file that the waiting time for many jobs decreases to under 0.1 seconds:

```
csci1100_hw4_waterz   upload: 23:00:06.078  start: 23:00:06.100  finish: 23:00:15.367  nice:  10  processor: 0     wait:    0.022 sec  grade:    9.267 sec
csci1100_hw4_jarvim   upload: 23:00:12.027  start: 23:00:12.100  finish: 23:00:21.248  nice:  10  processor: 1     wait:    0.073 sec  grade:    9.148 sec
csci1100_hw4_guerrh   upload: 23:00:14.069  start: 23:00:14.100  finish: 23:00:23.125  nice:  10  processor: 3     wait:    0.031 sec  grade:    9.025 sec
csci1100_hw4_sotoy    upload: 23:00:23.060  start: 23:00:23.100  finish: 23:00:33.553  nice:  10  processor: 1     wait:    0.040 sec  grade:   10.453 sec
csci1100_hw4_morrih   upload: 23:00:25.071  start: 23:00:25.100  finish: 23:00:36.347  nice:  10  processor: 3     wait:    0.029 sec  grade:   11.247 sec
csci1200_hw6_schmio   upload: 23:00:22.091  start: 23:00:22.100  finish: 23:00:46.144  nice:   4  processor: 0     wait:    0.009 sec  grade:   24.044 sec
csci2600_a3_mannt     upload: 23:00:13.064  start: 23:00:13.100  finish: 23:00:47.355  nice:   8  processor: 2     wait:    0.036 sec  grade:   34.255 sec
csci1100_hw4_meltoo   upload: 23:00:27.030  start: 23:00:46.200  finish: 23:00:56.013  nice:  10  processor: 0     wait:   19.170 sec  grade:    9.813 sec
csci1200_hw6_leachm   upload: 23:00:28.078  start: 23:00:33.600  finish: 23:01:00.028  nice:   4  processor: 1     wait:    5.522 sec  grade:   26.428 sec
csci2600_a3_schroq    upload: 23:00:29.073  start: 23:00:36.400  finish: 23:01:20.845  nice:   8  processor: 3     wait:    7.327 sec  grade:   44.445 sec
```

And we can also see in the overall statistics that the average wait times decrease, the time to empty the queue decreases, and the idle percentage decreases:

```
timestep:                      0.100 sec
job priority:        niceness, then FIFO
time to empty queue:          80.900 sec
maximum waiting time:         19.170 sec
average waiting time:          3.226 sec     10 jobs
nice=4  waiting time:          2.766 sec      2 jobs
nice=8  waiting time:          3.681 sec      2 jobs
nice=10 waiting time:          3.228 sec      6 jobs
idle percentage:              41.865   %
simulation running time:       0.003 sec
```

These trends continue if we push the timestep to 1/1000th of a second:

```
timestep:                      0.001 sec
job priority:        niceness, then FIFO
time to empty queue:          80.763 sec
maximum waiting time:         19.105 sec
average waiting time:          3.179 sec     10 jobs
nice=4  waiting time:          2.717 sec      2 jobs
nice=8  waiting time:          3.623 sec      2 jobs
nice=10 waiting time:          3.184 sec      6 jobs
idle percentage:              41.766   %
simulation running time:       0.204 sec
```

However, the running time of the simulation *increases*. This is called *busy waiting*. We are spending not insignificant resources with these excessively frequent status checks. Note that the length/size of the visualization file also grows significantly when a smaller timestep is used. Writing this larger file contributes to the slower simulation time. Can we do better? Yes! And it will involve another priority queue!

Reducing or Eliminating Busy Wait

The shorter autograding jobs on our system are usually running for about 10 seconds and the longer jobs may run for 10 minutes or more. If all processors are in the middle of running lengthy autograding jobs it does not make sense to poll each of them every 1/1000th of a second to ask *"Are you finished yet?"*. A better idea is to use the known running time of each autograding task to compute the expected finish time. *NOTE: In real-world simulations, the running time of each job may only be an estimate; thus some (infrequent) polling would still be necessary to confirm the status / availability of each processor.*

Your second task of this assignment is to use a priority queue to efficiently organize these job finish events. When we begin an autograding task, we calculate the expected finish time and make a job finish event record that knows both the finish time and on which processor the job is running. These events are placed in a priority queue with the earlier finish times at the top of the queue. Now when we are running the simulation we can simply check the top item in the finish time event priority queue and safely and efficiently make a large step forward in time. Here is the sample command line *without a timestep* that indicates that we should use the finish time event priority queue:

```
./simulate.out sample_jobs.txt 4 --vis sample_vis.txt --log sample_log.txt --nice
```

The statistics printed to `std::cout` match the 1/1000th of a second timestep simulation, except the simulation running time will be much faster:

```
timestep:             finish event queue
job priority:        niceness, then FIFO
time to empty queue:          80.763 sec
maximum waiting time:         19.105 sec
average waiting time:          3.179 sec     10 jobs
nice=4  waiting time:          2.717 sec      2 jobs
nice=8  waiting time:          3.623 sec      2 jobs
nice=10 waiting time:          3.184 sec      6 jobs
idle percentage:              41.766   %
simulation running time:       0.000 sec
```

Scalability Testing and Big O Notation Analysis

In your README.txt file, provide a Big O Notation complexity analysis for the simulation. Use j to represent the total number of student autograding submissions/jobs, use p for the total number of processors on the Submitty system, use r to be the average runtime of a single job, use q as the maximum number of jobs waiting to be graded at any time, and use t for the timestep. Discuss how the variables j, p, r, q, and t are related to each other.

Test and debug your implementation using the provided input files. Try different input file sizes, different numbers of processors, with and without the `--nice` options, different timesteps, and without the timestep. Present your data and summarize your conclusions in the README.txt file.

Discuss how the simulation results compare to your Big O Notation analysis. And as always, be sure to list your collaborators and all resources used in completing the assignment in your README.txt.

Congratulations! You have finished the final Data Structures homework!