# CSC 225 - SUMMER 2018
# ALGORITHMS AND DATA STRUCTURES I
# PROGRAMMING ASSIGNMENT 2
# UNIVERSITY OF VICTORIA

**Due**: Tuesday, July 3rd, 2018 before 11:55pm. **Late assignments will not be accepted.**

## 1 Assignment Overview: A `nice` simulator

In a multitasking operating system (such as the operating system on the machine you're using to read this), CPU time must be divided among a set of different processes such that each process's needs for resources are satisfied. Each process may have an associated *priority*, and processes with better priorities receive more computation time, while processes with less priority receive less. It can often be useful to deliberately give a process a low priority to run it unobtrusively in the background (and ensure that it does not interfere with other running tasks).

On Unix-like operating systems, a command called `nice` can be used to start a process with an arbitrary priority, and the command `renice` can be used to change the priority of a running process. For strange historical reasons[1], the priority of each process is called its *nice value*, and processes with **lower** nice values are given more processing time (that is, a 'high priority' task has a low 'nice level'). The apparent logic used by the creators was that the 'nicer' a process was, the more likely it would be to let other processes execute first, so a high 'nice level' correlates to low processing priority.

This programming assignment involves implementing a data structure for a simple CPU scheduling simulator. CPU scheduling is a complex topic, and is covered properly in courses like CSC 360 (Operating Systems). The goal of this assignment is to apply the concept of a priority queue to a specialized data structure which implements the operations of the simulator (and therefore requires more functionality than a basic priority queue), and our CPU scheduling simulation will therefore be relatively simple to allow the assignment to focus on the algorithms involved instead of the process model.

In deference to the Unix tradition, our simulator will use 'nice levels' to dictate priority, and give preference to tasks with lower nice levels over those with higher nice levels.

## 2 Simulation Description and Input Format

Two Java source files have been provided. The `Nice.java` program contains a `main()` method which parses input data, runs the simulation and generates output and the `NiceSimulator.java` file contains an (unimplemented) data structure for each of the simulated operations. You are expected to implement all of the methods of the `NiceSimulator` class to conform to this specification. It

1. See the Wikipedia entry at `https://en.wikipedia.org/wiki/Nice_(Unix)` for details on the origin of the name

should not be necessary to modify the `Nice.java` program at all, but you are welcome to do so, as long as your submitted result has **exactly** the same output format as the provided `Nice.java` program.

This section describes the simulator program which reads the input data and calls methods of the `NiceSimulator` data structure. Note that the entire simulator program described in this section has been provided to you in `Nice.java`, so you do not need to write code for any of the functionality described in this section. To read about the data structure you are required to implement, you can skip to Section 3. However, you will need to consult this section for information on what constitutes a valid test input.

Once compiled, the simulator can be run with the command

```
java Nice
```

Normally, the simulator will read its input from standard input (which defaults to user input). To provide input data from a file, use shell redirection, as in the example below.

```
java Nice < input_file.txt
```

Output will be produced to standard output (which defaults to the console). As with input, you can redirect output to a file if you want to store it for future use (such as automated comparison using the `diff` tool).

The program is intended to simulate a fictional CPU whose time must be divided among a set of tasks. Each task has a unique ID number, which must be a positive integer, an priority value (nice level), which may be any integer (including a negative value), and a time requirement, which must be a positive integer. At each step of the simulator, one task will be selected to run, and its total time requirement will decrease accordingly. Once a task's time requirement has reached zero, the task is complete and is removed from the system. Eventually, all tasks will be complete and the CPU will be idle.

The input to the simulator program is a set of commands, which may add, remove or reprioritize ('renice') tasks from the set of active tasks. After reading the input, the simulator uses the `NiceSimulator` data structure to repeatedly simulate single units of CPU time (or 'timesteps'), during which one of the active tasks will be run (and its total time requirement decremented) until a timestep occurs in which no active jobs are available and the simulated CPU is idle, at which point the simulation ends.

Input is provided in a simple text-based format. Note that an input file must meet the exact criteria described in this section or it will be considered invalid for the purposes of testing.

The first line of the input is an integer `max_tasks`, which must be positive. All task ID numbers must be in the range 0 - `max_tasks`$-1$ (inclusive). If any tasks in the input file have an ID outside the range, the input is invalid.

Each of the lines after the first line is a task operation, which may be either `add`, `kill` or `renice`. Note that a valid input must have at least one line (the `max_tasks` value), but there may be no additional lines (in which case the simulation stops after one step, since there are no tasks to simulate in such cases).

Note that the provided `Nice.java` program contains all of the code needed to parse and validate the input file, so you do not have to write any code to interpret the input data (but you should understand the structure of an input file).

The `add` operation has the form

```
<time> add <task ID> <time requirement>
```

and adds a task with the ID and time requirement provided immediately **before** the indicated time step. The time requirement must be a positive (and therefore non-zero) value or the input file will be considered invalid. If a task with the provided ID already exists, then the entire input file is considered invalid (since it is not possible to add a task in such cases).

The `kill` operation has the form

```
<time> kill <task ID>
```

and deletes the task with the provided ID immediately **before** the indicated time step. If a task with the provided ID does not exist, then the entire input file is considered invalid.

The `renice` operation has the form

```
<time> renice <task ID> <new priority>
```

and changes the priority of the task with the provided ID to the provided priority value immediately **before** the indicated time step. If a task with the provided ID does not exist, then the entire input file is considered invalid.

For the input file to be valid, the operations in the file must be ordered chronologically (so the operations can be applied in the simulation in the same order they appear in the input file). This implies that the timesteps of the operations in the file must appear in ascending order (so an operation set for time 6 must appear before an operation set for time 10).

## 3 The `NiceSimulator` data structure

The `NiceSimulator` data structure contains 8 methods, including a constructor. All 8 methods are empty and must be implemented before the simulator will work. You are permitted to add other methods (or classes) as needed. You are **not** permitted to change the method signature (name/argument list) or specification of **any** of the 8 provided methods. If you do so, you will receive no marks for the corresponding aspects of the asssignment.

An implementation-level description of each method is given in comments inside the `NiceSimulator.java` file. None of the methods in `NiceSimulator.java` may throw any exceptions during normal operation (and adding a `throws` clause to the method signature is not permitted). A high level description of each method is given below. For full marks, each of the methods must achieve the worst case running time given in the 'Target Time' column. However, as for assignment 1, you will receive no marks for running time unless your code is correct, so an asymptotically less-efficient implementation will be worth more marks than a broken (but allegedly 'fast') implementation. The parameter $n$ in the running time expressions is defined to be the number of active tasks (tasks which have at least one unit of time remaining in their requirement), not the maximum number of possible tasks.

| Method | Target Time | Description |
|---|---|---|
| `NiceSimulator(maxTasks)` | – | (Constructor) Initialize the data structure for the provided maximum number of tasks. The structure may assume that no task with an ID outside the range $[0, \texttt{maxTasks} - 1]$ will ever be used by any operation. There are no restrictions on the running time of this method. |
| `taskValid(ID)` | $O(1)$ | Return `true` if the provided ID is a valid task (with at least one unit of time remaining) and `false` otherwise. |
| `getPriority(ID)` | $O(1)$ | Return the current nice level for the provided task ID. |
| `getRemaining(ID)` | $O(1)$ | Return the time remaining for the provided task ID. |
| `add(ID, time)` | $O(\log_2 n)$ | Add a new task with the provided ID and time requirement. The new task will be assigned a priority of 0. |
| `kill(ID)` | $O(\log_2 n)$ | Delete the task with the provided ID. |
| `renice(ID,priority)` | $O(\log_2 n)$ | Set the priority of the provided task to the provided ID. |
| `simulate()` | $O(\log_2 n)$ | Identify the task $t$ which will run next (see rules below) and subtract one from the task's time requirement. If the requirement is decreased to zero, return the ID number of task $t$. Otherwise, return either the constant `SIMULATE_IDLE` (if no task $t$ was found) or `SIMULATE_NONE_FINISHED` (if $t$ was found but still has time remaining). |

In the `simulate()` method, the next task $t$ to run is selected by the following rules:

- If no active tasks exist, then no task $t$ can be found.
- Otherwise, if exactly one task has the **lowest** nice level, that task is selected as $t$.
- If multiple tasks all have the lowest nice level, then the task with the smallest task ID will be selected as $t$.

Since this assignment will not be evaluated by demos, you are encouraged to add comments to your code explaining the running time of each operation to ensure that you receive the marks for achieving the targeted running time.

## 4 Sample Input Files

Consider the input file below, which defines 4 jobs with several different execution times. Recall from the previous section that the initial priority of a job is always 0, so to assign a different initial priority to a job, it is necessary to use a `renice` operation before step 0.

| Input File | Effect |
|---|---|
| `32768` | Set `max_tasks` to 32768 |
| `0 add 6 3` | Add task 6 (with time requirement 3) before timestep 0 |
| `0 add 10 4` | Add task 10 (with time requirement 4) before timestep 0 |
| `0 add 17 5` | Add task 17 (with time requirement 5) before timestep 0 |
| `0 renice 6 -1` | Renice task 6 to nice level $-1$ before timestep 0 |
| `0 renice 10 100` | Renice task 10 to nice level 100 before timestep 0 |
| `0 renice 17 1` | Renice task 17 to nice level 1 before timestep 0 |
| `4 add 225 6` | Add task 225 (with time requirement 6) before timestep 4 |
| `7 kill 225` | Terminate task 225 before timestep 7 |
| `9 renice 10 -2` | Renice task 10 to nice level $-2$ before timestep 9 |

The table below shows the progression of a correct simulator on the input file above. Notice that the operations in the input file are always applied immediately before the provided timestep number and that the input operations do not consume any timesteps to perform. Only the `simulate()` method actually consumes simulation time.

| Time Step | Running Task | Job Status (after current timestep) Task 6 Priority | Task 6 Time Left | Task 10 Priority | Task 10 Time Left | Task 17 Priority | Task 17 Time Left | Task 225 Priority | Task 225 Time Left | Notes |
|---|---|---|---|---|---|---|---|---|---|---|
| **Before timestep 0**<br>Add task 6 with time requirement 3<br>Add task 10 with time requirement 4<br>Add task 17 with time requirement 5<br>Renice task 6 to priority -1<br>Renice task 10 to priority 100<br>Renice task 17 to priority 1 | | | | | | | | | | |
| - | - | −1 | 3 | 100 | 4 | 1 | 5 | - | - | Before timestep 0 |
| 0 | 6 | −1 | 2 | 100 | 4 | 1 | 5 | - | - | |
| 1 | 6 | −1 | 1 | 100 | 4 | 1 | 5 | - | - | |
| 2 | 6 | - | - | 100 | 4 | 1 | 5 | - | - | Task 6 finished. |
| 3 | 17 | - | - | 100 | 4 | 1 | 4 | - | - | |
| **Before timestep 4**<br>Add task 225 with time requirement 6<br>(Note that since task 225 is run next, the requirement immediately drops to 5) | | | | | | | | | | |
| 4 | 225 | - | - | 100 | 4 | 1 | 4 | 0 | 5 | |
| 5 | 225 | - | - | 100 | 4 | 1 | 4 | 0 | 4 | |
| 6 | 225 | - | - | 100 | 4 | 1 | 4 | 0 | 3 | |
| **Before timestep 7**<br>Kill task 225 | | | | | | | | | | |
| 7 | 17 | - | - | 100 | 4 | 1 | 3 | - | - | |
| 8 | 17 | - | - | 100 | 4 | 1 | 2 | - | - | |
| **Before timestep 9**<br>Renice task 10 to priority −2 | | | | | | | | | | |
| 9 | 10 | - | - | −2 | 3 | 1 | 2 | - | - | |
| 10 | 10 | - | - | −2 | 2 | 1 | 2 | - | - | |
| 11 | 10 | - | - | −2 | 1 | 1 | 2 | - | - | |
| 12 | 10 | - | - | - | - | 1 | 2 | - | - | Task 10 finished. |
| 13 | 17 | - | - | - | - | 1 | 1 | - | - | |
| 14 | 17 | - | - | - | - | - | - | - | - | Task 17 finished. |
| 15 | - | - | - | - | - | - | | - | - | CPU Idle. |
| Simulation Ends | | | | | | | | | | |

A second input file appears below. Notice that the same task ID is used for more than one task: Task 10 is added at timestep 2 and, after it finishes, another task with ID 10 is added at timestep 6. Task IDs may be reused an unlimited number of times within the program, as long as any previous task with the same ID has either finished or been killed.

| Input File | Effect |
|---|---|
| `16384` | Set `max_tasks` to 16384 |
| `0 add 100 3` | Add task 100 (with time requirement 3) before timestep 0 |
| `0 renice 100 1` | Renice task 100 to nice level 1 before timestep 0 |
| `0 add 2018 4` | Add task 2018 (with time requirement 4) before timestep 0 |
| `0 renice 2018 2` | Renice task 2018 to nice level 2 before timestep 0 |
| `2 add 10 3` | Add task 10 (with time requirement 3) before timestep 2 |
| `2 renice 10 -10` | Renice task 10 to nice level −10 before timestep 2 |
| `3 renice 2018 -1` | Renice task 2018 to nice level −1 before timestep 3 |
| `6 add 10 2` | Add task 10 (with time requirement 2) before timestep 6 |

The simulation progression for the input file above is shown below.

| Time Step | Running Job | Job Status (after current timestep) Task 10 Priority | Task 10 Time Left | Task 100 Priority | Task 100 Time Left | Task 2018 Priority | Task 2018 Time Left | Notes |
|---|---|---|---|---|---|---|---|---|
| **Before timestep 0**<br>Add task 100 with time requirement 3<br>Renice task 100 to priority 1<br>Add task 2018 with time requirement 4<br>Renice task 2018 to priority 2 | | | | | | | | |
| - | - | - | - | 1 | 3 | 2 | 4 | Before timestep 0 |
| 0 | 100 | - | - | 1 | 2 | 2 | 4 | |
| 1 | 100 | - | - | 1 | 1 | 2 | 4 | |
| **Before timestep 2**<br>Add task 10 with time requirement 3<br>Renice task 10 to priority -10 | | | | | | | | |
| 2 | 10 | −10 | 2 | 1 | 1 | 2 | 4 | |
| **Before timestep 3**<br>Renice task 2018 to priority -1 | | | | | | | | |
| 3 | 10 | −10 | 1 | 1 | 1 | −1 | 4 | |
| 4 | 10 | - | - | 1 | 1 | −1 | 4 | Job 10 finished. |
| 5 | 2018 | - | - | 1 | 1 | −1 | 3 | |
| **Before timestep 6**<br>Add task 10 with time requirement 2 | | | | | | | | |
| 6 | 2018 | 0 | 2 | 1 | 1 | −1 | 2 | |
| 7 | 2018 | 0 | 2 | 1 | 1 | −1 | 1 | |
| 8 | 2018 | 0 | 2 | 1 | 1 | - | - | Job 2018 finished. |
| 9 | 10 | 0 | 1 | 1 | 1 | - | - | |
| 10 | 10 | - | - | 1 | 1 | - | - | Job 10 finished. |
| 11 | 100 | - | - | - | - | - | - | Job 100 finished. |
| 12 | - | - | - | - | - | - | - | CPU Idle. |
| Simulation Ends | | | | | | | | |

## 5 Using Outside Code

You are encouraged to use the features of the Java Standard Library (including any of the data structures it provides) in your code. If you use a standard library data structure, make sure you are aware of the running times of the operations you use, since that will be important for determining the running time of your program.

There should be no need to use large volumes of code from other sources (such as outside libraries or the internet) in this assignment. If you believe that your implementation requires an outside library, talk to your instructor.

If you find a small snippet of code on the internet that you want to use (for example, in a StackOverflow thread), put a comment in your code indicating the source (including a complete URL). Remember that using code from an outside source without citation is plagiarism.

You are encouraged to discuss the assignment with your peers, and even to share implementation advice or algorithm ideas. However, you are not permitted to use any code from another CSC 225 student under any circumstances, nor are you permitted to share your code in any way with any other student (or the internet) until after the marking is complete. Sharing your code with others before marking is completed, or using another student's code for assistance (even if you do not directly copy it) is plagiarism.

## 6 Peer Testing

In addition to your code, you will be expected to include a single test input (in the format described in Section 2) in a text file called `test_input.txt`. Your test input must meet the following criteria to be considered valid (and any invalid test input will receive zero marks):

- The input must meet all of the conditions in Section 2 to be a valid input.
- The simulation must finish in at least 10 and at most 1000000 timesteps.
- The `max_tasks` value may be at most 65536.
- The input may contain at most 100000 lines.

After the due date (Tuesday, July 3rd, 2018), the results of testing every submitted implementation against every submitted test input will be posted to the Testing Database on conneX. No student numbers will be used in the published data (you will be identified by a randomly assigned numerical alias). Your complete test input will also be posted to assist your peers in their testing, but the code for your implementation will not be published (and you are reminded not to share it with any other students until after all marking is complete). Since the output of your program's testing will be publicly visible, please ensure that your program's output contains no identifying information.

After the results of peer testing are posted, you will have the opportunity to revise and resubmit your code until Sunday, July 8th, 2018. You will not be permitted to resubmit if you did not submit an implementation before the July 3rd deadline (and you will therefore receive a mark of zero). Since all test inputs will be published, you will also not be permitted to resubmit your test input.

For marking purposes, your original and revised submissions will be tested on a set of test inputs created by your instructor (you will not be graded based on the results of peer testing). The 8 marks for program function (see Section 7) will be computed by taking the average of the number of tests passed by your original submission and the number of tests passed by your revised submission.

## 7 Evaluation

Submit all `.java` files needed to compile your assignment electronically via conneX. Your code must compile and run correctly in the Linux environment in ECS 242. If your code does not compile as submitted, you will receive a mark of zero.

This assignment is worth 7% of your final grade and will be marked out of 14 with a combination of automated testing and human inspection. This assignment will not be evaluated by interactive demos.

The marks are distributed among the components of the assignment as follows.

| Marks | Component |
|---|---|
| 8 | The requirements in this document are met and the output of the program is correct on a variety of tested inputs. |
| 1 | Your submitted test input is valid (see Section 6). |
| 1 | The `taskValid`, `getPriority` and `getRemaining` methods **all** have worst case $O(1)$ running time. |
| 1 | The `add`, `kill` and `renice` methods **all** have worst case $O(\log_2 n)$ running time. |
| 1 | The `simulate` method has a worst case $O(\log_2 n)$ running time. |
| 2 | **All** of the required methods of the data structure (except the constructor) have a worst case $O(\log_2 n)$ running time. |

You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions or resubmissions will be accepted after the due date has passed. You will receive a mark of zero if you have not officially submitted your assignment (and received a confirmation email) before the due date.

Your data structure must be contained in `NiceSimulator.java` (and the main simulator program must be contained in `Nice.java`). You may use additional files if needed by your solution (as long as the program can be invoked from the command line using the syntax in Section 2). Ensure that each submitted file contains a comment with your name and student number.

Ensure that all code files needed to compile and run your code in ECS 242 are submitted. Only the files that you submit through conneX will be marked. The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. **If you do not receive such an email, you did not submit the assignment.** If you have problems with the submission process, send an email to the instructor **before** the due date.