Problem 2

Problem 2

Usage

Run the file simulator.py from either an IDE or the terminal. In order to work out imports correctly using the terminal, you should `cd` in the directory first and run the program using `python simulator.py`.

Run description

The simulator will then prompt you to insert an aging interval: the number of time slices after which a job waiting in the queue will have its priority updated. It's exactly the `x` parameter defined in the problem description.
From now on, the process of updating the priority of a waiting job will be referred to as *aging*.

Then, the simulator runs in an infinite loop, printing the current time slice as an incrementing integer which starts from 0.
In the first time slice it has no jobs to execute, so it simply asks if you want to add a new job to the queue. If the answer is affirmative, it then prompts you for the job name, priority and length.
If at a certain iteration the simulator has a job to execute, it does it before asking for a new job and prints the job name and the remaining time slices for this job.

The simulator will try to validate user input.

Solution description

The simulator uses the `LazyJobScheduler` class that you can find in the `schedulers` package.
It implements the `BaseJobScheduler` Abstract Base Class in the same package, which defines a simple Job class and provides the following behaviors typical of a priority queue:

- `get_max_priority_job`: returns the job with the highest priority and removes it from the waiting queue;
- `add_job`: adds a new job to the waiting queue;
- `increment_time`: it is used to let the scheduler know when a time slice has passed, which may then decide to do something, such as aging jobs.

Our LazyJobScheduler does not check if it needs to age jobs at each time slice: it only does it when it needs to yield the max priority job.
It does so by keeping track of the arrival time of the jobs, not of their waiting time, and of the current time. When it is time to provide a job, the scheduler evaluates all the jobs it is handling and checks whether they age and by how much - in fact, more than one aging interval may have passed for the waiting jobs since the last time they were updated. In the meanwhile, since it is already visiting all the jobs, it also keeps track of the minimum (so the highest) updated priority of the jobs, doing everything in one pass of the jobs.

Data structure description and complexity analysis

The LazyJobScheduler does not need any particular data structure to implement its behavior: a positional unsorted linked list was used.

Every time slice

In this way, the scheduler doesn't execute any actions but incrementing its time counter, so it only does a very simple $O(1)$ operation when no new job is requested.
If a job needs to be added, it is added in $O(1)$ too thanks to the unsorted linked list implementation.

Once a job has finished being processed

The scheduler visits all the jobs and does constant time operations for each one: it checks if it has passed an integer number of aging intervals in the queue and updates its priority and arrival time accordingly, just as if the jobs were updated every time slice. It then keeps track of the minimum during the visit.
At the end, it removes the selected job in constant time, thanks to the choice of the linked list.
Therefore, every time a job finishes processing, and only then, a $O(n)$ operation is executed.