Project 3: Completely Fair Scheduler

Due date: Midnight of Wednesday Mar 2, 2022

Project objective:

- Understand **Completely Fair Scheduler** by implementing it in Python.
- Simulate the CPU scheduling aspects of an Operating System kernel.
- Practice building OS simulations in Python and Jupyter Notebooks.
- Learn an important data structure: **Red-Black Trees**.

Project overview:

In this assignment, you will create simulations of the CPU scheduling aspects of an Operating System Kernel, and implement **Completely Fair Scheduler** using your own Red-Black Tree.

In addition to CFS, you will write code to calculate statistics and simulations. Finally, you will create a document in Jupyter Notebooks that acts as your **project report** where you run your tests, simulations, and create visualizations of each run. This project is a continuation of the previous project, so we will use the same file structure of *process.py*, *scheduler.py*, *operating_system.py*, and *Scheduling_Analyses3.ipynb*.

The Red-Black Tree file:

In the *RBTree.py* file you will implement a RBNode class and a RBTree class. The RBNode class is very simple and it includes only an __init__ method that takes a value and sets the following parameters for the node:

- Key (this will be the *vruntime* of a process in CFS)
- Reference to parent
- Reference to left child
- Reference to right child
- Boolean indicating if node is red

Implement getters and setters for each attribute (although Python keeps all attributes public, getters and setters reduce bugs in your code).

For the RBTree class, implement the following methods:

1. An __init__ method that does not take any parameters. The method initializes the following attributes:
    - A nil node with value zero, and color black (self.nil = RBNode(0)).
    - A root initially set to nil.

- A min_vruntime variable set to root initially.

2. An insert method that takes a value. The method should maintain a correct min_vruntime after each insert.
3. A fix_insert method that takes a node.
4. A rotate_left method that takes a node. The method implements the following pseudocode:

```
rotate_left(T,x)
   y = right[x]
   right[x] = left[y]
   p[left[y]] = x
   p[y] = p[x]


   if p[x] == nil[T] then root[T] = y
   else
      if x == left[p[x]] then left[p[x]] = y
      else
         right[p[x]] = y
   left[y] = x
   p[x] = y
```

5. A rotate_right method that takes a node. The method implements the following Pseudocode:

```
rotate_right(T,x)
    y = left[x]           // y now points to node to left of x
    left[x] = right[y]    // y's right subtree becomes x's left subtree
    p[right[y]] = x       // right subtree of y gets a new parent
    p[y] = p[x]           // y's parent is now x's parent

    // if x is at root then y becomes new root
    if p[x] == nil[T] then root[T] = y
    else
        // if x is a left child then adjust x's parent's left child or...
         if x == left[p[x]] then left[p[x]] = y
         else
         // adjust x's parent's right child
            right[p[x]] = y
    // the right child of y is now x
    right[y] = x
    // the parent of x is now y
    p[x] = y
```

6. A print_tree method that prints the tree inorder. This is helpful during debugging.
7. A remove_min_vruntime method that removes the node with the smallest vruntime in the tree and updates min_vruntime in constant time. The method should maintain all the properties of a

RB-Tree. This is a simplified remove method that works in our application, implementing the full RB-Tree remove method is a great extension.

The Process Class:

In the process file, you have to make the following modifications to the ***process*** class from project 2:

1. Add a new attribute (field) called ***vruntime*** to count virtual runtime**.**
2. Add a new attribute called ***weight*** that modulates vruntime according to process priority. The initial value for weight is 5.
3. Implement setters and getters for the new attributes.

This project will use the duty list explained in the previous project.

The scheduler file:

In your ***scheduler.py*** file from the previous project, add Completely Fair Scheduler. The scheduler takes a RB-Tree of ready processes stored in the tree according to their vruntime. The process with the minimum vruntime is selected to run for a dynamic quantum calculated using the following equation:

$$dynamic\_quantum = target\_latency / \#ready\_processes$$

Target latency defines the maximum response time for a process in the system. This dynamic_quantum guarantees fairness. If dynamic quantum is less than 1, then a value of 1 is chosen, and fairness is ignored as the system runs more processes than it can handle. Allow target latency to be an changeable attribute of your scheduler, and give it a value of 5 in your tests.

Update vruntime for each process according to how much it runs in the CPU multiplied by its weight, where weight is an integer that indicates priority. The highest priority is the value 1, and the lowest is 10. The default priority value for a process is 5. The equation for counting vruntime for each process is the following:

$$vruntime = t * weight$$

Where *t* is the time spent in the CPU, and weight is the priority value between 1 (high priority) and 10 (low priority).
As always, you are allowed to make modifications, helper functions, or any changes to the design of code, as long as it works as intended and your changes enhance efficiency, readability, and extendability of the code.

The operating_system file:

The ***operating_system.py*** file remains mostly the same from the previous project. Allow the kernel to take the new CFS as an argument, and use a RB-Tree as a ready tree instead of a list.

The Scheduling_Analyses3 Jupyter Notebook file:

Create a new notebook and name it ***Scheduling_Analyses3.ipynb***. In this file you will run your tests, visualize the results of CFS, run your simulations, and write your report. I will elaborate on each of the three parts separately.

Testing:
You are expected to run your CFS and generate data to verify that the scheduler works as intended. Let's use a small number of processes to verify your scheduler.

To run your scheduler, all you have to do is run the kernel with the name of the scheduler like the example below:

operating_system.kernel(scheduler.CFS_scheduler)

Since verbose is kept to the default true value, the line above should generate something like this:
process 0 starts: 0 ends: 5
process 1 starts: 5 ends: 9
process 2 starts: 9 ends: 10
process 3 starts: 10 ends: 16

Visualization:
Visualize your testing results using the same Gantt chart we have used previously. Make sure your visualization shows an understandable simulation.

Simulations:
Programmatically, generate 1,000 processes from which processes are split evenly according to their **duty** list:

- CPU-bound processes: Processes that spend a lot of time in the CPU, and very little time in I/O. The range of CPU time is between 8 and 12, while I/O is between 1 and 3.
- I/O-bound processes: Processes that spend a lot of time doing I/O, and very little time in the CPU. The range of CPU time is between 1 and 3, while I/O is between 8 and 12.

After you randomize the order of the I/O and CPU bound processes, compare the three metrics (response-time, wait-time, and turnaround-time) between I/O bound and CPU bound processes.

Report:
Organize your Jupyter Notebook as follows, maintaining a coherent document that includes markdown text and code to communicate the objective of your project:

1. Abstract: A brief summary of the project, in your own words. This should be no more than a few sentences. Give the reader context and identify the key purpose of the assignment.
2. Results: A section that goes over schedulers testing, visualization, and simulations.
    a. Testing
    b. Simulation
3. Discussion: A section that interprets and describes the significance of your findings focussing on the simulation results.
4. Extensions: Describe any extensions you undertook, including text output, graphs, tables, or images demonstrating those extensions.
5. References/Acknowledgements.

Extensions:

You can be creative here and come up with your own extensions. I will suggest the following ideas:

- Implement a ***visualization*** of your red-black tree using something like matplotlib.
- Implement a ***delete method*** for your red-black tree (delete any node in the tree, not just leftmost).
- Come up with your own **fair** scheduling algorithm and explain in which situations it might be useful/better.
- Compare CFS to previous schedulers in terms of the amount of time the scheduler itself takes to pick a process.
- Implement O(1) Scheduler and compare it to CFS.

Project submission:

- Add all your files (except *.ipynb_checkpoints* and *__pycache__*) to your ***project_3*** directory on Google Drive.
- Copy the rubric from Moodle to the project directory after you review it.