Project 7: Semaphores and Deadlock

Due date: Midnight of Wednesday Apr 13, 2022

Project objective:

- You will implement your own semaphore class in Python
- You will use your semaphore to solve the producer-consumer problem
- You will use your semaphore to simulate and implement a solution for the dining philosophers problem

Create your own Semaphore:

In a Jupyter Notebook, write a small class to implement your own semaphore using a counter and a condition variable. Here's the documentation for condition variables/objects in Python: LINK

Start by creating a class called *Semaphore*. This class is no more than 25 lines of code including spaces, so think more about what you are going to write. This class has three methods:

1. An __init__ method that takes a number, as an internal counter for the semaphore. The default value is 1 if no number is provided. The method also creates a condition variable called *condition*.
2. An *acquire* method that takes only self as an argument. The method acquires the lock for the condition variable before decrementing the counter by one, then it checks if the counter is below zero and sets the thread to sleep if true. Otherwise, it releases the lock.
3. A *release* method that takes only self as an argument. The method acquires the condition lock and increments the counter by one, notifies a single sleeping thread, and releases the lock.

To test your semaphore, use the *Buffer* example from class where we used Python semaphores and replace the Python semaphores with your own semaphores. The output should be similar to the python semaphore solution, in the sense that:

1. We see no errors, especially pop with empty list error.
2. The buffer never exceeds 10.
3. Every item added to the buffer is removed at the end of each run.

Dining Philosophers problem:

Start by demonstrating deadlock with a basic simulation of the dining philosopher problem. A good way to see deadlock in action is to implement the following structure for each philosopher. Each philosopher is a thread and each fork is a semaphore:

- For 5 iterations, the philosopher is going to do:
    - Think for 0.2 to 0.6 seconds (use time.sleep() and random.uniform())

    - Pick up fork on left, then pause for 0.1 seconds (increase the pause to increase chances of deadlock)
    - Pick up fork on right, then pause for 0.1 seconds
    - Eat for a random amount of time, then pause for a duration between 0.2 and 0.6 seconds
    - Put right fork back
    - Put left fork back
- Display a message when philosopher is finished

Your code should use 5 philosophers, but it should allow for any number of philosophers to be used easily (don't hard code any numbers).

In addition, your code should show print statements to show each stage. For example:

P0 is thinking…
P0 is taking left fork…
P0 is taking right fork…
P0 is eating…
P0 dropped right fork…
P0 dropped left fork…

You might not get a deadlock right away, it might need some adjustment to pause time and repeated runs to see it. Nonetheless, it is possible to happen. Your main task after that is to provide a solution for the deadlock problem, there are two good solutions:

1. **[Easy Solution]** Asymmetric solution: odd philosophers pick up left then right, even philosophers pick up right then left.
2. **[Less easy solution]** Allow philosopher to pick up forks if both are available (needs to pick up both in critical section)

Report:

Organize your notebook report as follows, maintaining a coherent document that includes screenshots and text to communicate the objective of your project:

1. Abstract: A brief summary of the project (including findings), in your own words. This should be no more than 150 words. Give the reader context and summarize the results of your assignment.

2. Results: A section that goes over the code you implemented and the performance of each synchronization solution.

3. Discussion: A section that interprets and describes the significance of your findings focussing on the 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:

- Programatically, provide an answer of one of the following questions:
    - Does adding more philosophers increase or decrease the chances for deadlock in the first simulation?
    - If we implement the random wait "solution" (Wifi solution), would that increase the total runtime of the program or decrease it (compared to your solution)?
    - Is there a number of philosophers that can never deadlock?
    - Does your solution slow down the simulation in comparison to no solution?
- Using NetworkX in Python, write a program that finds **all** cycles in a "wait-for" graph. Use the draw method to generate a visualization of your graph.
- Using NetworkX, create a simulation of deadlock detection and recovery.

Project submission:

- Add all your files (except *.ipynb_checkpoints* and *__pycache__*) to your ***project_7*** directory on Google Drive.
- Copy the rubric from Moodle to the project directory after you review it.