COMP3522 Lab #09: Looking Down from Space

Rahul Kukreja

rkukreja1@bcit.ca

BCIT CST - November 20, 2019

Welcome!

Welcome to the 8th COMP 3522 lab.

In today's lab, you will:

1. Get first hand experience implementing threads and concurrency
2. Learn how to use Locks to ensure Mutual Exclusion when accessing resources shared across threads.
3. Get a chance to work with a web API service that provides data about the International Space Station.

Grading

This lab and all future labs will be marked out of 5.

Figure 1: This lab is graded out of 5

For full marks this week, you must:

1. (3 points) Implement the Produce-Consumer problem correctly.
2. (2 points) Follow PEP-8 standards, best practices, OO Design principles, and write good code.

> 💡 **IMPORTANT:** This lab is due Thursday, 21st November at 11:59 PM. I will also grade anyone in class if they are finished early.

Requirements

> 💡 This is very **IMPORTANT**. Read through the whole document before writing code. This will ensure you understand the problem and requirements completely and don't waste any time writing code you don't need to.

In today's lab we are going to tackle the famous producer-consumer problem. In this problem, one or more threads are known as `producers`, that is they provide data to a common `buffer` (in our case, a `queue`).

And as you may expect, there are one or more threads known as `consumers`, who read and remove data from this common buffer and do something with it.

This is an extremely simple problem and a great way to get your hands dirty with writing multithreaded code.

If you haven't done so yet, install the requests module.

```
pip3 install requests
```

In today's lab we will be getting data from the `Open Notify` Project. The project specified a few `endpoints` (read: URL's or addresses) that each define a different API. Remember, an API is known as an `Application Programming Interface` , this is an interface that defines ways to interact with one or more behaviors. In this case, these are `Web API's` that allows us as developers to send requests to a Server and receive a response.

We will be querying one such endpoint for information about the **International Space Station (ISS).** We will be accessing the Overhead Pass Predictions for the

International Space Station. What this means is that we can provide the endpoint with a location (latitude and longitude) and find out when and for how long will the ISS be directly overhead that point. I don't know about you, but I thought this was really cool and interesting.

In this lab, you will be loading data about Canadian cities (their name, latitude and longitude) from the `city_locations.xlsx` file using pandas (**NOTE**: This is not an exhaustive list). The code to read and convert this data into usable objects has already been done for you.

Step 1: Get data from the endpoint.

The first thing you need to do is download the following files from D2L:

- `city_locations.xlsx`
- `city_locations_test.xlsx`
- `city_processing.py`

Go through the file and try running it. I have provided all the data structures and code you need to read the excel file and parse it into a database of cities. Let's break down the code shall we:

- **jprint**

  This is a function to help you debug the response you will be getting from the endpoint. This will make more sense as you read the referenced tutorial and start coding.

- **City**

  This data structure represents a single city. It stores the name, latitude and longitude of the city.

- **CityDatabase**

  This is the "Database" of cities that we will be processing .This class will read an excel file and parse it into a list of `City` objects.

- **OverheadPassEvent**

This is an object that represents a single instance of an event when the ISS station will be passing overhead a point. It consists of 2 attributes, the datetime of the event and the duration of the event.

- **CityOverheadTimes**

  This is a data structure that represents all the `OverheadPassEvent`'s for a city. It store a reference to the `City` object and a list of corresponding `OverheadPassEvent` for that city.

- **ISSDataRequest**

  This class acts as a `Facade` to the `Open Notify ISS Pass API`. This API accepts a latitude and a longitude as it's parameters and sends data about the next times the ISS will pass over the provided location.

Open up https://www.dataquest.io/blog/python-api-tutorial/ and read the tutorial on how to use `GET Requests` to get data from an endpoint. This is not difficult at all (especially since we don't need to provide a key and get authenticated). Once you have done this fill in the missing code in the `ISSDataRequest` class to query the endpoint for a response. Use the JSON response to create and return a `CityOverheadTimes` object for the specified city.

Write some code to test out and ensure that your code works. Use the `city_locations_test.xlsx` file for test data and print out the corresponding `CityOverheadTimes` objects for each city.

You are not allowed to modify any of the provided code except for the `get_overhead_pass` method in the `ISSDataRequest` class. Your code should not need to change any of the other existing classes to work.

Step 2: Creating our Buffer

Right, create a new module called `producer_consumer.py` . All code from this point onwards should be in this module.

The first step in our produce-consumer problem is to create a buffer that will hold the calculated `CityOverheadTimes` objects. We will use a `Queue` for this. Create a `CityOverheadTimeQueue` class with the following methods.

- `def __init__(self):`

  Instantiates an attribute called `data_queue` as an empty list.

  > 💡 Python provides a built-in queue class which is ThreadSafe. This means we don't need to worry about race conditions when using the data structure with multiple threads. For this lab, I do NOT want you to use this data structure. I want you to create a custom class to get a chance to work with Locks.

- `def put(self, overhead_time: city_processor.CityOverheadTimes) -> None:`

  This method is responsible for adding to the queue. Accept a `overhead_time` parameter and append it to the `data_queue` list.

- `def get(self) -> city_processor.CityOverheadTime:`

  This method is responsible for removing an element from a Queue. Remember a queue is a **FIFO** data structure, that is it is **F**irst **I**n **F**irst **O**ut. Each call to this method should return the element at index 0 and delete it from the list. Use the `del` keyword to delete the element as this will also automatically move all the other elements so there will be no empty spaces.

- `def __len__(self) -> int:`

  This magic method should return the length of the `data_queue`.

Write some code in `main` to test out this data structure. Make sure this is working as expected before proceeding.

Step 3: Creating a Producer and a Consumer Thread

We now need to create 2 classes that inherit from the `Thread` class found in the `threading` module. Import the threading module and create the following classes:

1. **ProducerThread**

   The ProducerThread class inherits from the `Thread` class. It has the following methods:

- `def __init__(self, cities: list, queue: CityOverheadTimeQueue):`
  This method initializes the class with a list of `City` Objects as well as a `CityOverheadTimeQueue`.

- `def run(self) -> None:`
  This method executes when the thread starts. It should loop over each `City` and pass it to the `ISSDataRequest.get_overheadpass()` method. It then proceeds to add the city to the queue. After reading in `5` cities, the thread should sleep for 1 second.

At this point, write some code in the main method to create, start and join a a ProducerThread and make sure this thread works as intended.

2. **ConsumerThread**

   The ConsumerThread is responsible for consuming data from the queue and printing it out to the console. It has the following methods:

   - `def __init__(self, queue: CityOverheadTimeQueue):`

     Initializes the ConsumerThread with the same queue as the one the producer has. It also implements a `data_incoming` boolean attribute that is set to True. This attribute should change to False after the producer thread has joined the main thread and finished processing all the cities.

   - `def run(self) -> None:`

     While `data__incoming` is true OR the length of the queue is > 0, this method should get an item from the queue and print it to the console and then sleep for `0.5` seconds. While processing the queue, if the queue is empty, put the thread to sleep for `0.75` seconds.

   Now write some more code in the main method to create a consumer thread and make sure it works as intended.

Step 4: Adding locks

Right now we are accessing a shared resource (the buffer) with 2 threads. Since the operating system can interrupt a thread and switch between them randomly, the queue may not be in sync. To ensure that we are accessing shared variables and resources safely we need to implement `Locks`.

Locks allow us to define areas of code for `Mutual Exclusion`. This means that only one thread can acquire the lock and access that block of code at a time. Mutual exclusion ensures that only one thread is modifying the queue at any given time. This avoid **Race Conditions**.

Race conditions are what happens when Locks are not implemented. These are situations where the order in which instructions are executed gets mixed up due to multiple switching threads and this causes errors.

Go read the `W12 Lab Race Condition Slides` on D2L to understand how locks work. I have also uploaded some sample code to show you how to implement them. It's really easy!

Add an attribute to our `CityOverheadTimeQueue` called `access_queue_lock` in it's `__init__` method. Use this lock to control access to the code in the `put` and `get` methods. Test and run the code again to ensure it works as expected.

You may want to use `city_locations_test.xlsx` for testing so you don't spend a lot of time waiting to process a 100 something cities. You can add more cities to this excel sheet to gradually increase the amount of cities as well.

Step 5: Adding more producers

The final step is to create 2 more producer thread. Split the cities from the city database across these three threads and start them. This should speed up your code and requests significantly.

> 💡 **NOTE:** You don't need to add a lock to the ISSDataRequest's get_overhead_pass method since it only works with local variables. Each thread keeps its own copy of local variables.

That's it for this lab! Only one more to go before the end of the semester.