Lab 10 - Looking Down from Space

Rahul Kukreja & Jeffrey Yim

jyim3@bcit.ca

Welcome!

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. Write code implementing the producer and consumer pattern

Grading

This lab will be marked out of 10.

For full marks this week, you must:

1. (2 point) Commit and push to GitHub after each non-trivial change to your code
2. (4 points) Successfully implement the requirements as described in this document
3. (2 points) Implement the Producer-Consumer pattern correctly
4. (2 points) Write code that is consistently commented and formatted correctly using good variable names, efficient design choices, atomic functions, constants instead of magic numbers, etc.

Requirements

Clone your repo using github classroom: https://classroom.github.com/a/kgo0u4mR

- [ ] 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 knowns 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.

In today's lab we will be getting data from a singleton called **ISSDataRequest**. Calling this singleton simulates retrieving data from an external source that introduces some time delay.

In this lab, you will be loading data about city locations (latitude and longitude) from **locations.txt**.

Step 1:

The first thing, clone the repo**.** I have provided several data structures.

- Location

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

- CityInfo
  This is a data structure that represents all the information about a city

- ISSDataRequest
  This singleton accepts a latitude and a longitude as its parameters and returns city information (CityInfo)

Write some code to test out and ensure that your code works. Use the **locations_test.txt** file for test data. Extract location objects from the test data, and pass them into ISSDataRequest to get the corresponding **CityInfo** objects for each city.

> You are not allowed to modify any of the provided code in the ISSDataRequest class

Step 2: Creating our Buffer

The first step in our producer-consumer problem is to create a buffer that will hold the extracted **CityInfo** objects. We will use a **Queue** for this. Examine the **CityInfoQueue** class.

- **dataQueue**
  Shared vector of CityInfos

- **accessQueueMutex**
  Mutex to protect dataQueue access

- **dataIncoming**
  Boolean to indicate if more data is being added to the data_queue. This attribute should change to **False** after the producer threads have joined the main thread and finished processing all the cities.

- **void put(CityInfo) -> None:**
  This method is responsible for adding to the queue. Accept a CityInfo parameter and append it to the dataQueue list.

- **CityInfo get() -> CityInfo:**
  This method is responsible for removing an element from a Queue. Remember a queue is a FIFO data structure, that is it is First In First Out. Each call to this method should return the element at index 0 and delete it from the vector.

Complete the incomplete put and get methods and test out this data structure in **main**. Make sure this is working as expected before proceeding

Step 3: Creating a Producer and a Consumer Thread

We now need 2 classes that our future **threads** will call. Examine the following classes:

1. Producer

   The Producer class has the following methods:

   - **Producer(vector<Location>locations, CityInfoQueue &queue)**

     queue is shared between the Producer and Consumer

   - **void operator()():**
     This method is called automatically when we attach this object to a thread. It should loop over each **Location** and pass it to the **ISSDataRequest.get_city_info()** method. It then proceeds to add the CityInfo 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 threads with the Producer objects and join them to ensure this thread works as intended.

2. Consumer

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

   - **Consumer(CityInfoQueue &q):**

     Initializes the ConsumerThread with the same queue as the one the Producer has.

   - **Void operator()():**

     While the queue's **dataIncoming** is true OR the queue is not empty, this method should get an item from the queue and print it to the console and then sleep for **0.5** seconds. If the queue is empty while processing, 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 queue) 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 avoids 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.

Examine the **CityInfoQueue** class and find the **accessQueueLock**. Use this lock to control access to the code in the **put** and **get** methods. Is there anywhere else the dataQueue is accessed and needs protection? Test and run the code again to ensure it works as expected

You may want to use **locations_test.txt** for testing so you don't spend a lot of time waiting to process 100+ cities. You can add more cities to this txt file to gradually increase the amount of cities as well.

Step 5: Adding more producers

The final step is to create 3 or more producer threads and at least 2 consumers. Split the locations across the producer threads and start them. This should speed up your code and requests significantly.

- Ensure you push your work to github classroom. I'd like to see sensible git commits comments, and commits must take place at logical points in development.

That's it for this lab!

Sample output when running the completed program with the included locations_test.txt file. There are **3 producer** and **2 consumer** threads:

```
Consumer 1 is sleeping since queue is empty
Consumer 2 is sleeping since queue is empty
Producer 2 is adding to the queue
Producer 3 is adding to the queue
```

```
Element added to queue! Queue has 1 elements
Producer 1 is adding to the queue
Element added to queue! Queue has 2 elements
Element added to queue! Queue has 3 elements
Producer 1 is adding to the queue
Element added to queue! Queue has 4 elements
Producer 3 is adding to the queue
Producer 2 is adding to the queue
Element added to queue! Queue has 5 elements
Element added to queue! Queue has 6 elements
Producer 3 is adding to the queue
Element added to queue! Queue has 7 elements
Consumer 2 is consuming from the queue
Consumer 1 is consuming from the queue
Element removed from queue! Queue has 6 elements left
latitude: 45.500000
longitude: -73.583300
timezoneId: America/Toronto
offset: -4
Element removed from queue! Queue has 5 elements left
countryCode: CA
latitude: 54.130100
longitude: -108.434700
timezoneId: America/Swift_Current
offset: -6
countryCode: CA
mapUrl: https://maps.google.com/maps?q=54.1301,-108.4347&z=4

mapUrl: https://maps.google.com/maps?q=45.5,-73.5833&z=4

Consumer Consumer 1 is consuming from the queue
2 is consuming from the queue
Element removed from queue! Queue has 4 elements left
latitude: 50.017100
longitude: -125.250000
timezoneId: America/Vancouver
offset: -7
countryCode: CA
mapUrl: https://maps.google.com/maps?q=50.0171,-125.25&z=4

Element removed from queue! Queue has 3 elements left
latitude: 62.442000
longitude: -114.397000
timezoneId: America/Yellowknife
offset: -6
countryCode: CA
mapUrl: https://maps.google.com/maps?q=62.442,-114.397&z=4

Consumer 1 is consuming from the queue
Element removed from queue! Queue has 2 elements left
latitude: 49.273400
```

```
longitude: -123.121600
timezoneId: America/Vancouver
offset: -7
countryCode: CA
mapUrl: https://maps.google.com/maps?q=49.2734,-123.1216&z=4

Consumer 2 is consuming from the queue
Element removed from queue! Queue has 1 elements left
latitude: 53.016700
longitude: -112.816600
timezoneId: America/Edmonton
offset: -6
countryCode: CA
mapUrl: https://maps.google.com/maps?q=53.0167,-112.8166&z=4

Consumer 1 is consuming from the queue
Element removed from queue! Queue has 0 elements left
latitude: 51.000500
longitude: -118.183300
timezoneId: America/Vancouver
offset: -7
countryCode: CA
mapUrl: https://maps.google.com/maps?q=51.0005,-118.1833&z=4
```