Lab5 (Oct 19)

The objective of this lab is to practice writing multithreading programs that need synchronization using python's semaphores and locks.

Academic honesty and standard:

This is an individual assignment. The work you submit must be your own work. Do not share any part of your code or your design thinking. Remind yourself of the UBC policies on Academic honesty and standard.

Submissions:

Follow the requested specification exactly. We may use automated tests as a part of grading, so every unit (e.g. functions or methods) must be testable fully on own. You are allowed to use inner functions, but for consistency, do not add additional functions or methods.

Submit your python3 file (.py extension) to the associated Canvas assignment dropbox by the deadline.

Do not forget to include your name and student number as comments at the beginning of the submitted python file.

Due: **Thu Oct 19, by 13:59 PM** (common to everyone); Please plan ahead, no late submissions are accepted.

For marking, we will test for functionality first (must work), and we will check the code design, readability, compliance with the specification, documentation ... Write the best code you can.

Lab room usage rule for Lab 5:

A TA will be in the lab during your scheduled lab time to help you, and I do recommend that you use the lab time wisely, however, for this lab we will not record attendance and there is no scheduled demo for it.

Producer-consumer problem

In this lab we will implement a multithreaded program with a specific python implementation of the producer-consumer synchronization problem.

The implementation of the circular buffer is given to you, to be used as is. It is implemented using a python class. We use the simple data type of `int` for the items that are produced and

consumed.

The producer and consumer (running as separate threads) will move items to and from a circular buffer that is synchronized with two semaphores and a mutex lock.

Alternative to the producer-consumer code in the lecture slides, we do not use a *counter* here. Instead we use two ***counting*** semaphores `full` and `empty`. As shown in the code, the `full` semaphore is initialized to 0 (at the beginning there is nothing in the buffer) and the `empty` buffer is initialized to the buffer `SIZE` (at the beginning the buffer is *completely* empty). Note that we are using these *counting* semaphores so that the producer and the consumer know whether they can insert an item in or remove an item from the buffer, respectively, so there is rather a symmetry between the producer and the consumer. In might be helpful to interpret that the producer is producing *full* buffers for the consumer and the consumer is producing *empty* buffers for the producer.

Note that in the provided code, we are using some random delay and random item value selection (during production) to create some randomness for the producer-consumer interaction (as it is the case normally as such, but limited for simplicity).

You are to complete the implementation of the `producer` function, the `consumer` function, and the main thread for instantiating the consumer and producer threads and to correctly use the two semaphores `full` and `empty` (used to keep track of the correct use of the circular buffer as explained above) and the `mutex` lock (to protect the critical sections in the threads, on insertion of a new item or removal of an item).

Consider the following code template:

```
#student name:
#student number:

import threading
import random #is used to cause some randomness
import time   #is used to cause some delay in item production/consumption

class circularBuffer:
    """
        This class implement a barebone circular buffer.
        Use as is.
    """
    def __init__ (self, size: int):
        """
            The size of the buffer is set by the initializer
            and remains fixed.
        """
        self._buffer = [0] * size   #initilize a list of length size
                                    #all zeroed (initial value doesn't matter)
        self._in_index = 0   #the in reference point
        self._out_index = 0  #the out reference point

    def insert(self, item: int):
        """
            Inserts the item in the buffer.
```

```
            The safeguard to make sure the item can be inserted
            is done externally.
        """
        self._buffer[self._in_index] = item
        self._in_index = (self._in_index + 1) % SIZE

    def remove(self) -> int:
        """
            Removes an item from the buffer and returns it.
            The safeguard to make sure an item can be removed
            is done externally.
        """
        item = self._buffer[self._out_index]
        self._out_index = (self._out_index + 1) % SIZE
        return item

def producer() -> None:
    """
        Implement the producer function to be used by the producer thread.
        It must correctly use full, empty and mutex.
    """
    def waitForItemToBeProduced() -> int: #inner function; use as is
        time.sleep(round(random.uniform(.1, .3), 2)) #a random delay (100 to 300 ms)
        return random.randint(1, 99)  #an item is produced

    for _ in range(SIZE * 2): #we just produce twice the buffer size for testing
        item = waitForItemToBeProduced()  #wait for an item to be produced
        print(f"DEBUG: {item} produced")
        #complete the function below here to correctly store the item in the circular buffer


def consumer() -> None:
    """
        Implement the consumer function to be used by the consumer thread.
        It must correctly use full, empty and mutex.
    """
    for _ in range(SIZE * 2): #we just consume twice the buffer size for testing
        #write the code below to correctly remove an item from the circular buffer

        #end of your implementation for this function
        #use the following code as is
        def waitForItemToBeConsumed(item) -> None: #inner function; use as is
            time.sleep(round(random.uniform(.1, .3), 2)) #a random delay (100 to 300 ms)
            #to simulate consumption, item is thrown away here by just ignoring it
        waitForItemToBeConsumed(item)  #wait for the item to be consumed
        print(f"DEBUG: {item} consumed")

if __name__ == "__main__":
    SIZE = 5  #buffer size
    buffer = circularBuffer(SIZE)  #initialize the buffer

    full = threading.Semaphore(0)         #full semaphore: number of full buffers
                                          #initial value set to 0
    empty = threading.Semaphore(SIZE)     #empty semaphore: number of empty buffers
                                          #initial value set to SIZE
    mutex = threading.Lock()  #lock for protecting data on insertion or removal

    #complete the producer-consumer thread creation below
```

Implement the two functions first, and then complete the code under the `if __name__ == "__main__":` section.

Notes and Hints:

- Use `Thread`, `start()` and `join()`.
- Correctly use the shared semaphores (`full` and `empty`) and `mutex` lock. Use them as intended (see description above). Since we are using threading, these will be shared with the threads that our program is creating (similar concept as the previous lab).
- We are not using any thread pools. A usual, follow the specification.
- All threads are non-daemonic.
- To make testing manageable, in the producer() method, currently twice the buffer size for testing is produced. You may change 2 to a different value for further testing.

Note that the terminology used, like "full buffers" and "empty buffers" are the usually used terms (they are not mine per se and they may look a bit strange at first). When we say "full buffers", it does NOT mean we have multiple circular buffers. There is one circular buffer in the code all along (the one I have provided and instantiated as a part of the provided code). So "full buffers" as in "full/taken buffer spaces" rather. The initial values of the semaphores are helpful in the understanding of what they do. At the beginning, empty is SIZE (as if counting the number of empty spaces) and full is 0 (as if counting the number of full/taken spaces).

A sample output from the program is:

```
DEBUG: 1 produced
DEBUG: 93 produced
DEBUG: 1 consumed
DEBUG: 4 produced
DEBUG: 61 produced
DEBUG: 93 consumed
DEBUG: 80 produced
DEBUG: 4 consumed
DEBUG: 84 produced
DEBUG: 61 consumed
DEBUG: 37 produced
DEBUG: 80 consumed
DEBUG: 91 produced
DEBUG: 84 consumed
DEBUG: 41 produced
DEBUG: 37 consumed
DEBUG: 2 produced
DEBUG: 91 consumed
DEBUG: 41 consumed
DEBUG: 2 consumed
```

Write the best code you can. For the marking, we check that the program works, has correct logic, follows the specs, and passes all our tests. The program should be readable (good choice of identifiers, acceptable structure and styling, useful comments wherever needed ...), and does not do any repetitive work or extra work unnecessarily.

See the submission dropbox for the marking detail.

If you are adding "extra" print() for any additional debugging purposes, please comment them out before submission to Canvas.