Lab4 (Oct 12)

The objective of this lab is to practice writing multithreading programs using python.

Academic honesty and standard:

This is an individual assignment. The work you submit must be your own work. Obviously you should be able to fully explain and describe possible alternatives for any line of code you include in your submission. Do not share any part of your code or your design thinking. Remind yourself of the UBC policies on Academic honesty and standard.

Submissions:

CPEN333B: You will submit one python3 file (.py extension) for the implementation of part1 only. Submit the file to the associated Canvas assignment dropbox by the deadline.
CPEN333A: You will submit two python3 files (.py extension), one for the implementation of part1 and one for the implementation of part2. Make sure to clearly include the words part1 or part2 in the name of your submitted files. Submit the files to the associated Canvas assignment dropbox by the deadline.

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.

You will submit a python3 file (.py extension). Submit the file 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 each .py file.

Due: **Thu Oct 12, by 8:00 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 4:

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.

Part 1: Multithreaded sorting program

[Everybody is to complete this part of the lab.]

In this lab we will implement a multithreaded sorting program that sorts a list of integers (assume with a length always divisible by two).

As threads within a process shared the data section, we are going to use four shared variables in our program: `testcase`, `sortedFirstHalf`, `sortedSecondHalf`, and `SortedFullList`.

The program uses three threads to complete the sorting task: two sorting threads and one merging thread. The two sorting threads use the method `sortingWorker` function. The merging thread uses the method `mergingWorker`.

Depending on the function argument `firstHalf`, the sorting threads sort either the first half of the list or the second half of the list, and to store the result in either `sortedFirstHalf`, or `sortedSecondHalf` respectively.

The sorting is ascending, so [12, -1, 7, 7, 3, 50, 6, 8] after sorting will be [-1, 3, 6, 7, 7, 8, 12, 50]. You can implement any sorting algorithm of your choice but YOU are to code the algorithm. That is, using the *sort()* method for lists, or *sorted(),* or the like is not allowed. Document your implementation: indicate what sorting algorithm you are using and explain your code well.

Consider the following code template:

```
#student name:
#student number:

import threading

def sortingWorker(firstHalf: bool) -> None:
    """
       If param firstHalf is True, the method
       takes the first half of the shared list testcase,
       and stores the sorted version of it in the shared
       variable sortedFirstHalf.
       Otherwise, it takes the second half of the shared list
       testcase, and stores the sorted version of it in
       the shared variable sortedSecondHalf.
       The sorting is ascending and you can choose any
       sorting algorithm of your choice and code it.
    """
    pass #to Implement

def mergingWorker() -> None:
    """ This function uses the two shared variables
        sortedFirstHalf and sortedSecondHalf, and merges/sorts
        them into a single sorted list that is stored in
        the shared variable sortedFullList.
    """
    pass #to Implement

if __name__ == "__main__":
    #shared variables
    testcase = [8,5,7,7,4,1,3,2]
    sortedFirstHalf: list = []
    sortedSecondHalf: list = []
    SortedFullList: list = []

    #to implement the rest of the code below, as specified


    #as a simple test, printing the final sorted list
    print("The final sorted list is ", SortedFullList)
```

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

Notes and Hints:

- use `import threading`, `threading.Thread()`, `start()` and `join()`.
- The shared variables are great help in allowing threads to work on the data and share the result. Note that the program is designed so that no two threads should work on the same shared data, so synchronization is not an issue here.

The topic of synchronization is discussed extensively separately (so no need for *locks* ... in this program).

- We are not using any thread pools.
- We are using procedural paradigm for the code in this lab (to focus on threading) but it would be easy to rewrite the code using OOP if we wanted to.
- All threads are non-daemonic.
- You may want to review the "scope of variables" topic in python. There are some good examples on this in the posted Jupyter notebook that I demoed in the first lecture.
- What we are implementing here itself is a stage of a simplified/modified merge sort. Our implementation only considers the list's two halves, then each half is sorted by one thread and a final third thread sort-merges the two sorted half lists into a final sorted list.
- The original *testcase* list must not be mutated (it goes without saying).

Note: we are ignoring the fact that, as they are defined now, the threads are CPU-bound. I have kept it as such for simplicity and focus on threading. However, if, for example, the list was being provided by a networked database (like via the Internet), then it would be IO.

Part 2: Comparing Multithreaded and Sequential File Downloads from the Internet

[see the Submissions section at the top of this page on whether you need to do part2 as well or not]

In this part of the lab, we will compare the time in takes to download a certain number of files from the Internet when we download the files sequentially (no threading for the 1st portion: one file after the other) versus multithreading (delegating the download of each image file to a separate thread).

The amount of code you are to write is actually quite limited. The objective is mainly to complete the multithreading portion left for you and to see that it is quite faster (even for a few files in this case) when we download from the Internet using threading than sequentially.

The skeleton code imports a few modules that we have not discussed (see the comments in the code for each of those). No worries though. You should study the code well, but you would mainly focus on the threading aspects to complete the code you are to write.

If you have installed anaconda (or use the lab computers), you do not need to install anything, that is, all the modules needed for this part come with anaconda by default. Regardless, to check if your anaconda has a specific module, for example to check if it has `requests`, simply open an anaconda powershell (Windows) or the terminal (MacOs) and issue: `conda list requests`. In the rare case that you do not have any of these modules, do not use pip with anaconda (it may be needed for specific modules), use the anaconda's own package manager to install.

The provided function, `getComic`, downloads a specific comic image file from http://xkcd.com and saves it. For example, if the parameter `comicNumber` is 55, it will download the comic image file at http://xkcd.com/55.

The provided 1st portion of the code sequentially downloads a number of randomly chosen comics (how many is set by `comicCount`). We also use the time module to get an estimate of the time duration for this operation.

You are to complete the 2nd portion of this program (see the space left for you) where we are to use python's threading to use one separate thread for downloading each of the comic images. The `comicNumber`s for this part are store in `list2`.

Important Notes:

- Be very careful not to make a large number of requests in a relatively short period of time. You do not want to be considered as a malicious client or attacker. Comment out momentarily what you do not need while developing or debugging, and space out the requests.
- When you run the code, it will download 10 comic images to the same directory as your python code. You would need to regularly delete them or you can momentarily comment out the portion of the code that saves the file to the disk or momentarily decrease the number of downloads (in either of these cases, do remember to reverse this before submitting your file) .

Use the following skeleton (read the included comments):

```
#student name:
#student number:

import random    # used to randomly choose comic numbers
import requests  # used to make HTTP get requests and to download comic images
import bs4       # used to parse the html file and locate the image file
import time      # used to estimate time duration to compare between the two methods
import threading # used in the second portion of the program

def getComic(comicNumber: int) -> None:
    """
    This function downloads the image file associated with a specific
      comicNumber on http://xkcd.com
    Use as is.
    """
    if not isinstance(comicNumber, int):             # a simple safeguard against non-i
nteger values
        raise ValueError("Invalid comic number")

    url = f"http://xkcd.com/{comicNumber}/"

    try:
        comic = requests.get(url)     # GET request to get the webpage
        print(f"Comic #{comicNumber} status code: {comic.status_code} (200 means succes
s)")
        # locate the image from the content
        webpageContent = bs4.BeautifulSoup(comic.text,'html.parser')
        comicImageUrl = f"http:{webpageContent.select('#comic img')[0].get('src')}"
        # GET request to get the image file
        imageFile = requests.get(comicImageUrl)
    except:
        print(f"Something is not right with {url}")
    else:
        # save the image file (same folder as this python file)
        with open(f"comic{comicNumber}.{comicImageUrl[-3:]}", "wb") as outputfile: #ass
uming 3-char image extension only
            outputfile.write(imageFile.content)

comicCount = 5   # number of comics we are to download for each of the following progra
m portions

##############################
# 1st portion: we get the comics sequentially (no multithreading)
# Use this portion as is
list1 = random.sample(range(1, 100, 2), comicCount) # odd numbers only, not to overlap
with the other part
start = time.time() # start time to get an estimate of the duration it takes to downloa
d the comics
for num in list1:
    getComic(num)
print()
print(f"Sequentially, it took {time.time() - start :.3f}s to download {comicCount} rand
om comics")
```

```
print()

##############################
# 2nd portion: use multithreading - one thread for downloading each of the comics
# You are to complete this portion
list2 = random.sample(range(2, 100, 2), comicCount) # even numbers only, not to overlap
with the other part
start = time.time()
# Your code come below here




# end of your code
print()
print(f"Using threading, it took {time.time() - start :.3f}s to download {comicCount} r
andom comics")
print()
```

==========

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.