Lab6 (Nov 2)

The objective of this lab is to practice writing multiprocessing programs with synchronization and to avoid deadlock.

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.

You will submit three python3 files (.py extension), one for part1, one for part2, and one for part3 (name the files you submit to clearly have the words `part1`, `part2,` or `part3`). Submit the files 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 Nov 2, by 13:59 PM** (common to everyone)

Please plan ahead, no late submissions are accepted.

Lab room usage rule for this lab:

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.

Dinning-philosophers problem

In this lab, a multiprocessing implementation of the dining-philosophers synchronization problem is provided to you. It is rather based on the code we discussed during the lecture, and as we explained, it is susceptible to deadlock (susceptible is a key word here, you may run the program many times and deadlock may not happen).

Note that in the provided code, we are using some random delay to simulate the time a philosopher takes to eat or to think.

Consider the following code:

```
#student name:
#student number:

import multiprocessing
import random #is used to cause some randomness
import time   #is used to cause some delay to simulate thinking or eating times

def philosopher(id: int, chopstick: list):
    """
       implements a thinking-eating philosopher
       id is used to identifier philosopher #id (id is between 0 to numberOfPhilosophers-1)
       chopstick is the list of semaphores associated with the chopsticks
    """
    def eatForAWhile():   #simulates philosopher eating time with a random delay
        print(f"DEBUG: philosopher{id} eating")
        time.sleep(round(random.uniform(.1, .3), 2)) #a random delay (100 to 300 ms)

    def thinkForAWhile(): #simulates philosopher thinking time with a random delay
        print(f"DEBUG: philosopher{id} thinking")
        time.sleep(round(random.uniform(.1, .3), 2)) #a random delay (100 to 300 ms)

    for _ in range(6): #to make testing easier, instead of a forever loop we use a finite loop
        leftChopstick = id
        rightChopstick = (id + 1) % 5      #5 is number of philosophers

        #to simplify, try statement not used here
        chopstick[leftChopstick].acquire()
        print(f"DEBUG: philosopher{id} has chopstick{leftChopstick}")
        chopstick[rightChopstick].acquire()
        print(f"DEBUG: philosopher{id} has chopstick{rightChopstick}")

        eatForAWhile()  #use this line as is

        print(f"DEBUG: philosopher{id} is to release chopstick{rightChopstick}")
        chopstick[rightChopstick].release()
        print(f"DEBUG: philosopher{id} is to release chopstick{leftChopstick}")
        chopstick[leftChopstick].release()

        thinkForAWhile()  #use this line as is

if __name__ == "__main__":
    semaphoreList = list()          #this list will hold one semaphore per chopstick
    numberOfPhilosophers = 5

    for i in range(numberOfPhilosophers):
        semaphoreList.append(multiprocessing.Semaphore(1))    #one semaphore per chopstick

    philosopherProcessList = list()
    for i in range(numberOfPhilosophers): #instantiate all processes representing philosophers
        philosopherProcessList.append(multiprocessing.Process(target=philosopher, args=(i, semap
horeList)))
    for j in range(numberOfPhilosophers): #start all child processes
        philosopherProcessList[j].start()
    for k in range(numberOfPhilosophers): #join all child processes
        philosopherProcessList[k].join()
```

========================

A sample output of the original program is:

```
DEBUG: philosopher2 has chopstick2
DEBUG: philosopher2 has chopstick3
DEBUG: philosopher2 eating
DEBUG: philosopher1 has chopstick1
DEBUG: philosopher0 has chopstick0
DEBUG: philosopher4 has chopstick4
DEBUG: philosopher2 is to release chopstick3
DEBUG: philosopher3 has chopstick3
DEBUG: philosopher2 is to release chopstick2
DEBUG: philosopher1 has chopstick2
DEBUG: philosopher1 eating
DEBUG: philosopher2 thinking
DEBUG: philosopher1 is to release chopstick2
DEBUG: philosopher1 is to release chopstick1
DEBUG: philosopher0 has chopstick1
DEBUG: philosopher1 thinking
DEBUG: philosopher0 eating
DEBUG: philosopher2 has chopstick2
DEBUG: philosopher0 is to release chopstick1
DEBUG: philosopher0 is to release chopstick0
DEBUG: philosopher4 has chopstick0
DEBUG: philosopher0 thinking
DEBUG: philosopher4 eating
DEBUG: philosopher1 has chopstick1
DEBUG: philosopher4 is to release chopstick0
DEBUG: philosopher0 has chopstick0
DEBUG: philosopher4 is to release chopstick4
DEBUG: philosopher3 has chopstick4
DEBUG: philosopher4 thinking
DEBUG: philosopher3 eating
DEBUG: philosopher3 is to release chopstick4
DEBUG: philosopher4 has chopstick4
DEBUG: philosopher3 is to release chopstick3
DEBUG: philosopher2 has chopstick3
DEBUG: philosopher3 thinking
DEBUG: philosopher2 eating
DEBUG: philosopher2 is to release chopstick3
DEBUG: philosopher2 is to release chopstick2
DEBUG: philosopher1 has chopstick2
DEBUG: philosopher2 thinking
DEBUG: philosopher1 eating
DEBUG: philosopher3 has chopstick3
DEBUG: philosopher1 is to release chopstick2
DEBUG: philosopher1 is to release chopstick1
DEBUG: philosopher0 has chopstick1
DEBUG: philosopher0 eating
DEBUG: philosopher1 thinking
DEBUG: philosopher2 has chopstick2
DEBUG: philosopher0 is to release chopstick1
DEBUG: philosopher1 has chopstick1
DEBUG: philosopher0 is to release chopstick0
DEBUG: philosopher4 has chopstick0
DEBUG: philosopher0 thinking
```

```
DEBUG: philosopher4 eating
DEBUG: philosopher4 is to release chopstick0
DEBUG: philosopher0 has chopstick0
DEBUG: philosopher4 is to release chopstick4
DEBUG: philosopher4 thinking
DEBUG: philosopher3 has chopstick4
DEBUG: philosopher3 eating
DEBUG: philosopher3 is to release chopstick4
DEBUG: philosopher3 is to release chopstick3
DEBUG: philosopher2 has chopstick3
DEBUG: philosopher3 thinking
DEBUG: philosopher2 eating
DEBUG: philosopher4 has chopstick4
DEBUG: philosopher2 is to release chopstick3
DEBUG: philosopher2 is to release chopstick2
DEBUG: philosopher1 has chopstick2
DEBUG: philosopher2 thinking
DEBUG: philosopher1 eating
DEBUG: philosopher3 has chopstick3
DEBUG: philosopher1 is to release chopstick2
DEBUG: philosopher2 has chopstick2
DEBUG: philosopher1 is to release chopstick1
DEBUG: philosopher0 has chopstick1
DEBUG: philosopher0 eating
DEBUG: philosopher1 thinking
DEBUG: philosopher0 is to release chopstick1
DEBUG: philosopher1 has chopstick1
DEBUG: philosopher0 is to release chopstick0
DEBUG: philosopher4 has chopstick0
DEBUG: philosopher0 thinking
DEBUG: philosopher4 eating
DEBUG: philosopher4 is to release chopstick0
DEBUG: philosopher4 is to release chopstick4
DEBUG: philosopher4 thinking
DEBUG: philosopher3 has chopstick4
DEBUG: philosopher3 eating
DEBUG: philosopher0 has chopstick0
DEBUG: philosopher3 is to release chopstick4
DEBUG: philosopher4 has chopstick4
DEBUG: philosopher3 is to release chopstick3
DEBUG: philosopher2 has chopstick3
DEBUG: philosopher3 thinking
DEBUG: philosopher2 eating
DEBUG: philosopher2 is to release chopstick3
DEBUG: philosopher3 has chopstick3
DEBUG: philosopher2 is to release chopstick2
DEBUG: philosopher1 has chopstick2
DEBUG: philosopher2 thinking
DEBUG: philosopher1 eating
DEBUG: philosopher1 is to release chopstick2
DEBUG: philosopher2 has chopstick2
DEBUG: philosopher1 is to release chopstick1
DEBUG: philosopher0 has chopstick1
DEBUG: philosopher0 eating
DEBUG: philosopher1 thinking
DEBUG: philosopher0 is to release chopstick1
DEBUG: philosopher1 has chopstick1
DEBUG: philosopher0 is to release chopstick0
DEBUG: philosopher0 thinking
DEBUG: philosopher4 has chopstick0
DEBUG: philosopher4 eating
DEBUG: philosopher4 is to release chopstick0
```

```
DEBUG: philosopher0 has chopstick0
DEBUG: philosopher4 is to release chopstick4
DEBUG: philosopher3 has chopstick4
DEBUG: philosopher3 eating
DEBUG: philosopher4 thinking
DEBUG: philosopher3 is to release chopstick4
DEBUG: philosopher4 has chopstick4
DEBUG: philosopher3 is to release chopstick3
DEBUG: philosopher2 has chopstick3
DEBUG: philosopher3 thinking
DEBUG: philosopher2 eating
DEBUG: philosopher2 is to release chopstick3
DEBUG: philosopher2 is to release chopstick2
DEBUG: philosopher1 has chopstick2
DEBUG: philosopher2 thinking
DEBUG: philosopher1 eating
DEBUG: philosopher3 has chopstick3
DEBUG: philosopher1 is to release chopstick2
DEBUG: philosopher2 has chopstick2
DEBUG: philosopher1 is to release chopstick1
DEBUG: philosopher0 has chopstick1
DEBUG: philosopher0 eating
DEBUG: philosopher1 thinking
DEBUG: philosopher0 is to release chopstick1
DEBUG: philosopher1 has chopstick1
DEBUG: philosopher0 is to release chopstick0
DEBUG: philosopher4 has chopstick0
DEBUG: philosopher0 thinking
DEBUG: philosopher4 eating
DEBUG: philosopher4 is to release chopstick0
DEBUG: philosopher4 is to release chopstick4
DEBUG: philosopher3 has chopstick4
DEBUG: philosopher4 thinking
DEBUG: philosopher3 eating
DEBUG: philosopher0 has chopstick0
DEBUG: philosopher3 is to release chopstick4
DEBUG: philosopher3 is to release chopstick3
DEBUG: philosopher2 has chopstick3
DEBUG: philosopher2 eating
DEBUG: philosopher3 thinking
DEBUG: philosopher4 has chopstick4
DEBUG: philosopher2 is to release chopstick3
DEBUG: philosopher3 has chopstick3
DEBUG: philosopher2 is to release chopstick2
DEBUG: philosopher1 has chopstick2
DEBUG: philosopher1 eating
DEBUG: philosopher2 thinking
DEBUG: philosopher1 is to release chopstick2
DEBUG: philosopher1 is to release chopstick1
DEBUG: philosopher0 has chopstick1
DEBUG: philosopher0 eating
DEBUG: philosopher1 thinking
DEBUG: philosopher0 is to release chopstick1
DEBUG: philosopher0 is to release chopstick0
DEBUG: philosopher4 has chopstick0
DEBUG: philosopher0 thinking
DEBUG: philosopher4 eating
DEBUG: philosopher4 is to release chopstick0
DEBUG: philosopher4 is to release chopstick4
DEBUG: philosopher4 thinking
DEBUG: philosopher3 has chopstick4
DEBUG: philosopher3 eating
```

```
DEBUG: philosopher3 is to release chopstick4
DEBUG: philosopher3 is to release chopstick3
DEBUG: philosopher3 thinking
```

==========================

Part 1:

As a possible solution to avoid deadlock, "our program as usual will spawn five processes to represent the five philosophers, but we only allow four philosophers to be sitting at the dining table (so to speak) to eat at any given time to limit the number of philosophers at the table. "

You are to modify the code to implement the solution based on the above statement. Submit a .py file for this part (make sure your filename include part1).

Depending on your approach, you are allowed to add any additional synchronization primitives (only the ones we have discussed in class obviously) that you may need. But, don't use more than what is obsoletely necessary. If used, they must be defined in the main process and be passed to the child processes as one of the `args` elements for their use (e.g. similar to the chopstick semaphoreList).

As usual, document very well your approach and reasoning by providing helpful comments.

==========================

Part 2:

We discussed that one possible solution to avoid deadlock is to "allow a philosopher to pick up her chopsticks only if both chopsticks are available".

You are to modify the code to implement the solution based on the above statement. Submit a .py file for this part (make sure your filename include part2).

Depending on your approach, you are allowed to add any additional synchronization primitives (only the ones we have discussed in class obviously) that you may need. But don't use more than what is obsoletely necessary. If used, they must be defined in the main process and be passed to the child processes as one of the `args` elements for their use (e.g. similar to the chopstick semaphoreList).

As usual, document very well your approach and reasoning by providing helpful comments.

==========================

Part 3:

We discussed that another possible solution to avoid deadlock is to "use an asymmetric solution, e.g. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right first and then her left chopstick".

You are to modify the code to implement the solution based on the above statement. Submit a .py file for this part (make sure your filename include part3).

As before, the philosophers are numbered 0 to 4, and for the philosopher id, her left chopstick is at index id, and her right chopstick is at (id+1)%5.

========================

Notes and Hints:

- Obviously, the shown sample output is just a possible sample.
- Each philosopher does this circularly, that is, eat-think-eat-think-eat-think ... as long as the loop runs and under the synchronization constraints.
- A good starting point would be to identify what condition creates a deadlock in the original code and then examine that your solution (for either part) avoids deadlock.
- The two inner functions are used solely for readability (so that the thinking state and the eating state stand out, instead of seeing the code used for simulating them). One can replace the function calls with the two liners in the body of those functions.

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. Write the best code you can.

See the submission dropbox for the marking detail.