IT5002 Computer Systems and Applications
Term Assignment

0. Instructions

This assignment is worth 20 marks. Fill in your answers in the enclosed answer book and submit to the Assignment Submission Folder on Canvas by **2359 on 6 November 2023.**

1. Introduction

Processes are a key part of operating systems, and all modern operating systems support multi-processing, where CPU time is divided rapidly between many processes, giving the illusion that these processes are running simultaneously.

In this assignment we will look at how to do multiprogramming in Python. It is largely introductory and will barely scratch the surface of what the multiprogramming libraries in Python offer.

2. Multiprogramming on Python

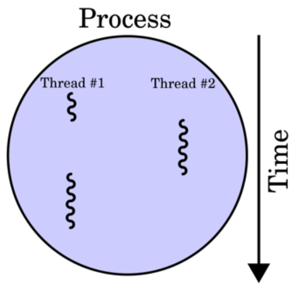
Python offers two ways to perform multiprogramming; either the “threading” library or the “multiprocessing” library. The “threading” library provides “multithreading” in which multiple “execution” threads execute within a single process, whereas the “multiprocessing” library creates multiple processes.

Since threads exist within a single process, they can share memory, and hence global variables. Multiple processes on the other hand exist in their own memory spaces, and hence cannot share global variables. We will explore this difference first.

Both the threading and multiprocessing libraries have almost identical interfaces.

3. The Python Threading Library

A “thread” is an independent sequence of instructions that executes within a process. The figure below illustrates this idea:



Recall that the Operating System allocates space for each process. All the threads in the process will share this memory, and hence are able to share global variables. This is in contrast with processes, which cannot share memory because the Operating System allocates a separate memory space for each process.

a. Creating Threads

In Python (like in most languages), the code for a thread is defined as a function. Create a program called "t1.py" and enter the following:

```
import threading
import time

x = 1

# First thread. We pass in one parameter
def thread1(param):
    global x
    while True:
        x = x + 1
        print("This is thread 1. x is %d" % x)
        time.sleep(param)

# Second thread. We pass in two parameters
def thread2(param1, param2):
    global x
    while True:
        x = x * 2
        print("This is thread 2. Param 2 is %d. x is %d" % (param2, x))
        time.sleep(param1)

# Create the first thread
th1 = threading.Thread(target = thread1, args = [1, ])
th1.start()
th2 = threading.Thread(target = thread2, args =  [2, 123])
th2.start()
```

Some explanation of this code:

i. We create two functions called "thread1" and "thread2" that run infinitely. Both print out a message and the value of a shared variable x. The "thread1" function takes one parameter, while the "thread2" function takes two parameters.

ii. To create the first thread we use the "Thread" function from the "threading" library. This returns a "handle" to the thread which we assign to th1:

    th1 = threading.Thread(target = thread1, args=[1,])

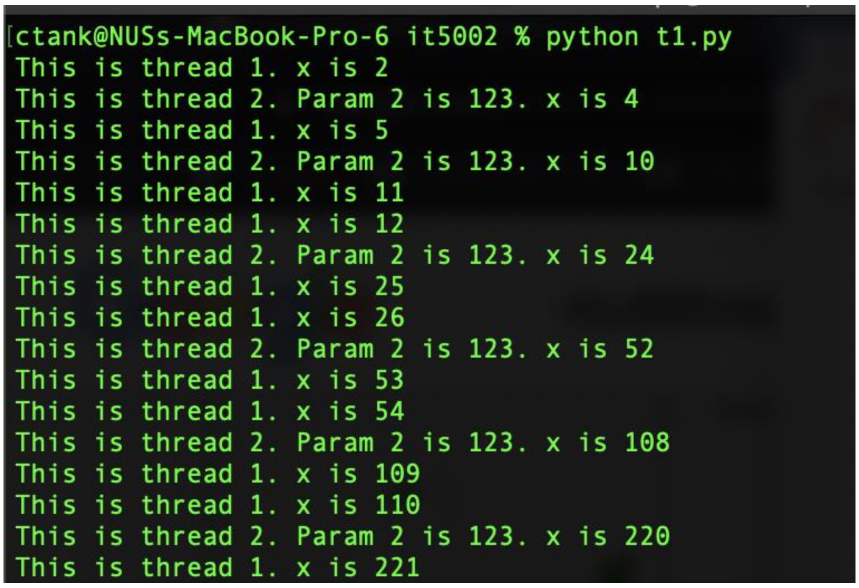
    The Thread function takes 2 parameters:

    target: The function that contains the code for the thread.
    args: Arguments to be passed to the thread on startup.

iii. Calling Thread only creates the thread, but does not start it. To start the thread:

    th1.run()

When run, we get:



```
[ctank@NUSs-MacBook-Pro-6 it5002 % python t1.py
This is thread 1. x is 2
This is thread 2. Param 2 is 123. x is 4
This is thread 1. x is 5
This is thread 2. Param 2 is 123. x is 10
This is thread 1. x is 11
This is thread 1. x is 12
This is thread 2. Param 2 is 123. x is 24
This is thread 1. x is 25
This is thread 1. x is 26
This is thread 2. Param 2 is 123. x is 52
This is thread 1. x is 53
This is thread 1. x is 54
This is thread 2. Param 2 is 123. x is 108
This is thread 1. x is 109
This is thread 1. x is 110
This is thread 2. Param 2 is 123. x is 220
This is thread 1. x is 221
```

Notice that x is accessible to both threads. The "thread1" thread increments it by one once per second, and "thread2" doubles x every 2 seconds.

Question 1. (3 marks)

Examine the code you have typed in, and explain how this code works. That is, why it produces two threads that print once per second and once every two seconds.

b. <u>Incorrect Thread Execution</u>

This parallel execution of threads is very cool, but can lead to incorrect execution. For example, the two threads below sum from 1 to 10 (to give an answer of 55). The first thread sums from 1 to 5 and the second from 6 to 10, and the main program will give the final result. We will store this in the shared variable "sum":

Create a file called "t2.py" and type in the following code:

```
import threading

# The number to add.
# Change this for your experiment
n = 10

# Result variable
result = 0

def thread1(count):
    global result

    x = 0
    for i in range(1, count + 1):
        x = x + i
    result = x
```

```
def thread2():
    global result
    result = result * 3


th1 = threading.Thread(target = thread1, args=[n, ])
th2 = threading.Thread(target = thread2)

th1.start()
th2.start()

correct = n * (n + 1) / 2 * 3
print("Correct result is %d." % correct)
print("Final result is %d." % result)

if result == correct:
    print("CORRECT!")
else:
    print("WRONG!")
```

In this program we create two threads; the first ("thread1") takes a parameter "count" and sums from 1 to count storing it in a global variable called "result". The second thread takes "result" computed by "thread1" and multiplies it by 3. The main routine creates the two threads:

```
th1 = threading.Thread(target = thread1, args=[n, ])
th2 = threading.Thread(target = thread2)
```

Here the th1 thread takes an argument "n" which sets tells "thread1" what to sum up to (i.e. "thread1" does 1 + 2 + 3 + … + n).

Since $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$, our final correct result should be $\frac{3n(n+1)}{2}$. The main routine calculates this and puts it into "correct".
When "thread1" and "thread2" finish running, our main routine will compare "result" against "correct" to see if the threads compute correctly.

Question 2a. (2 marks)

Run the code with the following values of "n", recording whether you get the correct result:

n = 10, Correct: Y / N
n = 1000, Correct: Y / N
n = 1000000, Correct: Y/ N
n = 10000000, Correct: Y/N

Question 2b. (4 marks)

Some of the answers in 2a are incorrect. Explain why you get incorrect results.

c. <u>Coordinating Threads</u>

We will now set n = 10000000 and see how we can coordinate the two threads using a structure called a Queue.

To create a queue, we import the queue library and make a Queue object. We call the "put" function to place a result in the queue, and "get" to get results placed in a queue.

We will now create a version of t2.py, using queues to coordinate the threads. Create a file called "t3.py" and enter the following code, and run the code.

```
import threading
from queue import Queue

resultQ = Queue()
finalQ = Queue()

# The number to add.
# Change this for your experiment
n = 10

def thread1(count):
    global resultQ

    x = 0
    for i in range(1, count + 1):
        x = x + i

    resultQ.put(x)

def thread2():

    global resultQ
    global finalQ

    result = resultQ.get()
    result = result * 3

    finalQ.put(result)


th1 = threading.Thread(target = thread1, args=[n, ])
th2 = threading.Thread(target = thread2)

th1.start()
th2.start()

correct = n * (n + 1) / 2 * 3
print("Correct result is %d." % correct)
```

```
finalResult = finalQ.get()
print("Final result is %d." % finalResult)

if finalResult == correct:
    print("CORRECT!")
else:
     print("WRONG!")
```

Run this program and answer the following question:

Question 3. (4 marks)

t3.py now shows the correct result even for large n. Explain, using the idea of queues, how this program works to give the correct result. You may Google for what queues are.

4. The Multiprocessing Library

In the Threading library we saw how to create threads of execution in a single process that run simultaneously. Since the threads are in a single process, they can share memory and hence global variables, as we have seen.

We will now look at how to use the multiprocessing library to create a program with multiple processes. Unlike threads however, each process has its own memory and multiple processes cannot share variables.

Create a file called "p1.py" and key in the following code. It is identical to t1.py except for the following:

i) We use the multiprocessing library instead of threading.
ii) We use the Process call to create new processes.
iii) We changed the names of thread1 and thread2 to process1 and process2, just for aesthetic reasons.
iv) We moved the code to create processes into a function called "main", then added the lines:

```
if __name__ == '__main__':
    main()
```

This causes main() to be run only if this script is run from the command line (e.g. "python p1.py") rather than being imported. This is necessary for the multiprocessing library to work correctly.

```
import multiprocessing
import time

x = 1

# First thread. We pass in one parameter
def process1(param):
    global x
    while True:
        x = x + 1
        print("This is process 1. x is %d" % x)
        time.sleep(param)

# Second process. We pass in two parameters
def process2(param1, param2):
    global x
    while True:
        x = x * 2
        print("This is process 2. Param 2 is %d. x is %d" % (param2, x))
        time.sleep(param1)

def main():
    # Create the first thread
    p1 = multiprocessing.Process(target = process1, args = [1, ])
    p1.start()
    p2 = multiprocessing.Process(target = process2, args =  [2, 123])
    p2.start()

if __name__ == '__main__':
    main()
```

Run the program and answer the following question:

Question 4. (2 marks)

Since p1.py and t1.py are essentially the same program, they should produce the same results, but do not. Explain why.

You can use queues to fix p1.py so that it produces the same result as t1.py. To do this:

a. Change process1 so that it increments x by 1 and sends it over to process2. It then waits for the result from process2.
b. Change process2 so that it waits for the result from process1, multiplies it by 2 then sends it back to process1.

IMPORTANT NOTE: For this to work correctly, you must use the multiprocessing version of Queue. So within your code, import Queue from multiprocessing instead of from queue. That is:

DO NOT DO THIS:

from queue import Queue

DO THIS INSTEAD:

from multiprocessing import Queue

Question 5. (5 marks)

Detail the changes you made to p1.py to make it give the same results at t1.py below and explain why it works.