Python Multithreading

Review of Terms

- Multitasking - running multiple tasks (programs) concurrently by switching back and forth between them; switching normally occurs if there is a higher-priority program that needs to run, or the current program blocks or runs out of its time quantum

- Multithreading - running multiple **functions/methods** of the **same program** concurrently by switching back and forth between them; switching normally occurs if there is a higher-priority thread that needs to run, or if the current thread blocks or runs out of its time quantum

- Interprocess Communication - communication between different programs running on the same system (e.g., copy and paste between different programs; communication via localhost)

- Network Communication - communication between different programs running on different systems (e.g., between your "A" machine and your VM)

Why Multithreading?

- The networked game in Lab 4 accepts one connection at a time and handles one command per connection
- This involves a lot of overhead due to setting up and tearing down connections (for sending 1 byte of data to the server and getting as little as 4 bytes back, at least 280 bytes of header information are exchanged)
- You can see this clearly in Wireshark!
- We can cut down on bandwidth by keeping the network connection open for the entirety of the game
- Problem: How do we deal with concurrent connections?
- Solution: Assign 1 thread per connection

Example Run

```
#!/usr/bin/python3.11

from threading import Thread
from time import sleep


def called_by_thread(name):
    while True:
        print(name)
        sleep(2)


for i in range(10):
    Thread(target=called_by_thread, args=(i,)).start()
```

0	0	0
1	6	2
2	7	6
3	9	9
4	5	5
5	8	8
6	3	1
8	2	7
9	4	3
7	1	4

Note that the order is not guaranteed!

Example Program

- Creates 10 threads, each of which runs its own version of *called_by_thread*
- The *name* will be different for each thread (i.e., 0 - 9)
- The order of thread execution is not guaranteed
- When creating the thread, *called_by_thread* must not have trailing *()*s; we are passing in the **name** of the function to *Thread*; we are **not** calling it at this time
- Arguments passed to *called_by_thread* must be in form of a tuple; because *called_by_thread* only needs a single argument, we must have a trailing comma to indicate that it is part of a tuple

Another Example

```
#!/usr/bin/python3.11

from threading import Thread

num = 0


def called_by_thread():
    global num
    for _ in range(1000000):
        num += 1


thread_list = []
for i in range(10):
    thread_list.append(Thread(target=called_by_thread))
    thread_list[-1].start()

# Wait until all threads have completed
for t in thread_list:
    t.join()

print(num)
```

Race Conditions

- This will cause a race condition (harder to reproduce in Python 3.11)
- The value should be 10,000,000, but due to race conditions, the result will almost always be less

Dockerfile

```
FROM python:3.7

RUN mkdir /program

COPY race_condition.py /program/

CMD [ "python", "/program/race_condition.py" ]
```

Docker Commands

- Build the container:
  sudo docker build -t race_condition .
- Run the container:
  sudo docker run --rm --name race_condition race_condition
- Run it multiple times; the output should differ every time

Normal Run: num += 1

Thread 1	Thread 2
tmp = num (e.g., 0)	
add 1 to tmp (1 + 0)	
save tmp to num (1)	
	tmp = num (1)
	add 1 to tmp (1 + 1)
	save tmp to num (2)

Conflict Run: num += 1

Thread 1	Thread 2
tmp = num (e.g., 0)	
	tmp = num (still 0)
	add 1 to tmp (1 + 0)
	save tmp to num (1)
add 1 to tmp (but scope of tmp is local to thread, so still 1 + 0)	
save tmp to num (1)	

Mutual Exclusion

- One solution is to block threads from interrupting each other while accessing a shared variable
- That way, the race condition cannot occur

Adding Mutual Exclusion

```
#!/usr/bin/python

from threading import Semaphore, Thread


lock = Semaphore()
num = 0


def called_by_thread():
    global num
    for _ in range(1000000):
        with lock:  # lock.acquire() and lock.release() also works
            num += 1


thread_list = []
for i in range(10):
    thread_list.append(Thread(target=called_by_thread))
    thread_list[-1].start()

for t in thread_list:
    t.join()

print(num)
```

Queues

- Another solution is to compute the value locally and then transmit the result via message passing
- So each thread would run tmp += 1 for 1,000,000 iterations, then send the result to a common server, which would add up the results locally
- Significantly faster than using mutual exclusion

Adding Queues

```
from threading import Thread
from queue import Queue

input_queue = Queue()
output_queue = Queue()

def called_by_thread(name):
    tmp = 0
    val = input_queue.get()
    for _ in range(val):
        tmp += 1
    output_queue.put(tmp)

thread_list = []
for i in range(10):
    thread_list.append(Thread(target=called_by_thread))
    thread_list[-1].start()
    input_queue.put(1000000)
for t in thread_list:
    t.join()

num = 0
while not output_queue.empty():
    num += output_queue.get()
print(num)
```

Exercises

- On your own:
  - Work on the questions in the *Threading* section of *Practice Questions and Solutions*

Lab 5

- Turn the server into a multithreaded server
- Be sure to use mutual exclusion when accessing the board!

Key Skills

- Explain multithreading, multitasking, interprocess communication, and network communication
- Write multithreaded networking programs in Python