Lab 7 - Toy Synchronization Problems (Hardware Solutions)

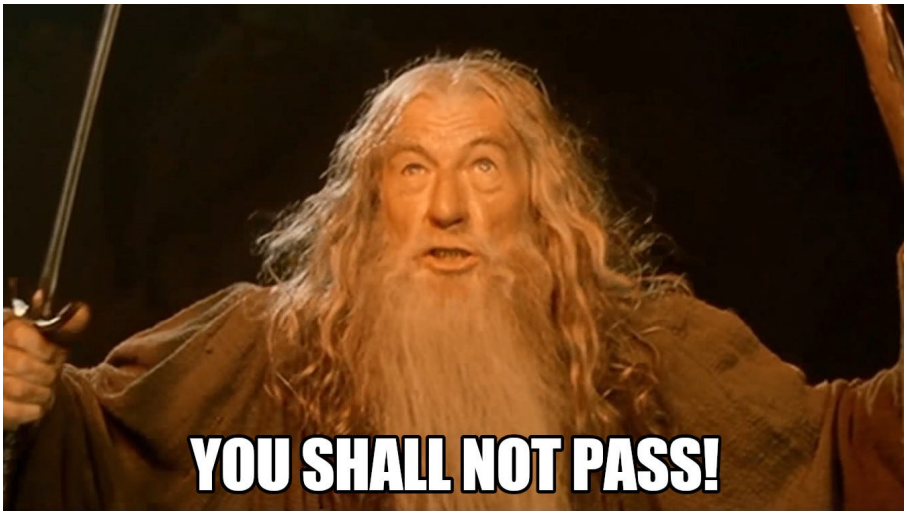

Figure 1: When threads need to work together, our mutex and sempahore heroes makes sure they do so smoothly and without problems

Welcome to this at-home lab where we're diving into some cool challenges. Imagine threads as busy little workers - sometimes they're writing to a database, other times they're researchers grabbing a bite, and sometimes they're even Mother Nature making water molecules! Below are some interesting and popular problems you can read about.

1. Reader-Writer Problem
2. Dining Philosophers Problem
3. Dining Savages Problem
4. Readers-Writers
5. Cigarette Smokers Problem
6. H2O Problem
7. Sleeping Barber Problem
8. Santa Claus Problem
9. Bridge Crossing Problem
10. Roller Coaster Problem

Remember, these are well-known problems with clear descriptions widely available. If you can't find them, just ask your teacher for help.

Just a reminder, you can't copy solutions from the internet. We want you to think and solve these puzzles yourself. It's fine to ask for a bit of help, but your solutions must be your own work. You can write a report or make a video explaining your solutions. Your work should show that you truly understand the problem. We've explained how to approach the bounded buffer problem step by step below. Your solution should also be built step by step, clearly and elegantly explained.

You can work in teams of 3-4 students

1 The Producer-Consumer Problem

We have a bit of intuition about this problem from our previous labs. Today, let us try to solve it in entirety.

Major Aspects

1. Producers: Generate items.
2. Consumers: Consume items.
3. Shared Buffer: Limited in size.
4. Challenge: Ensure producers wait when the buffer is full, and consumers wait when it's empty.

1.1 The most naive attempt

Below is the code we were working with so far, which had a problem of race conditions.

Listing 1: Producer-Consumer Problem with Race Conditions

```
import threading

# Shared variable
count = 0

# Producer function
def producer():
        global count
        for i in range(1000000):
                count += 1

# Consumer function
def consumer():
        global count
        for i in range(1000000):
                count -= 1

# Create producer and consumer threads
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

# Start the threads
producer_thread.start()
consumer_thread.start()

# Wait for both threads to finish
producer_thread.join()
consumer_thread.join()

print("Production and consumption finished, count=",count)
```

The above code demonstrates a race condition where two threads, a producer and a consumer, simultaneously modify a shared variable 'count' without synchronization. The producer increases 'count' by 1,000,000, while the consumer decreases it by the same amount. Due to the lack of synchronization, the final value of 'count' is unpredictable and varies each time the code runs. This showcases the challenges of concurrent programming without proper synchronization mechanisms.

1.2 Mutex to Rescue

The problem lies in the shared variable count being accessed simultaneously by both the producer and consumer threads. To fix this, we use a simple solution called a "mutex." A mutex acts like a traffic signal, allowing only

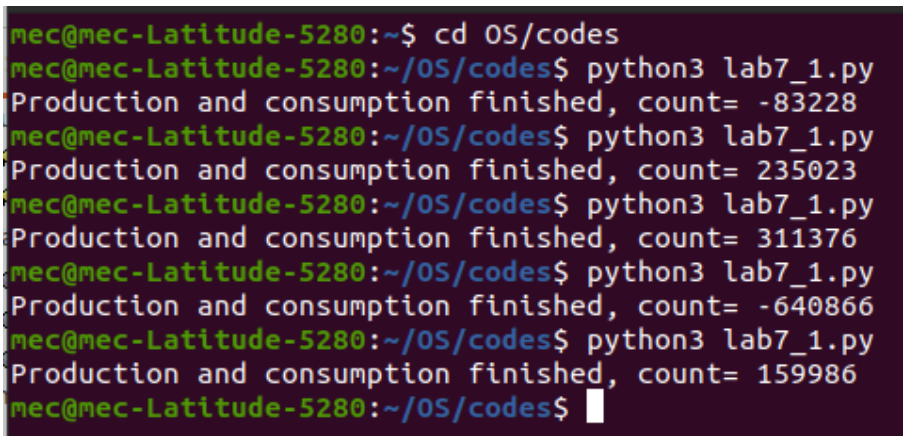

Figure 2: Output of listing 1

one thread to access the critical sections of code at a time. By using a mutex, we can ensure that the count++ operation for the producer and count– operation for the consumer occur without interference, making our program work smoothly.

In Python, creating a mutex is simple. You can do it like this:

```
# Creating a mutex (Lock) object
mutex = threading.Lock()
```

Then, to control access to critical sections, you use ‘with mutex‘ in your code. It’s as straightforward as that!

```
import threading

# Shared variable
count = 0
my_mutex = threading.Lock()

# Producer function
def producer():
        global count
        for i in range(1000000):
                with my_mutex:
                        count += 1

# Consumer function
def consumer():
        global count
        for i in range(1000000):
                with my_mutex:
                        count -= 1

# Create producer and consumer threads
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

# Start the threads
producer_thread.start()
consumer_thread.start()

# Wait for both threads to finish
producer_thread.join()
consumer_thread.join()

print("Production and consumption finished, count=",count)
```

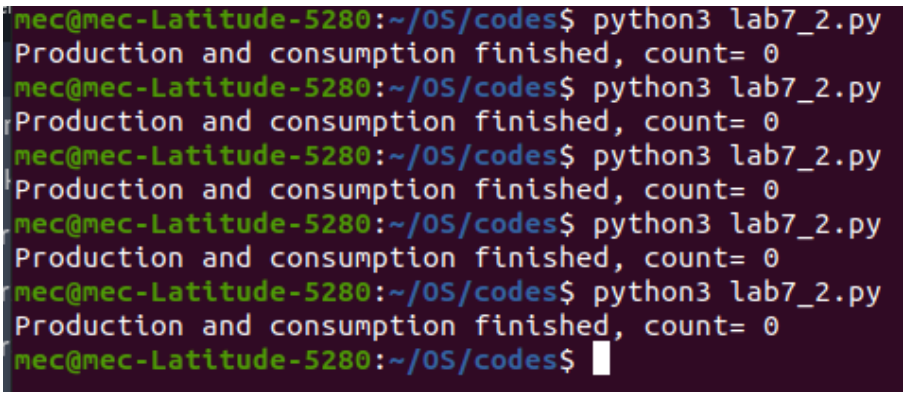

Figure 3: Output of listing 2

In this code, the `Lock` object (`mutex`) ensures that the `count` variable is accessed in a thread-safe manner, preventing race conditions. The `with mutex:` statement indicates the critical sections where `count` is being modified, ensuring mutual exclusion.

But, there's still a challenge. We need a place (a 'buffer') where items are stored and taken out. Let's keep it simple: imagine items as numbers from a counting loop. For example, if the loop is at 3, the item in the buffer is 3. You could make it more realistic using random numbers, but that takes more time. Now, both the producer and consumer can't mess with this buffer at the same time. We need to be careful about that too!

1.3 Let's make it Realistic

In this new version, we're using a list as our storage space, like a basket. The producer adds numbers by putting them in the basket (appending to the list), and the consumer takes them out (popping from the list). We don't need the 'count' variable anymore; the list itself keeps track of our items, just like a real shopping list!

```
import threading

# Shared variable
shared_buffer=[]
my_mutex = threading.Lock()

# Producer function
def producer():
        global count
        for i in range(400):
                with my_mutex:
                        shared_buffer.append(i)
                        print("inserted",i,"in buffer")
                        print("Buffer=",shared_buffer)

# Consumer function
def consumer():
        global count
        for i in range(400):
                with my_mutex:
                        j=shared_buffer.pop()
                        print("Retreived",j,"from buffer")
                        print("Buffer=",shared_buffer)


# Create producer and consumer threads
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

# Start the threads
```

```
producer_thread.start()
consumer_thread.start()

# Wait for both threads to finish
producer_thread.join()
consumer_thread.join()

print("Production and consumption finished, buffer=",shared_buffer)
```

You'd think that when the code finishes, the basket (or buffer) should be empty. But when you run this code, even with a small number of tries, you'll notice a problem, as we'll see next. This is the output received when the code is ran for 10 iterations. Note that you should run the code many times to notice a flaw.

```
    inserted 0 in buffer
Buffer= [0]
inserted 1 in buffer
Buffer= [0, 1]
inserted 2 in buffer
Buffer= [0, 1, 2]
inserted 3 in buffer
Buffer= [0, 1, 2, 3]
Retreived 3 from buffer
Buffer= [0, 1, 2]
Retreived 2 from buffer
Buffer= [0, 1]
Retreived 1 from buffer
Buffer= [0]
Retreived 0 from buffer
Buffer= []
Exception in thread Thread-2:
inserted 4 in buffer
Traceback (most recent call last):
Buffer= [4]
  File "/usr/lib/python3.8/threading.py", line 932, in _bootstrap_inner
inserted 5 in buffer
Buffer= [4, 5]
inserted 6 in buffer
Buffer= [4, 5, 6]
inserted 7 in buffer
Buffer= [4, 5, 6, 7]
inserted 8 in buffer
Buffer= [4, 5, 6, 7, 8]
inserted 9 in buffer
Buffer= [4, 5, 6, 7, 8, 9]
    self.run()
  File "/usr/lib/python3.8/threading.py", line 870, in run
    self._target(*self._args, **self._kwargs)
  File "lab7_3.py", line 21, in consumer
    j=shared_buffer.pop()
IndexError: pop from empty list
Production and consumption finished, buffer= [4, 5, 6, 7, 8, 9]
```

The issue was that the consumer tried to take something from an empty basket. One way to fix it is to make the consumer wait until there's something in the basket. But a smarter way is to use a semaphore. It's faster because it doesn't make the consumer constantly check if there's something in the basket.

1.4 Bounded buffer problem

We not only want the consumer to take items when the basket isn't empty but also want the producer to respect the basket's size. Now, we have what's called the 'bounded buffer problem.' Imagine a restaurant with only 10 seats or a basket that can only hold 10 items. If there are no items (basket is empty), the consumer can't take

anything. If there are already 10 items (basket is full), the producer can't add more. Now, we're shifting our attention from the mutex to the semaphore.

Here's the trick: when the producer adds an item, it's like saying 'go ahead' to the next customer. This operation is called 'signal' and it increases the semaphore value. When the consumer wants to take an item, it has to ask 'is there something?' by using 'wait.' This keeps things organized and prevents chaos in our little 'restaurant'.

You can use semaphores in Python with following simple commands.

```
semaphore = threading.Semaphore(starting_value)
semaphore.acquire()   #for wait()
semaphore.release()   #for signal()
```

Our code changes as following. Compare it with the previous code, only a few lines have been modified.

```
import threading

# Shared variable
shared_buffer=[]
my_mutex = threading.Lock()
num_items = threading.Semaphore(0)
# Producer function
def producer():
        global count
        for i in range(100):
                with my_mutex:
                        shared_buffer.append(i)
                        print("inserted",i,"in buffer")
                        print("Buffer=",shared_buffer)
                num_items.release()   #for signal()


# Consumer function
def consumer():
        global count
        for i in range(100):
                num_items.acquire()   #for wait()
                with my_mutex:
                        j=shared_buffer.pop()
                        print("Retreived",j,"from buffer")
                        print("Buffer=",shared_buffer)


# Create producer and consumer threads
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

# Start the threads
producer_thread.start()
consumer_thread.start()

# Wait for both threads to finish
producer_thread.join()
consumer_thread.join()

print("Production and consumption finished, buffer=",shared_buffer)
```

The code we discussed earlier solved the issue of consumer trying to take items from an empty basket. But we still haven't solved the problem of the basket becoming too full. We need to limit the maximum number of items it can hold. Also, the producer should 'wait' if there's no space in the basket (when there's no

room left), and the consumer should 'signal' when there's space (when some items are taken out). Think about how to do this!

1.5 The Final Solutin

So, below is the full fledged solution. When you run this code, you will not notice the size of the buffer beyond 10.

```
import threading

# Shared variable
shared_buffer=[]
my_mutex = threading.Lock()
num_items = threading.Semaphore(0)
empty_space = threading.Semaphore(10)           #10 is the size of the bounded buffer
# Producer function
def producer():
        global count
        for i in range(100):
                empty_space.acquire()
                with my_mutex:
                        shared_buffer.append(i)
                        print("inserted",i,"in buffer")
                        print("Buffer=",shared_buffer)
                num_items.release()   #for signal()


# Consumer function
def consumer():
        global count
        for i in range(100):
                num_items.acquire()   #for wait()
                with my_mutex:
                        j=shared_buffer.pop()
                        print("Retreived",j,"from buffer")
                        print("Buffer=",shared_buffer)
                empty_space.release()

# Create producer and consumer threads
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

# Start the threads
producer_thread.start()
consumer_thread.start()

# Wait for both threads to finish
producer_thread.join()
consumer_thread.join()

print("Production and consumption finished, buffer=",shared_buffer)
```