Threading in Python

Getting started First, make sure to import the threading module like below:

```
from threading import *
```

To create a thread called `t` with a target function `f`, use the line below:

```
t = Thread(target=f)
```

The target function contains the code that the thread will run. To start the thread, use its start method. Here is a program showing how to create and run a thread:

```
from threading import *

def f():
    print('hello from thread 1')

t1 = Thread(target=f)
t1.start()
```

A program with multiple threads Here is an example of a program that creates and runs two threads.

```
from threading import *

def f():
    print('hello from thread 1')

def g():
    print('hello from thread 2')

t1 = Thread(target=f)
t2 = Thread(target=g)
t1.start()
t2.start()
```

Note that each thread has a different target function. If both threads were to run the exact same code, it would be okay to give them both the same target function. Each thread would essentially get its own copy of that function.

Creating a list of threads Sometimes we want to create a bunch of threads. Rather than having a separate variable for each, we can create a list, like below:

```
from threading import *

def f(name):
    print(f'hello from thread {name}')

threads = []
for i in range(10):
    t = Thread(target=f, args=[i])
    t.start()
    threads.append(t)
```

The one tricky thing in the code above is the `args` keyword argument. For this program, to give each thread its own "name", the target function has a name parameter. In order to set that value when we create the thread, we use the `args` argument. The arguments are sent in a list. Note that if you run this, the printing will be all jumbled up. This could be fixed with locks.

The join method The join method is used to wait for a thread to finish. The code after the join method is called will not be run until after the thread is finished. Here is an example:

```
from threading import *

def f():
    print('hello from thread 1')

t1 = Thread(target=f)
t1.start()
t1.join()
print("This won't print until after the thread is done.")
```

Locks

To create a lock called `lock`, use the line below:

```
lock = Lock()
```

Locks have two methods we will use: acquire and release. Surround critical sections with acquire and release. The acquire method is called by a thread right before the critical section when it wants the lock. If the lock is free, then the thread can run the code in the critical section. If the lock is in use, the thread has to wait to run that code until the lock is free. The lock is freed by the release method.

In the example below, we take the list of threads program from the previous section and add a function called `safe_print`. The critical section in this program is calling the print function. We only ever want one thread at a time printing to the screen. To do this, we create a lock and surround the print statement with calls to acquire and release.

```
from threading import *

def safe_print(string):
    lock.acquire()
    print(string)
    lock.release()

def f(name):
    safe_print(f'hello from thread {name}')

lock = Lock()

threads = []
for i in range(10):
    t = Thread(target=f, args=(i,))
    t.start()
    threads.append(t)
```

Condition variables

To create a condition variable called `cond`, use the line below:

```
cond = Condition()
```

Condition variables are built on top of locks, and any time we want to do something with a condition variable, we have to surround the code with calls to acquire and release. The important methods of condition variables are these:

- `wait` — Pauses the current thread until it receives a signal from another.
- `notify` — Tell a waiting thread to stop waiting.
- `notify_all` — If multiple threads are waiting, tell all of them to stop waiting.

Here is an example. Thread 1 and Thread 2 each print hello. Thread 1 is always supposed to print hello before Thread 2 does. In the code, there is a sleep statement that causes a pause of 1 to 3 seconds. This is to simulate the fact that we might not know which thread will run first.

```
from threading import *
from time import sleep
from random import uniform

def f():
    global signal_sent
    sleep(uniform(1,3))
    print('hi from f')
    cond.acquire()
    cond.notify()
    signal_sent = True
    cond.release()


def g():
    sleep(uniform(1,3))
    if not signal_sent:
        cond.acquire()
        cond.wait()
        cond.release()
    print('hi from g')


signal_sent = False
cond = Condition()

t1 = Thread(target = f)
t2 = Thread(target = g)

t1.start()
t2.start()
```

Condition variables are used to make it so that Thread 1 prints hello before Thread 2 does. Once Thread 1 prints hello, it sends a signal to Thread 2. We also unfortunately need to have a global `signal_sent` variable. The reason we need it is that it might happen that Thread 1 runs and sends the signal to Thread 2 before Thread 2 has started running. In that case, the Thread 2 will completely miss the signal. When it starts running, it calls the wait method and will start waiting for a signal that has already been sent (and missed). This will cause it to wait forever. The boolean variable is set to true by Thread 1 when it sends the message, and Thread 2 checks the value of that variable before it waits. Using this `signal_sent` variable is not always needed with condition variables, but in this case it is.

Personally, I find condition variables a bit of a pain to work with. Sometimes, they are the right thing to use, but more often I prefer to work with semaphores.

Semaphores

Semaphores are a versatile tool for working with threads. They can be used as locks or as condition variables, and they can be used to limit access to a shared resource to a specific number of threads. Here is how to create a semaphore called `sem` with an initial value of 5:

```
sem = Semaphore(5)
```

Semaphores have two useful methods:

- `acquire` — Decreases semaphore value by 1. If the resulting value is negative, the caller will go to sleep.

- `release` — Increases the semaphore value by 1. If there are any threads waiting (i.e., the value of the semaphore is negative before the increase), one will be woken up.

Here is an example that does the same thing as the condition variable example in the previous section. We have two Threads, 1 and 2, where Thread 2 is not supposed to print hello until Thread 1 does.

```
from threading import *
from time import sleep
from random import uniform

def f():
    sleep(uniform(1,3))
    print('hi from f')
    semaphore.release()

def g():
    sleep(uniform(1,3))
    semaphore.acquire()
    print('hi from g')

semaphore = Semaphore(0)

t1 = Thread(target = f)
t2 = Thread(target = g)

t1.start()
t2.start()
```

The way this works is that the semaphore is initially set to 0. If Thread 2 runs first, when it calls acquire, it will have to wait until Thread 1 calls release.

The Producer-Consumer problem

This is a famous threading problem. The idea is there are two types of threads: producers and consumers. The producers produce random numbers and put them onto a buffer. Consumers read those values and remove them from the buffer.

We need to make sure that the consumers never try to remove something from an empty buffer and that producers never let the buffer get overfull. We can accomplish this using two semaphores.

Here is the code. It creates one producer and three consumers.

```
from threading import *
from random import randint, uniform
from time import sleep

def safe_print(s):
    plock.acquire()
    print(s)
    plock.release()

def produce(name):
    while True:
        sleep(uniform(.1, .5))
        r = randint(1, 100)
        full_semaphore.acquire()
        buffer.append(r)
        safe_print(f'Thread {name} produced {r}.\nBuffer = {buffer}')
        empty_semaphore.release()
```

```
def consume(name):
    while True:
        empty_semaphore.acquire()
        v = buffer.pop(0)
        full_semaphore.release()
        safe_print(f'Thread {name} consumed {v}.\nBuffer = {buffer}')
        sleep(uniform(1, 2))

plock = Lock()
empty_semaphore = Semaphore(0)
full_semaphore = Semaphore(3)
buffer = []

p1 = Thread(target=produce, args=['P1'])
c1 = Thread(target=consume, args=['C1'])
c2 = Thread(target=consume, args=['C2'])
c3 = Thread(target=consume, args=['C3'])

p1.start()
c1.start()
c2.start()
c3.start()
```

Here are the two semaphores that we use to solve this problem:

- `full_semaphore` — Initially set to 3. Producers call acquire on this when they want to put something onto the buffer. Consumers call release on this when they take something off of the buffer. Its job is to prevent the buffer from getting overfull (having more than 3 things on it).
- `empty_semaphore` — Initially set to 0. Consumers call acquire on this when they want to pull something off of the buffer. Producers call release on this when they put something on the buffer. Its job is to prevent consumers from trying to take something off of an empty buffer.

Working through the code, we first see the `safe_print` function. This is the same function from the section on locks. It's here to make it so that things print out nicely and not all jumbled.

The produce function loops forever. In that loop, the first thing it does is sleep for a random amount of time between .1 and .5 seconds. This is to simulate the time it takes to produce something. Then it calls acquire on the full semaphore. It is allowed to put something onto the buffer unless the buffer is full, in which case the semaphore's value will be 0. In that case, the producer will have to wait until one of the consumers pulls something off the buffer and calls release. Once the producer is done appending to the buffer, it calls release on the empty semaphore. This serves as a signal to any waiting consumers that there is something on the buffer.

The consume function is somewhat similar. It also loops forever. The consumer calls acquire on the empty semaphore which will cause it to wait until there is something on the buffer. Once it's allowed to, it pops something off the buffer and then calls release on the full semaphore to signal to producers that there is room on the buffer. After that, it sleeps for a random amount of time between .1 and .2 seconds to simulate the time it takes to consume the item.

It's worth running the code to see how it all works. In particular, try changing the amounts of time things sleep for to speed up or slow down the producers and consumers. You can also quickly add more producers and consumers. When running the program, because everything is in infinite while loops, you will need to restart the Python shell to stop it.

You might be asking what the reason is to use semaphores here. Why not just use some if statements that check if the buffer is full or empty? The reason is race conditions. If, for instance, a consumer gets interrupted in between checking the if condition and popping, another consumer might run in the meantime and pop from the buffer. When the original consumer resumes, it will end up popping from an empty buffer, which will cause an error.