Condition Variables and Semaphores

Condition variables

Condition variables are a tool used to synchronize actions between threads. Here is an example to demonstrate why they are needed.

Say we have a program with two threads. Thread #1 is supposed to print out the numbers 1 to 10. Then it should pause and wait for Thread #2. Thread #2 then prints out hello and finishes. Then Thread #1 picks up and prints out the numbers 11 to 20. The output should look like this (with . . . indicating some lines not shown):

```
1
2
3
...
10
Hello
11
12
...
20
```

One approach to this is to just have Thread 1 start Thread 2 itself, but for this simple example, we're just trying to understand how two threads that are running simultaneously can communicate with each other. When programming this, we have to be careful because we won't know for sure which thread the OS will run first. Even if we start one thread before another, it's possible that the one started later could run first. So we have to do something to make sure that Thread 2 doesn't print until it's turn. One approach is with a flag variable and a while loop in which Thread 2 continually checks the flag, waiting for Thread 1 to set it. But this means Thread 2 wastes cycles spinning in a similar way to a spin lock. Condition variables provide a better solution. Below is how to use condition variables to write the program. For now, just skim over the code. We'll talk about how it all works in a bit.

```
from threading import *
from time import sleep

def f():
    global notification_sent
    for i in range(1, 11):
        print(i)
    condition.acquire()
    notification_sent = True
    condition.notify()
    condition.wait()
    condition.release()
    for i in range(11, 21):
        print(i)

def g():
    condition.acquire()
    if notification_sent == False:
        condition.wait()
    print('Hello')
    condition.notify()
    condition.release()

notification_sent = False
condition = Condition()

t1 = Thread(target = f)
```

```
t2 = Thread(target = g)

t1.start()
t2.start()

t1.join()
t2.join()
print('Done!')
```

We'll look at what condition variables are and use this program to demonstrate how they work. Condition variables are built on top of locks. That's why in the code above there are lines where we call the acquire and release methods. Condition variables add the following methods to the lock:

- wait — When a thread calls this, it gets put onto a queue and goes to sleep.
- notify — A thread calls this to wake up another thread on the queue.
- notifyAll — This is like notify, but all threads on the queue are woken up.

Recall that in the program above, one thread is supposed to print the numbers 1 to 10, then wait, and then print out 11 to 20. The other thread is supposed to wait for the signal from the first thread, print out hello, and then signal the other thread that it can finish. Here is the code for both threads side-by-side.

```
def f():                              def g():
    global notification_sent              condition.acquire()
    for i in range(1, 11):                if notification_sent==False
        print(i)                              condition.wait()
    condition.acquire()                   print('Hello')
    notification_sent = True              condition.notify()
    condition.notify()                    condition.release()
    condition.wait()
    condition.release()
    for i in range(11, 21):
        print(i)
```

Ignore the `notification_sent` variable for the moment. Besides that, we see that both methods have to call `acquire` and `release` before doing anything with condition variables. We will always need these when working with condition variables. They are needed because of the lock that condition variables are built on.

After Thread #1 prints out 1 to 10, it calls `notify` to let the other thread know it's okay to run. Then Thread #1 calls `wait` to wait for the signal from Thread #2. Thread #2, on the other hand, initially calls `wait` to wait for the signal for the first thread that it's okay to do its thing. Once it's done printing hello, it calls `notify` to let the other thread know it's okay to continue.

Now let's talk about the `notification_sent` variable. Suppose we did not have it at all. Notice in the code that we start Thread #1 before we start Thread #2. Now it might happen that Thread #1 runs as soon as it's started and Thread #2 doesn't run for awhile. After Thread #1 prints 1 to 10, it sends a notification to Thread #2. However, if Thread #2 has not yet gotten a chance to run, it won't yet be waiting for that notification. That notification will basically be sent off into space and no one will get it. Then Thread #1 goes to sleep and waits for a notification from Thread #2. Eventually Thread #2 runs and calls `wait`. It will now be waiting for a notification that was already sent that it missed. Now both threads will be sitting there waiting forever for signals from the other one.

The `notification_sent` variable is a global variable that Thread #1 sets to true when it sends the notification. Thread #2 will check the value of that variable before it calls wait. If it's true, then that means it already missed the notification and it should not wait.It's worth going through and adding a call to Python's `sleep` method in one or the other thread to make it so that a particular thread runs first. Try doing this and removing the `notification_sent` variable and watching the program fail.

Semaphores

Like condition variables, semaphores are a tool used for communication between threads. You might be familiar with the term *semaphore* from real life. It's a system of using flags to communicate, often on boats. Semaphores are in a sense more general than locks and condition variables, as we can use them to implement both locks and condition variables. A semaphore has a numerical value, which is an integer. A semaphore also has two 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.

Semaphores are often used to limit concurrent access to no more than a certain number of threads. For instance, if we want at most 2 threads to have access to a resource at a time, then we can set the semaphore to have an initial value of 2. When the first thread accesses the resource by calling acquire, it decreases the value to 1, meaning 1 more thread is allowed to access the resource. The second thread to access it will decrease the value to 0, meaning no more threads can access the resource until one of the two threads calls release. If another thread tries to access the resource before then, it will set the value to -1. This causes the thread to sleep. Suppose one more thread calls acquire. This will set the value to -2, meaning there are two threads waiting for the semaphore to become free. If the first thread then calls release, that will set the semaphore to -1, and it will also wake up a thread. Most likely, that will be the first thread that had to sleep.

In general, if the value of the semaphore is negative, then it represents how many threads are waiting on the semaphore. If the value is not negative, then it represents how many "slots" remain free before callers have to sleep.

Note: In place of acquire and release, some people use the terms wait and signal, wait and post, P and V, or down and up.

Example Suppose we have a semaphore with initial value 2. The table below shows a sequence of calls to acquire and release by various threads. The second and third columns show the value of the semaphore before and after the call. The last column shows what, if anything, happens. We'll assume that sleeping threads are put on a queue and are woken up in the order of when they got put on the queue.

action	before	after	special action
A acquire	2	1	none
B acquire	1	0	none
C acquire	0	−1	C sleeps
D acquire	−1	−2	D sleeps
A release	−2	−1	C is woken up
C release	−1	0	D is woken up
B release	0	1	none
E acquire	1	0	none
F acquire	0	−1	F sleeps
B release	−1	0	F is woken up
E release	0	1	none
F release	1	2	none

Using a semaphore as a lock To use a semaphore as a lock, set the initial value to 1. Threads call acquire before entering the critical section and they call release when exiting. If the value is 1 when acquire is called, then the thread goes into the critical section and sets the semaphore to 0. If another thread then tries to access the critical section, it will have to sleep because the semaphore is 0. On leaving the critical section, a thread calls release, which adds 1 to the value, causing the lock to open back up.

Here is an example of using semaphores as locks in the counter program from the last set of notes.

```python
from threading import *

def f():
    global count
    for i in range(1000000):
        semaphore.acquire()
        count += 1
        semaphore.release()

def g():
    global count
    for i in range(1000000):
        semaphore.acquire()
        count += 1
        semaphore.release()

count = 0
semaphore = Semaphore(1)

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

t1.start()
t2.start()

t1.join()
t2.join()
print(count)
```

Using a semaphore as a condition variable To use a semaphore as a condition variable, set the semaphore's initial value to 0. If a thread calls release on this while its value is 0, it will have to sleep. In other words it has to wait until there is a "notification", which is accomplished by another thread calling acquire and increasing the semaphore's value to 1. Let's use the example from earlier. Thread #1 is supposed to print the numbers 1 to 10, then pause. At that point, Thread #2 is supposed to print hello, and then Thread #1 picks back up and prints 11 to 20. Here is the code using semaphores.

```python
from threading import *
from time import sleep

def f():
    for i in range(1, 11):
        print(i)

    semaphore1.release()
    semaphore2.acquire()

    for i in range(11, 21):
        print(i)

def g():
    semaphore1.acquire()
    print('Hello')
    semaphore2.release()

semaphore1 = Semaphore(0)
semaphore2 = Semaphore(0)

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

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

t1.join()
t2.join()
print('Done!')
```

We use two semaphores, both initially set to 0. Thread #2 calls `semaphore1.acquire()` as the first thing it does. Since the semaphore's value is initially 0, that means it will sleep until Thread #1 calls release on that semaphore. Similarly, Thread #1 calls `semaphore2.acquire()` when it needs to wait for Thread #2 to finish. When Thread #2 is finished, it releases that semaphore, which wakes up Thread #1.

Using a semaphore to limit concurrent access to a certain number of threads Probably the most important use of semaphores is to allow no more than a certain number of threads to have access to a shared resource. In the example below, we have 15 threads that each want access to a file in order to write to it. A semaphore is used to limit it so that no more than 5 threads can use the file at a time.

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

out_file = open('threadwriter.txt', 'w')

def f(num):
    semaphore.acquire()
    for i in range(10):
        print(f'Hello from Thread {num}', file=out_file)
        sleep(uniform(.1,.5))
    semaphore.release()

semaphore = Semaphore(5)

threads = []
for i in range(15):
    threads.append(Thread(target=f, args=[i]))

for i in range(15):
    threads[i].start()

for i in range(15):
    threads[i].join()

out_file.close()
```

This code creates 15 threads, all with the same target. We'll talk more about the syntax for this in a later section. Let's focus instead on the semaphore. It has an initial value of 5, which is what will limit things to only 5 threads using the file at a time. Each time a thread wants to write to the file, it calls `semaphore.acquire()`. This decreases the semaphore's count by 1. Once that value becomes 0, we have reached our limit of threads using the file, and any other thread will have to wait until other ones ahead of it call release on the semaphore.

The program doesn't output anything. To see what it does, run it, and open the file threadwriter.txt. Notice that the higher numbered threads don't start writing to the file until later on in the file. This is because the five spots get filled up by the earlier threads.