Implementing Semaphore

Learn how to design and implement a simple semaphore class in Python.

Implementing Semaphore

Python does provide its own implementation of **`Semaphore`** and **`BoundedSemaphore`**, however, we want to implement a semaphore with a slight twist.

Briefly, a semaphore is a construct that allows some threads to access a fixed set of resources in parallel. Always think of a semaphore as having a fixed number of permits to give out. Once all the permits are given out, requesting threads, need to wait for a permit to be returned before proceeding forward.

Your task is to implement a semaphore which takes in its constructor the maximum number of permits allowed and is also initialized with the same number of permits. Additionally, if all the permits have been given out, the semaphore blocks threads attempting to acquire it.

Solution

Given the above definition we can now start to think of what functions our Semaphore class will need to expose. We need a function to "gain the permit" and a function to "return the permit".

1. **`acquire()`** function to simulate gaining a permit
2. **`release()`** function to simulate releasing a permit

The constructor accepts an integer parameter defining the number of permits available with the semaphore. Internally we need to store a count which keeps track of the permits given out so far.

The skeleton for our Semaphore class looks something like this so far.

```
class CountSemaphore():

    def __init__(self, permits):
        self.max_permits = permits
        self.given_out = 0

    def acquire(self):
        pass

    def release(self):
        pass
```

Realize that the two methods **`acquire()`** and **`release()`** can be invoked at the same time by different threads and therefore we need to guard the logic inside of them by some kind of lock. We can use a simple **`Lock`** object for this purpose. Hold on to that thought as we'll shortly see that we'd need something more than just a lock.

Now let us fill in the implementation for our **`acquire()`** method. When can a thread not be allowed to acquire a semaphore? When all the permits are out! This implies we'll need to wait for a permit to become available when **`given_out == max_permits`**. If this condition isn't true we simply increment **`given_out`** to simulate giving out a permit. The fact that we need to *wait* for a permit if none is available should ring a bell. What synchronization primitive allows us to wait on a condition? A **Condition** object! The condition object also comes with an implicit or explicitly passed-in lock that can be used to guard the critical section of the two methods.

The implementation of the **acquire()** method appears below. Note that we are also **notifyAll()-ing** at the end of the method. We'll talk shortly, about why we need it.

```
    def acquire(self):
        self.cond_var.acquire()
        while self.given_out == self.max_permits:
            self.cond_var.wait()

        self.given_out += 1
        self.cond_var.notifyAll()
        self.cond_var.release()
```

Implementing the **release()** method should require a simple decrement of the **given_out** variable. However, when should we block a thread from proceeding forward like we did in **acquire()** method? If **given_out == 0** then it won't make sense to decrement **given_out** and we should block at this condition.

This might seem counter-intuitive, you might ask why would someone call **release()** before calling **acquire()** - This is entirely possible since semaphore can also be used for signaling between threads. A thread can call **release()** on a semaphore object before another thread calls **acquire()** on the same semaphore object. There is no concept of ownership for a semaphore! Hence different threads can call acquire or release methods as they deem fit.

This also means that whenever we decrement or increment the **given_out** variable we need to call **notifyAll()** so that any waiting thread in the other method is able to move forward. The full implementation appears below:

```
class CountSemaphore():

    def __init__(self, permits):
        self.max_permits = permits
        self.given_out = 0
        self.cond_var = Condition()

    def acquire(self):
        self.cond_var.acquire()
        while self.given_out == self.max_permits:
            self.cond_var.wait()

        self.given_out += 1
        self.cond_var.notifyAll()
        self.cond_var.release()

    def release(self):

        self.cond_var.acquire()

        while self.given_out == 0:
            self.cond_var.wait()

        self.given_out -= 1
        self.cond_var.notifyAll()
        self.cond_var.release()
```

Follow up

Note that in both the methods, we increment or decrement the variable **`given_out`** before invoking **`notifyAll()`** on the condition variable. Does it matter if we switch the order of the two statements in the two methods? The answer is no. A thread inside either of the methods holds the lock associated with the condition variable until the point the thread invokes **`release()`** on the condition variable, which is the last

statement in both the methods. A waiting thread once notified must acquire the lock associated with the condition variable before resuming execution, therefore, the order of the two statements doesn't matter.

Test Case

In the example below, we have two threads, one always acquires the semaphore while the other always releases it. The execution of the two threads is staggered. We are using the semaphore as a signaling mechanism between the two threads.

```

def task2(sem):
    time.sleep(2)
    print("releasing")
    sem.release()

    time.sleep(2)
    print("releasing")
    sem.release()

    time.sleep(2)
    print("releasing")
    sem.release()


if __name__ == "__main__":
    sem = CountSemaphore(1)

    t1 = Thread(target=task1, args=(sem,))
    t2 = Thread(target=task2, args=(sem,))

    t1.start()
    time.sleep(1);
    t2.start()

    t1.join()
    t2.join()

```

Back

Thread Safe Deferred Callback

Next

Read Write Lock

✓ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/implementing-semaphore__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)