Locks & Reentrant Lock

This lesson discusses the Lock and Reentrant Lock versions of the multiprocessing module.

Locks & Reentrant Lock

Similar to the **`threading`** module, synchronization primitives exist for the **`multiprocessing`** module to coordinate among multiple processes. The primitives between the two modules have a lot of commonalities.

Because of the similarities, we just reproduce the examples from the threading section using the primitives from the multiprocessing module.

Lock

Lock is a non-recursive object and shares the same DNA as the **`threading.Lock`** class. In the section on using Queues and Pipes we introduced a snippet with a bug, that could potentially hang depending on the order in which the two processes consumed objects placed on the queue. We can change that example snippet to be process-safe. First let's see the faulty code:

Faulty queue sharing code

```
def child_process(q):
    count = 0
    while not q.empty():
        print(q.get())
        count += 1

    print("child process {0} processed {1} items from the queu
e".format(current_process().name, count), flush=True)

if __name__ == '__main__':
    multiprocessing.set_start_method("forkserver")
    q = Queue()

    random.seed()
    for _ in range(100):
        q.put(random.randrange(10))

    p1 = Process(target=child_process, args=(q,))
    p2 = Process(target=child_process, args=(q,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

The problem with the above snippet is that a process can find the `q.empty()` to be false but before it gets a chance to take the last item off the queue, another process may have already removed it. Thus, our process ends up getting blocked on the get call because the queue is empty.

Using locks can we avoid the described situation? The key is to make the emptiness check and the get call atomic, i.e. when a process executes the two calls, no other process should be able to make the same calls. The fix

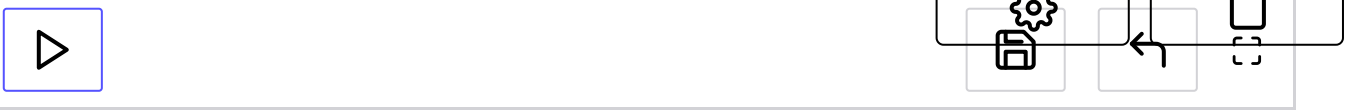
is shown below:

Fixed Code

```
def child_process(q, lock):
    count = 0
    keep_going = True

    while keep_going:
        lock.acquire()
        if not q.empty():
            print(q.get())
            count += 1
        else:
            keep_going = False
        lock.release()

    print("child process {0} processed {1} items from the queu
e".format(current_process().name, count), flush=True)


if __name__ == '__main__':

    multiprocessing.set_start_method("forkserver")
    lock = Lock()
    q = Queue()

    random.seed()
    for _ in range(100):
        q.put(random.randrange(10))

    p1 = Process(target=child_process, args=(q, lock))
    p2 = Process(target=child_process, args=(q, lock))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

Note, we had to change the code slightly to make the two calls atomic but the overall intent of the code is unchanged.

```
from multiprocessing import Process, Queue, current_process, Lock
import multiprocessing, sys
import random
import time

def child_process(q, lock):
    count = 0
    keep_going = True

    while keep_going:
        lock.acquire()
        if not q.empty():
            print(q.get())
            count += 1
        else:
            keep_going = False
        lock.release()
        # Added this sleep so that not all items get processed by
        # a single process
        time.sleep(0.001)

    print("child process {0} processed {1} items from the queue".format(current_pro


if __name__ == '__main__':

    multiprocessing.set_start_method("forkserver")
    print("This machine has {0} CPUs".format(str(multiprocessing.cpu_count())))
    lock = Lock()
    q = Queue()

    random.seed()
    for _ in range(100):
        q.put(random.randrange(10))

    p1 = Process(target=child_process, args=(q, lock))
    p2 = Process(target=child_process, args=(q, lock))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```



RLock

A reentrant lock can be acquired multiple times by a process without blocking. But remember to release the lock just as many times as it has been acquired. In the below example, we create two processes and each one of them acquires and releases an **`RLock`** five times.

Note that the main process acquires the lock first and waits for 3 seconds before releasing it, which is why the output from this program appears on the console after a slight delay. This emphasizes that the RLock object is being truly shared between the two child processes and the main process.

Using multiprocess.RLock

```
def child_task(rlock):
    for _ in range(0, 5):
        rlock.acquire()
        print("I am child process {0}".format(current_process
().name))

    for _ in range(0, 5):
        rlock.release()

if __name__ == '__main__':
    multiprocessing.set_start_method('fork')

    rlock = RLock()
    rlock.acquire()

    process1 = Process(target=child_task, args=(rlock,))
    process1.start()

    process2 = Process(target=child_task, args=(rlock,))
    process2.start()

    # sleep 3 seconds before releasing the lock
    time.sleep(3)
    rlock.release()

    process1.join()
    process2.join()
```

```
from multiprocessing import Process, RLock, current_process
import multiprocessing
import time


def child_task(rlock):
    for _ in range(0, 5):
        rlock.acquire()
        print("I am child process {0}".format(current_process().name))
        time.sleep(0.01)

    for _ in range(0, 5):
        rlock.release()

if __name__ == '__main__':
    multiprocessing.set_start_method('fork')

    rlock = RLock()
    rlock.acquire()

    process1 = Process(target=child_task, args=(rlock,))
    process1.start()

    process2 = Process(target=child_task, args=(rlock,))
    process2.start()

    # sleep 3 seconds before releasing the lock
    time.sleep(3)
    rlock.release()

    process1.join()
    process2.join()
```

Back

Sharing State

Next

Barrier, Semaphore, Condition Variable

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/locks-reentrant-lock__multiprocessing__python-concurrency-for-senior-engineering-interviews)