Locks and Synchronization

When you have more than one thread, then you may find yourself needing to consider how to avoid conflicts. What I mean by this is that you may have a use case where more than one thread will need to access the same resource at the same time. If you don't think about these issues and plan accordingly, then you will end up with some issues that always happen at the worst of times and usually in production.

The solution is to use locks. A lock is provided by Python's threading module and can be held by either a single thread or no thread at all. Should a thread try to acquire a lock on a resource that is already locked, that thread will basically pause until the lock is released. Let's look at a fairly typical example of some code that doesn't have any locking functionality but that should have it added:

```
import threading

total = 0

def update_total(amount):
    """
    Updates the total by the given amount
    """
    global total
    total += amount
    print (total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

What would make this an even more interesting example would be to add a **time.sleep** call that is of varying length. Regardless, the issue here is that one thread might call **update_total** and before it's done updating it, another

thread might call it and attempt to update it too. Depending on the order of operations, the value might only get added to once.

Let's add a lock to the function. There are two ways to do this. The first way would be to use a **try/finally** as we want to ensure that the lock is always released. Here's an example:

```
import threading

total = 0
lock = threading.Lock()

def update_total(amount):
    """
    Updates the total by the given amount
    """
    global total
    lock.acquire()
    try:
        total += amount
    finally:
        lock.release()
    print (total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

Here we just acquire the lock before we do anything else. Then we attempt to update the total and finally, we release the lock and print the current total. We can actually eliminate a lot of this boilerplate using Python's **with** statement:

```
import threading

total = 0
lock = threading.Lock()

def update_total(amount):
    """
    Updates the total by the given amount
    """
    global total
    with lock:
        total += amount
    print (total)

if __name__ == '__main__':

    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

As you can see, we no longer need the **try/finally** as the context manager that is provided by the **with** statement does all of that for us.

Of course you will also find yourself writing code where you need multiple threads accessing multiple functions. When you first start writing concurrent code, you might do something like this:

```
import threading

total = 0
lock = threading.Lock()


def do_something():
    lock.acquire()

    try:
        print('Lock acquired in the do_something function')
    finally:
        lock.release()
        print('Lock released in the do_something function')

    return "Done doing something"

def do_something_else():
    lock.acquire()

    try:
        print('Lock acquired in the do_something_else function')
    finally:
        lock.release()
        print('Lock released in the do_something_else function')

    return "Finished something else"

if __name__ == '__main__':
    result_one = do_something()
    result_two = do_something_else()
```

This works alright in this circumstance, but suppose you have multiple threads calling both of these functions. While one thread is running over the

functions, another one could be modifying the data too and you'll end up with some incorrect results. The problem is that you might not even notice the results are wrong immediately. What's the solution? Let's try to figure that out.

A common first thought would be to add a lock around the two function calls. Let's try modifying the example above to look like the following:

```
import threading

total = 0
lock = threading.RLock()

def do_something():

    with lock:
        print('Lock acquired in the do_something function')
    print('Lock released in the do_something function')

    return "Done doing something"

def do_something_else():
    with lock:
        print('Lock acquired in the do_something_else function')
    print('Lock released in the do_something_else function')

    return "Finished something else"


def main():
    with lock:
        result_one = do_something()
        result_two = do_something_else()

    print (result_one)
    print (result_two)

if __name__ == '__main__':
    main()
```

When you actually go to run this code, you will find that it just hangs. The reason is that we just told the threading module to acquire the lock. So when we call the first function, it finds that the lock is already held and blocks. It will continue to block until the lock is released, which will never happen.

The real solution here is to use a **Re-Entrant Lock**. Python's threading module

The real solution here is to use a **Re-Entrant Lock**. Python's threading module provides one via the **RLock** function. Just change the line **lock** =

threading.Lock() to **lock** = **threading.RLock()** and try re-running the code. Your code should work now!

If you want to try the code above with actual threads, then we can replace the call to **main** with the following:

```
if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=main)
        my_thread.start()
```

This will run the **main** function in each thread, which will in turn call the other two functions. You'll end up with 10 sets of output too.