Lock

Python's Lock is the equivalent of a Mutex and this lesson looks at its various uses.

Lock

Lock is the most basic or primitive synchronization construct available in Python. It offers two methods: **`acquire()`** and **`release()`**. A Lock object can only be in two states: **locked** or **unlocked**. A Lock object can only be unlocked by a thread that locked it in the first place.

A **`Lock`** object is equivalent of a mutex that you read about in operating systems theory.

Acquire

Whenever a Lock object is created it is initialized with the unlocked state. Any thread can invoke **`acquire()`** on the lock object to lock it. Advanced readers should note that **`acquire()`** can only be invoked by a single thread at any point because the GIL ensures that only one thread is being executed by the interpreter. This is in contrast to other programming languages with more robust threading models where multiple threads could be executing on different cores and theoretically attempt to acquire a lock at exactly the same time.

If a **`Lock`** object is already acquired/locked and a thread attempts to **`acquire()`** it, the thread will be blocked till the **`Lock`** object is released. If the caller doesn't want to be blocked indefinitely, a floating point

timeout value can be passed in to the **`acquire()`** method. The method returns true if the lock is successfully acquired and false if not.

Release

The **`release()`** method will change the state of the Lock object to unlocked and give a chance to other waiting threads to acquire the lock. If multiple threads are already blocked on the acquire call then only one arbitrarily chosen (varies across implementations) thread is allowed to acquire the Lock object and proceed.

Example

An example is presented below where two threads share a list and one of the threads tries to modify the list while the other attempts to read it. Using a lock object we make sure that the two threads share the list in a thread-safe manner.

```
sharedState = [1, 2, 3]
my_lock = Lock()


def thread1_operations():
    my_lock.acquire()
    print("{0} has acquired the lock".format(current_thread().
getName()))

    time.sleep(3)
    sharedState[0] = 777

    print("{0} about to release the lock".format(current_threa
d().getName()))
    my_lock.release()
    print("{0} released the lock".format(current_thread().getN
ame()))


def thread2_operations():
    print("{0} is attempting to acquire the lock".format(curre
nt_thread().getName()))
    my_lock.acquire()
    print("{0} has acquired the lock".format(current_thread().
getName()))

    print(sharedState[0])
    print("{0} about to release the lock".format(current_threa
d().getName()))
    my_lock.release()
    print("{0} released the lock".format(current_thread().getN
ame()))

if __name__ == "__main__":
    # create and run the two threads
    thread1 = Thread(target=thread1_operations, name="thread1"
)
    thread1.start()
```

```
    thread2 = Thread(target=thread2_operations, name="thread2"
)
    thread2.start()

    # wait for the two threads to complete
    thread1.join()
    thread2.join()
```

```
import time
from threading import Lock
from threading import Thread
from threading import current_thread

sharedState = [1, 2, 3]
my_lock = Lock()


def thread1_operations():
    my_lock.acquire()
    print("{0} has acquired the lock".format(current_thread().getName()))

    time.sleep(3)
    sharedState[0] = 777

    print("{0} about to release the lock".format(current_thread().getName()))
    my_lock.release()
    print("{0} released the lock".format(current_thread().getName()))


def thread2_operations():
    print("{0} is attempting to acquire the lock".format(current_thread().getName()
    my_lock.acquire()
    print("{0} has acquired the lock".format(current_thread().getName()))

    print(sharedState[0])
    print("{0} about to release the lock".format(current_thread().getName()))
    my_lock.release()
    print("{0} released the lock".format(current_thread().getName()))

if __name__ == "__main__":
    # create and run the two threads
    thread1 = Thread(target=thread1_operations, name="thread1")
    thread1.start()

    thread2 = Thread(target=thread2_operations, name="thread2")
    thread2.start()

    # wait for the two threads to complete
    thread1.join()
    thread2.join()
```

If you examine the output from the above snippet, you'll notice that thread2 blocks on acquire method and waits for thread1 to release it before it can successfully acquire it.

Deadlock Example

Consider the example below, where two threads are instantiated and each tries to invoke **`release()`** on the lock acquired by the other thread, resulting in a deadlock.

```
from threading import *
import time


def thread_one(lock1, lock2):
    lock1.acquire()
    time.sleep(1)
    lock2.release()


def thread_two(lock1, lock2):
    lock2.acquire()
    time.sleep(1)
    lock1.release()


if __name__ == "__main__":
    lock1 = Lock()
    lock2 = Lock()

    t1 = Thread(target=thread_one, args=(lock1, lock2))
    t2 = Thread(target=thread_one, args=(lock1, lock2))

    t1.start()
    t2.start()

    t1.join()
    t2.join()
```

> The above example demonstrates that a thread can't release a lock it has not locked. Furthermore, trying to release an unacquired lock will result in an exception.

Back

Daemon Thread

Next

RLock

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/lock__threading-module__python-concurrency-for-senior-engineering-interviews)