Lesson 11 Python Multithreading

1. Process and Thread

A process is the carrier to execute the program. A process is started whenever we open software and game, and execute Python script.

A thread is the smallest unit that performs operations in a process, and an entity in a process. It also is a basic unit that is independently scheduled and dispatched by the system. For example, the production in the workshop is a process, and each assembly line is one of its threads.

A thread does not have system resources, and only has a few resources that are essential for running. However it can share all the resources owned by the process with other threads belonging to the same process. A thread can create and cancel another thread, and multiple threads in the same process can execute concurrently.

The process provides the pre-requirements for the thread to execute the program. And the thread executes the program with the reorganized resources. These two modules are commonly used in Python3 threads.

1) _thread

2) threading(recommend)

thread module has been abandoned, which can be replaced by threading. Therefore “thread” module cannot be used in Python3. For compatibility, thread is renamed as “_thread” by Python3.

2. Thread Object

```
class threading.Thread(grounp=None,target=None,name=None,args=(),kwargs={})
```

1) group: it should be None, and is reserved for ThreadGroup class extension in the future.

2) target: The object can be called by run(). It is set as None by default, which means that no method needs to be called.

3) name: The thread name. By default, use the format of "Thread-N" to create a unique name, where N is a small decimal number.

4) args: It is used to call the parameter tuple of the target function, () by default.

5) kwargs: It is used to call dictionary of keyword parameters of the target function, {} by default.

3. Thread Method

Method Name	Explanation	Usage
start	Start thread	start()
run	Thread activity	run()
join	Block until the thread execution ends. It blocks the thread calling this method until the thread that calls join() ends. No matter normal ending, unhandled exception or a timeout, the timeout is optional. Ensure to call is_alive() after Join() to determine if a	Join(timeout=None)

	timeout has occurred. if the thread is still alive, join() times out. And a thread can be join() many times.	
getName	Obtain the name of the thread	getName()
setName	Set the name of the thread	setName(name)
is_alive	Judge whether the thread is alive or not	Is_alive()
setDaemon	Daemon the thread	setDaemon(True)

4 Create Thread Object

4.1 Thread Directly Create Thread

Pass a function object from the class constructor, which is the callback function used to handle the task.

```
from threading import Thread

def task():
    print('i am sub_thread')
if __name__ == '__main__':
    t = Thread(target=task)
    t.start()
    print('i am main-thread')
```

```
i am sub_thread
i am main-thread
```

4.2 Inherit Thread Class

Write a custom class to inherit Thread, then rewrite the run() method, that is write the task processing code in it, and then create a child class of this

Thread.

```
from threading import Thread

def task():
    print('i am sub_thread')

class MyThread(Thread):
    def run(self):
        task()

if __name__ == '__main__':
    t = MyThread()
    t.start()
    print('i am main-thread')
```

```
i am sub_thread
i am main-thread
```

5. Multithreading

Multithreading is similar to workshop production where the efficiency of production can be improved by the simultaneous operation of several assembly lines. Create Thread objects and let them run. Each Thread object represents a thread. In each thread, the programs can handle different tasks, which is multi-threaded programming.

For example, create 7 threads to execute tasks, and then call join method to wait for the thread execution to end, which is faster than a single main thread execution.

```python
import threading

def func(n):
    while n>0:
        print("Number of current threads:",threading.activeCount())
        n -= 1

threads = []
for x in range(5):
    t = threading.Thread(target=func, args=(2,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("main thread",threading.current_thread().name)
```

```
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
Number of current threads: 2
main thread: MainThread
```

6. Thread Synchronization

Multiple threads modifying a data together will lead to unpredictable results. In order to ensure the accuracy of data, multiple threads need to be synchronized. Simple thread synchronization can be achieved by using the Lock and Rlock of the Thread object.

6.1 LOCK

1) acquire(blocking=True, timeout=-1):: Puts the thread into a synchronously blocked state to acquire a lock. If the parameter blocking is set as True during calling, blocking will last until the lock is released, then lock the LOCK and return True. When the parameter blocking is set as False, no

blocking will occur.

2) release()::elease the lock. The lock must be acquired before using the thread, otherwise an exception will be thrown.

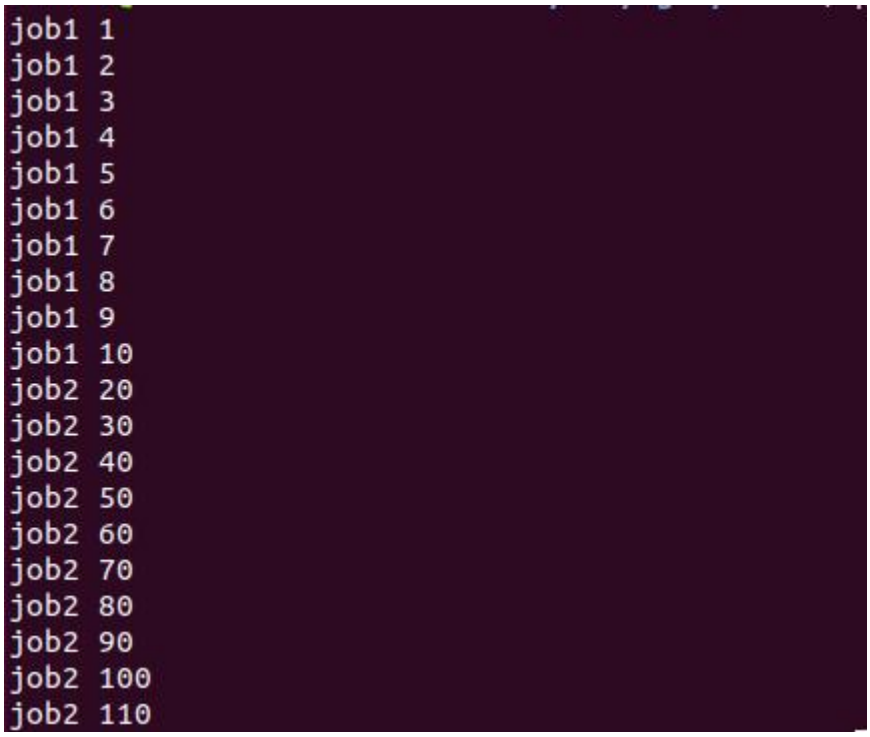
release() is only called under the locked state. It changes the state to unlocked and returns immediately. A RuntimeError exception will be triggered if an attempt is made to release a non-locking lock. For the Lock object, if a thread releases twice in a row, it will cause deadlock. Therefore, Lock is not commonly used, while Rlock is generally used to set thread locks.

```
import threading

def job1():
    global A, lock
    lock.acquire()
    for i in range(10):
        A += 1
        print('job1', A)
    lock.release()

def job2():
    global A, lock
    lock.acquire()
    for i in range(10):
        A += 10
        print('job2', A)
    lock.release()

if __name__ == '__main__':
    lock = threading.Lock()
    A = 0
    t1 = threading.Thread(target=job1)
    t2 = threading.Thread(target=job2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```



After execution, t1 first obtains the lock, and then releases the lock after the execution ends. t2 obtains the lock again, and continues to execute before releasing the lock, so as to avoid simultaneous processing and data errors.

6.2 RLock

1) acquire(blocking=True, timeout=-1): The lock can be acquired either blocking or non-blocking. When no parameter is called and if the thread already owns the lock, the recursion level is increased by one, and it returns immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked.

Once the lock is unlocked (not owned by any thread), grab ownership, set the recursion level to one, and return. If multiple threads are blocked, wait for the lock to be unlocked and only one thread at a time can grab ownership of the lock. In this case, there is no return value.

2) release(): Release the lock and decrement the recursion level. If it is reduced to zero, the lock is reset to the unlocked state (not owned by any

thread). And, if other threads are blocked waiting for unlocking, only one of the threads is allowed to continue. If the recursion level is still non-zero after the decrement, the lock remains locked, still owned by the calling thread.

```
import threading

lock = threading.RLock()

ret = lock.acquire()
print(ret)
ret = lock.acquire(timeout=3)
print(ret)
ret = lock.acquire(True)
print(ret)
ret = lock.acquire(False)
print(ret)

lock.release()
lock.release()
lock.release()
lock.release()
```

```
True
True
True
True
```

RLock (reentrant lock) is a synchronous instruction that can be requested multiple times by the same thread. RLock uses the concepts of "owned thread" and "recursion level". When in the locked state, RLock is owned by a thread. The thread that owns the RLock can call acquire() again, and call release() the same number of times when releasing the lock.