Threading

created by David Song

Contents

Creating a thread

Subclassing Thread

```
class MyThread(threading.Thread):

    def run(self):
        logging.debug('running')
        return

t = MyThread()
t.start()
```

Deligate to target

The simplest way to use a `Thread` is to instantiate it with a target function and call `start()` to let it begin working.

```
def worker():
    print 'Worker'
    return

t = threading.Thread(target=worker)
t.start()
```

with arguments:

```
threading.Thread(target=worker, name=str(i), args=(s, pool))
```

Determining the Current Thread

```
1. Direct output

   print threading.currentThread().getName()

2. Use logging
   logging.basicConfig(level=logging.DEBUG,
                     format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                     )logging.debug('Starting')

   The output: [DEBUG] (worker) Starting

   Note: if something goes wrong, such as create a thread using target as
   function call, logging will show only main thread name

   Example:

        t = threading.Thread(target=worker())
        t.start()
```

Enumerate

```
for t in threading.enumerate():
    if t is main_thread:
        continue
    logging.debug('joining %s', t.getName())
    t.join()
```

A simple way to print all thread is using `threading._active`

Daemon Thread

A daemon thread is running background. The entire Python program exits when only daemon threads are left, thus it's referred as daemon.

```
def worker():
    logging.debug('worker started')
    sleep(5)
    logging.info('worker finished')

w = threading.Thread(target=worker, name='worker')
w.setDaemon(True)
w.start()

logging.info('Main finished')
```

with daemon flag set to True, we won't see output from worker thread, since main program is finished and closed console.

Join method

This blocks the calling thread until the thread whose join() method is called terminates

Below is an example, main thread waits for worker2, while work2 wait for worker to finish:

```
def worker():

    logging.debug('worker started')
    sleep(5)
    logging.info('worker finished')


w = threading.Thread(target=worker, name='worker')
w.setDaemon(True)
w.start()

def worker2():
    logging.info('worker2 started')
    # wait for worker to finish
    logging.info('waiting for worker to finish')
    w.join()
    logging.info('worker2 finished')

w2 = threading.Thread(target=worker2, name='w2')
```

```
w2.start()


logging.info('Main: Waiting for thread worker 2 to terminate...')
# wait for worker2 to finish.
w2.join()
logging.info('Main: finished')
```

Implementation on the join method: each thread has a condition variable, the join method simply put current thread on the waiting thead's condition's waiting list. Once the waiting thread finished, it will call condition.notify_all. Internally, each thread worker method is put into a wrapper, in the wrapper's finally block, it set the flag to signal the thread is finished and call condition notify all method.

Console output:

```
[worker]: worker started
[w2]: worker2 started
[w2]: waiting for worker to finish
[MainThread]: Main: Waiting for thread worker 2 to terminate...
[worker]: worker finished
[w2]: worker2 finished
[MainThread]: Main: finished
```

Timer Threads

One example of a reason to subclass `Thread` is provided by `Timer`, also included in threading. A `Timer` starts its work after a delay, and can be canceled at any point within that delay time period.

```
t1 = threading.Timer(3, target)
t1.cancel()
```

Threads Control

In addition to use join, and timer, there are many ways to control threads. Most commonly used are Lock, RLock, Event, Semaphore. Condition and Barrier. Only RLock and Condition are the key components, the rest are implemented based on top of RLock and Condition.

Signaling Between Threads (Event)

An `Event` manages an internal flag that callers can either `set()` or `clear()`. Other threads can `wait()` for the flag to be `set()`, effectively blocking progress until allowed to continue.

Unlike Lock, when a lock is release, only one thread can acquire the lock and gain access, when an event is set, all thread can continue.

Below example shows that, once an event object is set by thread 3, then thread 1 and 2 all continue their executions.

```python
import threading
import time
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )

event_object = threading.Event()

def f1():
    logging.info('F1 started, and wait for event')
    event_object.wait()
    logging.info('F1 finished')


def f2():
    logging.info('F2 started, and wait for event')
    event_object.wait()
    logging.info('F2 finished')


def f3():
    logging.info('F3 started, and set event')
    event_object.set()
    logging.info('F3 finished')


T1 = threading.Thread(target=f1)
T2 = threading.Thread(target=f2)
T3 = threading.Thread(target=f3)

# starting threads
T1.start()
T2.start()
T3.start()
```

Internally, Event is implemented via a Condition, simple calls condition.wait and condition.notify_all for event wait and set operations.

Controlling Access to Resources (Lock, RLock)

```
To guard against simultaneous access to an object, use a Lock object.

lock.acquire(),
lock.release()
```

Thread has acquired the lock without holding up the current thread, pass False for the *blocking* argument to `acquire()`.

Caution: check if the lock is locked before acquire and release, otherwise, acquire may cause dead lock, release may throw exceptions.

RLock: A reentrant lock, which means once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it. Each time acquire gets called, the RLock internal count variable gets increased. Release must gets called the same number of acquire called, to get the lock totally released and become available to other thread.

RLock is implemented from Lock which is primitive.

Below example shows that, a dead lock will happen when using Lock, and no issue when using RLock.

```
import threading
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )


#lock = threading.Lock()
lock = threading.RLock()

def f():
    logging.info('I am f')
    with lock:
        g()
        h()

def g():
    with lock:
        h()
        logging.info('I am g')


def h():
    logging.info('I am h')

worker = threading.Thread(target=f)
worker.start()
```

Lock object can be checked `lock.locked()`

or give a time in acquire method call, if the lock is not available it returns false, like below :

```
lock.acquire() True
lock.acquire(0) False
```

`RLock doesn't have locked method, it can be checked by using lock.acquire(0),` it will return 1 if the locked is being used. Or use `lock._is_owned` to check if current thread owns this lock. When acquire get called within the same thread, internal count will be incremented

```
>>>lock
<_RLock owner='MainThread' count=5>
```

Synchronizing Threads (Condition)

Thinking condition as a signal, waiting for a signal and sending a signal. A condition object maintains a waiting list, once a resource becomes available or a task is done, it will send a signal wake up thread(s) in the waiting list. It gets more interesting when you consider that you can have several different Conditions over the same underlying lock. The `Condition` uses a `Lock`, it can be tied to a shared resource. This allows threads to wait for the resource to be updated. The easy way is to use it as a context manager:

```
with cond:
     cond.wait()

with cond:
     cond.notifyAll()
```

Note that, wait and notify methods need to be called within a lock context, meaning acquire a lock first. This is because those operations will release the lock. This is implementation preference, the whole idea of condition object is to utilize the underlying lock, without the lock, multithreads can mess the context. Without acquiring the lock first, wait and notify operations can cause runtime issues, perhaps deadlock, race condition or cannot release a lock without holding it. Condition is implemented on Top of RLock, each thread waiting on the condition will be assigned a new lock and waiting for its own lock to be released. Without acquiring the underlying lock, the waiting queue can be messed up.

The wait() operation will release the condition's underlying lock. Below example shows that the 2nd thread gets blocked on acquiring the condition until the 1st thread calling wait():

```
condition = Condition()


def worker_1():
    with condition:
        logging.info('worker 1 will sleep, holding the condition ')
        time.sleep(10)
        logging.info('waiting for ')
        condition.wait()
        logging.info('done waiting ')

def worker_2():
    logging.info('worker 2 will start checking condition ')
    with condition:
        # lock count = 1, with thread 1
        logging.info('waiting for  {}'.format(condition))
        condition.wait(1)
        # lock count = 1 release the lock and restore count so the lock
        # can be released once the codition context if fiished.
        logging.info('done waiting  {}'.format(condition))



def main():


    w1 = threading.Thread(target=worker_1)
```

```
    w2 = threading.Thread(target=worker_2)

    w1.start()
    time.sleep(1)
    w2.start()


if __name__ == '__main__':

    main()
```

To illustrate how condition works, think condition structure briefly like below:

```
condition {
    lock1 - condition lock
    waiter locks -- list of locks (lock2 for simple)
}
```

assume there are only one thread in the waiting list, say it's lock2, below is the workflow:

1	condition.acquire()	`<- lock lock1, or wait till available`
2	condition.wait()	`<- lock lock2, release lock1`
3	time.sleep(10) *# do some work*	`<- once lock2 gets released from anther thread by calling notify, lock lock1, resume execution from here. If lock 1 is not available, back to step 1`
4	condition.release()	`<- release lock1`

Limiting Concurrent Access to Resources (Semaphore)

A `Semaphore` is one way

```
s = threading.Semaphore(2)
with s:
     name = threading.currentThread().getName()
```

Barrier (From Python 3)

A Barrier allows multiple threads to wait on the same barrier object instance (e.g. at the same point in code) until a predefined fixed number of threads arrive (e.g. the barrier is full), after which all threads are then notified and released to continue their execution.

Below example shows 4 threads wait for a barrier:

```
num = 4
barrier= threading.Barrier(num)

def worker():
    working_time = randrange(2, 5)
    logging.info('will work {} seconds'.format(working_time))
    time.sleep(randrange(2, 5))
    logging.info('done my work, reached the barrier')
    barrier.wait()
    logging.info('finished waiting on barrier')
```

```
logging.info("Race starts now…")
for i in range(num):
   t = threading.Thread(target=worker)
   t.start()
```

Thread-specific Data (local)

The `local()` function creates an object capable of hiding values from view in separate threads. This local object is empty, it cannot be assign any value at creation, but can be assigned to any data on the fly and it's specific to the thread.

```
local_data = threading.local()
local_data.value = 1000
```

To initialize the settings so all threads start with the same value, use a subclass and set the attributes in `__init__()`.

```
class MyLocal(threading.local):
    def __init__(self, value):
        self.value = value

local = MyLocal(100)


def task(increment):


    local.value = local.value + increment
    time.sleep(3)

    logging.info(local.value)


T1 = threading.Thread(target=task, name='T1', args=(1,))
T2 = threading.Thread(target=task, name='T2', args=(2,))

T1.start()
T2.start()

T1.join()
T2.join()

logging.info(local.value)
```

From above example, we can see that local object are distinct between 3 threads (2 workers and the main thread).

Low-level threading API

The **thread** module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks)The **threading** module provides an easier to use and higher-level threading API built on top of this module.

The module is optional. Most operations are related to lock. There are a few powerful operations:

`start_new_thread`
`interrupt_main` -- A subthread can use this function to interrupt the main thread.
exit -- this will cause the thread to exit silently.
`stack_size`

Note that, from Python 3, thread module is changed to _thread

Threading Practice

Print ABC in order

Use lock:

```
import threading
from threading import Lock
from time import sleep
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )


locks = (Lock(), Lock(), Lock())
locks[0].acquire()
locks[1].acquire()
locks[2].acquire()

iter_num = 5

def first(iter_num):

    for i in range(iter_num):
        while locks[0].locked():
            sleep(0.1)

        locks[0].acquire()
        logging.info('A')

        locks[1].release()


def second(iter_num):
    for i in range(iter_num):
        while locks[1].locked():
            sleep(0.1)
```

```
        locks[1].acquire()
        logging.info('B')

        locks[2].release()


def third(iter_num):
    for i in range(iter_num):
        while locks[2].locked():
            sleep(0.1)

        locks[2].acquire()
        logging.info('C')

        locks[0].release()

def main():

    locks[0].release()
    a = threading.Thread(target=first, args=(iter_num,))
    b = threading.Thread(target=second, args=(iter_num,))
    c = threading.Thread(target=third, args=(iter_num,))

    a.start()
    b.start()
    c.start()

if __name__ == '__main__':

    main()
```

Use condition

```
import threading
from threading import Condition
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )


condition = Condition()

thread_done=[False,False,False]

iter_num = 5
total_thread = 3

def printer(iter_num, thread_num, print_letter):
    pre_thread=(thread_num+2)%total_thread
    print (thread_num)
    for i in range(iter_num):
        with condition:
            while not thread_done[pre_thread]:
                condition.wait()

            logging.info(print_letter)
```

```
            thread_done[thread_num] = True
            thread_done[pre_thread] = False
            condition.notify()


def main():


    a = threading.Thread(target=printer, args=(iter_num,0,'A'))
    b = threading.Thread(target=printer, args=(iter_num, 1, 'B'))
    c = threading.Thread(target=printer, args=(iter_num,2,'C'))
    thread_done[2] = True
    a.start()
    b.start()
    c.start()


if __name__ == '__main__':

    main()
```

Start from Python 3.2, wait_for method can be used instead of the while loop wait:

```
    with condition:
        condition.wait_for(lambda : thread_done[pre_thread])
```

Producer-Consumer

Producer-Consumer with Conditions

```
condition = threading.Condition()
queue = []
queue_max_size = 5

def producer(condition, items, max_size):

    while True:
        with condition:
            if len(queue)>=max_size:
                logging.info('queue is full, waiting for an empty slot')
                condition.wait()
            else:
                time.sleep(2) # hard working
                num = randrange(1,10)
                logging.info('generate {}'.format(str(num)))
                queue.append(num)


def consumer(condition, items):

    while True:
        with condition:
            if len(queue) == 0:
                logging.info('No item to consume, waiting for an item')
                condition.wait()
            else:
                time.sleep(2)  # hard working
```

```python
                num = queue.pop()
                logging.info('Consume {}'.format(str(num)))


for i in range(2):
    threading.Thread(target=producer, name='Producer {}'.format(str(i)),
args=(condition, queue, queue_max_size)).start()


for i in range(3):
    threading.Thread(target=consumer, name='Consumer {}'.format(str(i)),
args=(condition, queue)).start()
```

Producer-Consumer with Queue

```python
'''
     Queue is  A synchronized queue class. The Queue object encapsulates the blocking
     and notification.

The Queue module has been renamed to queue in Python 3
'''
import threading
import Queue
import time
from random import randrange
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )


condition = threading.Condition()
queue = Queue.Queue(5)


def producer(condition, items):

    while True:
        with condition:
            if items.full():
                logging.info('queue is full, waiting for an empty slot')
                condition.wait()
            else:
                time.sleep(2) # hard working
                num = randrange(1,10)
                logging.info('generate {}'.format(str(num)))
                items.put(num)


def consumer(condition, items):

    while True:
        with condition:
            if items.empty():
                logging.info('No item to consume, waiting for an item')
                condition.wait()
            else:
                time.sleep(2)  # hard working
```

```
                num = items.get()
                logging.info('Consume {}'.format(str(num)))


for i in range(2):
    threading.Thread(target=producer, name='Producer {}'.format(str(i)),
args=(condition, queue)).start()


for i in range(3):
    threading.Thread(target=consumer, name='Consumer {}'.format(str(i)),
args=(condition, queue)).start()
```

Read write lock

Read write lock with writers starvation

```
'''

A lock allows one writer, multiple readers

This class encapsulates a condition object

A writer
 1. acquire the condition lock successfully, release it once the writer is done
 2. If it cannot acquire the lock, means another writer is holding it, or a reader is
temporary updating the readers count, then put the writer into waiting list

A reader:
1. will get blocked upon acquiring the condition, which means there is a writer
holding it.

drawback:
since there is only 1 lock, readers and writers can be blocked on, it can slow down a
bit on performance

Notice that the conditions underlying lock blocking list is a FIFO, if there many
readers, this may cause a writer starvation issue.

'''


import threading
import Queue


import logging

logging.basicConfig(level=logging.DEBUG,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )


class ReadWriteLock:

    def __init__(self):
        self._readers = 0
        self._monitor = threading.Condition()

    def acquire_read(self):

        with self._monitor: # if a reader can get the lock, it means no writer
```

```
            self._readers += 1

    def release_read(self):

        with self._monitor:
            self._readers -=1
            self._monitor.notify_all() # wakeup writers

    def acquire_write(self):

        self._monitor.acquire()  # no other writers
        while self._readers >0:
            #
            # wait for readers to finish, once it's waked up from the last reader,
            # the writer will restore it's lock and continue,
            # it could be blocked if there are other readers and writers are already
            # blocked on the condition's underlying lock.
            #

            self._monitor.wait()

    def release_write(self):

        self._monitor.release()  # release the condition lock

    def wait(self):
        self._monitor.wait()
```

Testing program:

```
read_write_lock = ReadWriteLock()
items =[1,2,3,4,5]


def writer(read_write_lock, items):

    while True:
        read_write_lock.acquire_write()

        if len(items)>5:
            logging.info('queue is full, waiting for an empty slot')
            read_write_lock.wait()
        else:
            time.sleep(2) # hard working
            num = randrange(1,10)
            logging.info('Write {}'.format(str(num)))
            pos = randrange(0, 5)
            items[pos]=num

        read_write_lock.release_write()

def reader(read_write_lock, items):

    while True:
        read_write_lock.acquire_read()
        if len(items)<1:
```

```
            logging.info('No item to consume, waiting for an item')
            read_write_lock.wait()
        else:
            time.sleep(2)  # hard working
            pos = randrange(0, 5)
            num = items[pos]
            logging.info('Read {}'.format(str(num)))
        read_write_lock.release_read()


for i in range(2):
    threading.Thread(target=writer, name='Writer {}'.format(str(i)),
args=(read_write_lock, items)).start()


for i in range(5):
    threading.Thread(target=reader, name='Reader {}'.format(str(i)),
args=(read_write_lock, items)).start()
```

Read write lock favors to writers

```
'''

A read write lock that permits multiple readers to run concurrently, blocking if there
is a writer.

This implementation gives writers higher priority over readers, means if there are
readers and writers in the waiting list, let writers to run. This makes sense in real
world, since writer is more important; and readers can get the latest information
after writer finished.

'''


import threading
import logging
logging.basicConfig(level=logging.INFO,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )


class ReadWriteLock:

  def __init__(self):

    self._monitor = threading.Condition()

    #
    # number of readers are currently running
    #
    self._readers = 0

    #
    # number of writers, 1 is running, the rest (n-1) are waiting
    #
    self._writers = 0


  def acquire_read(self):
    logging.debug('acquire read')
    #
```

```
    # Acquire a read lock. Blocks only if a thread has
    # acquired the write lock.
    #
    self._monitor.acquire()
    try:
      while self._writers > 0:
        self._monitor.wait()
      self._readers += 1
    finally:
      self._monitor.release()

  def release_read(self):
    logging.debug('release_read')

    self._monitor.acquire()
    try:
      self._readers -= 1
      if not self._readers:
        self._monitor.notifyAll()
    finally:
      self._monitor.release()

  def acquire_write(self):
    logging.debug('acquire write')
    #
    # Acquire a write lock. Blocks until there are no
    # acquired read or write locks.
    #
    self._monitor.acquire()
    self._writers += 1
    while self._readers > 0:
      self._monitor.wait()

  def release_write(self):
    logging.debug('release write')

    self._writers -= 1
    self._monitor.notifyAll()
    self._monitor.release()
```

Read write lock with Context manager

```
'''

Use a wrapper to encapsulate the read write lock implementation, and with read lock
and write lock class.

'''

import threading
import logging
logging.basicConfig(level=logging.INFO,
                    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
                    )

class ReadWriteLock:
  class _ReadWriteLock:
```

```python
  def __init__(self):

    self._monitor = threading.Condition()

    #
    # number of readers are currently running
    #
    self._readers = 0

    #
    # number of writers, 1 is running, the rest (n-1) are waiting
    #
    self._writers = 0


  def acquire_read(self):
    logging.debug('acquire read')
    #
    # Acquire a read lock. Blocks only if a thread has
    # acquired the write lock.
    #
    self._monitor.acquire()
    try:
      while self._writers > 0:
        self._monitor.wait()
      self._readers += 1
    finally:
      self._monitor.release()

  def release_read(self):
    logging.debug('release_read')

    self._monitor.acquire()
    try:
      self._readers -= 1
      if not self._readers:
        self._monitor.notifyAll()
    finally:
      self._monitor.release()

  def acquire_write(self):
    logging.debug('acquire write')
    #
    # Acquire a write lock. Blocks until there are no
    # acquired read or write locks.
    #
    self._monitor.acquire()
    self._writers += 1
    while self._readers > 0:
      self._monitor.wait()

  def release_write(self):
    logging.debug('release write')

    self._writers -= 1
    self._monitor.notifyAll()
    self._monitor.release()

#
# Context manager classes
#
class ReadLock:
```

```
    def __init__(self, readWriteLock):
      self.readWriteLock = readWriteLock

    def __enter__(self):
      self.readWriteLock.acquire_read()


    def __exit__(self, exc_type, exc_value, traceback):
      self.readWriteLock.release_read()


  class WriteLock:

    def __init__(self, readWriteLock):
      self.readWriteLock = readWriteLock

    def __enter__(self):
      self.readWriteLock.acquire_write()


    def __exit__(self, exc_type, exc_value, traceback):
      self.readWriteLock.release_write()


  readWritelock = _ReadWriteLock()
  readLock = ReadLock(readWritelock)
  writelock = WriteLock(readWritelock)
```

Test Read write lock with Context manager

```
items =[1,2,3,4,5]

lock = ReadWriteLock()


def writer(lock, items):

    while True:
        with lock.writelock:
            time.sleep(2) # hard working
            num = randrange(1,10)
            logging.info('Write {}'.format(str(num)))
            pos = randrange(0, 5)
            items[pos]=num



def reader(lock, items):

    while True:
        with lock.readLock:
            time.sleep(1)  # hard working
            pos = randrange(0, 5)
            num = items[pos]
            logging.info('Read {}'.format(str(num)))
```

```
for i in range(2):
    threading.Thread(target=writer, name='Writer {}'.format(str(i)), args=(lock,
items)).start()


for i in range(2):
    threading.Thread(target=reader, name='Reader {}'.format(str(i)), args=(lock,
items)).start()
```