

2

Thread-based Parallelism

In this chapter, we will cover the following recipes:

- ▶ How to use the Python threading module
- ▶ How to define a thread
- ▶ How to determine the current thread
- ▶ How to use a thread in a subclass
- ▶ Thread synchronization with Lock and RLock
- ▶ Thread synchronization with semaphores
- ▶ Thread synchronization with a condition
- ▶ Thread synchronization with an event
- ▶ How to use the `with` statement
- ▶ Thread communication using a queue
- ▶ Evaluating the performance of multithread applications
- ▶ The criticality of multithreaded programming

Introduction

Currently, the most widely used programming paradigm for the management of concurrence in software applications is based on multithreading. Generally, an application is made by a single process that is divided into multiple independent threads, which represent activities of different types that run parallel and compete with each other.

Although such a style of programming can lead to disadvantages of use and problems that need to be solved, modern applications with the mechanism of multithreading are still used quite widely.

Practically, all the existing operating systems support multithreading, and in almost all programming languages, there are mechanisms that you can use to implement concurrent applications through the use of threads.

Therefore, multithreaded programming is definitely a good choice to achieve concurrent applications. However, it is not the only choice available—there are several other alternatives, some of which, *inter alia*, perform better on the definition of thread.

A thread is an independent execution flow that can be executed parallelly and concurrently with other threads in the system. Multiple threads can share data and resources, taking advantage of the so-called space of shared information. The specific implementation of threads and processes depends on the operating system on which you plan to run the application, but, in general, it can be stated that a thread is contained inside a process and that different threads in the same process conditions share some resources. In contrast to this, different processes do not share their own resources with other processes.

Each thread appears to be mainly composed of three elements: program counter, registers, and stack. Shared resources with other threads of the same process essentially include data and operating system resources. Similar to what happens to the processes, even the threads have their own state of execution and can synchronize with each other. The states of execution of a thread are generally called ready, running, and blocked. A typical application of a thread is certainly parallelization of an application software, especially, to take advantage of modern multi-core processors, where each core can run a single thread. The advantage of threads over the use of processes lies in the performance, as the context switch between processes turns out to be much heavier than the switch context between threads that belong to the same process.

Multithreaded programming prefers a communication method between threads using the space of shared information. This choice requires that the major problem that is to be addressed by programming with threads is related to the management of that space.

Using the Python threading module

Python manages a thread via the `threading` package that is provided by the Python standard library. This module provides some very interesting features that make the threading-based approach a whole lot easier; in fact, the threading module provides several synchronization mechanisms that are very simple to implement.

The major components of the threading module are:

- ▶ The thread object
- ▶ The Lock object
- ▶ The RLock object
- ▶ The semaphore object
- ▶ The condition object
- ▶ The event object

In the following recipes, we examine the features offered by the threading library with different application examples. For the examples that follow, we will refer to the Python distribution 3.3 (even though Python 2.7 could be used).

How to define a thread

The simplest way to use a thread is to instantiate it with a target function and then call the `start()` method to let it begin its work. The Python module `threading` has the `Thread()` method that is used to run processes and functions in a different thread:

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

In the preceding code:

- ▶ `group`: This is the value of `group` that should be `None`; this is reserved for future implementations
- ▶ `target`: This is the function that is to be executed when you start a thread activity
- ▶ `name`: This is the name of the thread; by default, a unique name of the form `Thread-N` is assigned to it

- ▶ `args`: This is the tuple of arguments that are to be passed to a target
- ▶ `kwargs`: This is the dictionary of keyword arguments that are to be used for the target function

It is useful to spawn a thread and pass arguments to it that tell it what work to do. This example passes a number, which is the thread number, and then prints out the result.

How to do it...

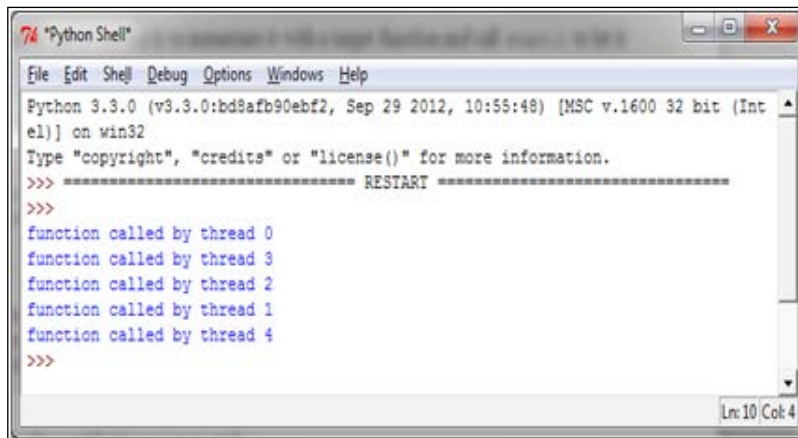
Let's see how to define a thread with the `threading` module, for this, a few lines of code are necessary:

```
import threading

def function(i):
    print ("function called by thread %i\n" %i)
    return

threads = []
for i in range(5):
    t = threading.Thread(target=function , args=(i,))
    threads.append(t)
    t.start()
    t.join()
```

The output of the preceding code should be, as follows:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
function called by thread 0
function called by thread 3
function called by thread 2
function called by thread 1
function called by thread 4
>>>
```

We should also point out that the output could be achieved in a different manner; in fact, multiple threads might print the result back to `stdout` at the same time, so the output order cannot be predetermined.

How it works...

To import the threading module, we simply use the Python command:

```
import threading
```

In the main program, we instantiate a thread, using the `Thread` object with a target function called `function`. Also, we pass an argument to the function that will be included in the output message:

```
t = threading.Thread(target=function , args=(i,))
```

The thread does not start running until the `start()` method is called, and that `join()` makes the calling thread wait until the thread has finished the execution:

```
t.start()
t.join()
```

How to determine the current thread

Using arguments to identify or name the thread is cumbersome and unnecessary. Each `Thread` instance has a name with a default value that can be changed as the thread is created. Naming threads is useful in server processes with multiple service threads that handle different operations.

How to do it...

To determine which thread is running, we create three target functions and import the `time` module to introduce a suspend execution of two seconds:

```
import threading
import time

def first_function():
    print (threading.currentThread().getName()+\
          str(' is Starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+\
          str( ' is Exiting \n'))
    return

def second_function():
    print (threading.currentThread().getName()+\
          str(' is Starting \n'))
    time.sleep(2)
```

```
        print (threading.currentThread().getName()+\
                str( ' is Exiting \n'))
        return

def third_function():
    print (threading.currentThread().getName()+\
            str(' is Starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+\
            str( ' is Exiting \n'))
    return

if __name__ == "__main__":

    t1 = threading.Thread\
        (name='first_function', target=first_function)
    t2 = threading.Thread\
        (name='second_function', target=second_function)
    t3 = threading.Thread\
        (name='third_function',target=third_function)

    t1.start()
    t2.start()
    t3.start()
```

The output of this should be, as follows:

How it works...

We instantiate a thread with a target function. Also, we pass the name that is to be printed and if it is not defined, the default name will be used:

```
t1 = threading.Thread(name='first_function', target=first_function)
t2 = threading.Thread(name='second_function', target=second_function)
t3 = threading.Thread(target=third_function)
```

Then, we call the `start()` and `join()` methods on them:

```
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

How to use a thread in a subclass

To implement a new thread using the `threading` module, you have to do the following:

- ▶ Define a new subclass of the `Thread` class
- ▶ Override the `__init__(self [,args])` method to add additional arguments
- ▶ Then, you need to override the `run(self [,args])` method to implement what the thread should do when it is started

Once you have created the new `Thread` subclass, you can create an instance of it and then start a new thread by invoking the `start()` method, which will, in turn, call the `run()` method.

How to do it...

To implement a thread in a subclass, we define the `myThread` class. It has two methods that must be overridden with the thread's arguments:

```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
```

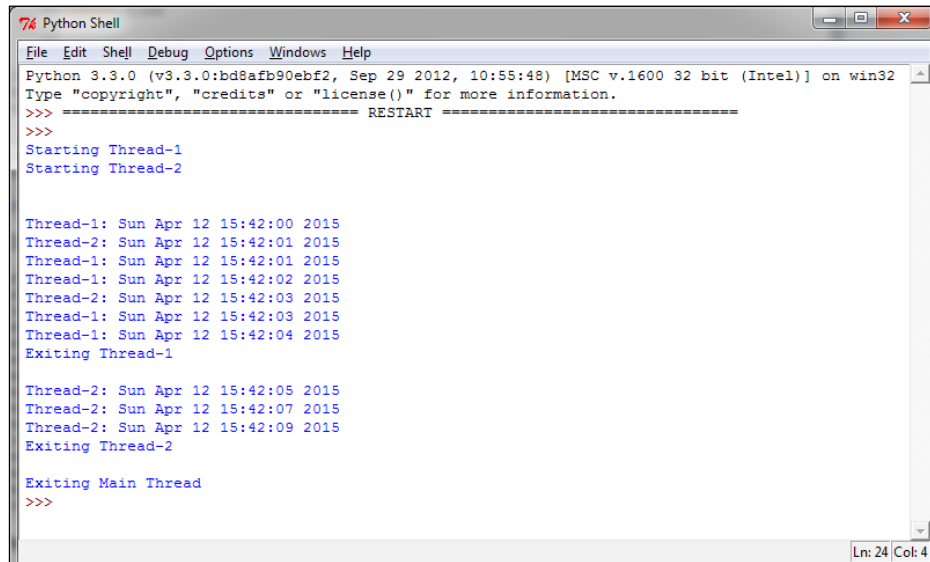
```
        self.name = name
        self.counter = counter
    def run(self):
        print ("Starting " + self.name)
        print_time(self.name, self.counter, 5)
        print ("Exiting " + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print ("%s: %s" %\
                (threadName, time.ctime(time.time())))
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()
print ("Exiting Main Thread")
```

When the previous code is executed, it produces the following result:



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting Thread-1
Starting Thread-2

Thread-1: Sun Apr 12 15:42:00 2015
Thread-2: Sun Apr 12 15:42:01 2015
Thread-1: Sun Apr 12 15:42:01 2015
Thread-1: Sun Apr 12 15:42:02 2015
Thread-2: Sun Apr 12 15:42:03 2015
Thread-1: Sun Apr 12 15:42:03 2015
Thread-1: Sun Apr 12 15:42:04 2015
Exiting Thread-1

Thread-2: Sun Apr 12 15:42:05 2015
Thread-2: Sun Apr 12 15:42:07 2015
Thread-2: Sun Apr 12 15:42:09 2015
Exiting Thread-2

Exiting Main Thread
>>>
```

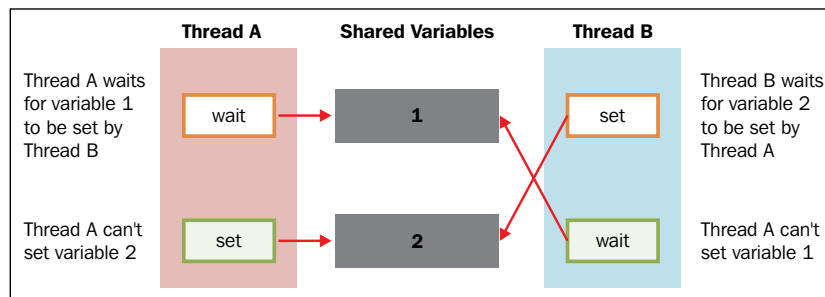

How it works...

The threading module is the preferred form for creating and managing threads. Each thread is represented by a class that extends the `Thread` class and overrides its `run()` method. Then, this method becomes the starting point of the thread. In the main program, we create several objects of the `myThread` type; the execution of the thread begins when the `start()` method is called. Calling the constructor of the `Thread` class is mandatory—using it, we can redefine some properties of the thread as the name or group of the thread. The thread is placed in the active state of the call to `start()` and remains there until it ends the `run()` method or you throw an unhandled exception to it. The program ends when all the threads are terminated.

The `join()` command just handles the termination of threads.

Thread synchronization with Lock and RLock

When two or more operations belonging to concurrent threads try to access the shared memory and at least one of them has the power to change the status of the data without a proper synchronization mechanism a race condition can occur and it can produce invalid code execution and bugs and unexpected behavior. The easiest way to get around the race conditions is the use of a lock. The operation of a lock is simple; when a thread wants to access a portion of shared memory, it must necessarily acquire a lock on that portion prior to using it. In addition to this, after completing its operation, the thread must release the lock that was previously obtained so that a portion of the shared memory is available for any other threads that want to use it. In this way, it is evident that the impossibility of incurring races is critical as the need of the lock for the thread requires that at a given instant, only a given thread can use this part of the shared memory. Despite their simplicity, the use of a lock works. However, in practice, we can see how this approach can often lead the execution to a bad situation of deadlock. A deadlock occurs due to the acquisition of a lock from different threads; it is impossible to proceed with the execution of operations since the various locks between them block access to the resources.



Deadlock

For the sake of simplicity, let's think of a situation wherein there are two concurrent threads (**Thread A** and **Thread B**) who have at their disposal resources **1** and **2**. Suppose **Thread A** requires resource **1** and **Thread B** requires resource **2**. In this case, both threads require their own lock and up to this point, everything proceeds smoothly. Imagine, however, that subsequently, before releasing the lock, **Thread A** requires a lock on resource **2** and **Thread B** requires a lock on resource **1**, which is now necessary for both the processes. Since both resources are locked, the two threads are blocked and waiting each other until the occupied resource is released. This situation is the most emblematic example of the occurrence of a deadlock situation. As said, therefore, showing the use of locks to ensure synchronization so that you can access the shared memory on one hand is a working solution, but, on the other hand, it is potentially destructive in certain cases.

In this recipe, we describe the Python threading synchronization mechanism called `lock()`. It allows us to restrict the access of a shared resource to a single thread or a single type of thread at a time. Before accessing the shared resource of the program, the thread must acquire the lock and must then allow any other threads access to the same resource.

How to do it...

The following example demonstrates how you can manage a thread through the mechanism of `lock()`. In this code, we have two functions: `increment()` and `decrement()`, respectively. The first function increments the value of the shared resource, while the second function decrements the value, where each function is inserted in a suitable thread. In addition to this, each function has a loop in which the increase or decrease is repeated. We want to make sure, through the proper management of the shared resources, that the result of the execution is equal to the value of the shared variable that is initialized to zero.

The sample code is shown, as follows, where each feature within the sample code is properly commented:

```
import threading

shared_resource_with_lock      = 0
shared_resource_with_no_lock   = 0
COUNT = 100000
shared_resource_lock = threading.Lock()

####LOCK MANAGEMENT##
def increment_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
```

```
        shared_resource_lock.acquire()
        shared_resource_with_lock += 1
        shared_resource_lock.release()

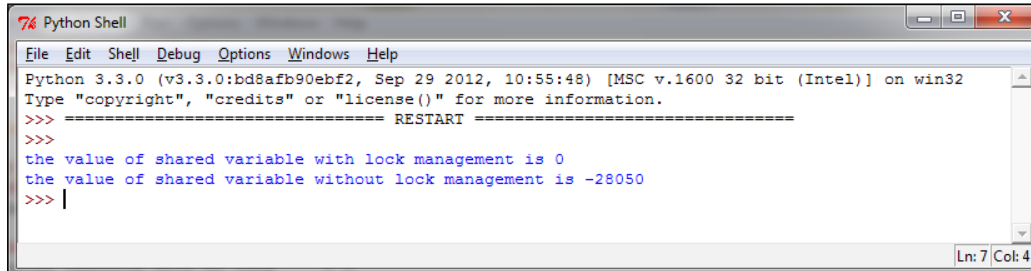
def decrement_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
        shared_resource_lock.acquire()
        shared_resource_with_lock -= 1
        shared_resource_lock.release()

####NO LOCK MANAGEMENT ##
def increment_without_lock():
    global shared_resource_with_no_lock
    for i in range(COUNT):
        shared_resource_with_no_lock += 1

def decrement_without_lock():
    global shared_resource_with_no_lock
    for i in range(COUNT):
        shared_resource_with_no_lock -= 1

####the Main program
if __name__ == "__main__":
    t1 = threading.Thread(target = increment_with_lock)
    t2 = threading.Thread(target = decrement_with_lock)
    t3 = threading.Thread(target = increment_without_lock)
    t4 = threading.Thread(target = decrement_without_lock)
    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
    print ("the value of shared variable with lock management is %s"\
          %shared_resource_with_lock)
    print ("the value of shared variable with race condition is %s"\
          %shared_resource_with_no_lock)
```

This is the result that you get after a single run:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
the value of shared variable with lock management is 0
the value of shared variable without lock management is -28050
>>> |
```

As you can see, we have the correct result with the appropriate management and lock instructions. Note again that the result for the shared variable without lock management could differ from the result shown.

How it works...

In the main method, we have the following procedures:

```
t1 = threading.Thread(target = increment_with_lock)
```

```
t2 = threading.Thread(target = decrement_with_lock)
```

For thread starting, use:

```
t1.start()
```

```
t2.start()
```

For thread joining, use:

```
t1.join()
```

```
t2.join()
```

In the `increment_with_lock()` and `decrement_with_lock()` functions, you can see how to use lock management. When you need to access the resource, call `acquire()` to hold the lock (this will wait for the lock to be released, if necessary) and call `release()` to release it:

```
shared_resource_lock.acquire()
```

```
shared_resource_with_lock -= 1
```

```
shared_resource_lock.release()
```

Let's recap:

- ▶ Locks have two states: locked and unlocked
- ▶ We have two methods that are used to manipulate the locks: `acquire()` and `release()`

The following are the rules:

- ▶ If the state is unlocked, a call to `acquire()` changes the state to locked
- ▶ If the state is locked, a call to `acquire()` blocks until another thread calls `release()`
- ▶ If the state is unlocked, a call to `release()` raises a `RuntimeError` exception
- ▶ If the state is locked, a call to `release()` changes the state to unlocked

There's more...

Despite their theoretical smooth running, the locks are not only subject to harmful situations of deadlock, but also have many other negative aspects for the application as a whole. This is a conservative approach which, by its nature, often introduces unnecessary overhead; it also limits the scalability of the code and its readability. Furthermore, the use of a lock is decidedly in conflict with the possible need to impose the priority of access to the memory shared by the various processes. Finally, from a practical point of view, an application containing a lock presents considerable difficulties when searching for errors (debugging). In conclusion, it would be appropriate to use alternative methods to ensure synchronized access to shared memory and avoid race conditions.

Thread synchronization with RLock

If we want only the thread that acquires a lock to release it, we must use a `RLock()` object. Similar to the `Lock()` object, the `RLock()` object has two methods: `acquire()` and `release()`. `RLock()` is useful when you want to have a thread-safe access from outside the class and use the same methods from inside the class.

How to do it...

In the sample code, we introduced the `Box` class, which has the methods `add()` and `remove()`, respectively, that provide us access to the `execute()` method so that we can perform the action of adding or deleting an item, respectively. Access to the `execute()` method is regulated by `RLock()`:

```
import threading
import time
```

```
class Box(object):
    lock = threading.RLock()
    def __init__(self):
        self.total_items = 0
    def execute(self,n):
        Box.lock.acquire()
        self.total_items += n
        Box.lock.release()
    def add(self):
        Box.lock.acquire()
        self.execute(1)
        Box.lock.release()
    def remove(self):
        Box.lock.acquire()
        self.execute(-1)
        Box.lock.release()

## These two functions run n in separate
## threads and call the Box's methods

def adder(box,items):
    while items > 0:
        print ("adding 1 item in the box\n")
        box.add()
        time.sleep(5)
        items -= 1

def remover(box,items):
    while items > 0:
        print ("removing 1 item in the box")
        box.remove()
        time.sleep(5)
        items -= 1

## the main program build some
## threads and make sure it works
if __name__ == "__main__":
    items = 5
    print ("putting %s items in the box " % items)
    box = Box()
    t1 = threading.Thread(target=adder,args=(box,items))
    t2 = threading.Thread(target=remover,args=(box,items))
    t1.start()
    t2.start()
```

```

t1.join()
t2.join()
print ("%s items still remain in the box " % box.total_items)

```

How it works...

In the main program, we repeated what was written in the preceding example; the two threads `t1` and `t2` are with the associated functions `add()` and `remove()`. The functions are active when the number of items is greater than zero. The call to `RLock()` is carried out inside the `Box` class:

```

class Box(object):
    lock = threading.RLock()

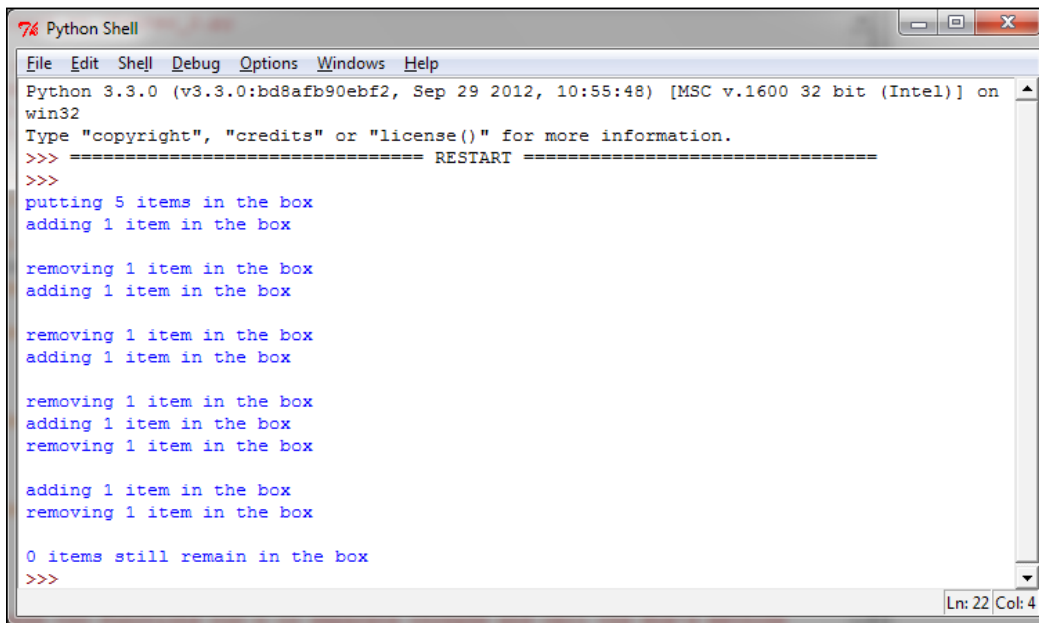
```

The two functions `add()` and `remove()` interact with the items of the `Box` class, respectively, and call the `Box` class methods: `add()` and `remove()`. In each method call, a resource is captured and then released. As for the object `lock()`, `RLock()` owns the `acquire()` and `release()` methods to acquire and release the resource; then for each method, we have the following function calls:

```

Box.lock.acquire()
#...do something
Box.lock.release()

```



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
putting 5 items in the box
adding 1 item in the box

removing 1 item in the box
adding 1 item in the box

removing 1 item in the box
adding 1 item in the box

removing 1 item in the box
adding 1 item in the box
removing 1 item in the box

adding 1 item in the box
removing 1 item in the box

0 items still remain in the box
>>>
Ln: 22 Col: 4

```

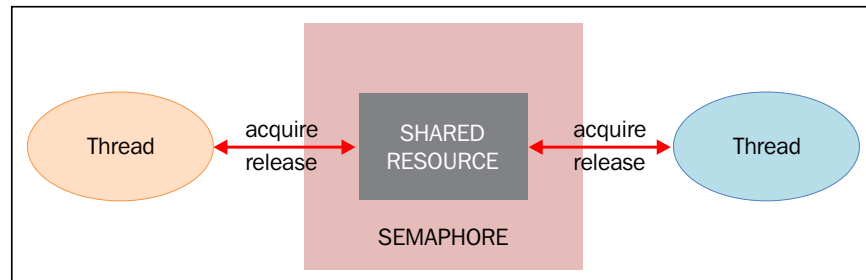
The execution result of the `RLock()` object's example

Thread synchronization with semaphores

Invented by E. Dijkstra and used for the first time in the operating system, a semaphore is an abstract data type managed by the operating system, used to synchronize the access by multiple threads to shared resources and data. Essentially, a semaphore is constituted of an internal variable that identifies the number of concurrent access to a resource to which it is associated.

Also, in the threading module, the operation of a semaphore is based on the two functions `acquire()` and `release()`, as explained:

- ▶ Whenever a thread wants to access a resource that is associated with a semaphore, it must invoke the `acquire()` operation, which decreases the internal variable of the semaphore and allows access to the resource if the value of this variable appears to be non-negative. If the value is negative, the thread would be suspended and the release of the resource by another thread will be placed on hold.
- ▶ Whenever a thread has finished using the data or shared resource, it must release the resource through the `release()` operation. In this way, the internal variable of the semaphore is incremented, and the first waiting thread in the semaphore's queue will have access to the shared resource.



Thread synchronization with semaphores

Although at first glance the mechanism of semaphores does not present obvious problems, it works properly only if the wait and signal operations are performed in atomic blocks. If not, or if one of the two operations is stopped, this could arise unpleasant situations.

Suppose that two threads execute simultaneously, the operation waits on a semaphore, whose internal variable has the value 1. Also assume that after the first thread has the semaphore decremented from 1 to 0, the control goes to the second thread, which decrements the light from 0 to -1 and waits as the negative value of the internal variable. At this point, with the control that returns to the first thread, the semaphore has a negative value and therefore, the first thread also waits.

Therefore, despite the semaphore having access to a thread, the fact that the wait operation was not performed in atomic terms has led to a solution of the stall.

Getting ready

The next code describes the problem, where we have two threads, `producer()` and `consumer()` that share a common resource, which is the item. The task of `producer()` is to generate the item while the `consumer()` thread's task is to use the item produced.

If the item has not yet produced the `consumer()` thread, it has to wait. As soon as the item is produced, the `producer()` thread notifies the consumer that the resource should be used.

How to do it...

In the following example, we use the consumer-producer model to show you the synchronization via semaphores. When the producer creates an item, it releases the semaphore. Also, the consumer acquires it and consumes the shared resource. The synchronization process done via the semaphores is shown in the following code:

```
###Using a Semaphore to synchronize threads

import threading
import time
import random

##The optional argument gives the initial value for the internal
##counter;
##it defaults to 1.
##If the value given is less than 0, ValueError is raised.
semaphore = threading.Semaphore(0)

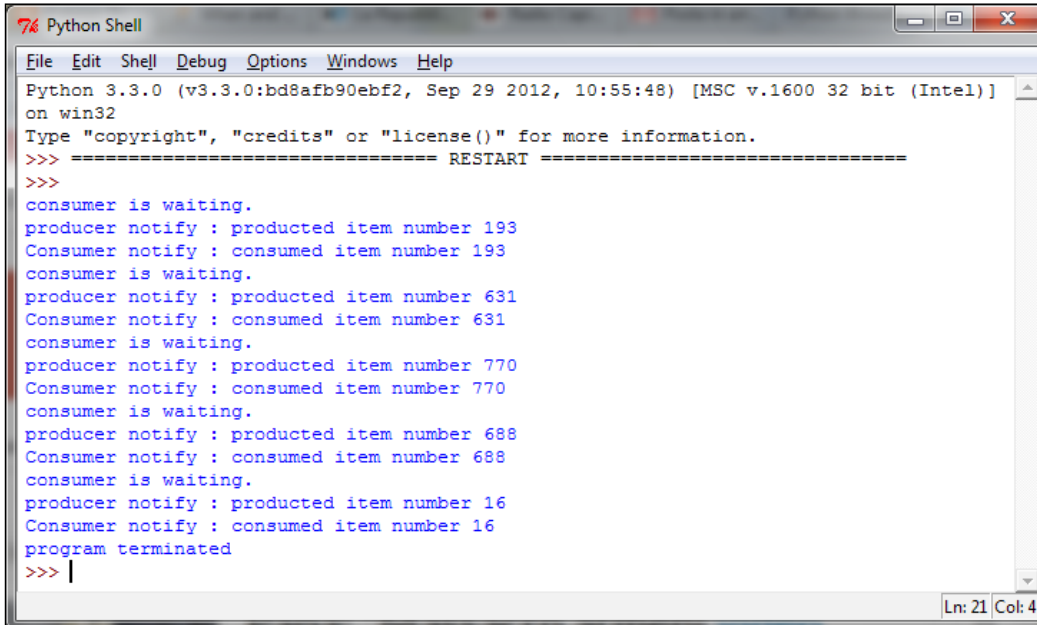
def consumer():
    print ("consumer is waiting.")
    ##Acquire a semaphore
    semaphore.acquire()
    ##The consumer have access to the shared resource
    print ("Consumer notify : consumed item number %s " %item)

def producer():
    global item
    time.sleep(10)
    ##create a random item
    item = random.randint(0,1000)
    print ("producer notify : produced item number %s" %item)
```

```
##Release a semaphore, incrementing the internal counter by one.
##When it is zero on entry and another thread is waiting for it
##to become larger than zero again, wake up that thread.
semaphore.release()

#Main program
if __name__ == '__main__':
    for i in range (0,5) :
        t1 = threading.Thread(target=producer)
        t2 = threading.Thread(target=consumer)
        t1.start()
        t2.start()
        t1.join()
        t2.join()
    print ("program terminated")
```

This is the result that we get after five runs:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
consumer is waiting.
producer notify : producted item number 193
Consumer notify : consumed item number 193
consumer is waiting.
producer notify : producted item number 631
Consumer notify : consumed item number 631
consumer is waiting.
producer notify : producted item number 770
Consumer notify : consumed item number 770
consumer is waiting.
producer notify : producted item number 688
Consumer notify : consumed item number 688
consumer is waiting.
producer notify : producted item number 16
Consumer notify : consumed item number 16
program terminated
>>> |
```

How it works...

Initializing a semaphore to 0, we obtain a so-called semaphore event whose sole purpose is to synchronize the computation of two or more threads. Here, a thread must necessarily make use of data or common resources simultaneously:

```
semaphore = threading.Semaphore(0)
```

This operation is very similar to that described in the lock mechanism of the lock. The `producer()` thread creates the item and after that, frees the resource by calling:

```
semaphore.release()
```

The semaphore's `release()` method increments the counter and then notifies the other thread. Similarly, the `consumer()` method acquires the data by:

```
semaphore.acquire()
```

If the semaphore's counter is equal to 0, it blocks the condition's `acquire()` method until it gets notified by a different thread. If the semaphore's counter is greater than 0, it decrements the value.

Finally, the acquired data is then printed on the standard output:

```
print ("Consumer notify : consumed item number %s " %item)
```

There's more...

A particular use of semaphores is the mutex. A mutex is nothing but a semaphore with an internal variable initialized to the value 1, which allows the realization of mutual exclusion in access to data and resources.

Semaphores are still commonly used in programming languages that are multithreaded; however, using them you can run into situations of deadlock. For example, there is a deadlock situation created when the thread `t1` executes a wait on the semaphore `s1`, while the `t2` thread executes a wait on the semaphore `s1`, and then `t1`, and then executes a wait on `s2` and `t2`, and then executes a wait on `s1`.

Thread synchronization with a condition

A condition identifies a change of state in the application. This is a synchronization mechanism where a thread waits for a specific condition and another thread notifies that this condition has taken place. Once the condition takes place, the thread acquires the lock to get exclusive access to the shared resource.

Getting ready

A good way to illustrate this mechanism is by looking again at a producer/consumer problem. The class producer writes to a buffer as long as it is not full, and the class consumer takes the data from the buffer (eliminating them from the latter), as long as the buffer is full. The class producer will notify the consumer that the buffer is not empty, while the consumer will report to the producer that the buffer is not full.

How to do it...

To show you the condition mechanism, we will again use the consumer producer model:

```
from threading import Thread, Condition
import time

items = []
condition = Condition()

class consumer(Thread):
    def __init__(self):
        Thread.__init__(self)

    def consume(self):
        global condition
        global items

        condition.acquire()
        if len(items) == 0:
            condition.wait()
            print("Consumer notify : no item to consume")
        items.pop()
        print("Consumer notify : consumed 1 item")
        print("Consumer notify : items to consume are "\
              + str(len(items)))
```

```
        condition.notify()
        condition.release()

    def run(self):
        for i in range(0,20):
            time.sleep(10)
            self.consume()

class producer(Thread):
    def __init__(self):
        Thread.__init__(self)

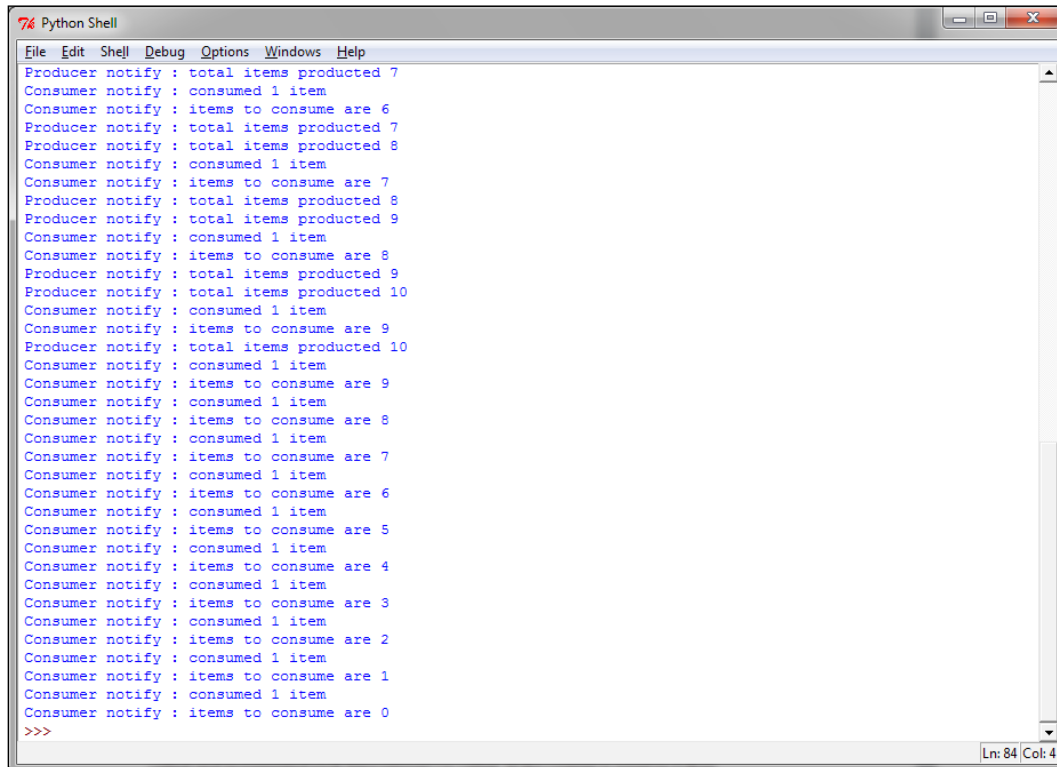
    def produce(self):
        global condition
        global items

        condition.acquire()
        if len(items) == 10:
            condition.wait()
            print("Producer notify : items produced are "\
                  + str(len(items)))
            print("Producer notify : stop the production!!")
            items.append(1)
            print("Producer notify : total items produced "\
                  + str(len(items)))
            condition.notify()
            condition.release()

    def run(self):
        for i in range(0,20):
            time.sleep(5)
            self.produce()

if __name__ == "__main__":
    producer = producer()
    consumer = consumer()
    producer.start()
    consumer.start()
    producer.join()
    consumer.join()
```

This is the result that we get after a single run:



```
Python Shell
File Edit Shell Debug Options Windows Help
Producer notify : total items produced 7
Consumer notify : consumed 1 item
Consumer notify : items to consume are 6
Producer notify : total items produced 7
Producer notify : total items produced 8
Consumer notify : consumed 1 item
Consumer notify : items to consume are 7
Producer notify : total items produced 8
Producer notify : total items produced 9
Consumer notify : consumed 1 item
Consumer notify : items to consume are 8
Producer notify : total items produced 9
Producer notify : total items produced 10
Consumer notify : consumed 1 item
Consumer notify : items to consume are 9
Producer notify : total items produced 10
Consumer notify : consumed 1 item
Consumer notify : items to consume are 9
Consumer notify : consumed 1 item
Consumer notify : items to consume are 8
Consumer notify : consumed 1 item
Consumer notify : items to consume are 7
Consumer notify : consumed 1 item
Consumer notify : items to consume are 6
Consumer notify : consumed 1 item
Consumer notify : items to consume are 5
Consumer notify : consumed 1 item
Consumer notify : items to consume are 4
Consumer notify : consumed 1 item
Consumer notify : items to consume are 3
Consumer notify : consumed 1 item
Consumer notify : items to consume are 2
Consumer notify : consumed 1 item
Consumer notify : items to consume are 1
Consumer notify : consumed 1 item
Consumer notify : items to consume are 0
>>>
Ln: 84 Col: 4
```

How it works...

The class consumer acquires the shared resource that is modeled through the list `items []`:

```
condition.acquire()
```

If the length of the list is equal to 0, the consumer is placed in a waiting state:

```
if len(items) == 0:
    condition.wait()
```

Otherwise, it makes a `pop` operation from the items list:

```
items.pop()
```

So, the consumer's state is notified to the producer and the shared resource is released:

```
condition.notify()
condition.release()
```

The class producer acquires the shared resource and then it verifies that the list is completely full (in our example, we place the maximum number of items, 10, that can be contained in the items list). If the list is full, then the producer is placed in the wait state until the list is consumed:

```
condition.acquire()
if len(items) == 10:
    condition.wait()
```

If the list is not full, a single item is added. The state is notified and the resource is released:

```
condition.notify()
condition.release()
```

There's more...

It's interesting to see the Python internals for the condition synchronizations mechanism. The internal class `_Condition` creates a `RLock()` object if no existing lock is passed to the class's constructor. Also, the lock will be managed when `acquire()` and `released()` are called:

```
class _Condition(_Verbose):
    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self.__lock = lock
```

Thread synchronization with an event

Events are objects that are used for communication between threads. A thread waits for a signal while another thread outputs it. Basically, an event object manages an internal flag that can be set to `true` with the `set()` method and reset to `false` with the `clear()` method. The `wait()` method blocks until the flag is `true`.

How to do it...

To understand the thread synchronization through the event object, let's take a look again at the producer/consumer problem:

```
import time
from threading import Thread, Event
import random

items = []
event = Event()

class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        while True:
            time.sleep(2)
            self.event.wait()
            item = self.items.pop()
            print ('Consumer notify : %d popped from list by %s'\
                  %(item, self.name))

class producer(Thread):
    def __init__(self, integers, event):
        Thread.__init__(self)
        self.items = integers
        self.event = event

    def run(self):
        global item
        for i in range(100):
            time.sleep(2)
            item = random.randint(0, 256)
            self.items.append(item)
            print ('Producer notify : item N° %d appended \
                  to list by %s'\
                  % (item, self.name))
            print ('Producer notify : event set by %s'\
                  % self.name)
```



```

        self.event.set()
        print ('Produce notify : event cleared by %s \n\'
              % self.name)
        self.event.clear()

if __name__ == '__main__':
    t1 = producer(items, event)
    t2 = consumer(items, event)
    t1.start()
    t2.start()
    t1.join()
    t2.join()

```

This is the output that we get when we run the program. The `t1` thread appends a value to the list and then sets the event to notify the consumer. The consumer's call to `wait()` stops blocking and the integer is retrieved from the list.

```

*Python Shell*
File Edit Shell Debug Options Windows Help
Producer notify : item 204 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 204 popped from list by Thread-2

Producer notify : item 98 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1

Consumer notify : 98 popped from list by Thread-2
Producer notify : item 90 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 90 popped from list by Thread-2

Producer notify : item 3 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 3 popped from list by Thread-2

Producer notify : item 162 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 162 popped from list by Thread-2

Producer notify : item 208 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 208 popped from list by Thread-2

Producer notify : item 97 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 97 popped from list by Thread-2

Producer notify : item 233 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 233 popped from list by Thread-2
Ln: 480 Col: 0

```

How it works...

The `producer` class is initialized with the list of items and the `Event()` function. Unlike the example with condition objects, the item list is not global, but it is passed as a parameter:

```
class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event
```

In the `run` method for each item that is created, the `producer` class appends it to the list of items and then notifies the event. There are two steps that you need to take for this and the first step is as follows:

```
self.event.set()
```

The second step is:

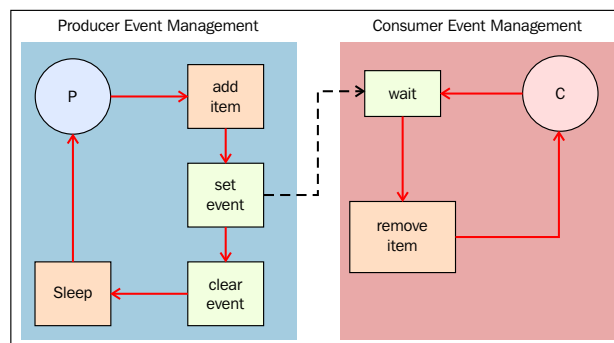
```
self.event.clear()
```

The `consumer` class is initialized with the list of items and the `Event()` function.

In the `run` method, the consumer waits for a new item to consume. When the item arrives, it is popped from the item list:

```
def run(self):
    while True:
        time.sleep(2)
        self.event.wait()
        item = self.items.pop()
        print ('Consumer notify : %d popped from list by %s' %
              (item, self.name))
```

All the operations between the `producer` and the `consumer` classes can be easily resumed with the help of the following schema:



Thread synchronization with event objects

Using the with statement

Python's `with` statement was introduced in Python 2.5. It's useful when you have two related operations that must be executed as a pair with a block of code in between. Also, with the `with` statement, you can allocate and release some resource exactly where you need it; for this reason, the `with` statement is called a context manager. In the `threading` module, all the objects provided by the `acquire()` and `release()` methods may be used in a `with` statement block.

So the following objects can be used as context managers for a `with` statement:

- ▶ Lock
- ▶ RLock
- ▶ Condition
- ▶ Semaphore

Getting ready

In this example, we simply test all the objects using the `with` statement.

How to do it...

This example shows the basic use of the `with` statement. We have a set with the most important synchronization primitives. So, we test them by calling each one with the `with` statement:

```
import threading
import logging

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

def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with' %statement)

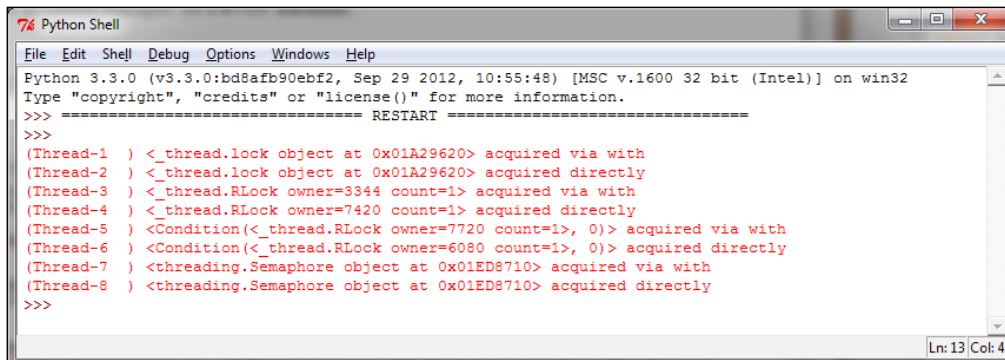
def threading_not_with(statement):
    statement.acquire()
    try:
        logging.debug('%s acquired directly' %statement )
    finally:
        statement.release()
```

```
if __name__ == '__main__':

    #let's create a test battery
    lock = threading.Lock()
    rlock = threading.RLock()
    condition = threading.Condition()
    mutex = threading.Semaphore(1)
    threading_synchronization_list = \
        [lock, rlock, condition, mutex]

    #in the for cycle we call the threading_with
    # e threading_no_with function
    for statement in threading_synchronization_list :
        t1 = threading.Thread(target=threading_with,
                               args=(statement,))
        t2 = threading.Thread(target=threading_not_with,
                               args=(statement,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

The output shows the use of the `with` statement for each function and also where it is not used:



```
Python 3.3.0 (v3.3.0:bd8a9b90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
(Thread-1 ) <_thread.lock object at 0x01A29620> acquired via with
(Thread-2 ) <_thread.lock object at 0x01A29620> acquired directly
(Thread-3 ) <_thread.RLock owner=3344 count=1> acquired via with
(Thread-4 ) <_thread.RLock owner=7420 count=1> acquired directly
(Thread-5 ) <Condition(<_thread.RLock owner=7720 count=1>, 0)> acquired via with
(Thread-6 ) <Condition(<_thread.RLock owner=6080 count=1>, 0)> acquired directly
(Thread-7 ) <threading.Semaphore object at 0x01ED8710> acquired via with
(Thread-8 ) <threading.Semaphore object at 0x01ED8710> acquired directly
>>>
```

How it works...

In the main program, we have defined a list, `threading_synchronization_list`, of thread communication directives that are to be tested:

```
lock = threading.Lock()
rlock = threading.RLock()
condition = threading.Condition()
mutex = threading.Semaphore(1)
threading_synchronization_list = \

    [lock, rlock, condition, mutex]
```

After defining them, we pass each object in the `for` cycle:

```
for statement in threading_synchronization_list :
    t1 = threading.Thread(target=threading_with,
        args=(statement,))
    t2 = threading.Thread(target=threading_not_with,
        args=(statement,))
```

Finally, we have two target functions, in which the `threading_with` tests the `with` statement:

```
def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with' %statement)
```

There's more...

In the following example we have used the Python support for logging, as we can see:

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

It embeds the thread name in every log message using the formatter code's `%(threadName)s` statement. The logging module is thread-safe, so the messages from different threads are kept distinct in the output.

Thread communication using a queue

As discussed earlier, threading can be complicated when threads need to share data or resources. As we saw, the Python threading module provides many synchronization primitives, including semaphores, condition variables, events, and locks. While these options exist, it is considered a best practice to instead concentrate on using the module queue. Queues are much easier to deal with and make threaded programming considerably safer, as they effectively funnel all access to a resource of a single thread and allow a cleaner and more readable design pattern.

We will simply consider these four queue methods:

- ▶ `put()`: This puts an item in the queue
- ▶ `get()`: This removes and returns an item from the queue
- ▶ `task_done()`: This needs to be called each time an item has been processed
- ▶ `join()`: This blocks until all items have been processed

How to do it...

In this example, we will see how to use the threading module with the queue module. Also, we have two entities here that try to share a common resource, a queue. The code is as follows:

```
from threading import Thread, Event
from queue import Queue
import time
import random
```

```
class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
```

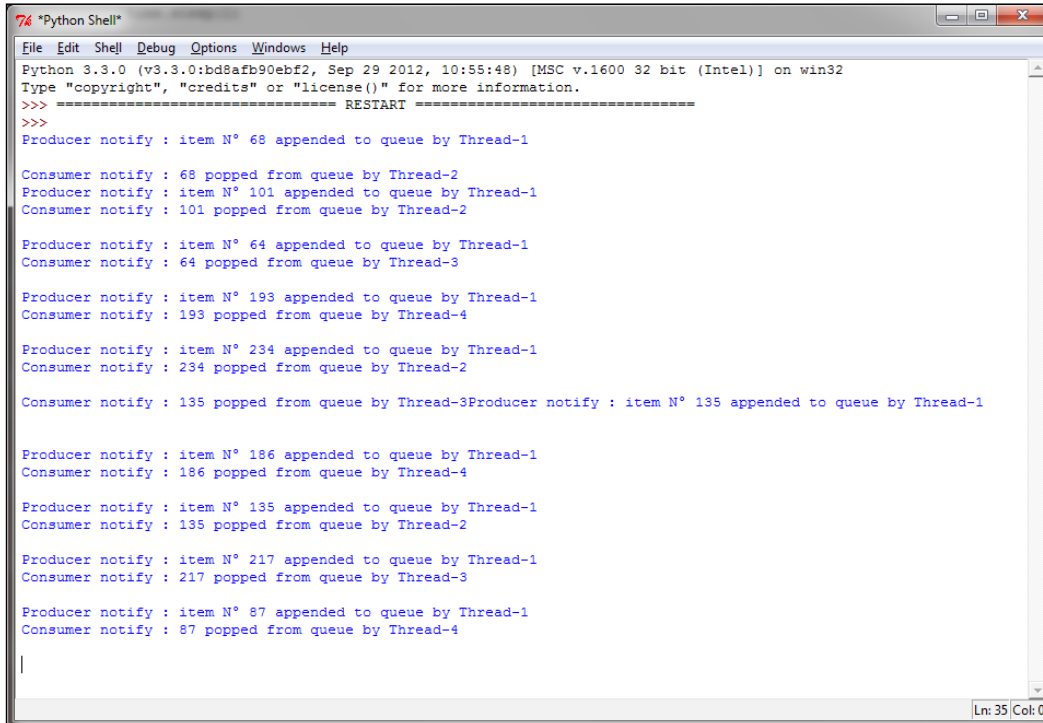
```
        item = random.randint(0, 256)
        self.queue.put(item)
        print ('Producer notify: item N°%d appended to queue by %s'
              '\n'
              % (item, self.name))
        time.sleep(1)

class consumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            item = self.queue.get()
            print ('Consumer notify : %d popped from queue by %s'
                  % (item, self.name))
            self.queue.task_done()

if __name__ == '__main__':
    queue = Queue()
    t1 = producer(queue)
    t2 = consumer(queue)
    t3 = consumer(queue)
    t4 = consumer(queue)
    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
```

After running the code, you should have an output similar to this:



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8a9b90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Producer notify : item N° 68 appended to queue by Thread-1
Consumer notify : 68 popped from queue by Thread-2
Producer notify : item N° 101 appended to queue by Thread-1
Consumer notify : 101 popped from queue by Thread-2
Producer notify : item N° 64 appended to queue by Thread-1
Consumer notify : 64 popped from queue by Thread-3
Producer notify : item N° 193 appended to queue by Thread-1
Consumer notify : 193 popped from queue by Thread-4
Producer notify : item N° 234 appended to queue by Thread-1
Consumer notify : 234 popped from queue by Thread-2
Consumer notify : 135 popped from queue by Thread-3Producer notify : item N° 135 appended to queue by Thread-1
Producer notify : item N° 186 appended to queue by Thread-1
Consumer notify : 186 popped from queue by Thread-4
Producer notify : item N° 135 appended to queue by Thread-1
Consumer notify : 135 popped from queue by Thread-2
Producer notify : item N° 217 appended to queue by Thread-1
Consumer notify : 217 popped from queue by Thread-3
Producer notify : item N° 87 appended to queue by Thread-1
Consumer notify : 87 popped from queue by Thread-4
|
Ln: 35 Col: 0
```

How it works...

First, the producer class. We don't need to pass the integers list because we use the queue to store the integers that are generated:

```
class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue
```


The thread in the `producer` class generates integers and puts them in the queue in a `for` loop:

```
def run(self) :
    for i in range(100):
        item = random.randint(0, 256)
        self.queue.put(item)
```

The producer uses `Queue.put(item[, block[, timeout]])` to insert data into the queue. It has the logic to acquire the lock before inserting data in a queue.

There are two possibilities:

- ▶ If optional args `block` is `true` and `timeout` is `None` (this is the default case that we used in the example), it is necessary for us to block until a free slot is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises the full exception if no free slot is available within that time.
- ▶ If the block is `false`, put an item in the queue if a free slot is immediately available; otherwise, raise the full exception (`timeout` is ignored in this case). Here, `put()` checks whether the queue is full and then calls `wait()` internally and after this, the producer starts waiting.

Next is the `consumer` class. The thread gets the integer from the queue and indicates that it is done working on it using `task_done()`:

```
def run(self):
    while True:
        item = self.queue.get()
        self.queue.task_done()
```

The consumer uses `Queue.get([block[, timeout]])` and acquires the lock before removing data from the queue. If the queue is empty, it puts the consumer in a waiting state.

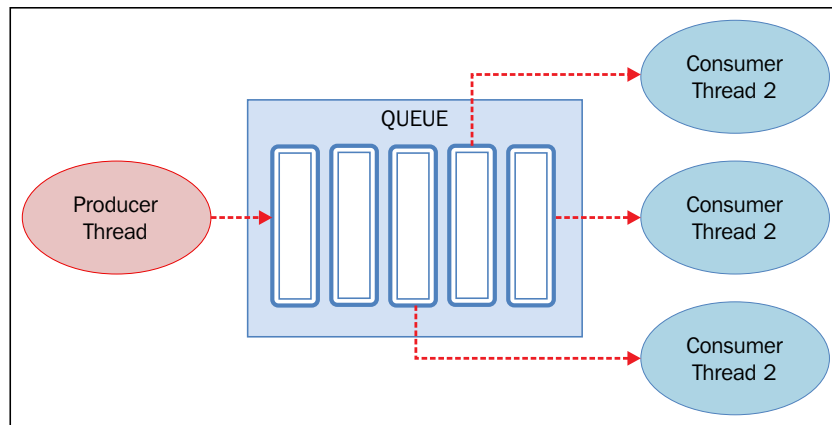
Finally, in the main, we create the `t` thread for the producer and three threads, `t1`, `t2`, and `t3` for the consumer class:

```
if __name__ == '__main__':
    queue = Queue()
    t = producer(queue)
    t1 = consumer(queue)
    t2 = consumer(queue)
    t3 = consumer(queue)
```

```
t.start()
t1.start()
t2.start()
t3.start()

t.join()
t1.join()
t2.join()
t3.join()
```

All the operations between the `producer` class and the `consumer` class can easily be resumed with the following schema:



Thread synchronization with the queue module

Evaluating the performance of multithread applications

In this recipe, we will verify the impact of the GIL, evaluating the performance of a multithread application. The GIL, as described in the previous chapter, is the lock introduced by the CPython interpreter. The GIL prevents parallel execution of multiple threads in the interpreter. Before being executed each thread must wait for the GIL to release the thread that is running. In fact, the interpreter forces the executing thread to acquire the GIL before it accesses anything on the interpreter itself as the stack and instances of Python objects. This is precisely the purpose of GIL—it prevents concurrent access to Python objects from different threads. The GIL then protects the memory of the interpreter and makes the garbage work in the right manner. The fact is that the GIL prevents the programmer from improving the performance by executing threads in parallel. If we remove the GIL from the CPython interpreter, the threads would be executed in parallel. The GIL does not prevent a process from running on a different processor, it simply allows only one thread at a time to turn inside the interpreter.

How to do it...

The next code is a simple tool that is used to evaluate the performance of a multithreaded application. Each test calls a function only once in a hundred loop iterations. Then, we will see the fastest among the hundred calls. In the `for` loop, we call the `non_threaded` and `threaded` functions. Also, we iterate the tests that increase the number of calls and threads. We will try with 1, 2, 3, 4, and 8 at the end of calls threads. In the non-threaded execution, we simply call the function sequentially the same number of times corresponding to those threads that we would use. To keep things simple, all the measurements of the speed of execution are provided by the Python's module `timer`.

This module is designed to evaluate the performance of pieces of Python code, which are generally single statements.

The code is as follows:

```
from threading import Thread

class threads_object(Thread):
    def run(self):
        function_to_run()

class nothreads_object(object):
    def run(self):
        function_to_run()

def non_threaded(num_iter):
    funcs = []
    for i in range(int(num_iter)):
        funcs.append(nothreads_object())
    for i in funcs:
        i.run()

def threaded(num_threads):
    funcs = []
    for i in range(int(num_threads)):
        funcs.append(threads_object())
    for i in funcs:
        i.start()
    for i in funcs:
        i.join()

def function_to_run():
    pass
```

```
def show_results(func_name, results):
    print ("%23s %4.6f seconds" % (func_name, results))

if __name__ == "__main__":
    import sys
    from timeit import Timer

    repeat = 100
    number = 1
    num_threads = [ 1, 2, 4, 8]

    print ('Starting tests')
    for i in num_threads:
        t = Timer("non_threaded(%s)" \
                  % i, "from __main__ import non_threaded")
        best_result = \
            min(t.repeat(repeat=repeat, number=number))
        show_results("non_threaded (%s iters)" \
                    % i, best_result)

        t = Timer("threaded(%s)" \
                  % i, "from __main__ import threaded")
        best_result = \
            min(t.repeat(repeat=repeat, number=number))
        show_results("threaded (%s threads)" \
                    % i, best_result)

    print ('Iterations complete')
```

How it works...

We performed a total of three tests and for each head, we used a different function, changing the function code `function_to_run()` defined in the sample code.

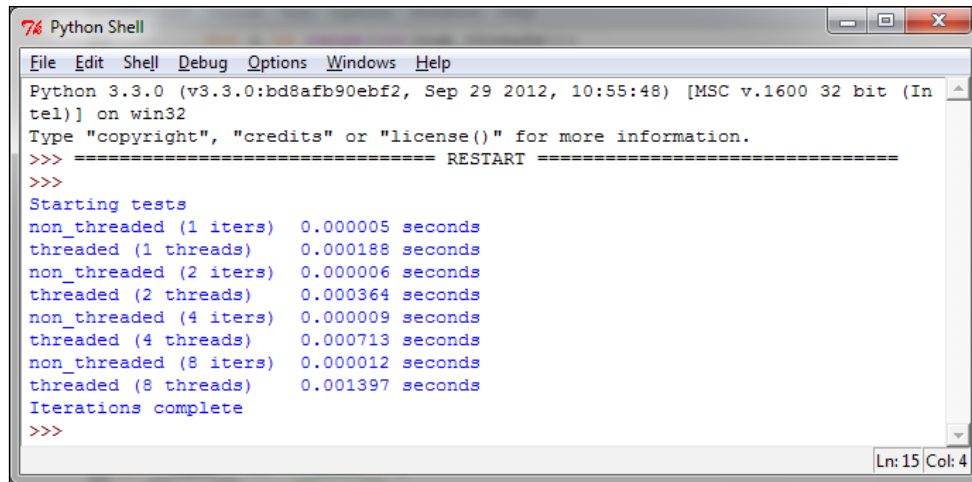
The machine used for these tests is a Core 2 Duo CPU – 2.33Ghz.

The first test

In this test, we simply evaluate the empty function:

```
def function_to_run():
    pass
```

It will show us the overhead associated with each mechanism that we are testing:



The screenshot shows a Python Shell window with the following output:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters) 0.000005 seconds
threaded (1 threads) 0.000188 seconds
non_threaded (2 iters) 0.000006 seconds
threaded (2 threads) 0.000364 seconds
non_threaded (4 iters) 0.000009 seconds
threaded (4 threads) 0.000713 seconds
non_threaded (8 iters) 0.000012 seconds
threaded (8 threads) 0.001397 seconds
Iterations complete
>>>
```

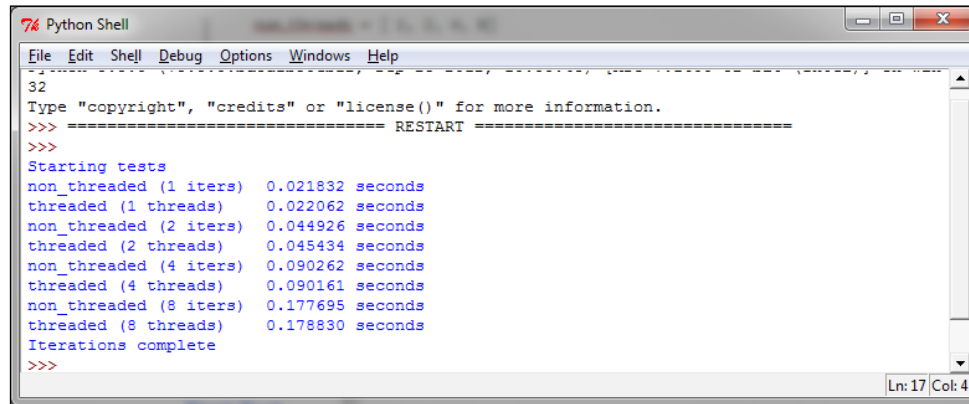
If we look at the results, we see how the thread calls are more expensive than the calls without threads. In particular, we also note how the cost of adding the thread is proportional to their number; in our example, we have four threads with 0.0007143 seconds, while with eight threads, we employ 0.001397 seconds.

The second test

A typical example of threaded applications is the processing of numbers. Let's take a simple method to calculate the brute force of the Fibonacci sequence; note that there is no sharing of the state here, just try to include more tasks that generate sequences of numbers:

```
def function_to_run():
    a, b = 0, 1
    for i in range(10000):
        a, b = b, a + b
```

This is the output:



```
Python Shell
File Edit Shell Debug Options Windows Help
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters) 0.021832 seconds
threaded (1 threads) 0.022062 seconds
non_threaded (2 iters) 0.044926 seconds
threaded (2 threads) 0.045434 seconds
non_threaded (4 iters) 0.090262 seconds
threaded (4 threads) 0.090161 seconds
non_threaded (8 iters) 0.177695 seconds
threaded (8 threads) 0.178830 seconds
Iterations complete
>>>
```

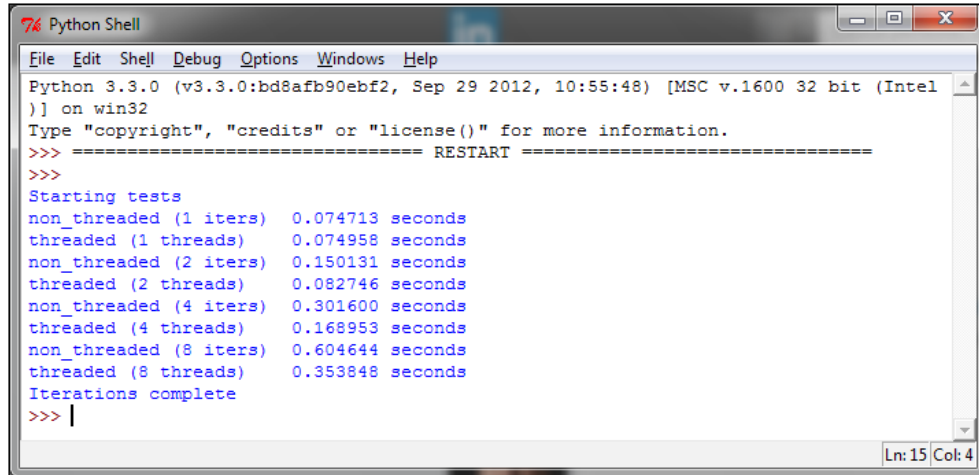
As we can see from the output, we get no advantage by increasing the number of threads. The function is executed in Python and because of the overhead for creating threads and GIL, the multithreaded example can never be faster than the non-threaded example. Again, let's remember that the GIL allows only one thread at a time to access the interpreter.

The third test

The following test consists in reading 1,000 times a block of data (1Kb) from the `test.dat` file. The function tested is as follows:

```
def function_to_run():
    fh=open("C:\\CookBookFileExamples\\test.dat","rb")
    size = 1024
    for i in range(1000):
        fh.read(size)
```

These are the results of the test:



```

Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters) 0.074713 seconds
threaded (1 threads) 0.074958 seconds
non_threaded (2 iters) 0.150131 seconds
threaded (2 threads) 0.082746 seconds
non_threaded (4 iters) 0.301600 seconds
threaded (4 threads) 0.168953 seconds
non_threaded (8 iters) 0.604644 seconds
threaded (8 threads) 0.353848 seconds
Iterations complete
>>> |

```

We have begun to see a better result in the multithreading case. In particular, we've noted how the threaded execution is half time-consuming if we compare it with the `non_threaded` one. Let's remember that in real life, we would not use threads as a benchmark. Typically, we would put the threads in a queue, pull them out, and perform other tasks. Having multiple threads that execute the same function although useful in certain cases, is not a common use case for a concurrent program, unless it divides the data in the input.

The fourth test

In the final example, we use `urllib.request`, a Python module for fetching URL's. This module based on the `socket` module, is written in C and is thread-safe.

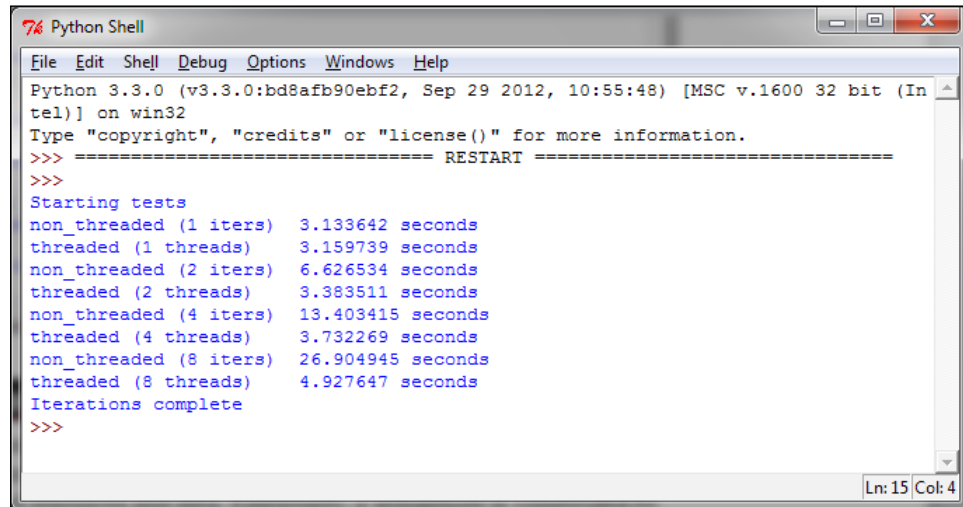
The following script tries to get to the `https://www.packtpub.com/` main page and simply read the first 1k bytes of it:

```

def function_to_run():
    import urllib.request
    for i in range(10):
        with urllib.request.urlopen("https://www.packtpub.com/") as f:
            f.read(1024)

```

The following is the result of the preceding code:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters) 3.133642 seconds
threaded (1 threads) 3.159739 seconds
non_threaded (2 iters) 6.626534 seconds
threaded (2 threads) 3.383511 seconds
non_threaded (4 iters) 13.403415 seconds
threaded (4 threads) 3.732269 seconds
non_threaded (8 iters) 26.904945 seconds
threaded (8 threads) 4.927647 seconds
Iterations complete
>>>
```

As you can see, during the I/O, the GIL is released. The multithreading execution becomes faster than the single-threaded execution. Since many applications perform a certain amount of work in the I/O, the GIL does not prevent a programmer from creating a multithreading work that concurrently increases the speed of execution.

There's more...

Let's remember that you do not add threads to speed up the startup time of an application, but to add support to the concurrence. For example, it's useful to create a pool of threads once and then reuse the worker. This allows us to split a big dataset and run the same function on different parts (the producer/consumer model). So, although it is not the norm for concurrent applications, these tests are designed to be simple. Is the GIL an obstacle for those who work on pure Python and try to exploit multi-core hardware architectures? Yes it does. While threads are a language construct, the CPython interpreter is the bridge between the threads and operating system. This is why Jython, IronPython, and others interpreters do not possess GIL, as it was simply not necessary and it has not been reimplemented in the interpreter.