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

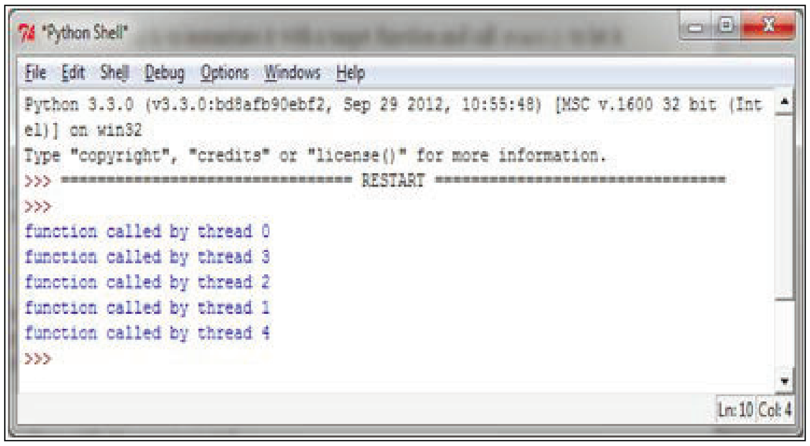
It is useful to spawn a thread and pass arguments to it that tell it what work to do. This example passes a number, which is the thread number, and then prints out the result.

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 (Int
el)] 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.

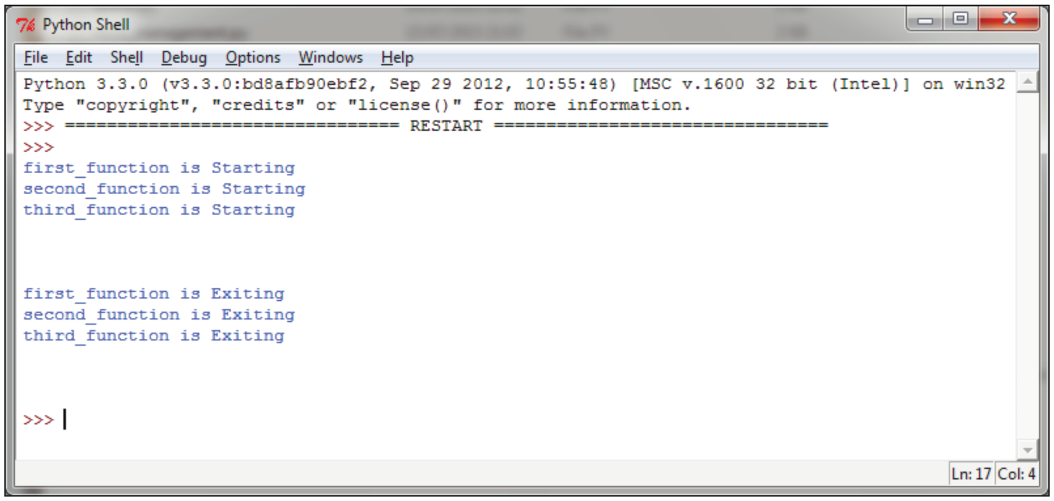
How to do it...

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:



```
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 ================================
>>>
first_function is Starting
second_function is Starting
third_function is Starting



first_function is Exiting
second_function is Exiting
third_function is Exiting



>>> 
```

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.

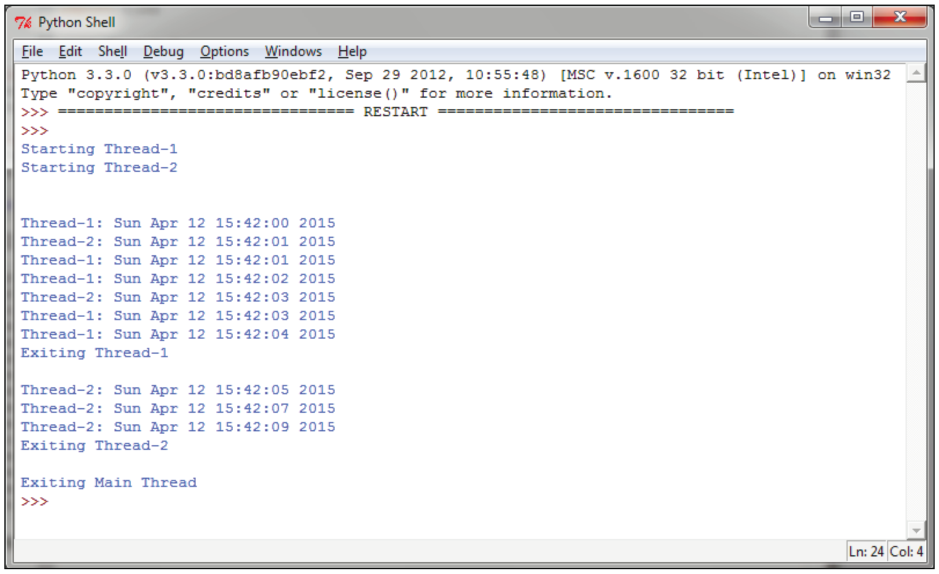
How to do it...

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 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
>>>
```

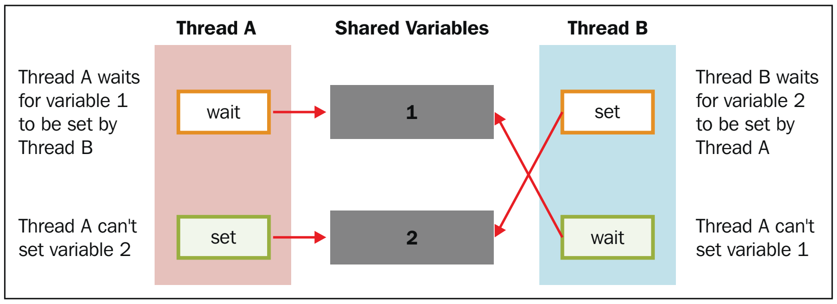
How it works...

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.


Thread A
Shared Variables
Thread B
Thread A waits for variable 1 to be set by Thread B
wait
1
set
Thread B waits for variable 2 to be set by Thread A
Thread A can't set variable 2
set
2
wait
Thread A can't set variable 1

Deadlock