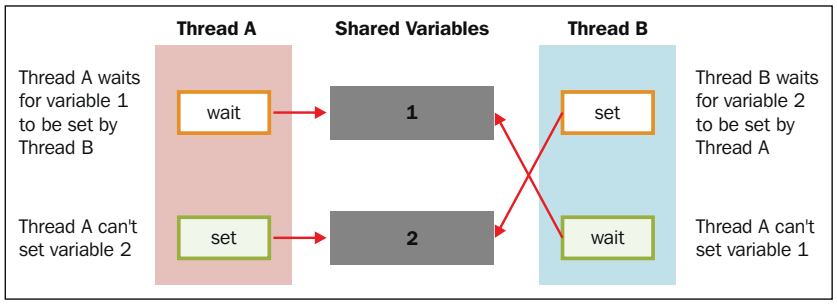
How it works...

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...

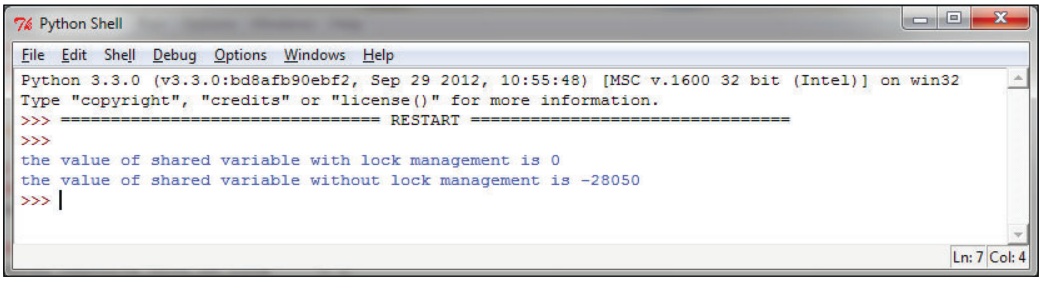
The following example demonstrates how you can manage a thread through the mechanism of `lock()`. In this code, we have two functions: `increment()` and `decrement()`, respectively. The first function increments the value of the shared resource, while the second function decrements the value, where each 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:



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)
```

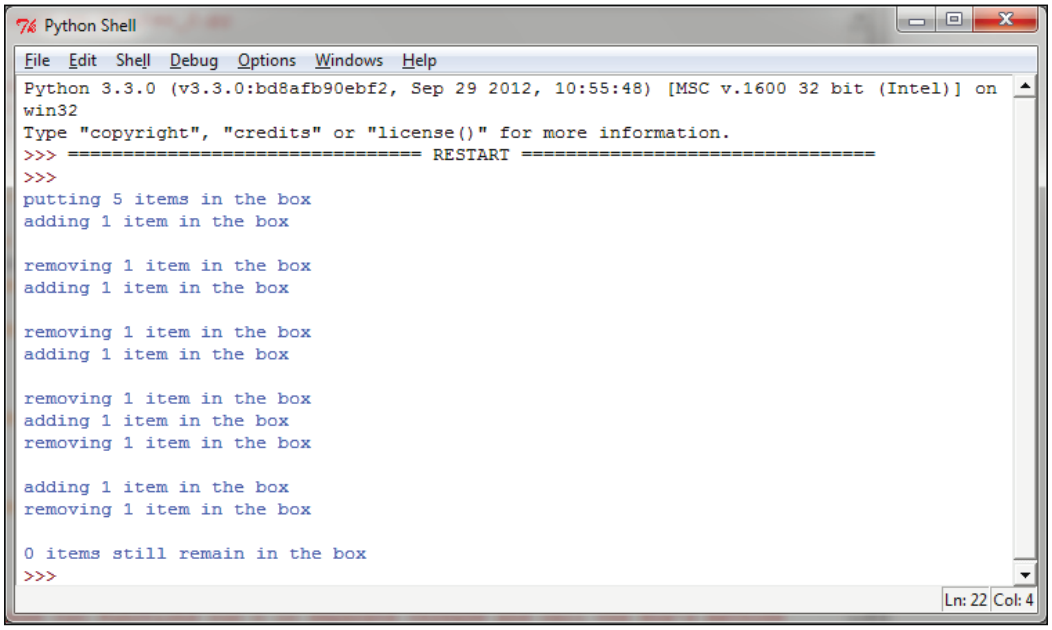
How it works...

In the main program, we repeated what was written in the preceding example; the two threads `t1` and `t2` are with the associated functions `adder()` and `remover()`. 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 `adder()` and `remover()` 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()
```



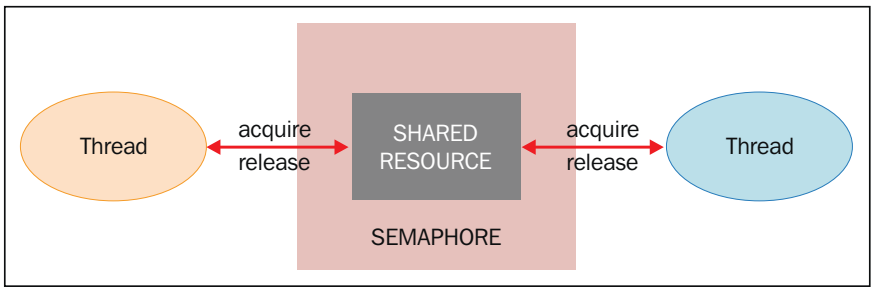
The execution result of the RLock() object's example

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: