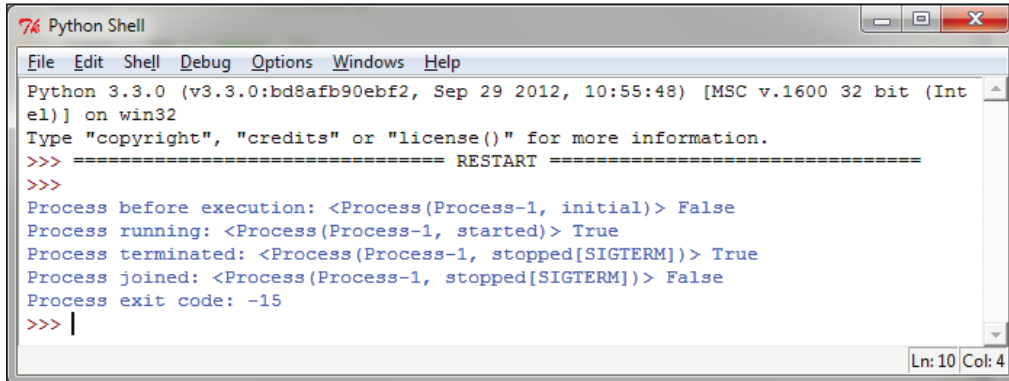


The following is the output we get when we use the preceding command:



```

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 =====
>>>
Process before execution: <Process(Process-1, initial)> False
Process running: <Process(Process-1, started)> True
Process terminated: <Process(Process-1, stopped[SIGTERM])> True
Process joined: <Process(Process-1, stopped[SIGTERM])> False
Process exit code: -15
>>> |
Ln: 10 Col: 4

```

## How it works...

We create the process and then monitor its lifetime by the `is_alive()` method. Then, we finish it with a call to `terminate()`:

```
p.terminate()
```

Finally, we verify the status code when the process is finished, and read the attribute of the `ExitCode` process. The possible values of `ExitCode` are, as follows:

- ▶ == 0: This means that no error was produced
- ▶ > 0: This means that the process had an error and exited that code
- ▶ < 0: This means that the process was killed with a signal of  $-1 * \text{ExitCode}$

For our example, the output value of the `ExitCode` code is equal to `-15`. The negative value `-15` indicates that the child was terminated by an interrupt signal identified by the number `15`.

## How to use a process in a subclass

To implement a custom subclass and process, we must:

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

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

## How to do it...

We will rewrite the first example in this manner:

```
#Using a process in a subclass Chapter 3: Process Based #Parallelism

import multiprocessing

class MyProcess(multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' %self.name)
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = MyProcess ()
        jobs.append(p)
        p.start ()
    p.join()
```

To run the script from the Command Prompt, type the following command:

```
python subclass_process.py
```

The result of the preceding command is as follows:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python subclass_process.py
```

```
called run method in process: MyProcess-1
called run method in process: MyProcess-2
called run method in process: MyProcess-3
called run method in process: MyProcess-4
called run method in process: MyProcess-5
```

## How it works...

Each Process subclass could be represented by a class that extends the Process class and overrides its run() method. This method is the starting point of Process:

```
class MyProcess (multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' %self.name)
        return
```

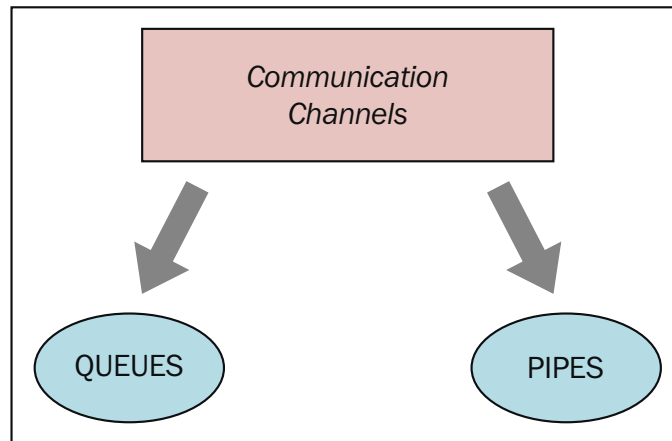
In the main program, we create several objects of the type `MyProcess()`. The execution of the thread begins when the `start()` method is called:

```
p = MyProcess()  
p.start()
```

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

## How to exchange objects between processes

The development of parallel applications has the need for the exchange of data between processes. The multiprocessing library has two communication channels with which it can manage the exchange of objects: queues and pipes.



Communication channels in the multiprocessing module

## Using queue to exchange objects

As explained before, it is possible for us to share data with the queue data structure.

A queue returns a process shared queue, is thread and process safe, and any serializable object (Python serializes an object using the `pickle` module) can be exchanged through it.

## How to do it...

In the following example, we show you how to use a queue for a producer-consumer problem. The `producer` class creates the item and queues and then, the `consumer` class provides the facility to remove the inserted item:

```
import multiprocessing
import random
import time

class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ("Process Producer : item %d appended to queue %s"\
                  % (item,self.name))
            time.sleep(1)
            print ("The size of queue is %s"\
                  % self.queue.qsize())

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

    def run(self):
        while True:
            if (self.queue.empty()):
                print("the queue is empty")
                break
            else :
                time.sleep(2)
                item = self.queue.get()
                print ('Process Consumer : item %d popped from by %s \n'\
                      % (item, self.name))
                time.sleep(1)
```

```
if __name__ == '__main__':
    queue = multiprocessing.Queue()
    process_producer = producer(queue)
    process_consumer = consumer(queue)
    process_producer.start()
    process_consumer.start()
    process_producer.join()
    process_consumer.join()
```

This is the output that we get after the execution:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python using_queue.py
```

```
Process Producer : item 69 appended to queue producer-1
The size of queue is 1
Process Producer : item 168 appended to queue producer-1
The size of queue is 2
Process Consumer : item 69 popped from by consumer-2
Process Producer : item 235 appended to queue producer-1
The size of queue is 2
Process Producer : item 152 appended to queue producer-1
The size of queue is 3
Process Producer : item 213 appended to queue producer-1
Process Consumer : item 168 popped from by consumer-2
The size of queue is 3
Process Producer : item 35 appended to queue producer-1
The size of queue is 4
Process Producer : item 218 appended to queue producer-1
The size of queue is 5
Process Producer : item 175 appended to queue producer-1
Process Consumer : item 235 popped from by consumer-2
The size of queue is 5
Process Producer : item 140 appended to queue producer-1
The size of queue is 6
Process Producer : item 241 appended to queue producer-1
The size of queue is 7
Process Consumer : item 152 popped from by consumer-2
Process Consumer : item 213 popped from by consumer-2
```

```

Process Consumer : item 35 popped from by consumer-2
Process Consumer : item 218 popped from by consumer-2
Process Consumer : item 175 popped from by consumer-2
Process Consumer : item 140 popped from by consumer-2
Process Consumer : item 241 popped from by consumer-2
the queue is empty

```

## How it works...

The multiprocessing class has its `Queue` object instantiated in the main program:

```

if __name__ == '__main__':
    queue = multiprocessing.Queue()

```

Then, we create the two processes, `producer` and `consumer`, with the `Queue` object as an attribute:

```

process_producer = producer(queue)
process_consumer = consumer(queue)

```

The process `producer` is responsible for entering 10 items in the queue using its `put()` method:

```

for i in range(10):
    item = random.randint(0, 256)
    self.queue.put(item)

```

The process `consumer` has the task of removing the items from the queue (using the `get` method) and verifying that the queue is not empty. If this happens, the flow inside the `while` loop ends with a `break` statement:

```

def run(self):
    while True:
        if (self.queue.empty()):
            print("the queue is empty")
            break
        else :
            time.sleep(2)
            item = self.queue.get()
            print ('Process Consumer : item %d popped from by %s
\n' \
                  % (item, self.name))
            time.sleep(1)

```

## There's more...

A queue has the `JoinableQueue` subclass. It has the following two additional methods:

- ▶ `task_done()`: This indicates that a task is complete, for example, after the `get()` method is used to fetch items from the queue. So, it must be used only by queue consumers.
- ▶ `join()`: This blocks the processes until all the items in the queue have been achieved and processed.

## Using pipes to exchange objects

The second communication channel is the pipe data structure.

A pipe does the following:

- ▶ Returns a pair of connection objects connected by a pipe
- ▶ In this, every object has send/receive methods to communicate between processes

## How to do it...

Here is a simple example with pipes. We have one process pipe the gives out numbers from 0 to 9 and another process that takes the numbers and squares them:

```
import multiprocessing

def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()

def multiply_items(pipe_1, pipe_2):
    close, input_pipe = pipe_1
    close.close()
    output_pipe, _ = pipe_2
    try:
        while True:
            item = input_pipe.recv()
            output_pipe.send(item * item)
    except EOFError:
        output_pipe.close()
```

```
if __name__ == '__main__':

    #First process pipe with numbers from 0 to 9
    pipe_1 = multiprocessing.Pipe(True)
    process_pipe_1 = \
        multiprocessing.Process\
            (target=create_items, args=(pipe_1,))
    process_pipe_1.start()

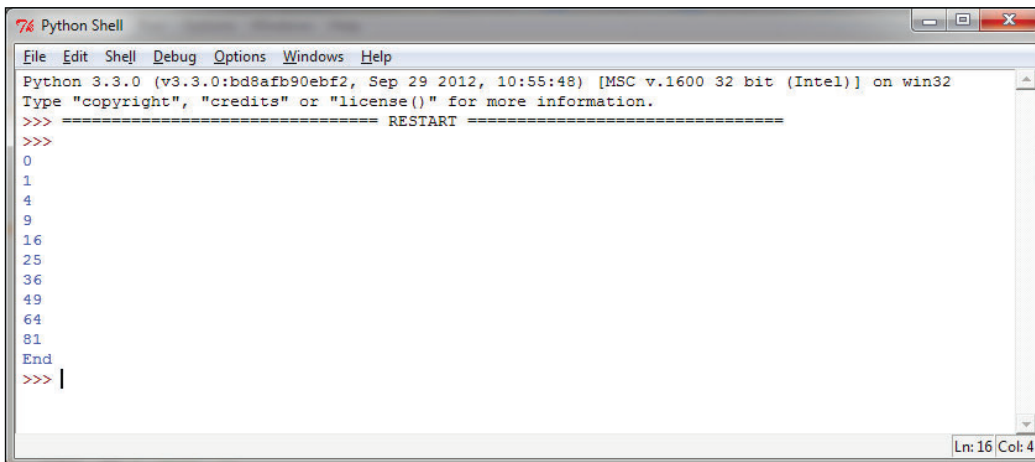
    #second pipe,
    pipe_2 = multiprocessing.Pipe(True)
    process_pipe_2 = \
        multiprocessing.Process\
            (target=multiply_items, args=(pipe_1, pipe_2,))
    process_pipe_2.start()

    pipe_1[0].close()
    pipe_2[0].close()

    try:
        while True:

            print (pipe_2[1].recv())
    except EOFError:
        print("End")
```

The output obtained is as follows:



```
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 =====
>>>
0
1
4
9
16
25
36
49
64
81
End
>>> |
```



## How it works...

Let's remember that the `pipe()` function returns a pair of connection objects connected by a two way pipe. In the example, `out_pipe` contains the numbers from 0 to 9, generated by the target function `create_items()`:

```
def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()
```

In the second process, we have two pipes: the input pipe and final output pipe that contains the results:

```
process_pipe_2 = multiprocessing.Process(target=multiply_items,
                                       args=(pipe_1, pipe_2,))
```

These are finally printed as:

```
try:
    while True:
        print (pipe_2[1].recv())
except EOFError:
    print ("End")
```

## How to synchronize processes

Multiple processes can work together to perform a given task. Usually, they share data. It is important that the access to shared data by various processes does not produce inconsistent data. Processes that cooperate by sharing data must therefore act in an orderly manner in order to access that data. Synchronization primitives are quite similar to those encountered for the library and threading.

They are as follows:

- ▶ **Lock:** This object can be in one of the states: locked and unlocked. A lock object has two methods, `acquire()` and `release()`, to manage the access to a shared resource.
- ▶ **Event:** This realizes simple communication between processes, one process signals an event and the other processes wait for it. An `Event` object has two methods, `set()` and `clear()`, to manage its own internal flag.
- ▶ **Condition:** This object is used to synchronize parts of a workflow, in sequential or parallel processes. It has two basic methods, `wait()` is used to wait for a condition and `notify_all()` is used to communicate the condition that was applied.

- ▶ **Semaphore:** This is used to share a common resource, for example, to support a fixed number of simultaneous connections.
- ▶ **RLock:** This defines the recursive `lock` object. The methods and functionality for `RLock` are the same as the `Threading` module.
- ▶ **Barrier:** This divides a program into phases as it requires all of the processes to reach it before any of them proceeds. Code that is executed after a barrier cannot be concurrent with the code executed before the barrier.

## How to do it...

The example here shows the use of `barrier()` to synchronize two processes. We have four processes, wherein `process1` and `process2` are managed by a barrier statement, while `process3` and `process4` have no synchronizations directives:

```
import multiprocessing
from multiprocessing import Barrier, Lock, Process
from time import time
from datetime import datetime

def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
    now = time()
    with serializer:
        print("process %s ----> %s" \
              %(name,datetime.fromtimestamp(now)))

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print("process %s ----> %s" \
          %(name ,datetime.fromtimestamp(now)))

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p3 - test_without_barrier'\
```

```
        ,target=test_without_barrier).start()  
Process(name='p4 - test_without_barrier'\  
        ,target=test_without_barrier).start()
```

By running the script, we can see that process1 and process2 print out the same timestamps:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes  
Chapter 3>python process_barrier.py
```

```
process p1 - test_with_barrier ----> 2015-05-09 11:11:33.291229  
process p2 - test_with_barrier ----> 2015-05-09 11:11:33.291229  
process p3 - test_without_barrier ----> 2015-05-09 11:11:33.310230  
process p4 - test_without_barrier ----> 2015-05-09 11:11:33.333231
```

## How it works...

In the main program, we created four processes; however, we also need a barrier and lock primitive. The parameter 2 in the barrier statement stands for the total number of process that are to be managed:

```
if __name__ == '__main__':  
    synchronizer = Barrier(2)  
    serializer = Lock()  
    Process(name='p1 - test_with_barrier'\  
            ,target=test_with_barrier,\  
            args=(synchronizer,serializer)).start()  
    Process(name='p2 - test_with_barrier'\  
            ,target=test_with_barrier,\  
            args=(synchronizer,serializer)).start()
```

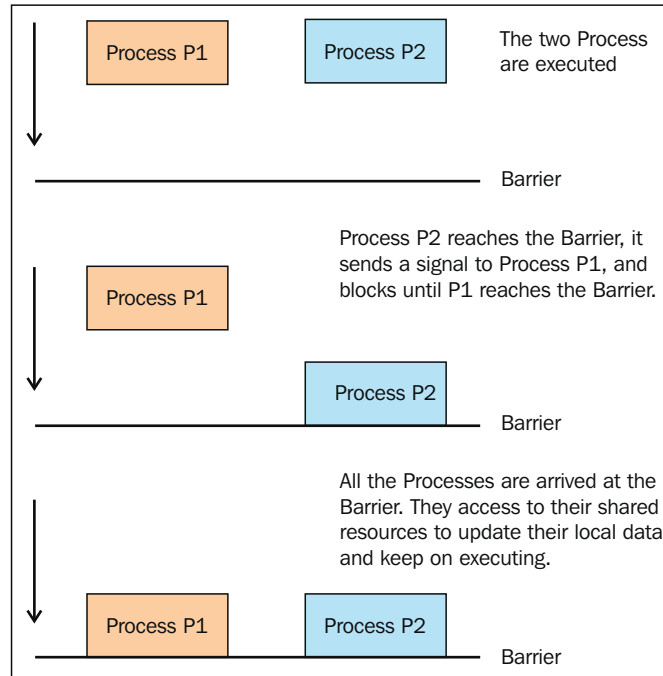
The `test_with_barrier_function` executes the barrier's `wait()` method:

```
def test_with_barrier(synchronizer, serializer):  
    name = multiprocessing.current_process().name  
    synchronizer.wait()
```

When the two processes have called the `wait()` method, they are released simultaneously:

```
now = time()  
with serializer:  
    print("process %s ----> %s" %(name \  
        ,datetime.fromtimestamp(now)))
```

The following figure shows you how a barrier works with the two processes:



Process management with a barrier

## How to manage a state between processes

Python multiprocessing provides a manager to coordinate shared information between all its users. A manager object controls a server process that holds Python objects and allows other processes to manipulate them.

A manager has the following properties:

- ▶ It controls the server process that manages a shared object
- ▶ It makes sure the shared object gets updated in all processes when anyone modifies it

## How to do it...

Let's see an example of how to share a state between processes:

1. First, the program creates a manager list, shares it between  $n$  number of `taskWorkers`, and every worker updates an index.
2. After all workers finish, the new list is printed to `stdout`:

```
import multiprocessing

def worker(dictionary, key, item):
    dictionary[key] = item

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    dictionary = mgr.dict()
    jobs = [ multiprocessing.Process\
              (target=worker, args=(dictionary, i, i*2))
            for i in range(10)
            ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print ('Results:', dictionary)
```

The output is as follows:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python manager.py
key = 0 value = 0
key = 2 value = 4
key = 6 value = 12
key = 4 value = 8
key = 8 value = 16
key = 7 value = 14
key = 3 value = 6
key = 1 value = 2
key = 5 value = 10
key = 9 value = 18
Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9:
18}
```

## How it works...

We declare the manager with the following statement:

```
mgr = multiprocessing.Manager()
```

In the next statement, a data structure of the type dictionary is created:

```
dictionary = mgr.dict()
```

Then, the multiprocess is launched:

```
jobs = [multiprocessing.Process \
        (target=taskWorker, args=(dictionary, i, i*2))
        for i in range(10)
        ]

for j in jobs:
    j.start()
```

Here, the target function `taskWorker` adds an item to the data structure dictionary:

```
def taskWorker(dictionary, key, item):
    dictionary[key] = value
```

Finally, we get the output and all the dictionaries are printed out:

```
for j in jobs:
    j.join()
    print ('Results:', d)
```

## How to use a process pool

The multiprocessing library provides the `Pool` class for simple parallel processing tasks. The `Pool` class has the following methods:

- ▶ `apply()`: It blocks until the result is ready.
- ▶ `apply_async()`: This is a variant of the `apply()` method, which returns a result object. It is an asynchronous operation that will not lock the main thread until all the child classes are executed.
- ▶ `map()`: This is the parallel equivalent of the `map()` built-in function. It blocks until the result is ready, this method chops the iterable data in a number of chunks that submits to the process pool as separate tasks.

- ▶ `map_async()`: This is a variant of the `map()` method, which returns a result object. If a callback is specified, then it should be callable, which accepts a single argument. When the result becomes ready, a callback is applied to it (unless the call failed). A callback should be completed immediately; otherwise, the thread that handles the results will get blocked.

## How to do it...

This example shows you how to implement a process pool to perform a parallel application. We create a pool of four processes and then we use the pool's `map` method to perform a simple calculation:

```
def function_square(data):
    result = data*data
    return result

if __name__ == '__main__':
    inputs = list(range(100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)
    pool.close()
    pool.join()
    print ('Pool      :', pool_outputs)
```

This is the result that we get after completing the calculation:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>\python process_pool.py
Pool      : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784,
841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704,
2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096,
4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776,
5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744,
7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
```