Getting ready

The `current.Futures` module provides two subclasses of the `Executor` class, respectively, which manipulates a pool of threads and a pool of processes asynchronously. The two subclasses are as follows:

- `concurrent.futures.ThreadPoolExecutor(max_workers)`
- `concurrent.futures.ProcessPoolExecutor(max_workers)`

The `max_workers` parameter identifies the max number of workers that execute the call asynchronously.

How to do it...

The following example shows you the functionality of process and thread pooling. The task to be performed is that we have a list of numbers from one to 10, `number_list`. For each element of the list, a count is made up to 10,000,000 (just to waste time) and then the latter number is multiplied with the *i*th element of the list.

By doing this, the following cases are evaluated:

- Sequential execution
- A thread pool with 5 workers

Consider the following code:

```
#
# Concurrent.Futures Pooling - Chapter 4 Asynchronous Programming
#

import concurrent.futures
import time

number_list = [1,2,3,4,5,6,7,8,9,10]

def evaluate_item(x):
    #count...just to make an operation
    result_item = count(x)
    #print the input item and the result
    print ("item " + str(x) + " result " + str(result_item))

def count(number) :
    for i in range(0,10000000):
        i=i+1
    return i*number
```

```
if __name__ == "__main__":

##Sequential Execution
    start_time = time.clock()
    for item in number_list:
        evaluate_item(item)
    print ("Sequential execution in " + \
           str(time.clock() - start_time), "seconds")

    ##Thread pool Execution
    start_time_1 = time.clock()
    with concurrent.futures.ThreadPoolExecutor(max_workers=5)\
         as executor:
        for item in number_list:
            executor.submit(evaluate_item, item)
    print ("Thread pool execution in " + \
           str(time.clock() - start_time_1), "seconds")

    ##Process pool Execution
    start_time_2 = time.clock()
    with concurrent.futures.ProcessPoolExecutor(max_workers=5)\
         as executor:
        for item in number_list:
            executor.submit(evaluate_item, item)
    print ("Process pool execution in " + \
           str(time.clock() - start_time_2), "seconds")
```

After running the code, we have the following results with the execution time:

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\ >python
Process_pool_with_concurrent_futures.py
item 1 result 10000000
item 2 result 20000000
item 3 result 30000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 8 result 80000000
item 9 result 90000000
item 10 result 100000000
Sequential execution in 17.241238674183425 seconds
```

```
item 4 result 40000000
item 2 result 20000000
item 1 result 10000000
item 5 result 50000000
item 3 result 30000000
item 7 result 70000000
item 6 result 60000000
item 8 result 80000000
item 10 result 100000000
item 9 result 90000000
Thread pool execution in 17.14648646290675 seconds

item 3 result 30000000
item 1 result 10000000
item 2 result 20000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 9 result 90000000
item 8 result 80000000
item 10 result 100000000
Process pool execution in 9.913172716938618 seconds
```

How it works...

We build a list of numbers stored in `number_list` and for each element in the list, we operate the counting procedure until 100,000,000 iterations. Then, we multiply the resulting value for 100,000,000:

```
def evaluate_item(x):
    #count...just to make an operation
    result_item = count(x)

def count(number) :
    for i in range(0,10000000):
        i=i+1
    return i*number
```

In the main program, we execute the task that will be performed in a sequential mode:

```
if __name__ == "__main__":
   for item in number_list:
       evaluate_item(item)
```

Also, in a parallel mode, we will use the `concurrent.futures` module's pooling capability for a thread pool:

```
with concurrent.futures.ThreadPoolExecutor(max_workers=5)\
     as executor:
    for item in number_list:
        executor.submit(evaluate_item, item)
```

The `ThreadPoolExecutor` executes the given task using one of its internally pooled threads. It manages five threads working on its pool. Each thread takes a job out from the pool and executes it. When the job is executed, it takes the next job to be processed from the thread pool.

When all the jobs are processed, the execution time is printed:

```
print ("Thread pool execution in " + \
           str(time.clock() - start_time_1), "seconds")
```

For the process pooling implemented by the `ProcessPoolExecutor` class, we have:

```
with concurrent.futures.ProcessPoolExecutor(max_workers=5)\
     as executor:
    for item in number_list:
        executor.submit(evaluate_item, item)
```

Like `ThreadPoolExecutor`, the `ProcessPoolExecutor` class is an executor subclass that uses a pool of processes to execute calls asynchronously. However, unlike `ThreadPoolExecutor`, the `ProcessPoolExecutor` uses the multiprocessing module, which allows us to outflank the global interpreter lock and obtain a shorter execution time.

There's more...

The pooling is used in almost all server applications, where there is a need to handle more simultaneous requests from any number of clients. Many other applications, however, require that each task should be performed instantly or you have more control over the thread that executes it. In this case, pooling is not the best choice.

Event loop management with Asyncio

The Python module Asyncio provides facilities to manage events, coroutines, tasks and threads, and synchronization primitives to write concurrent code. The main components and concepts of this module are:

- **An event loop**: The Asyncio module allows a single event loop per process
- **Coroutines**: This is the generalization of the subroutine concept. Also, a coroutine can be suspended during the execution so that it waits for external processing (some routine in I/O) and returns from the point at which it had stopped when the external processing was done.
- **Futures**: This defines the `Future` object, such as the `concurrent.futures` module that represents a computation that has still not been accomplished.
- **Tasks**: This is a subclass of Asyncio that is used to encapsulate and manage coroutines in a parallel mode.

In this recipe, the focus is on handling events. In fact, in the context of asynchronous programming, events are very important since they are inherently asynchronous.

What is an event loop

Within a computational system, the entity that can generate events is called an event source, while the entity that negotiates to manage an event is called the event handler. Sometimes, there may be a third entity called an event loop. It realizes the functionality to manage all the events in a computational code. More precisely, the event loop acts cyclically during the whole execution of the program and keeps track of events that have occurred within a data structure to queue and then process them one at a time by invoking the event handler if the main thread is free. Finally, we report a pseudocode of an event loop manager:

```
while (1) {
events = getEvents();
for (e in events)
processEvent(e);
}
```

All the events in the `while` loop are caught and then processed by the event handler. The handler that processes an event is the only activity that takes place in the system. When the handler has ended, the control is passed on to the next event that is scheduled.

Getting ready

Asyncio provides the following methods that are used to manage an event loop:

- `loop = get_event_loop()`: Using this, you can get the event loop for the current context.
- `loop.call_later(time_delay,callback,argument)`: This arranges for the callback that is to be called after the given `time_delay` seconds.
- `loop.call_soon(callback,argument)`: This arranges for a callback that is to be called as soon as possible. The callback is called after `call_soon()` returns and when the control returns to the event loop.
- `loop.time()`: This returns the current time, as a float value, according to the event loop's internal clock.
- `asyncio.set_event_loop()`: This sets the event loop for the current context to loop.
- `asyncio.new_event_loop()`: This creates and returns a new event loop object according to this policy's rules.
- `loop.run_forever()`: This runs until `stop()` is called.

How to do it...

In this example, we show you how to use the loop event statements provided by the Asyncio library to build an application that works in an asynchronous mode. Let's consider the following code:

```
import asyncio
import datetime
import time

def function_1(end_time, loop):
    print ("function_1 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()
```

```
def function_2(end_time, loop):
    print ("function_2 called ")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_3, end_time, loop)
    else:
        loop.stop()

def function_3(end_time, loop):
    print ("function_3 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_1, end_time, loop)
    else:
        loop.stop()

def function_4(end_time, loop):
    print ("function_5 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_4, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

end_loop = loop.time() + 9.0
loop.call_soon(function_1, end_loop, loop)
#loop.call_soon(function_4, end_loop, loop)

loop.run_forever()
loop.close()
```

The output of the preceding code is as follows:

```
C:\Python Parallel Programming INDEX\Chapter 4- Asynchronous
Programming >python asyncio_loop.py
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
```