Chapter 34
Concurrency with AsyncIO

34.1 Introduction

The Async IO facilities in Python are relatively recent additions originally introduced in Python 3.4 and evolving up to and including Python 3.7. They are comprised (as of Python 3.7) of two new keywords `async` and `await` (introduced in Python 3.7) and the Async IO Python package.

In this chapter we first discuss Asynchronous IO before introducing the `async` and `await` keywords. We then present Async IO Tasks, how they are created used and managed.

34.2 Asynchronous IO

Asynchronous IO (or Async IO) is a language agnostic concurrent programming model (or paradigm) that has been implemented in several different programming language (such as C# and Scala) as well as in Python.

Asynchronous IO is another way in which you can build concurrent applications in Python. It is in many ways an alternative to the facilities provided by the Threading library in Python. However, were as the Threading library is more susceptible to issues associated with the GIL (The Global Interpreter Lock) which can affect performance, the Async IO facilities are better insulated from this issue.

The way in which Async IO operates is also lighter weight then the facilities provide day the `multiprocessing` library since the asynchronous tasks in Async IO run within a single process rather than requiring separate processes to be spawned on the underlying hardware.

Async IO is therefore another alternative way of implementing concurrent solutions to problems. It should be noted that it does not build on either Threading or Multi Processing; instead Async IO is based on the idea of cooperative multitasking. These cooperating tasks operate asynchronously; by this we mean that the tasks:

- are able to operate separately from other tasks,
- are able to wait for another task to return a result when required,
- and are thus able to allow other tasks to run while they are waiting.

The IO (Input/Output) aspect of the name Async IO is because this form of concurrent program is best suited to I/O bound tasks.

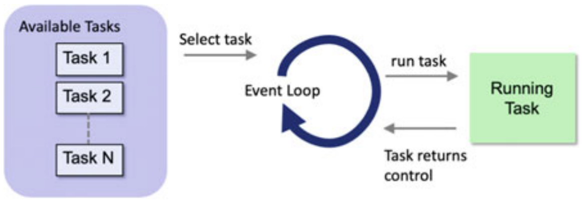
In an I/O bound task a program spends most of its time sending data to, or reading data from, some form of external device (for example a database or set of files etc.). This communication is time consuming and means that the program spends most of its time waiting for a response from the external device.

One way in which such I/O bound applications can (appear to) speed up is to overlap the execution of different tasks; thus, while one task is waiting for a database to respond with some data, another task can be writing data to a log file etc.

34.3 Async IO Event Loop

When you are developing code using the Async IO facilities you do not need to worry about how the internals of the Async IO library work; however at least at the conceptual level it is useful to understand one key concept; that of the Async IO *Event Loop*; This loop controls how and when each task gets run. For the purposes of this discussion a task represents some work that can be run independently of other pieces of work.

The Event Loop knows about each task to be run and what the state of the task currently is (for example whether it is waiting for something to happen/complete). It selects a task that is *ready to run* from the list of available tasks and executes it. This task has complete control of the CPU until it either completes its work or hands back control to the Event Loop (for example, because it must now wait for some data to be supplied from a database). The Event Loop now checks to see if any of the waiting tasks are ready to continue executing and makes a note of their status. The Event Loop then selects another task that is ready to run and starts that task off. This loop continues until all the tasks have finished. This is illustrated below:



An important point to note in the above description is that a task does not give up the processor unless it decides to, for example by having to wait for something else. They never get interrupted in the middle of an operation; this avoids the problem that two threads might have when being time sliced by a separate scheduler as they may both be sharing the same resource. This can greatly simplify your code.

34.4 The Async and Await Keywords

The `async` keyword, introduced in Python 3.7 is used to mark a function as being something that uses the `await` keyword (we will come back to this below as there is one other use of the `async` keyword). A function that uses the `await` keyword can be run as a separate task and can give up control of the processor when it calls `await` against another `async` function and must *wait* for that function to complete. The invoked `async` function can then run as a separate task etc.

To invoke an `async` function it is necessary to start the Async IO Event Loop and for that function to be treated as a task by the Event Loop. This is done by calling the `asyncio.run()` method and passing in the root async function.

The `asyncio.run()` function was introduced in Python 3.7 (older versions of Python such as Python 3.6 required you to explicitly obtain a reference to the Event Loop and to run the root async function via that). One point to note about this function is that it has been marked as being provisional in Python 3.7. This means that future versions of Python may or may not support the function or may modify the function in some way. You should therefore check the documentation for the version of Python you are using to see whether the run method has been altered or not.

34.4.1 Using Async and Await

We will examine a very simple Async IO program from the top down. The `main()` function for the program is given below:

```
def main() :
    print('Main - Starting')
    asyncio.run(do_something())
    print('Main - Done')

if __name__ == '__main__':
    main()
```

The `main()` function is the entry point for the program and calls:

```
asyncio.run(do_something())
```

This starts the Async IO Event Loop running and results in the `do_something()` function being wrapped up in a `Task` that is managed by the loop. Note that you do not explicitly create a `Task` in Async IO; they are always created by some function however it is useful to be aware of Tasks as you can interact with them to check their status or to retrieve a result.

The `do_something()` function is marked with the keyword `async`:

```
async def do_something():
    print('do_something - will wait for worker')
    result = await worker()
    print('do_something - result:', result)
```

As previously mentioned this indicates that it can be run as a separate Task and that it can use the keyword `await` to wait for some other function or behaviour to complete. In this case the `do_something()` asynchronous function must wait for the `worker()` function to complete.

The `await` keyword does more than merely indicate that the `do_something ()` function must wait for the worker to complete. It triggers another `Task` to be created that will execute the `worker()` function and releases the processor allowing the Event Loop to select the next task to execute (which may or may not be the task running the `worker()` function). The status of the `do_something` task is now *waiting* while the status of the `worker()` task is *ready* (to run).

The code for the worker task is given below:

```
async def worker():
    print('worker - will take some time')
    time.sleep(3)
    print('worker - Done it')
    return 42
```

The `async` keyword again indicates that this function can be run as a separate task. However, this time the body of the function does not use the `await` keyword. This is because this is a special case known as an Async IO *coroutine* function. This is a function that returns a value from a `Task` (it is related to the idea of a standard Python coroutine which is a data consumer).

Sadly, Computer Science has many examples where the same term has been used for different things as well as examples where different terms have been used for the same thing. In this case to avoid confusion just stick with Async IO coroutines are functions marked with async that can be run as a separate task and *may* call await.

The full listing for the program is given below:

```
import asyncio
import time

async def worker():
    print('worker - will take some time')
    time.sleep(3)
    print('worker - done it')
    return 42

async def do_something():
    print('do_something - will wait for worker')
    result = await worker()
    print('do_something - result:', result)

def main():
    print('Main - Starting')
    asyncio.run(do_something())
    print('Main - Done')
if __name__ == '__main__':
    main()
```

When this program is executed the output is:

```
Main - Starting
do_something - will wait for worker
worker - will take some time
worker - done it
do_something - result: 42
Main - Done
```

When this is run there is a pause between the two worker printouts as it sleeps.

Although it is not completely obvious here, the `do_something()` function was run as one task, this task then waited when it got to the `worker()` function which was run as another Task. Once the worker task completed the `do_something` task could continue and complete its operation. Once this happened the Async IO Event Loop could then terminate as no further tasks were available.

34.5 Async IO Tasks

Tasks are used to execute functions marked with the `async` keyword concurrently. Tasks are never created directly instead they are created implicitly via the keyword `await` or through functions such as `asyncio.run` described above or asyncio.create_task(), asyncio.gather() and asyncio.as_-completed(). These additional task creation functions are described below:

- asyncio.create_task() This function takes a function marked with async and wraps it inside a Task and schedules it for execution by the Async IO Event Loop. This function was added in Python 3.7.
- asyncio.gather(*aws) This function runs all the async functions passed to it as separate Tasks. It gathers the results of each separate task together and returns them as a list. The order of the results corresponds to the order of the async functions in the aws list.
- asyncio.as_completed(aws) Runs each of the async functions passed to it.

A Task object supports several useful methods

- cancel() cancels a running task. Calling this method will cause the Task to throw a CancelledError exception.
- cancelled() returns True if the Task has been cancelled.
- done() returns True if the task has completed, raised an exception or was cancelled.
- result() returns the result of the Task if it is done. If the Tasks result is not yet available, then the method raises the InvalidStateError exception.
- exception() return an exception if one was raised by the Task. If the task was cancelled then raises the CancelledError exception. If the task is not yet *done*, then raises an InvalidStateError exception.

It is also possible to add a callback function to invoke once the task has completed (or to remove such a function if it has been added):

- add_done_callback(callback) Add a callback to be run when the Task is done.
- remove_done_callback(callback) Remove callback from the callbacks list.

Note that the method is called ‘add’ rather than ‘set’ implying that there can be multiple functions called when the task has completed (if required).

The following example illustrates some of the above:

```
import asyncio
async def worker():
    print('worker - will take some time')
    await asyncio.sleep(1)
    print('worker - Done it')
    return 42

def print_it(task):
    print('print_it result:', task.result())
```

```
async def do_something():
    print('do_something - create task for worker')
    task = asyncio.create_task(worker())
    print('do_something - add a callback')
    task.add_done_callback(print_it)
    await task
    # Information on task
    print('do_something - task.cancelled():',
task.cancelled())
    print('do_something - task.done():', task.done())
    print('do_something - task.result():', task.result())
    print('do_something - task.exception():',
task.exception())
    print('do_something - finished')

def main() :
    print('Main - Starting')
    asyncio.run(do_something())
    print('Main - Done')

if __name__ == '__main__':
    main()
```

In this example, the `worker()` function is wrapped within a task object that is returned from the `asyncio.create_task(worker())` call.

A function (`print_it()`) is registered as a callback on the task using the `asyncio.create_task(worker())` function. Note that the worker is passed the task that has completed as a parameter. This allows it to obtain information from the task such as any result generated.

In this example the async function `do_something()` explicitly waits on the task to complete. Once this happens several different methods are used to obtain information about the task (such as whether it was cancelled or not).

One other point to note about this listing is that in the `worker()` function we have added an `await` using the `asyncio.sleep(1)` function; this allows the worker to sleep and wait for the triggered task to complete; it is an Async IO alternative to `time.sleep(1).`

The output from this program is:

```
Main - Starting
do_something - create task for worker
do_something - add a callback
worker - will take some time
worker - Done it
print_it result: 42
do_something - task.cancelled(): False
do_something - task.done(): True
do_something - task.result(): 42
do_something - task.exception(): None
do_something - finished
Main - Done
```

34.6 Running Multiple Tasks

In many cases it is useful to be able to run several tasks concurrently. There are two options provided for this the `asyncio.gather()` and the `asyncio.as_completed()` function; we will look at both in this section.

34.6.1 Collating Results from Multiple Tasks

It is often useful to collect all the results from a set of tasks together and to continue only once all the results have been obtained. When using Threads or Processes this can be achieved by starting multiple Threads or Processes and then using some other object such as a Barrier to wait for all the results to be available before continuing. Within the Async IO library all that is required is to use the `asyncio.gather()` function with a list of the async functions to run, for example:

```
import asyncio
import random

async def worker():
    print('Worker - will take some time')
    await asyncio.sleep(1)
    result = random.randint(1,10)
    print('Worker - Done it')
    return result

async def do_something():
    print('do_something - will wait for worker')
    # Run three calls to worker concurrently and collect
results
    results = await asyncio.gather(worker(), worker(),
worker())
    print('results from calls:', results)

def main() :
    print('Main - Starting')
    asyncio.run(do_something())
    print('Main - Done')

if __name__ == '__main__':
    main()
```

In this program the `do_something()` function uses

```
results = await asyncio.gather(worker(), worker(), worker())
```

to run three invocations of the `worker()` function in three separate Tasks and to wait for the results of all three to be made available before they are returned as a list of values and stored in the `results` variable.

This makes is very easy to work with multiple concurrent tasks and to collate their results.

Note that in this code example the worker `async` function returns a random number between 1 and 10.

The output from this program is:

```
 Main - Starting
do_something - will wait for worker
Worker - will take some time
Worker - will take some time
Worker - will take some time
Worker - Done it
Worker - Done it
Worker - Done it
results from calls: [5, 3, 4]
Main – Done
```

As you can see from this all three of the worker invocations are started but then release the processor while they sleep. After this the three tasks wake up and complete before the results are collected together and printed out.

34.6.2 Handling Task Results as They Are Made Available

Another option when running multiple Tasks is to handle the results as they become available, rather than wait for all the results to be provided before continuing. This option is supported by the `asyncio.as_completed()` function. This function returns an iterator of async functions which will be served up as soon as they have completed their work.

The `for-loop` construct can be used with the iterator returned by the function; however within the for loop the code must call await on the async functions returned so that the result of the task can be obtained. For example:

```
async def do_something():
    print('do_something - will wait for worker')
    # Run three calls to worker concurrently and collect 
results
    for async_func in asyncio.as_completed((worker('A'),
                                           worker('B'),
                                           worker('C'))):
        result = await async_func
        print('do_something - result:', result)
```

Note that the `asyncio.as_completed()` function takes a container such as a tuple of async functions.

We have also modified the worker function slightly so that a label is added to the random number generated so that it is clear which invocation of the worker function return which result:

```
async def worker(label):
    print('Worker - will take some time')
    await asyncio.sleep(1)
    result = random.randint(1,10)
    print('Worker - Done it')
    return label + str(result)
```

When we run this program

```
def main() :
    print('Main - Starting')
    asyncio.run(do_something())
    print('Main - Done')
```

The output is

```
Main - Starting
do_something - will wait for worker
Worker - will take some time
Worker - will take some time
Worker - will take some time
Worker - Done it
Worker - Done it
Worker - Done it
do_something - result: C2
do_something - result: A1
do_something - result: B10
Main - Done
```

As you can see from this, the results are not returned in the order that the tasks are created, task 'C' completes first followed by 'A' and 'B'. This illustrates the behaviour of the `asyncio.as_completed()` function.

34.7 Online Resources

See the following online resources for information on Futures:

- https://docs.python.org/3/library/asyncio-task.html The Python standard Library documentation on AsyncIO.
- https://pymotw.com/3/asyncio The Python Module of the Week page on AsyncIO.
- https://pythonprogramming.net/asyncio-basics-intermediate-python-tutorial An AsyncIO tutorial.

34.8 Exercises

This exercise will use the facilities in the AsyncIO library to calculate a set of factorial numbers.

The factorial of a positive integer is the product of all positive integers less than or equal to n. For example,

```
5! = 5 x 4 x 3 x 2 x 1 = 120
```

Note that the value of 0! is 1,

Create an application that will use the `async` and `await` keywords to calculate the factorials of a set of numbers. The factorial function should await for 0.1 of a second (using `asyncio.sleep(0.1)`) each time round the loop used to calculate the factorial of a number.

You can use with `asyncio.as_completed()` or `asyncio.gather()` to collect the results up.

You might also use a list comprehension to create the list of calls to the factorial function.

The main function might look like:

```
def main():
    print('Main - Starting')
    asyncio.run(calculate_factorials([5, 7, 3, 6]))
    print('Main - Done')

if __name__ == '__main__':
    main()
```