Chapter 31
Multiprocessing

31.1 Introduction

The `multiprocessing` library supports the generation of separate (operating system level) processes to execute behaviour (such as functions or methods) using an API that is similar to the Threading API presented in the last chapter.

It can be used to avoid the limitation introduced by the Global Interpreter Lock (the GIL) by using separate operating system processes rather than lightweight threads (which run within a single process).

This means that the `multiprocessing` library allows developers to fully exploit the multiple processor environment of modern computer hardware which typically has multiple processor cores allowing multiple operations/behaviours to run in parallel; this can be very significant for data analytics, image processing, animation and games applications.

The multiprocessing library also introduces some new features, most notably the `Pool` object for parallelising execution of a callable object (e.g. functions and methods) that has no equivalent within the Threading API.

31.2 The Process Class

The `Process` class is the `multiprocessing` library's equivalent to the `Thread` class in the `threading` library. It can be used to run a callable object such as a function in a separate process. To do this it is necessary to create a new instance of the `Process` class and then call the `start()` method on it. Methods such as `join()` are also available so that one process can wait for another process to complete before continuing etc.

The main difference is that when a new `Process` is created it runs within a separate process on the underlying operating systems (such as Window, Linux or

© Springer Nature Switzerland AG 2019
J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_31

Mac OS). In contrast a `Thread` runs within the same process as the original program. This means that the process is managed and executed directly by the operating system on one of the processors that are part of the underlying computer hardware.

The up side of this is that you are able to exploit the underlying parallelism inherent in the physical computer hardware. The downside is that a `Process` takes more work to set up than the lighter weight Threads.

The constructor for the `Process` class provides the same set of arguments as the `Thread` class, namely:

```
class multiprocessing.Process(group=None,
                              target=None,
                              name=None,
                              args=(),
                              kwargs={},
                              daemon=None)
```

- `group` should always be `None`; it exists solely for compatibility with the Threading API.
- `target` is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called.
- `name` is the process name.
- `args` is the argument tuple for the target invocation.
- `kwargs` is a dictionary of keyword arguments for the target invocation.
- `daemon` argument sets the process daemon flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

As with the `Thread` class, the `Process` constructor should *always* be called using *keyword* arguments.

The `Process` class also provides a similar set of methods to the `Thread` class

- `start()` Start the process's activity. This must be called at *most once* per process object. It arranges for the object's `run()` method to be invoked in a separate process.
- `join([timeout])` If the optional argument timeout is None (the default), the method blocks until the joined process terminates. If timeout is a positive number, it blocks at most timeout seconds. Note that the method returns `None` if its process terminates or if the method times out.

- `is_alive()` Return whether the process is alive. Roughly, a process object is alive from the moment the `start()` method returns until the child process terminates.

The process class also has several attributes:

- `name` The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name. It can be useful for debugging purposes.
- `daemon` The process's daemon flag, a boolean value. This must be set before `start()` is called. The default value is inherited from the creating process. When a process exits, it attempts to terminate all of its daemonic child processes. Note that a daemonic process is not allowed to create child processes.
- `pid` Return the process ID. Before the process is spawned, this will be `None`.
- `exitcode` The process exit code. This will be `None` if the process has not yet terminated. A negative value -N indicates that the child was terminated by signal N.

In addition to these methods and attributes, the `Process` class also defines additional process related methods including:

- `terminate()` Terminate the process.
- `kill()` Same as `terminate()` except that on Unix the SIGKILL signal is used instead of the SIGTERM signal.
- `close()` Close the `Process` object, releasing all resources associated with it. `ValueError` is raised if the underlying process is still running. Once `close()` returns successfully, most of the other methods and attributes of the `Process` object will raise a `ValueError`.

31.3 Working with the Process Class

The following simple program creates three `Process` objects; each runs the function `worker()`, with the string arguments A, B and C respectively. These three process objects are then started using the `start()` method.

```
from multiprocessing import Process
from time import sleep

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

print('Starting')

t2 = Process(target=worker, args='A')
t3 = Process(target=worker, args='B')
t4 = Process(target=worker, args='C')

t2.start()
t3.start()
t4.start()

print('Done')
```

It is essentially the same as the equivalent program for threads but with the `Process` class being used instead of the `Thread` class.

The output from this application is given below:

```
Starting
Done
ABCABCABCABCABCABCABCACBACBACB
```

The main difference between the `Thread` and `Process` versions is that the `Process` version runs the `worker` function in separate processes whereas in the `Thread` version all the Threads share the same process.

31.4 Alternative Ways to Start a Process

When the `start()` method is called on a `Process`, three different approaches to starting the underlying process are available. These approaches can be set using the `multiprocessing.set_start_method()` which takes a string indicating the approach to use. The actual process initiation mechanisms available depend on the underlying operating system:

- '`spawn`' The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process objects `run()` method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using fork or forkserver. Available on Unix and Windows. This is the default on Windows.

- '`fork`' The parent process uses `os.fork()` to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Available only on Unix type operating systems. This is the default on Unix, Linux and Mac OS.
- '`forkserver`' In this case a server process is started. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded so it is safe for it to use `os.fork()`. No unnecessary resources are inherited. Available on Unix style platforms which support passing file descriptors over Unix pipes.

The `set_start_method()` should be used to set the start method (and this should only be set once within a program).

This is illustrated below, where the *spawn* start method is specified:

```
from multiprocessing import Process
from multiprocessing import set_start_method
from time import sleep
import os

def worker(msg):
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

def main():
    print('Starting')
    print('Root application process id:', os.getpid())
    set_start_method('spawn')
    t = Process(target=worker, args='A')
    t.start()

    print('Done')

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

The output from this is shown below:

```
Starting
Root application process id: 6281
Done
```

```
module name: __main__
parent process: 6281
process id: 6283
AAAAAAAAAA
```

Note that the parent process and current process ids are printed out for the `worker()` function, while the `main()` method prints out only its own id. This shows that the *main* application process id is the same as the worker process parents' id.

Alternatively, it is possible to use the `get_context()` method to obtain a context object. Context objects have the same API as the `multiprocessing` module and allow you to use multiple start methods in the same program, for example:

```
ctx = multiprocessing.get_context('spawn')
q = ctx.Queue()
p = ctx.Process(target = foo, args = (q,))
```

31.5 Using a Pool

Creating Processes is expensive in terms of computer resources. It would therefore be useful to be able to reuse processes within an application. The `Pool` class provides such reusable processes.

The `Pool` class represents a pool of worker processes that can be used to perform a set of concurrent, parallel operations. The `Pool` provides methods which allow tasks to be offloaded to these worker processes.

The `Pool` class provides a constructor which takes a number of arguments:

```
class multiprocessing.pool.Pool(processes,
                                initializer, initargs,
                                maxtasksperchild,
                                context)
```

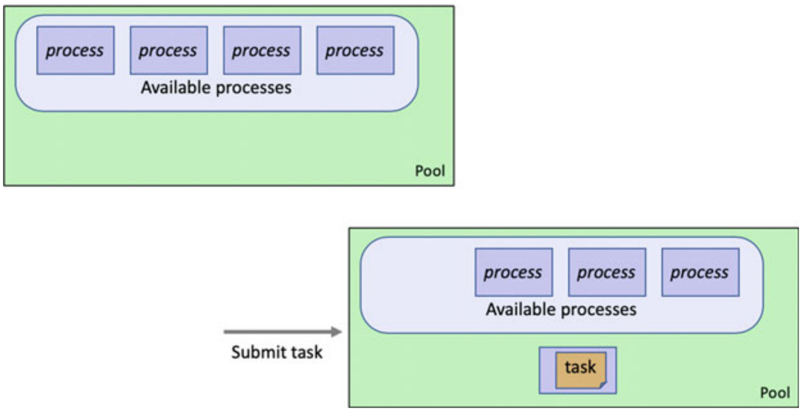
These represent:

- `processes` is the number of worker processes to use. If processes is `None` then the number returned by `os.cpu_count()` is used.
- `initializer` If `initializer` is not `None` then each worker process will call `initializer(*initargs)` when it starts.
- `maxtasksperchild` is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default `maxtasksperchild` is `None`, which means worker processes will live as long as the pool.

- `context` can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()`. Alternatively the pool can be created using the `Pool()` method of a context object.

The `Pool` class provides a range of methods that can be used to submit work to the worker processes managed by the pool. Note that the methods of the `Pool` object should only be called by the process which created the pool.

The following diagram illustrates the effect of submitting some work or task to the pool. From the list of available processes, one process is selected and the task is passed to the process. The process will then execute the task. On completion any results are returned and the process is returned to the available list. If when a task is submitted to the pool there are no available processes then the task will be added to a wait queue until such time as a process is available to handle the task.



The simplest of the methods provided by the `Pool` for work submission is the `map` method:

```
pool.map(func, iterable, chunksize=None)
```

This method returns a list of the results obtained by executing the function in parallel against each of the items in the `iterable` parameter.

- The `func` parameter is the callable object to be executed (such as a function or a method).
- The `iteratable` is used to pass in any parameters to the function.
- This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting `chunksize` to a positive integer. The method blocks until the result is ready.

The following sample program illustrates the basic use of the `Pool` and the `map()` method.

```
    from multiprocessing import Pool

    def worker(x):
        print('In worker with: ', x)
        return x * x

  def main():
      with Pool(processes=4) as pool:
          print(pool.map(worker, [0, 1, 2, 3, 4, 5]))

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

Note that the `Pool` object must be closed once you have finished with it; we are therefore using the '`with as`' statement described earlier in this book to handle the `Pool` resource cleanly (it will ensure the `Pool` is closed when the block of code within the `with as` statement is completed).

The output from this program is

```
In worker with: 0
In worker with: 1
In worker with: 2
In worker with: 3
In worker with: 4
In worker with: 5
[0, 1, 4, 9, 16, 25]
```

As can be seen from this output the `map()` function is used to run six different instances of the `worker()` function with the values provided by the list of integers. Each instance is executed by a *worker* process managed by the `Pool`.

However, note that the `Pool` only has 4 worker processes, this means that the last two instances of the worker function must wait until two of the worker Processes have finished the work they are doing and can be reused. This can act as a way of throttling, or controlling, how much work is done in parallel.

A variant on the `map()` method is the `imap_unordered()` method. This method also applies a given function to an iterable but does not attempt to maintain the order of the results. The results are accessible via the iterable returned by the function. This may improve the performance of the resulting program.

The following program modified the `worker()` function to return its result rather than print it. These results are then accessible by iterating over them as they are produced via a `for` loop:

```
from multiprocessing import Pool

def worker(x):
    print('In worker with: ', x)
    return x * x

def main():
    with Pool(processes=4) as pool:
        for result in pool.imap_unordered(worker,
                                          [0, 1, 2, 3, 4, 5]):
            print(result)

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

As the new method obtains results as soon as they are available, the order in which the results are returned may be different, as shown below:

```
In worker with: 0
In worker with: 1
In worker with: 3
In worker with: 2
In worker with: 4
In worker with: 5
0
1
9
16
4
25
```

A further method available on the `Pool` class is the `Pool.apply_async()` method. This method allows operations/functions to be executed asynchronously allowing the method calls to return immediately. That is as soon as the method call is made, control is returned to the calling code which can continue immediately. Any results to be collected from the asynchronous operations can be obtained either by providing a callback function or by using the blocking `get()` method to obtain a result.

Two examples are shown below, the first uses the blocking `get()` method. This method will wait until a result is available before continuing. The second approach uses a *callback* function. The callback function is called when a result is available; the result is passed into the function.

```
from multiprocessing import Pool

def collect_results(result):
    print('In collect_results: ', result)

def worker(x):
    print('In worker with: ', x)
    return x * x

def main():
    with Pool(processes=2) as pool:
        # get based example
        res = pool.apply_async(worker, [6])
        print('Result from async: ', res.get(timeout=1))

    with Pool(processes=2) as pool:
        # callback based example
        pool.apply_async(worker, args=[4],
                         callback=collect_results)

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

The output from this is:

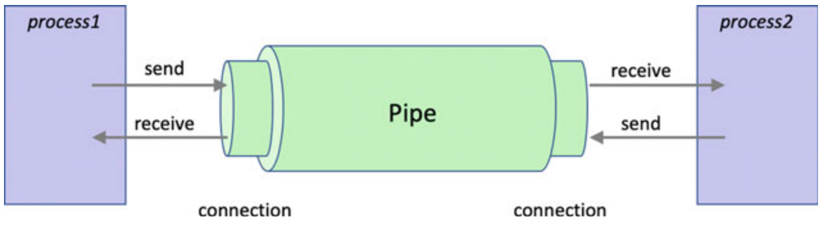
```
In worker with: 6
Result from async: 36
In worker with: 4
In collect_results: 16
```

31.6 Exchanging Data Between Processes

In some situations it is necessary for two processes to exchange data. However, the two process objects do not share memory as they are running in separate operating system level *processes*. To get around this the multiprocessing library provides the `Pipe()` function.

The `Pipe()` function returns a pair of `connection.Connection` objects connected by a pipe which by default is duplex (two-way).

The two connection objects returned by `Pipe()` represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). This allows one process to send data via the `send()` method of one end of the connection object. In turn a second process can receive that data via the `receive()` method of the other connection object. This is illustrated below:



Once a program has finished with a connection is should be closed using `close ()`.

The following program illustrates how pipe connections are used:

```
from multiprocessing import Process, Pipe
from time import sleep

def worker(conn):
    print('Worker - started now sleeping for 1 second')
    sleep(1)
    print('Worker - sending data via Pipe')
    conn.send('hello')
    print('Worker - closing worker end of connection')
    conn.close()

def main():
    print('Main - Starting, creating the Pipe')
    main_connection, worker_connection = Pipe()
    print('Main - Setting up the process')
    p = Process(target=worker, args=[worker_connection])
    print('Main - Starting the process')
    p.start()
    print('Main - Wait for a response from the child
process')
    print(main_connection.recv())
    print('Main - closing parent process end of connection')
    parent_connection.close()
    print('Main - Done')

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

The output from this `Pipe` example is:

```
Main - Starting, creating the Pipe
Main - Setting up the process
Main - Starting the process
Main - Wait for a response from the child process
Worker - started now sleeping for 1 second
Worker - sending data via Pipe
Worker - closing worker end of connection
hello
```

```
Main - closing parent process end of connection
Main - Done
```

Note that data in a pipe may become corrupted if two processes try to read from or write to the same end of the pipe at the same time. However, there is no risk of corruption from processes using different ends of the pipe at the same time.

31.7 Sharing State Between Processes

In general, if it can be avoided, then you should not share state between separate processes. However, if it is unavoidable then the `mutiprocessing` library provides two ways in which state (data) can be shared, these are Shared Memory (as supported by `multiprocessing.Value` and `multiprocessing.Array`) and Server Process.

31.7.1 Process Shared Memory

Data can be stored in a shared memory map using a `multiprocessing.Value` or `multiprocessing.Array`. This data can be accessed by multiple processes.

The constructor for the `multiprocessing.Value` type is:

```
  multiprocessing.Value
(typecode_or_type, *args, lock = True)
```

Where:

- `typecode_or_type` determines the type of the returned object: it is either a ctypes type or a one character typecode. For example, 'd' indicates a double precision float and 'i' indicates a signed integer.
- `*args` is passed on to the constructor for the type.
- `lock` If lock is `True` (the default) then a new recursive lock object is created to synchronise access to the value. If lock is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be *process-safe*.

The constructor for `multiprocessing.Array` is

```
        multiprocessing.Array
 multiprocessing.Array(typecode_or_type,
                       size_or_initializer,
                       lock=True)
```

Where:

- `typecode_or_type` determines the type of the elements of the returned array.
- `size_or_initializer` If size_or_initializer is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, size_or_initializer is a sequence which is used to initialise the array and whose length determines the length of the array.
- If `lock` is `True` (the default) then a new lock object is created to synchronise access to the value. If lock is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

An example using both the `Value` and `Array` type is given below:

```
from multiprocessing import Process, Value, Array

def worker(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

def main():
    print('Starting')
    num = Value('d', 0.0)
    arr = Array('i', range(10))
    p = Process(target=worker, args=(num, arr))
    p.start()
    p.join()
    print(num.value)
    print(*arr)
    print('Done')

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

31.8 Online Resources

See the following online resources for information on multiprocessing:

- https://docs.python.org/3/library/multiprocessing.html The Python standard Library documentation on MultiProcessing.
- https://pymotw.com/3/multiprocessing The Python Module of the Week page on MultiProcessing.
- https://pythonprogramming.net/multiprocessing-python-intermediate-python-tutorial Tutorial on Python's MultiProcessing module.

31.9 Exercises

Write a program that can find the factorial of any given number. For example, find the factorial of the number 5 (often written as 5!) which is 1 * 2 * 3 * 4 * 5 and equals 120.

The factorial is not defined for negative numbers and the factorial of Zero is 1; that is 0! = 1.

Next modify the program to run multiple factorial calculations in parallel.

Collect all the results together in a list and print that list out.

You an use whichever approach you like to running multiple processes although a `Pool` could be a good approach to use.

Your program should compute the factorials of 5, 8, 10, 15, 3, 6, and 4 in parallel.