Asynchronous Programming

And

Reactive Design Pattern

# Content:

1. The Beginning: We will cover the general design pattern that we generally use and how thread can be used to run concurrent task in parallel
2. Paradigm Shift: Leap from synchronous thread based to and asynchronous callback style programming
3. Python library: Twisted and usage of deferred
4. The Conversion technique: Conversion of synchronous library into asynchronous library

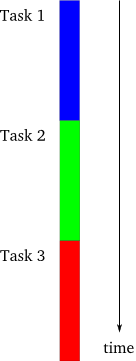
# The Beginning:

The Models:

We will start by reviewing two familiar models in order to contrast them with the asynchronous model. By way of illustration we will imagine a program that consists of three conceptually distinct tasks which must be performed to complete the program. We will make these tasks more concrete later on, but for now we won’t say anything about them except the program must perform them

The Synchronous Model:

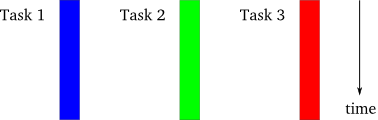
The first model is single-threaded synchronous model where each task execute sequentially one at a time. And if the tasks are always performed in a definite order, the implementation of a later task can assume that all earlier tasks have finished without errors.



The Threaded Model:

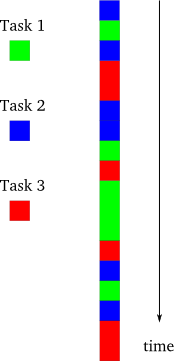
In this model each task is performed in separate thread of control. The threads are managed by the operating system and may, on a system with multiple processors or multiple cores, run truly concurrently, or may be interleaved together on a single processor. In the threaded model the details of execution are handled by the OS and the programmer simply thinks in terms of independent instruction streams which may run simultaneously.

Note: Some programmer implements parallelism using multiple process instead of multiple threads.



The Asynchronous Model:

In this model, the tasks are interleaved with one another, but in a single thread of control. This is simpler than the threaded case because the programmer always knows that when one task is executing, another task is not. Although in a single-processor system a threaded program will also execute in an interleaved pattern, a programmer using threads should still think in terms of Figure 2, not Figure 3, lest the program work incorrectly when moved to a multi-processor system. But a single-threaded asynchronous system will always execute with interleaving, even on a multi-processor system.



Note:

There is another difference between the asynchronous and threaded models. In a threaded system the decision to suspend one thread and execute another is largely outside of the programmer’s control. Rather, it is under the control of the operating system, and the programmer must assume that a thread may be suspended and replaced with another at almost any time. In contrast, under the asynchronous model a task will continue to run until it explicitly give up control to other tasks.

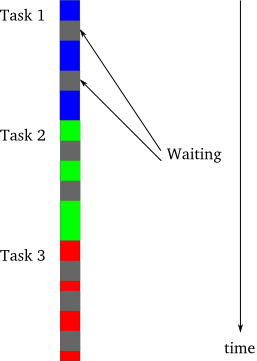
The Motivation:

The asynchronous model is clearly more complex than the synchronous case. The programmer must organize each task as a sequence of smaller steps that execute intermittently. Since there is no actual parallelism, it appears from our diagrams that an asynchronous program will take just as long to execute as a synchronous one.

So why would you choose to use the asynchronous model?

If one or more of the tasks are executed at a same time i.e. long running cpu task or I/O, then by interleaving the tasks together the system can remain responsive to user input while still performing other work in the “background”. So while the background tasks may not execute any faster, the system will be more pleasant and responsive for the person using it.

However, there is a condition under which an asynchronous system will simply outperform a synchronous one, sometimes dramatically so, in the sense of performing all of its tasks in an overall shorter time. This condition holds when tasks are forced to wait, or *block*, as illustrated in the figure.



In the figure, the gray sections represent periods of time when a particular task is waiting (blocking) and thus cannot make any progress. Why would a task be blocked? A frequent reason is that it is waiting to perform I/O, to transfer data to or from an external device. Thus, a synchronous program that is doing lots of I/O will spend much of its time blocked while a disk or network catches up. Such a synchronous program is also called a blocking program for that reason.A blocking program, looks a bit like Figure 3, an asynchronous program. This is not a coincidence. The fundamental idea behind the asynchronous model is that an asynchronous program, when faced with a task that would normally block in a synchronous program, will instead execute some other task that can still make progress. So an asynchronous program only “blocks” when no task can make progress and is thus called a non-blocking program. And each switch from one task to another corresponds to the first task either finishing, or coming to a point where it would have to block. With a large number of potentially blocking tasks, an asynchronous program can outperform a synchronous one by spending less overall time waiting, while devoting a roughly equal amount of time to real work on the individual tasks.

Comparison:

1. There are a large number of tasks so there is likely always at least one task that can make progress.
2. The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running.
3. The tasks are largely independent from one another so there is little need for inter-task communication (and thus for one task to wait upon another).

These conditions almost perfectly characterize a typical busy network server (like a web server) in a client-server environment. Each task represents one client request with I/O in the form of receiving the request and sending the reply. And client requests (being mostly reads) are largely independent.

## Paradigm Shift

Many enthusiast programmers have followed asynchronous style programming for long time. It may not be surprising to see that Javascript actually execute all of its code in single thread. This design pattern, that the language is constructed on known as reactive pattern.

There is huge popularity of nodejs which a javascript framework built on top of Google (Chrome) V8 engine for building web services for the application. This design is achieved with the adoption of event loop.

Event Loop and Thread:

To understand the difference in behavior of the two solutions, we simply need to understand the thread and event loop.

**Synchronous Style Programing**:

|  |
| --- |
| def print\_hello():  while True:  print("{} - Hello world!".format(int(time())))  sleep(3)  def read\_and\_process\_input():  while True:  n = int(input())  print('fib({}) = {}'.format(n, fib(n)))  def main():  # Second thread will print the hello message. Starting as a daemon means  # the thread will not prevent the process from exiting.  t = Thread(target=print\_hello)  t.daemon = True  t.start()  # Main thread will read and process input  read\_and\_process\_input() |

**Asynchronous Style Programming:**

|  |
| --- |
| def process\_input(stream):  text = stream.readline()  n = int(text.strip())  print('fib({}) = {}'.format(n, timed\_fib(n)))  def print\_hello():  print("{} - Hello world!".format(int(time())))  if \_\_name\_\_ == '\_\_main\_\_':  selector = selectors.DefaultSelector()  # Register the selector to poll for "read" readiness on stdin  selector.register(sys.stdin, selectors.EVENT\_READ)  last\_hello = 0 # Setting to 0 means the timer will start right away  while True:  # Wait at most 100 milliseconds for  # input to be available  for event, mask in selector.select(0.1):  process\_input(event.fileobj)  if time() - last\_hello > 3:  last\_hello = time()  print\_hello() |

Let’s start with thread. Think of a thread as a single sequence of instructions and the CPU's current state in executing them (CPU state refers to e.g. register values, in particular the next instruction register).

Think of a thread as a single sequence of instructions and the CPU's current state in executing them (CPU state refers to e.g. register values, in particular the next instruction register).

A simple synchronous program often runs on a single thread, which is why if an operation needs to wait for something, say an IO operation or a timer, the execution of the program is paused until the operation is finished. One of the simplest blocking operations is sleep. In fact, that's all sleep does, namely blocking the thread it is executed on for the given length of time. A process can have multiple threads running in it. Threads in the same process share the same process-level resources, such as memory and its address space, file descriptors, etc.

Open the solution **example/threaded\_hello.py**, the Python solution is clearly multi-threaded. This explains why the two tasks are run concurrently, and why the calculation of the large Fibonacci number, which is CPU intensive, is not blocking the execution of the other thread.

In Asynchronous programming we can use event loop (a queue) to register the task that need to be executed whenever the thread is available. As far as the operating system is concerned your application is running in a single thread that was this design pattern is all about.

First, we will need a way to poll **stdin** for input availability, that is, a system call that asks if a file descriptor (in this case stdin) has input available for reading or not.

Once we have the polling functionality, our event loop will be very simple: in each iteration of the loop, we check to see if there's input available for reading, and if so we read and process it. After that, we check to see if more than three seconds has passed since the last printing of "Hello world!" and if yes, we print it. You can find the simple implementation in example/evt\_loop.py

Event Loop with Callback:

A natural generalization of the previous section's event loop is to allow for generic event handlers. This can be relatively easily achieved using callbacks: for each event type (in our case, we only have two of them, input on stdin and timers going off), allow the user to add arbitrary functions as event handlers. The implementation of event loop with callback can be found in example/evtloop\_callback.py

**Python Library:**

Twisted is a high performance asynchronous networking library, which support many protocol to TCP, UDP, and SNMP etc. to interact with remote service asynchronously.

You can find basic example in [Twisted](https://twistedmatrix.com/trac/).

I’ll include any basic implementation of asynchronous programming using twisted. I’ll add a small application that is created with [Klein](http://klein.readthedocs.io/en/latest/), which a micro framework built on top of, twisted to create asynchronous web server. The api is similar to [Flask](http://flask.pocoo.org/) which is another Micro web framework but not asynchronous in this github **link,** although I’ll not explain anything related to Klein since it’s usage is similar to flask but walk through the basic concept and api that is used both in twisted and flask.

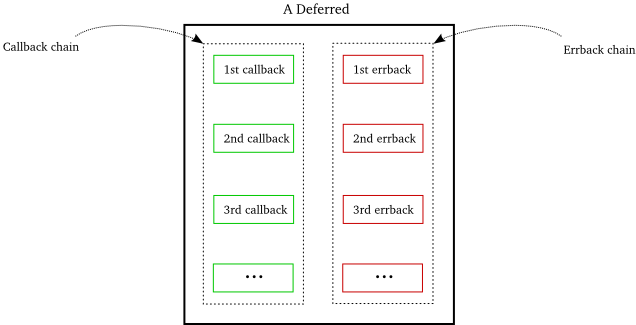
Before proceeding I’ll introduce you few concept, which are Deferred, Co-routine and Why it is needed?

Managing callback is difficult you have to pass function pointer as an argument to the function, which will be call later. Consider in regular program flow where callback is executed once the previous line execute successfully but what if there is an error. The program will stop it’s execution further if not handled gracefully.

Deferred:

Twisted provide an abstractions over callback known as deferred which provide two chain

1. Callback
2. Errback

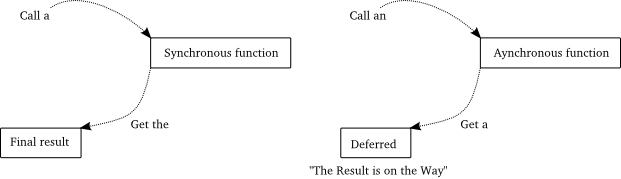


|  |
| --- |
| from \_\_future\_\_ import print\_function  from twisted.internet.defer import Deferred  def google(request):  d = treq.get('https://www.google.com' + request.uri)  d.addCallback(treq.content)  return d  def got\_poem(res):  print 'Your poem is served:'  print res  def poem\_failed(err):  print err.\_\_class\_\_  print err  print 'No poetry for you.'  if \_\_name\_\_ == '\_\_main\_\_':  d = Deferred()  # add a callback/errback pair to the chain  d.addCallbacks(got\_poem, poem\_failed)  # fire the chain with a normal result  d.callback('This poem is short.')  print("Finished")  # fire the chain with an error result  d.errback(Exception('I have failed.')) |

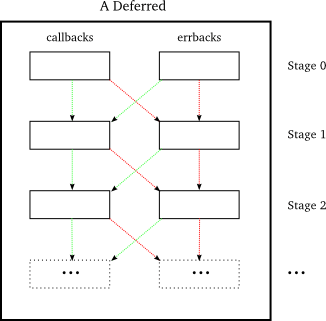
Consider the above method that is requesting a page from <http://www.google.com> you might notice the code look synchronous but it is returning deferred that can be to used to attached another callback method using addCallback method which technically formed a chain. And finally once the page has been downloaded programmer can call the d.callback() function to execute the callback chain, which means all the callback attached to that deferred object will get invoked sequentially.

d.addCallback method of the deferred class takes 2 arguments a callback and a failure function. You can also invoke an exception using d.errback and wrap an Exception object inside it.

d.addBoth is another method which can be used to register method that will be invoked both in case of success and error which is similar like finally statement in try catch block.



Flow of control in deferred:



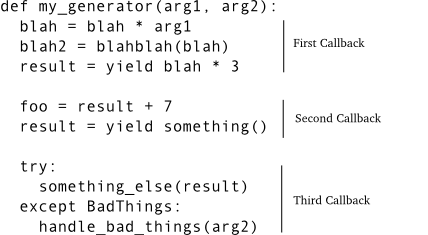
The purpose of introducing deferred so that it can facilitate us in writing code in synchronous way as we used to do. Consider the flow of execution in native python programming whenever we raise an exception the exception get passed to the method calling it. Similar goes with deferred whenever an unhandled exception occur or raise an exception, the Exception object get wrapped in a twisted **Failure** object that will be passed down the chain. If the exception is handled gracefully it will be handled by the next callback as shown by green line else the flow of control will be handled by the next errback in the chain.

Co-routine:

Before proceeding in deeper details with co-routine, let me introduce you the magic of generator function. Generator is function that can be pause it’s flow of execution and can be continued later whenever you called next or send on the generator function, it’s created using yield function and return an iterator object that can be called using next method.

|  |
| --- |
| def my\_generator():  print 'starting up'  yield 1  print "working'"  yield 2  print "still working'"  yield 3  print 'done'  for n in my\_generator():  print n |

Co-routine provide us another alternative for writing asynchronous program using yield statement you can pause of the execution of code and can later invoke the remaining statement whenever output is available for processing.



Now when a series of callbacks is chained together in a deferred, each callback receives the result from the one prior. That’s easy enough to do with a generator — just send the value you got from the previous run of the generator (the value it yielded) the next time you restart it.

So imagine that our generator yields a deferred object instead of an ordinary Python value. The generator is now “paused”, and that’s automatic; generators always pause after every yield statement until they are explicitly restarted. So we can delay restarting the generator until the deferred fires, at which point we either send the value (if the deferred succeeds) or throw the exception (if the deferred fails). That would make our generator a genuine sequence of asynchronous callbacks and that’s the idea behind the inlineCallbacks function in twisted.internet.defer.

**Inline Callback:**

The whole purpose of inlineCallbacks is turn a generator into a series of asynchronous callbacks according to the scheme we outlined before.

Second, when we invoke an inlineCallbacks-decorated function, we don’t need to call next or send or throw ourselves. The decorator takes care of those details for us and ensures the generator will run to the end (assuming it doesn’t raise an exception).

Third, if we yield a non-deferred value from the generator, it is immediately restarted with that same value as the result of the yield.

And finally, if we yield a deferred from the generator, it will not be restarted until that deferred fires. If the deferred succeeds, the result of the yield is just the result from the deferred. And if the deferred fails, the yield statement raises the exception. Note the exception is just an ordinary Exception object, rather than a Failure, and we can catch it with a try/except statement around the yield expression.

|  |
| --- |
| from twisted.internet.defer import inlineCallbacks, Deferred  @inlineCallbacks  def my\_callbacks():  from twisted.internet import reactor  print 'first callback'  result = yield 1 # yielded values that aren't deferred come right back  print 'second callback got', result  d = Deferred()  reactor.callLater(5, d.callback, 2)  result = yield d # yielded deferreds will pause the generator  print 'third callback got', result # the result of the deferred  d = Deferred()  reactor.callLater(5, d.errback, Exception(3))  try:  yield d  except Exception, e:  result = e  print 'fourth callback got', repr(result) # the exception from the deferred  reactor.stop()  from twisted.internet import reactor  reactor.callWhenRunning(my\_callbacks)  reactor.run() |

**Summary:**

I have covered the basic of reactive design pattern the benefits of using this pattern in contrast to synchronous programming, basic introduction to programming models Synchronous, Asynchronous and Thread Model. An elementary introduction of writing asynchronous code using callback and co-routine.

Also I have walk through twisted – a python asynchronous library and some of it’s features Deferred and inlineCallback which facilitate us in writing asynchronous code in synchronous patter.

Please go through the example code which I have added it contain explanation of the basic asynchronous function that library like Twisted, Gevent provide, the demo application which I have prepared for the sake of this tutorial using Klein can be found the tutklein folder.

Download the code from [this](https://github.com/sandyz1000/twistedklein) link.