Mini-Guild - Build your own Actor Core

Michael Sparks

August 24, 2022

Contents

1 Introduction

Back in the day we'd periodically have relatively novice python developers working with Kamaelia. In order to bring them up to speed, we created a set of targetted tutorials. These focussed on two things. One aspect the python concepts they needed to know. The other was "how does Kamelia's core 'Axon' work". To that end it essentially walked them through building a simplified core. This tutorial was called "mini-axon".

This turned out to be a very effective way of introducing developers. It also turned out to be a very effective way of building micro-versions that could be used in specific targetted ways. For example when we shifted to zero copy, or threaded versions of kamaelia components, these experiments started as targeted mini-axon examples.

To this end, this mini-guild "build your own guide" aims to provide:

- A means of illustrating the concepts around actors and actor systems
- A basis for explaining some key ideas, specifically in the context of Guild
- A basis for experimentation around lightweight actor implementations
- Clear user code and clear core code

1.1 Concurrency & Parallelism

This a is a huge topic, so this section defines how I am using these words here.

Concurrency

Concurrency normally means "multiple things going on at once". This is generally conceptual and relates to activities. Reality is that CPU may be spending some time on this, then some time on that, and the some more on something else. Some problems in life are inherently concurrent - such as muliple browser fetches for a webpage or a webserver having to deal with 1000 concurrent users.

Parallelism

Parallel normally refers to "things actually occurring simultaneously". Examples:

- Eating, breathing and your heart beating at the same time.
- A CPU doing lots of work on the 12 discrete cores it has.
- 12 physical web servers handling network traffic
- Calculations regarding shading of a scene being handled by 2000 physical shaders on a GPU

Of note here is that in all the above, there needs to be multiple physical things capable of doing work. (I'll refer to these as devices in the next section) Not all concurrent activities need be parallel. Parallel activities are always concurrent. Despite this, not all concurrent activities are parallelisable.

Usage

- I use concurrent to refer to tools and techniques for managing the inherent complexity in a problem space.
- I use parallel, when we are referring to techniques for to spreading load over multiple devices.

Actors are good for orchestrating and managing concurrency - as such, this is the focus of Guild and this tutorial

Techniques (like GPU offloading, SIMD, etc) for intentionally boosting parallelism are out of scope for this tutorial. A developer friendly metaphor that would be more suitable for GPU offloading, SIMD and similar parallelism related aspects would be Entity Component Systems. (Indeed a follow on document on how ECS and Actors can work together is likely to be written)

1.2 Core Aspects of Actors

1.2.1 General perspective

As defined by Carl Hewitt, actors:

- Are concurrent things
- That can send messages to each other
- Can decide what to do and what to next based on those messages
- Can create new actors

If you're implementing actors, you have to decide how each of these things work. Some decisions you can make are:

- To implement concurrent thing we have a number of options from high level network servers, all the way down to processes, threads, green threads, coroutines, state machines. These maintain some sort of state and that state is not shared directly at all.
- Actors interact by sending messages to each other in a concurrent safe form. This can be any/all of: REST API, RPC, MPI, methods, mailboxes, queues, lists, STM, mutexes, semaphones, etc
- Can decide what to do and what to next based on those messages - variables, switches, methods, statemachines, etc
- Can create new actors - spin up a server/process/thread/etc

It's because of this very loose description and the very many different ways of implementing it that personally I think of actors as a pattern of implementation rather than a specific single implementation style.

1.2.2 Tutorial perspective

In this tutorial, we make the following decisions around concurrency, messages and behaviour.

Concurrency

- Our unit of concurrency is a **restricted co-routine**
- These are managed by a single threaded **scheduler**
- After creating new actors, we will ask the scheduler to run it

Messages

- Our message system **is a method call**
- Method calls result in named references to methods being **queued**
- The queue is simply a list

Behaviour

- The actor performs actions based on messages in the queue
- **The behaviour of an actor is a basic class.**

1.2.3 Limitations

- If we were using a functional programming language our messages would be immutable. As a result the rule applies: If you've handed it off to someone else, you cannot assume that the content will not change, except within the limited restriction of a single piece of code.

Things in guild, that are not in this tutorial. These will be discussed at the end.

- **Promises/futures** - a means for an actor to reply back to the sender
- **Actor-functions with exceptions** - Send a message, wait for a response and if an error occurred, propogate the error
- **Simplified STM** - software transactional memory (though this will be added later)

2 Getting things Started

2.1 Concurrency

Our basic unit of concurrency is a restricted co-routine. In python this specifically means a python generator. A python generator can be viewed as a single function that has the ability to suspend and restart.

A simple fibonacci generator looks like this:

```
def fib(basecase=1):
    a, b = basecase, basecase
    while True:
        yield a
        a,b = b, a+b
```

This function creates a generator when called.

```
>>> f = fib()
>>> f
<generator object fib at 0x7efd3b9046d0>
>>>
```

We can repeatedly ask for the next[1] values:

```
>>> for _ in range(5):
...     print(next(f))
...
1
1
2
3
5
```

We can use the same approach to stuff the values into a list using a list comprehension:

```
>>> f = fib()
>>> [ next(f) for _ in range(5) ]
[1, 1, 2, 3, 5]
```

NB, we made out fib() function configurable, so a different base case gives different results:

```
>>> f = fib(2)
>>> [ next(f) for _ in range(5) ]
[2, 2, 4, 6, 10]
```

We could create 5 different generator objects like this, using different base cases to tell them apart:

```
>>> fibs = [ fib(i) for i in range(5) ]
>>> fibs
[<generator object fib at 0x7fc888c53610>, <generator object fib at
0x7fc888b587b0>, <generator object fib at 0x7fc888b58820>, <generator
object fib at 0x7fc888b58890>, <generator object fib at 0x7fc888b58900>]
```

A compact way of getting the first 5 values out of each of these concurrently looks like this:

```
fibs = [ fib(i) for i in range(1, 6) ]
for i in range(5):
    x = [next(f) for f in fibs]
    print(x)
```

With the result:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[2, 4, 6, 8, 10]
[3, 6, 9, 12, 15]
[5, 10, 15, 20, 25]
```

Note that the various versions of fibonacci form vertical slices. Effectively each call to next() gave each generator a timeslice.

2.2 MicroActors

It should be clear that generators give us a way of having "something that can be given a bit of time to run". We can then run something else.

We'll start start there and create a `MicroActor`. We can then create a `MicroScheduler` that can run these `MicroActors`. We can then create a couple of toy examples.

In later section's we'll flesh out `MicroActora` to `Actors`.

2.2.1 Exercise: Write a `MicroActor` baseclass

Definition of done:

- Create a class called `MicroActor`
- Give it a main method.
- That main method must `yield 1` to the caller. This makes it generator. It also means the first value returned will be 1. Requesting further values would raise the exception `StopIteration` - since that's what happens with generators when you fall of the bottom of them.

Expected usage/results:

```
>>> m = MicroActor()
>>> g = m.main()
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Answer

```
class MicroActor:
    def main(self):
        yield 1


m = MicroActor()
g = m.main()
```

[1] next() is a helper function that calls the __next__() method on a generator.

```
next(g)
next(g)
```

2.2.2 Exercise: Write a `MicroScheduler` class

We'll generally create 1 `MicroScheduler` (or `Scheduler`) instance in this tutorial. However you could imagine having mutiple schedulers (one per CPU for example). We expect to create one instance, then add MicroActors to it and then run them. A detail is that while MicroActors are running, they may ask to add more MicroActors to the runqueue. (This means our runqueue may change while we're looping through it)

Definition of done:

- Create a class called `MicroScheduler`
- Add an `__init__` method with anything you feel necessary.
- Add a `schedule(self, microactor)` method
  - This accepts an instance of `MicroActor` called `microactor` as an argument
  - Call `microactor.main()` to give you a generator `g`
  - Add this generator `g` to the list of `microactors` to give CPU time
- Add a `run(self)` method – When this is called (with no arguments)
  - If there is nothing to run, it exits, otherwise it loops round everything below
    - it repeatedly runs through all `microactors`' generators
    - For each generator `g`, call `next(g)` to give it a time slice.
    - The yielded value is ignored (for now)
    - `next(g)` may raise a `StopIteration` exceptin:
      - If it doesn't - add it to the runqueue to run again next time
      - If it does - do anything (meaning it will not be run again)
    - Tip: iterate through the runqueue with a while loop and explicit index.

Expected usage/results:

```
>>> class LimitedFibonacciPrinter(MicroActor):
...     def __init__(self, basecase=1):
...         self.basecase = basecase
...     def main(self):
...         a = b = self.basecase
...         for _ in range(5):
...             yield 1
...             print("FIB:", a)
...             a, b = b, a+b
...
>>> lfp = LimitedFibonacciPrinter()
>>> s = MicroScheduler()
>>> s.runqueue
[]
>>> s.add(lfp)
>>> s.runqueue
[<generator object LimitedFibonacciPrinter.main at 0x7f7953447610>]
>>> s.run()
1
1
2
3
5
>>> s.runqueue
[]
```

Answer

The simplest possible answer that will work looks like this:

```
class MicroScheduler:
    def __init__(self):
        self.runqueue = []
    def add(self, microactor):
        g = microactor.main()
        self.runqueue.append(g)
    def run(self):
        while len(self.runqueue) > 0:
            new_runqueue= []
            i = 0
            while i < len(self.runqueue):
                g = self.runqueue[i]
                try:
                    next(g)
                    new_runqueue.append(g)
                except StopIteration:
                    # Expected - generator has exitted
                    pass
                i += 1
            self.runqueue = new_runqueue
```

2.2.3 Toy Example Usage

Suppose we have 3 MicroActor classes:

```
class LimitedFibonacciPrinter(MicroActor):
    def __init__(self, basecase=1):
        self.basecase = basecase
    def main(self):
        a = b = self.basecase
        for _ in range(5):
            yield 1
            print("FIB:", a)
            a, b = b, a+b


class LimitedTrianglesPrinter(MicroActor):
    def __init__(self, basecase=1):
        self.basecase = basecase
    def main(self):
        a = self.basecase
        n = a +1
        for _ in range(5):
            yield 1
            print("TRIANGLE:", a)
            a = a + n
            n = n + 1

def isprime(n):
    for i in range(2,n):
        if n % i == 0:
            return False
    return True

class LimitedPrimesPrinter(MicroActor):
    def __init__(self, basecase=1):
        self.basecase = basecase
    def main(self):
        a =  self.basecase
        for _ in range(5):
            while not isprime(a):
                a += 1
            yield 1
            print("PRIME:", a)
            a += 1
```

We can now run create instances of these `MicroActors` and run them:

```
>>> lfp = LimitedFibonacciPrinter(1)
>>> ltp = LimitedTrianglesPrinter(1)
>>> lpp = LimitedPrimesPrinter(99)
>>>
>>> s = MicroScheduler()
>>> s.add(lfp)
>>> s.add(ltp)
>>> s.add(lpp)
>>> s.run()
FIB: 1
TRIANGLE: 1
PRIME: 101
FIB: 1
TRIANGLE: 3
PRIME: 103
FIB: 2
TRIANGLE: 6
PRIME: 107
FIB: 3
TRIANGLE: 10
PRIME: 109
FIB: 5
TRIANGLE: 15
PRIME: 113
```

2.3 From MicroActors to Actors

It’s clear from the last example that we’re creating 3 little MicroActors and they all run conccurently and they all send stuff to the console. They are all however all printing to the console. Since they’re all part of the same thread of control this isn’t a problem, but if these were actually separate threads this would be a complete mess. Indeed the output could look something like this:

```
TRIPFIB: 1
RIME: 101
ANGFIB: 1
LE: 1
TRIANPRIME: 103
GLE: 3
TRIAPFIB: 2
RIME: 107
NGLE: 6
TRFIB: 3
IPRIME: 109
ANGLE: 10
TRFIB: 5
IPRIME: 113
ANGLE: 15
```

The next steps in this section therefore are:

- Change the 3 MicroActors so that they send a message to another actor asking it to print to the console.
- Tweak our MicroActors and Scheduler to allow the system to exit
- Tweak the system so that it doesn't eat all the system CPU
- Split off the behaviour from the concurrency model.

At that point we'll have something that could be reasonably called a simple actor system.

2.3.1 Exercise: Simplest Possible MicroActor with a Mailbox

So now, we have some decisions to make.

- How do we send messages to other MicroActors?
  - Sender must be placing the messages where the recipient is looking
- How do we ensure these messages are well formed?
- We need to ensure that these are processed in order.

```
class PrintActor(MicroActor):
    def __init__(self):
        self.inbox = []
    def _print(self, *args):
        print(*args)
    def print(self, *args):
        self.inbox.append(("print", args))

    def main(self):
        while True:
            while self.inbox:
                msg = self.inbox.pop(0)
                func, args = msg
                if func == "print":
                    self._print(*args)
            yield 1
```

We can modify our example `MicroActors` to use a `PrinterActor` instance for printing:

```
class LimitedFibonacciPrinter(MicroActor):
    def __init__(self, printer, basecase=1):  # NOTE: Changed
        self.basecase = basecase
        self.printer = printer # NOTE: Added
    def main(self):
        a = b = self.basecase
        for _ in range(5):
            yield 1
            self.printer.print("FIB:", a)  # NOTE: Changed
            a, b = b, a+b
```

```
class LimitedTrianglesPrinter(MicroActor):
    def __init__(self, printer, basecase=1):  # NOTE: Changed
        self.basecase = basecase
        self.printer = printer # NOTE: Added
    def main(self):
        a = self.basecase
        n = a +1
        for _ in range(5):
            yield 1
            self.printer.print("TRIANGLE:", a)  # NOTE: Changed
            a = a + n
            n = n + 1

def isprime(n):
    for i in range(2,n):
        if n % i == 0:
            return False
    return True

class LimitedPrimesPrinter(MicroActor):
    def __init__(self, printer, basecase=1): # NOTE: Changed
        self.basecase = basecase
        self.printer = printer # NOTE: Added
    def main(self):
        a =  self.basecase
        for _ in range(5):
            while not isprime(a):
                a += 1
            yield 1
            self.printer.print("PRIME:", a)  # NOTE: Changed
            a += 1
```

Now we can run this:

```
pa = PrintActor()
lfp = LimitedFibonacciPrinter(pa, 1)
ltp = LimitedTrianglesPrinter(pa, 1)
lpp = LimitedPrimesPrinter(pa, 99)

s = MicroScheduler()
s.add(pa)
s.add(lfp)
s.add(ltp)
```

```
s.add(lpp)
s.run()
```

And we find that the output is as expected. Not only that if we switched the microactors to using threads and from a list to a threadsafe qeue like `Queue.Queue`, we would find that this still works as expected.