Introduction to Asynchronous Python

Hello!

I’m Doug Farrell

I’ve been developing with Python for quite a while in a few different domains and industries.

What we'll be talking about

Asynchronous World

We live in an asynchronous world and we react to it as people by reacting to events because we're relatively good task switchers.

We can delegate activities to run on their own and switch our personal context between them as needed.

We can drive a car, talk with the passenger and remember the next route change to take.

We can handle these activities because we're reacting to the events of each activity as they happen.

Asynchronous Misconceptions

Asynchronous behavior in a computer program is not parallelism, where things are actually happening at the same time on multiple CPUs.

It's more like task-switching concurrency, where multiple things are getting done, but not necessarily at the same time, and on a single CPU.

Parenting Analogy

You might be a parent, or certainly have parents, so you can relate to what we'll be using as an analogy to work our way towards asynchronous programming with Python.

Being a parent means handling many things at once; watching the kids, balancing the checkbook, moving the laundry to and from the washer and dryer, among lots of other stuff.

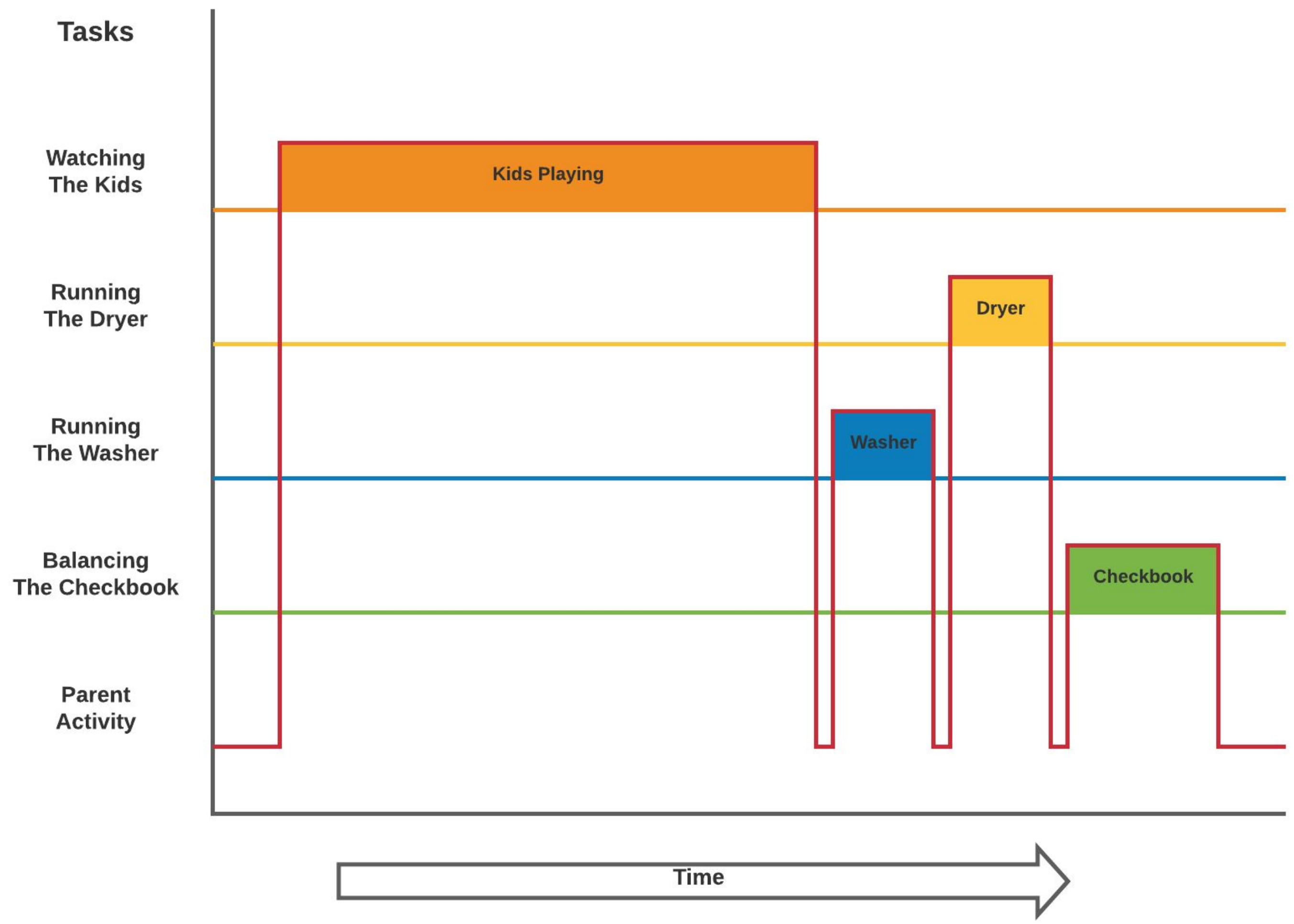
We're going to talk about "programming" a parent to handle these things as a way to see how well, or poorly, the different approaches work.

1.

The Synchronous Parent

This parent wants to get everything done, but has to stay with a task all the way through to completion. This means, as a good parent, minding the kids all the way to bedtime when something else can happen

Here we find the synchronous parent attending to various tasks. Because a task has to be completed before starting another, all the tasks are run serially, one after another.



Synchronous Results

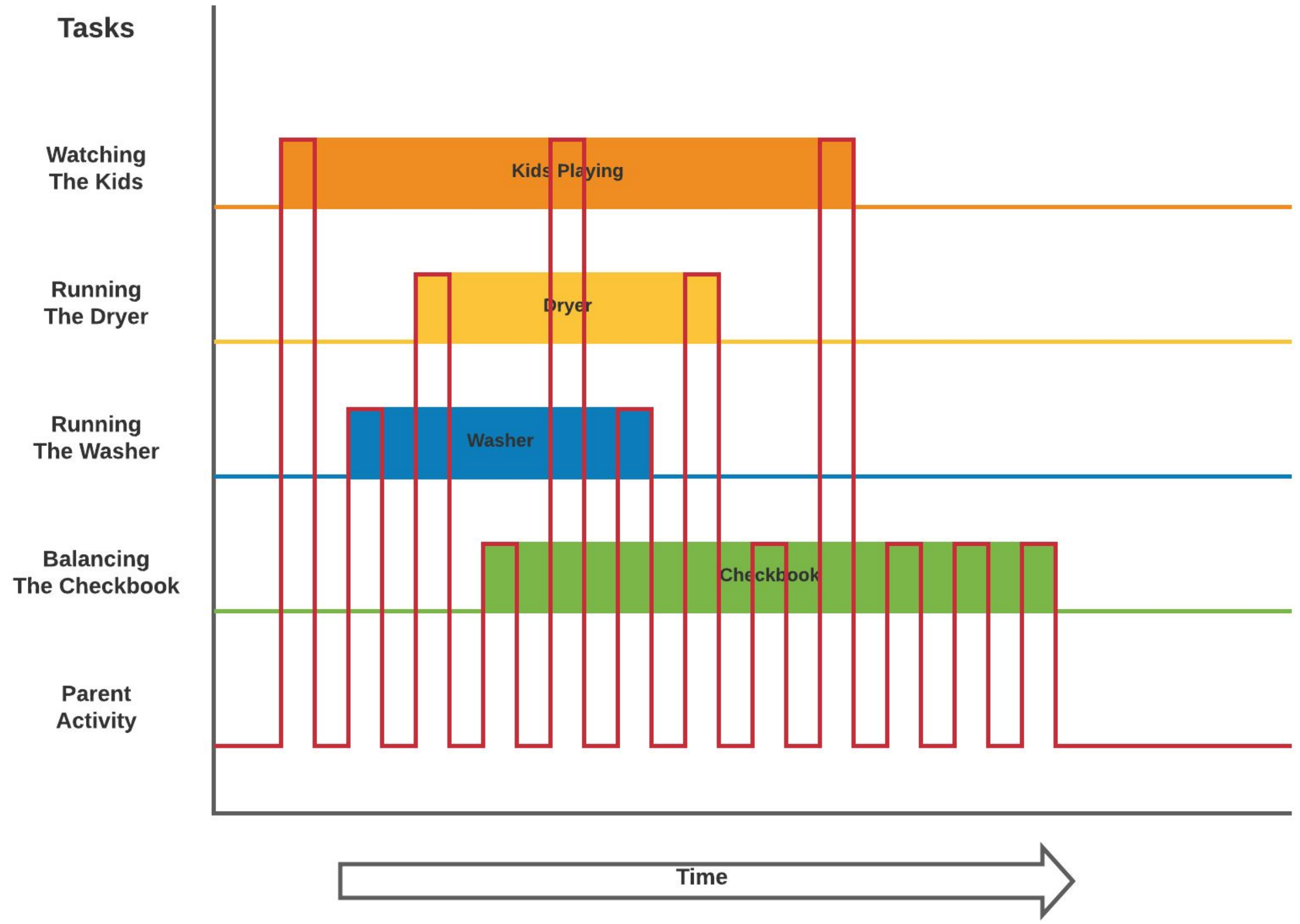
From the execution timeline you can see performing the tasks serially makes for a very long day. Parents spend most of their time watching the kids.
Only after the kids go to bed is the parent able to start another task, but even then the serialized nature of this approach means waiting for tasks like running the washing machine to completion before starting the dryer.
I think you can see that a few weeks of living like this and most parents would abandon ship.

2.

The Polling Parent

This parent wants to get everything done too, and does so by breaking away from any current task on a regular basis to see if any other task needs attention.

The polling parent is able to move all the tasks towards completion, but the activity of polling makes each task taker longer. In addition the delay between activity on a task might be unacceptable, as with watching the kids playing.



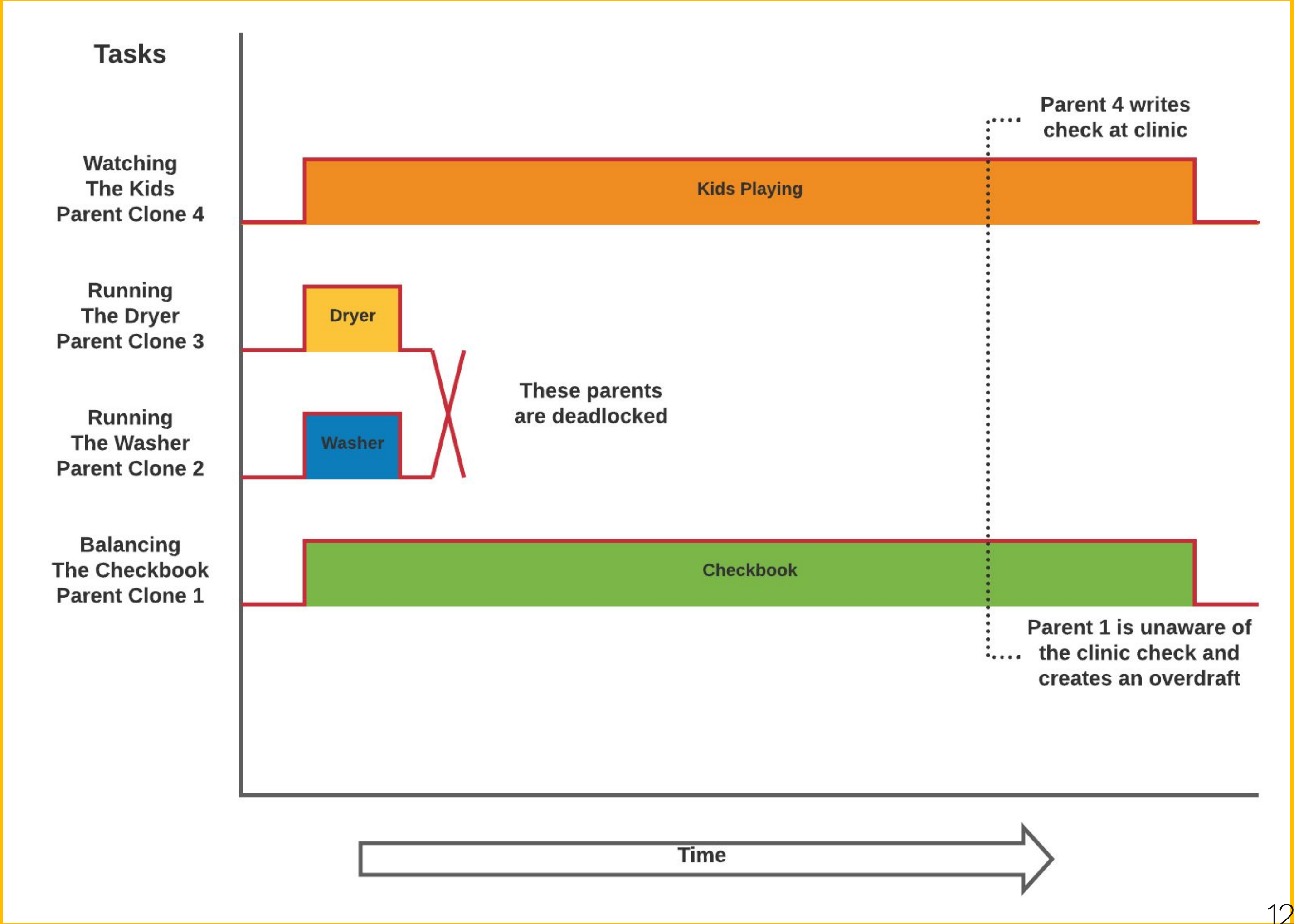
Polling Results

The execution timeline shows that breaking away from any particular task on a periodic basis to work on another is a way of making progress on all the tasks.
However, this has some limitations. Depending on the polling frequency a significant amount of time is spent waiting for tasks to complete.
Worse, priority tasks like the kids might not get attention for a multiple of the polling frequency. This fails if a child is hurt and needs attention.
This problem could be partially addressed by increasing the polling frequency, but then more time is spent context switching between tasks.
Again, a couple weeks of this and we're back to the ship and abandonment options.

3.
The Threaded Parent

This parent wants to get everything done as well, and does so by realizing the dream of all parents; to clone themselves. Each cloned parent attends to a single task.

The threading model creates parent clones, each assigned a separate task. The parents need to talk with each other in order to avoid deadlocking on resources one or more parents want, and not change a shared resource.


Tasks
Watching The Kids Parent Clone 4
Running The Dryer Parent Clone 3
Running The Washer Parent Clone 2
Balancing The Checkbook Parent Clone 1
Kids Playing
Dryer
Washer
Checkbook
These parents are deadlocked
Parent 4 writes check at clinic
Parent 1 is unaware of the clinic check and creates an overdraft
Time

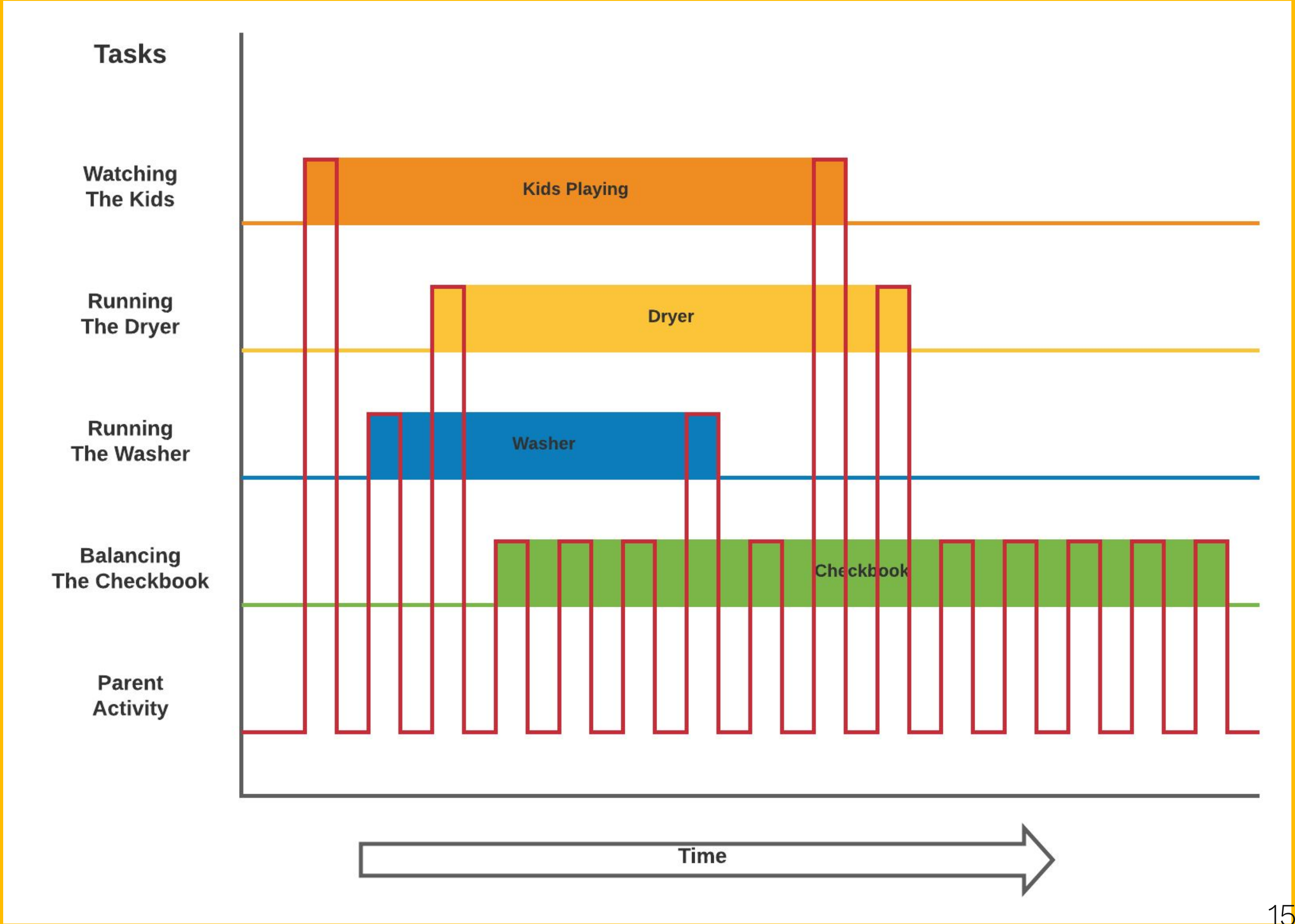
Threading Results

The results of threading looks promising, all the tasks have the full attention of a parent clone and proceed to completion.
However, there are complexities that need to be handled. For instance, if the washing machine parent is using the washer and wants to acquire the dryer, and at the same time the dryer parent is using the dryer and wants to use acquire the washer. Unless handled, these two parents are deadlocked and both tasks are stalled.
Another possible issue is sharing a resource like the family budget. If a child is hurt and has to go to a clinic and that parent has to write a check, how does the checkbook balancing parent know about this and avoid an overdraft?

4. The Asynchronous Parent

Like the other parents, this parent wants to complete tasks, and does so by delegating tasks that can run on their own. This parent can then complete tasks that require full attention to move forward and respond to events the delegated tasks generate.

The asynchronous parent is able to initiate multiple tasks and then move on till those tasks send an event needing attention. The CPU bound task (the checkbook) explicitly cooperates with the event loop to act on the events from the other tasks.



Asynchronous Results

Based on the execution timeline the asynchronous parent handles the tasks pretty well, and models the behavior of a real parent closely.
The success of this model is the ability to delegate tasks and respond to events they produce. The events are like the washer and dryer making a tone when the cycles are done, or like the kids getting louder because of something that needs attention.
This behavior frees the parent to attend to the checkbook, a task that can't be delegated, but can be paused to break away and check for events.

Asynchronous Python

An asynchronous style of programming is possible in Python by taking advantage of the orders of magnitude differences between CPU and IO bound processing.
CPU bound tasks are like long running calculations, compression algorithms, code that requires the CPU to be executing instructions.
IO bound tasks are like making an HTTP request, reading/writing a file, code that makes a call to the OS, but the CPU has to wait for the IO to complete.
If IO processing can be delegated, and an event raised when the operation is complete, the CPU is freed to process application instructions that can't be delegated.
This is exactly what asynchronous Python applications do, by using the asyncio module to create an event loop and modules that support asynchronous IO operations that generate events.
This allows the application to create and manage many relatively long IO operations, while at the same time allowing the CPU to process application instructions.

Example Programs

The example programs follow a pattern that moves towards demonstrations of asynchronous programming with Python.

- There is a io_task and cpu_task function that takes time to complete based on the parameter passed to it.
- A queue contains the tasks to run, along with the parameters to pass to the tasks.
- There are one or more worker functions that pull tasks from the queue and run them with the associated parameters. The worker(s) will continue to pull work from the task queue until it is empty.
- The main function is the engine that runs the worker(s) until the task queue is empty.

The end goal is to show how asynchronous programming lets relatively long running IO tasks execute concurrently.

Example Programs List

* Example 1 - A single worker processes all the task synchronously
* Example 2 - Two workers are created, but only one does all the work because there is no context switching to run other workers
* Example 3 - The two workers are now generator functions and can yield control (context switch), but the tasks still block the CPU, so there is no net gain.
* Example 4 - The tasks are now asynchronous allowing the two workers to context switch. This allows the tasks to run concurrently and the total execution time is cut in half.
* Example 5 - The io_task performs HTTP IO work in a synchronous manner, so the workers can context switch, but run one after the other with no net gain.
* Example 6 - The io_task performs HTTP IO work in an asynchronous manner, the workers can context switch and the IO runs concurrently, with a gain in total performance.

Example 7 - A bonus example that adds file reading to the task work.

Example Programs

The example programs were developed and run in Python version 3.9.6.

You can clone the example program repository from here:

https://github.com/writeson/asynchronous_python_presentation

Asynchronous Use Cases

Pros

If your application is mixture of CPU and IO operations, it could benefit from using the asynchronous model

If your application needs to perform multiple IO operations simultaneously

It can be simpler to develop an asynchronous application over solutions that use polling or threading

Cons

If your application is primarily CPU intensive, there's little benefit using an asynchronous model

The asynchronous model doesn't take advantage of multiple CPU cores

It is more complex to develop an asynchronous application over a synchronous one

Thanks!

Any questions?

You can find me at

@writeson

doug.farrell@gmail.com

linkedin.com/in/dougfarrell/

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Unsplash