Part 1: Concurrent Programming

Bookmark this page

Part 1: Concurrent Programming

What does it mean to do something “Concurrently”? It means multiple tasks are being done at the same time. Is Concurrency the same thing as “parallelism”? Not exactly:

- **Parallelism** is about processing multiple things at the same time – true parallelism requires multiple processors (or cores).
- **Concurrency** is about handling multiple things at the same time – things that may or may not actually be running in the processor at the same time (such as network requests).

So, parallelism needs concurrency, but concurrency needs not be in parallel.

“Concurrency is not parallelism” – Rob Pike: https://vimeo.com/49718712

Despite Rob Pike using an example about burning books, I recommend listening to at least the first half of his talk.

In that talk, Rob makes a key point: Breaking down tasks into concurrent subtasks only allows parallelism, it’s the scheduling of these subtasks that creates it.

Once you have a set of subtasks, they can be scheduled in a truly parallel fashion, or managed asynchronously in a single thread (concurrent, but not parallel).

Asynchrony

We now know parallelism is not the same as concurrency. What about “asynchrony”?

- **Concurrency:** Having different code running at the same time, or kind of the same time.
- **Asynchrony**: The occurrence of events independent of the main program flow and the ways in which such events are dealt with.

Asynchrony and Concurrency are really two different things – you can do either one without the other – but they are closely related, and often used together. They solve different problems, but the problems they solve and the solutions tend to overlap.

Types of Concurrency

There are different ways to implement concurrency in Python:

- **Multithreading**: Multiple code paths sharing memory – one Python interpreter, one set of Python objects.
- **Multiprocessing**: Multiple code paths with separate memory space – completely separate Python interpreter.
- **Asyncronous programming**: Multiple “jobs” run at “arbitrary” times – but usually in one thread – i.e. only one code path, one interpreter.

Several different packages for these, both in the standard library and 3rd party libraries.

Which one should you use?

- IO bound vs. CPU bound – CPU bound requires multiprocessing (at least with pure Python).
- Event driven cooperative multitasking vs. preemptive multitasking.
- Callbacks vs coroutines + scheduler/event loop.

Motivations for parallel execution

Why would you want your code to use execute in parallel?

- Performance - Limited by “Amdahl’s Law”: http://en.wikipedia.org/wiki/Amdahl%27s_law
  - CPUs aren’t getting much faster.
- Event handling - If a system handles asynchronous events, a separate thread of execution could handle those events

- Event handling - If a system handles asynchronous events, a separate thread of execution could handle those events and let other threads do other work.
  - Examples:
    - Network applications.
    - User interfaces.

Parallel programming can be hard! If your problem can be solved sequentially, consider the costs and benefits before going parallel.

‹ Previous Next ›

Parallelization Strategy for Performance

Once you have determined that you want to execute your code in parallel, there are certain steps you need to complete to take advantage of parallel execution:

© All Rights Reserved

1. Break the problem down into chunks.
2. Execute chunks in parallel.
3. Reassemble output of chunks into a result.

Remember:

- Not every problem is parallelizable.
- There is an optimal number of threads for each problem in each environment, so make it tunable.
- Working concurrently opens up synchronization issues.

Methods for synchronizing threads:

- locks.

© 2024 University of Washington | Seattle, WA. All rights reserved.

edX, Open edX and the edX and Open edX logos are trademarks or registered trademarks of edX Inc.