Asynchronous Processing

Overview and Applicability in Python

Rui Teixeira - rui.teixeira.eng@gmail.com

Summary

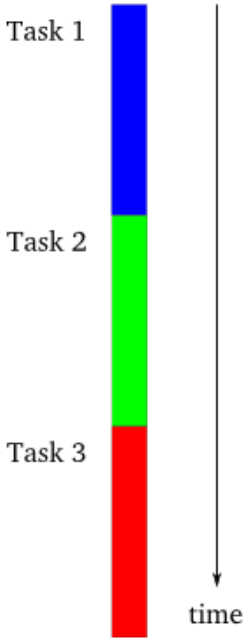
- What is Asynchronous Processing?
- Why (and when to) use it?
- How to do it?
- Some real world implementations.

What is Asynchronous Processing?

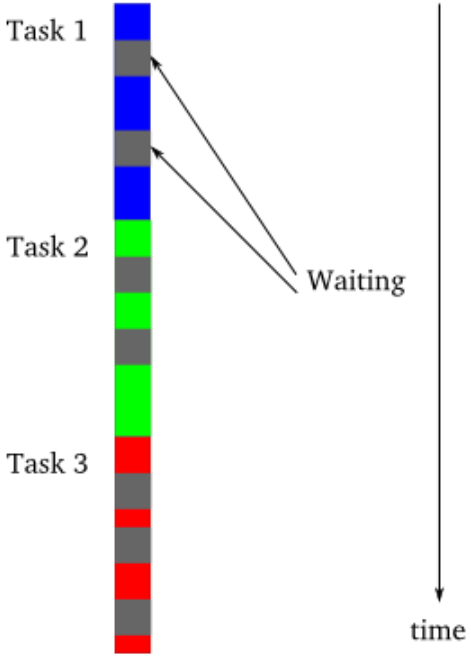
What is Asynchronous Processing? (Some caveats)

1. The term Asynchronous I/O is somewhat ambiguous, it falls under:
    a. an execution model for network programming, that scales to connections rather concurrent OS threads. Usually improving scalability (10.000s of connections vs 100s of threads)
    b. a programming model based on explicit cooperative multi-threading rather that OS-based preemptive multi-threading.
2. Do to practical real-world reasons, the two are frequently intertwined.
    a. (This presentation will reflect that fact.)

Synchronous Processing?



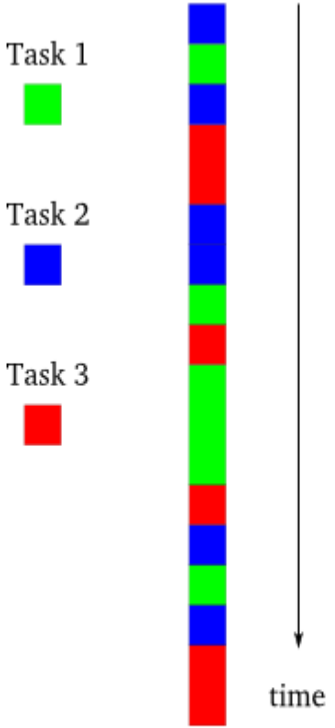
Sequential Process - 1 Thread, 3 Tasks CPU Bound



Sequential Process - 1 Thread, 3 Tasks I/O Bound

So what is Asynchronous Processing? (Async)

- Running on a single thread;
  - (No need to manage access or memory)
- There is free CPU time available between; tasks (I/O bound process);
- Using Non-blocking I/O calls;
- Tracks event notifications;
- Use event loop to resume tasks.



Why when use Async?

- **#1 Reason - Scalability (the 10kc problem!)**;
- The application:
  - is I/O bound;
  - serves a large number of clients;
  - is able to initiate requests as they became available.
- Connections:
  - are mostly idle;
  - long lived.
- Async will save a lot of memory (shared state process)

!Why && !when use Async?

- **If you don’t need it - It’s the right medicine for the wrong problem**;
- The application:
  - is CPU bound
  - serves a few clients;
- It’s a short project (no time to learn the paradigm)
- There is not async driver
  - ex: MySQL;

How to Implement Asynchronous Processes?

- Async using threads (not really Asynchronous);
- Async using Green Threads;
- Async using Coroutines.

Async: Using Threads (not really async)

- Threading module is easy to use;
- Python threads are real OS threads;
- OS scheduler decides which thread runs and for how long:
  - threads use priority;
  - switches when thread does IO;
  - switches when the thread expends it’s time;
- Results are “similar” to async approach
  - Because of the GIL.
- There is some overhead in using threads;
- Number of threads is limited by memory (100s-1000s)

Bonus: GIL - Global Interpreter Lock

- CPython's memory management is not thread-safe;
- The GIL is a mutex that prevents multiple native threads from executing Python bytecodes at once.
  - (Can be an issue for CPU bound tasks, single-core only)
- the GIL is implemented in cPtyhon;
- Alternatives are:
  - PyPy-STM (JIT compiler using Software Transactional Memory)
    - unfinished but promising
  - Jython - Python interpreter written in Java
    - No C extensions support breaks compatibility with many packages;
  - IronPython - Python interpreter written in C#
    - python =< 2.6 only

Async: Green threads

- Also called user-processes, micro-threads, light-weight threads;
- Not managed by OS;
- Several green threads per OS thread;
- Smaller overhead than OS threads.

Async: Green threads in Python

- primitive micro-thread;
- no implicit scheduling;
- implemented in stackless-python
- implemented as a C extension (not pure Python);
- C implementation of greenlets are hard to debug (no stacktrace available);
- Scheduler, event loop, and non-blocking calls are still necessary;
- (Used in several frameworks):
  - Eventlet, Gevent, Tornado, etc?

Async Implementation: Coroutines

Async: Coroutines

- Functions that can be suspended/resumed (without losing state);
- Each task decides (program logic) when to suspend;
- Event loop decides when to resume tasks (also program logic);
- Ideal for I/O bound tasks (coroutine can suspend instead of blocking);
- Not ideal for CPU bound tasks (starvation may occur, some scheduling may be required);

Async: Python coroutines

- Implemented in pure Python using generators (PEP 342);
- Coroutine stops execution using “yield from”;
- Can behave similar to callbacks.

Async Frameworks & Engines

Async Engine: Twisted

- "Twisted is an event-driven networking engine written in python".
- High-performance asynchronous architecture using “deferred” (or futures);
- Support for network, system and timed calls;
- Transport and protocol abstractions;
- Programed using framework supported callbacks;
- Suitable for low-level applications.
- Complex framework and a steep learning curve.

Async Frameworks: asyncio

- Pure python module providing async support;
- Standardized event loop interface;
- Async “flavour” similar to “threading”;
- Support for network, system and timed calls
- Transport and protocol abstractions (similar to those in Twisted);
- Very recent (Python >= 3.4);

Async Frameworks: Tornado

- web framework and asynchronous networking library,
- Can scale to tens of thousands open connections;
- Ideal for long polling, WebSockets, and other applications that require a long-lived connection to each user.

Sources - Docs

- multiprocessing — Process-based parallelism
- PEP 3156 -- Asynchronous IO Support Rebooted: the "asyncio" Module
- asyncio – Asynchronous I/O, event loop, coroutines and tasks
- Python's Hardest Problem, Revisited
- The C10K Problem
- PEP 342 -- Coroutines via Enhanced Generators
- An Introduction to Asynchronous Programming and Twisted

Sources - Talks

- Python Concurrency From the Ground Up: Live! PyCon'2015
- G. Peretin - Greenlet based concurrency
- What Is Async, How Does It Work, and When Should I Use It? (PyCon APAC 2014)
- Architecting an event-driven networking engine: Twisted Python

Questions?

In Conclusion

It really depends on the problem.

That’s why it’s called engineering.

Bonus: Multiprocessing

- Multiprocessing trades threads for processes;
- No more GIL constraints;
- Same interface as threads module;
- Unlike threads, no data sharing between processes;
- Uses Queues and Pipes for communication
  - Implies normal data synchronization issues
- Larger overhead in creating processes and sharing data.