Concurrency

CS 339R (Python) — Chapter 20
Concurrency Modules

- multiprocessing
- threading
Processes vs. Threads

- A “process” is an independently running *program*
  - managed by the O/S
  - It has its own process space
- Threads run *inside* a single process
  - a process can have many threads
  - threads *share* the process space
- Which is better?
  - Depends!
  - The answer in Python is usually *processes*
Parallel Processing

* All the rage (We live in the age of multiple cores)
* True parallel processing requires *separable code*
* `multiprocessing.Pool.map()` *parallelizes* separable code
* See `multiproc0.py`
Concurrent Processes

- `multiprocessing.Process`
  - launches a new **Python** interpreter *process*
  - you pass it a *callable* to run
  - launch overhead: Java: 500 ms, Python: 150ms, Go: 50ms
- You can *wait* for it to complete or not
- You can *communicate* with it
Selected Process Methods

- **Process**(target=<function>, args=<arg-tuple>,...)
- **is_alive()**
- **start()**
- **run()** (override option to **start()**)
- **terminate()** (use should be rare)
- **See multiproc1.py**
Message Passing

- A technique for *inter-task communication* (IPC)

- An alternative to *in-process shared data*
  - aka “critical sections” via locks/mutexes
  - those are gnarly and error prone

- Some threading models use message passing
  - Erlang, Scala, D (D actually allows *both*)

- Message Passing Options: *pipes, queues*
  - They *pickle* the data
**Shared Process Queues**

- `multiprocessing.Queue`

- Methods:
  - `empty()`, `full()`, `qsize()`
  - `get()`, `put(<item>)`
  - `close()`, `join_thread()`

- See `multiproc3.py`
  - Check the logs
Processes vs. Threads

Python supports both message passing and thread-based concurrent programming on most systems. Although most programmers tend to be familiar with the thread interface, Python threads are actually rather restricted. Although minimally thread-safe, the Python interpreter uses an internal global interpreter lock (the GIL) that only allows a single Python thread to execute at any given moment. This restricts Python programs to run on a single processor regardless of how many CPU cores might be available on the system. Although the GIL is often a heated source of debate in the Python community, it is unlikely to be removed at any time in the foreseeable future.

The presence of the GIL has a direct impact on how many Python programmers address concurrent programming problems. If an application is mostly I/O bound, it is generally fine to use threads because extra processors aren’t going to do much to help a program that spends most of its time waiting for events. For applications that involve heavy amounts of CPU processing, using threads to subdivide work doesn’t provide any benefit and will make the program run slower (often much slower than you would guess). For this, you’ll want to rely on subprocesses and message passing. (p. 414)
The Peril of the GIL

- Only one thread can run at a time
- Therefore, only one processor is busy
- ... no real concurrency!
- Recall `multiproc2.py`
Using Pipes

- Pipes can be one or two-way

- `conn1, conn2 = multiprocessing.Pipe()`
  - pass `conn2` to a child process
  - `conn1` receives, `conn2` sends in non-duplex mode

- Use `send` to write, `recv` to read

- See `multiproc4.py`

- See `primes1.py` vs. `primes2.py` (use `n = 100,000`)
Threads

* Independent paths of execution in a *single process*

* As stated before, not useful for multiple, CPU-intensive threads because of the GIL in Python

* Synchronize shared data with *locks*

* Coordinate communication with *events*

* See *prodcons_t.py*