Concurrency in Python

Summary

This is an overview of various approaches to concurrency with an emphasis on network servers (like nginx and Apache) and frameworks (like Tornado Web and gevent).

Find out how stacks, locks, and callbacks each provide their own challenges!

About Me

Shannon -jj Behrens <jjinux@gmail.com>

http://jjinux.blogspot.com

Motivation: Writing a Web Server

- Concurrency is doing multiple things at the same time
- Imagine a non-concurrent Web server

What I Won't Focus On

- Parallel computing
  - MPI
- Distributed computing
  - Creating a Google-like server farm
  - MapReduce
  - Grid computing
  - Erlang (which takes these ideas to their logical conclusion!)

I/O vs. CPU Throughput

- Do you have a lot of requests or a lot of data to crunch?
- The solutions sometimes blur this distinction
- Threads try to address both
- I'm going to focus on I/O throughput

Approaches to Multitasking

- Time slicing
- Non-preemptive schedulers
- Preemptive schedulers

Time Slicing

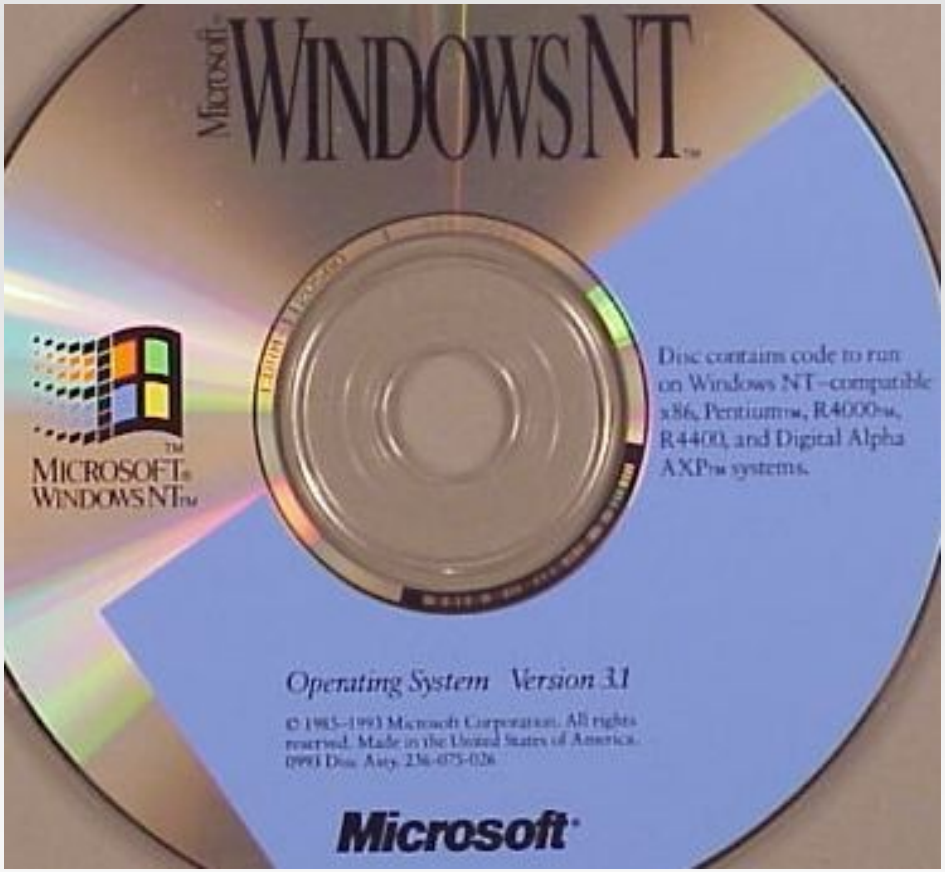
- A way of getting one CPU to pretend like many CPUs

Non-preemptive Schedulers

Non-preemptive Schedulers

- One program does something and then "yields" to the others
- If one program gets wedged, then the whole system is wedged
- It's easy to reason about, because it's more deterministic

Preemptive Schedulers



Preemptive Schedulers

- You can be interrupted at any time
- No one program is going to block your whole system

Locking

- Race conditions
- Fine grained vs. coarse grained locking
- Lockless data structures

The Canonical Race Condition

- Process A sees that John's balance is $25
- Process B sees that John's balance is $25
- Process A deducts $25 and sets balance to $0
- Process B deducts $25 and sets balance to $0
- John gets a free $25
- ...which he promptly loses on a horse race

Locking with ”with”

```
with lock:
    if balance > amt:
        deduct(amt)
    else:
        raise ValueError('Insufficient funds')
```

Fine vs. Coarse Grained Locking

Fine vs. Coarse Grained Locking

- How many locks do you need?
- FreeBSD 4 ("giant") and Python ("the GIL") are coarse grained
- FreeBSD 5 and especially BeOS are fine grained
- Fine grained has some real benefits, but can be hard and expensive

Lockless Data Structures

- Some abstract data type implementations can do atomic updates without needing locks

Threads vs. Processes

- When UNIX first came out, it was just processes
- Processes don't share the same address space
- Multiple threads within a process do share the same address space

Which Should You Use?

- Apache 1 didn't use threads
- Apache 2 lets you mix and match processes and threads
- Processes are heavy
- Threads aren't free either

When Do You Need to Lock?

- Let's remember that you only need locking when there is sharing
- Immutable data doesn't need locking
- If each Web request is independent, you have fewer locking issues

Native Threads vs. Green Threads

- Native threads are implemented by the OS
- Green threads are implemented by the interpreter or VM
- OS threads are heavier
- Green threads can't use multiple CPUs
- Erlang uses multiple OS threads under the covers, but provides green threads to the user

Python's Native Threads

- Python has native threads, but there's a catch!
- The Global Interpreter Lock (GIL) only allows one thread at a time to ”use Python”
  - Blocking I/O doesn't ”use Python”
- Supporting native threads in a thread-safe way is hard

When to use Python Threads

- Python threads are great for I/O concurrency
- They're useless if you're CPU-bound on Python code
- Using Python threads for CPU-bound code on a multicore system will make your code *slower*!
- David Beazley explains why and how: http://blip.tv/file/2232410

Asynchronous Programming

- What happens if you have thousands of sockets that are in the wait state?
- "Call my callback when you actually have something for me"

Asynchronous Programming

- select, poll, kqueue, aio, and epoll
- Squid, nginx
- asyncore, Twisted, Tornado Web
- Stackless Python, eventlet, gevent

Notice that the biggest difference between nginx and Apache is that nginx is asynchronous whereas Apache uses processes or threads.

A Synchronous Example

```
def handle_request(request):
    data = talk_to_database()
    print "Processing request with data from database.
```

An Asynchronous Example

```
def handle_request(request):

    def use_data(data):
        print "Processing request with data from database."

    deferred = talk_to_database()
    deferred.addCallback(use_data)
```

An Example from Tornado Web

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        http = tornado.httpclient.AsyncHTTPClient()
        http.fetch("http://friendfeed-api.com/v2/feed/bret",
                   callback=self.async_callback(self.on_response))

    def on_response(self, response):
        if response.error: raise tornado.web.HTTPError(500)
        json = tornado.escape.json_decode(response.body)
        self.write("Fetched " + str(len(json["entries"])) + " entries "
                   "from the FriendFeed API")
        self.finish()
```

Continuation Passing Style

- Continuation passing style is when you have to use callbacks everywhere instead of using the stack
- It can be a bit painful
- It's pervasive throughout the code

WSGI

Continuation passing style and WSGI don't mix.

”Tornado comes with limited support for WSGI. However, since WSGI does not support non-blocking requests, you cannot use any of the asynchronous/non-blocking features of Tornado in your application if you choose to use WSGI instead of Tornado's HTTP server."

Subroutines vs. Coroutines

- Coroutines are just like subroutines, but they don't return
  - f() calls g() which calls h() which returns to g() which returns to f()
  - f() yields to g() which yields to h() which yields to f()
- Very simple cooperative multitasking

Generator-based Coroutines

```
import itertools

def my_coro(name):
    count = 0
    while True:
        count += 1
        print "%s %s" % (name, count)
        yield

coros = [my_coro('coro1'), my_coro('coro2')]
for coro in itertools.cycle(coros):  # A round-robin scheduler :)
    coro.next()

# Produces:
#
# coro1 1
# coro2 1
# coro1 2
# coro2 2
# ...
```

Generator-based Limitations

- You can only yield in the generator
- What happens if my_coro calls f and f wants to yield?
- http://jjinux.blogspot.com/2006/08/python-limitations-of-coroutines-via.html

Stackless

Stackless

- A stackless interpreter does not use the C stack in order to implement its own stack
- Some interpreters are, some aren't
- Python isn't, but, of course, there's a version that is stackless

Coping When you Aren't

- Getting around the generator limitations
- Stack copying
- Greenlets is a library to do stack copying
- Implementing your own green thread libraries

Custom Thread Libraries

- Why on Earth would you want a custom thread library?
- I/O-based schedulers
  - Looks like threads
  - Behaves like asynchronous code
  - Locking is simplified
- Yahoo Groups (i.e. eGroups), IronPort, Slide, and EVE Online all use this trick

gevent

- gevent is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libevent event loop.
  - convenient API around greenlets
  - socket module that cooperates
  - WSGI server on top of libevent-http
  - monkey patching utility to get pure Python modules to cooperate

An Example from gevent

```
import gevent
gevent.monkey.patch_socket()
import urllib2

def print_head(url):
    print 'Starting %s' % url
    data = urllib2.urlopen(url).read()  # No callback needed!
    print '%s: %s bytes: %r' % (url, len(data), data[:50])

urls = ['http://www.google.com', 'http://www.yandex.ru']
jobs = [gevent.spawn(print_head, url) for url in urls]
gevent.joinall(jobs)
```

Comet

Comet

”In web development, Comet is a neologism to describe a web application model in which a long-held HTTP request allows a web server to push data to a browser, without the browser explicitly requesting it.” [Wikipedia]

Building a Chat Application

- Whenever you implement comet, generally the first demo application you write is a chat application
- Most asynchronous web frameworks have such a demo
- This is something you can't do if you give each request its own thread or process–you can't have 10,000 threads or processes on the same box

Building a Chat Application

- Tornado Web's example: http://github.com/facebook/tornado/tree/master/der
- gevent's example: http://code.google.com/p/gevent/source/browse/#h (built on top of Django!)
- I built one back in 2006 using Aquarium, Dojo, and stackless Python ;)

Coping with MySQL

Coping with MySQL

- Generally, Web applications have to talk to a backend data store such as MySQL
- This is a real challenge for asynchronous frameworks since MySQL's database driver is written in C and isn't asynchronous
- If you block while talking to a database, you block *all* the requests, not just yours!

Coping with MySQL

- Twisted handles this by putting the MySQL connection in its own thread
- I don't see anything in Tornado Web to handle this, but I know they're using MySQL synchronously
- At IronPort, we had a proprietary Python driver built on top of Stackless Python
- MySQL Connector/Python is a pure Python driver that might be able to work asynchronously in gevent

Thanks for Coming

Thanks for Coming

The slides are already online:

http://jjinux.blogspot.com/2009/12/python-concurrency.html

Creative Commons Images

http://flickr.com/photos/19838221@N00/442349474/
http://www.flickr.com/photos/rene-germany/15571970/
http://www.flickr.com/photos/dvorscak/262939592/
http://www.flickr.com/photos/strandloper/360831841/
http://www.flickr.com/photos/fallenpegasus/2500679208/