An Introduction to Python Concurrency

David Beazley
http://www.dabeaz.com

Presented at USENIX Technical Conference
San Diego, June, 2009

This Tutorial

- Python : An interpreted high-level programming language that has a lot of support for "systems programming" and which integrates well with existing software in other languages.
- Concurrency : Doing more than one thing at a time. Of particular interest to programmers writing code for running on big iron, but also of interest for users of multicore PCs. Usually a bad idea--except when it's not.

Support Files

- Code samples and support files for this class

http://www.dabeaz.com/usenix2009/concurrent/

- Please go there and follow along

An Overview

- We're going to explore the state of concurrent programming idioms being used in Python
- A look at tradeoffs and limitations
- Hopefully provide some clarity
- A tour of various parts of the standard library
- Goal is to go beyond the user manual and tie everything together into a "bigger picture."

Disclaimers

- The primary focus is on Python
- This is not a tutorial on how to write concurrent programs or parallel algorithms
- No mathematical proofs involving "dining philosophers" or anything like that
- I will assume that you have had some prior exposure to topics such as threads, message passing, network programming, etc.

Disclaimers

- I like Python programming, but this tutorial is not meant to be an advocacy talk
- In fact, we're going to be covering some pretty ugly (e.g., "sucky") aspects of Python
- You might not even want to use Python by the end of this presentation
- That's fine... education is my main agenda.

Part I

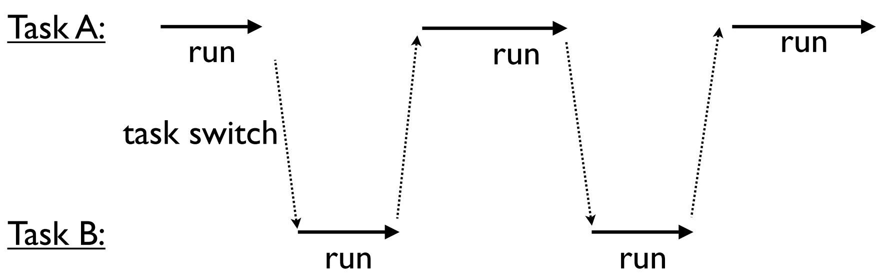
Some Basic Concepts

Concurrent Programming

- Creation of programs that can work on more than one thing at a time
- Example : A network server that communicates with several hundred clients all connected at once
- Example : A big number crunching job that spreads its work across multiple CPUs

Multitasking

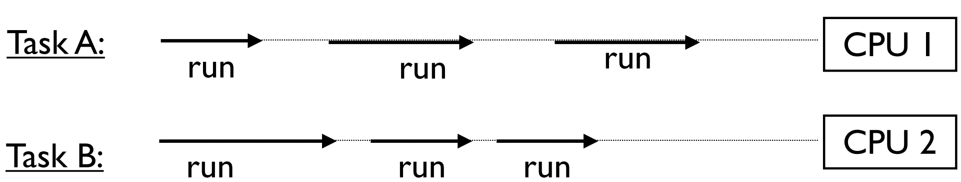
- Concurrency typically implies "multitasking"



- If only one CPU is available, the only way it can run multiple tasks is by rapidly switching between them

Parallel Processing

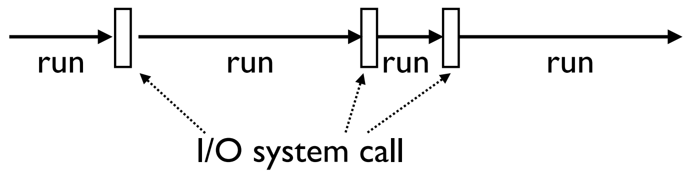
- You may have parallelism (many CPUs)
- Here, you often get <u>simultaneous</u> task execution



- Note: If the total number of tasks exceeds the number of CPUs, then each CPU also multitasks

Task Execution

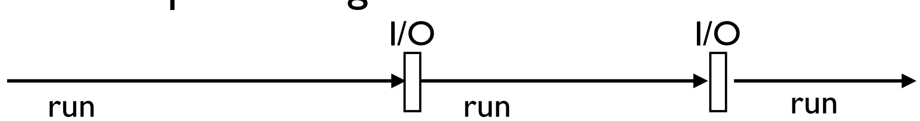
- All tasks execute by alternating between CPU processing and I/O handling



- For I/O, tasks must wait (sleep)
- Behind the scenes, the underlying system will carry out the I/O operation and wake the task when it's finished

CPU Bound Tasks

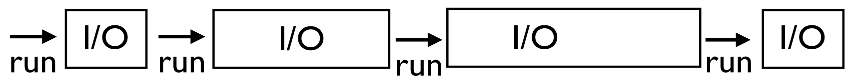
- A task is "CPU Bound" if it spends most of its time processing with little I/O



- Examples:
  - Crunching big matrices
  - Image processing

I/O Bound Tasks

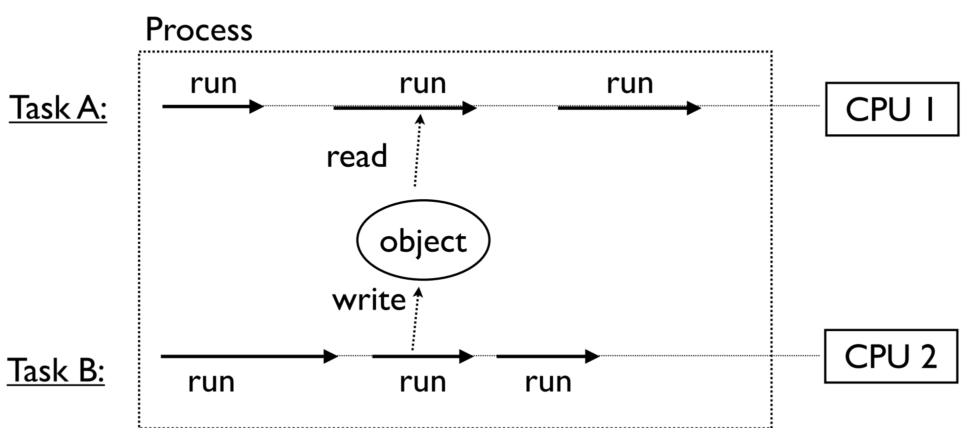
- A task is "I/O Bound" if it spends most of its time waiting for I/O



- Examples:
  - Reading input from the user
  - Networking
  - File processing
- Most "normal" programs are I/O bound

Shared Memory

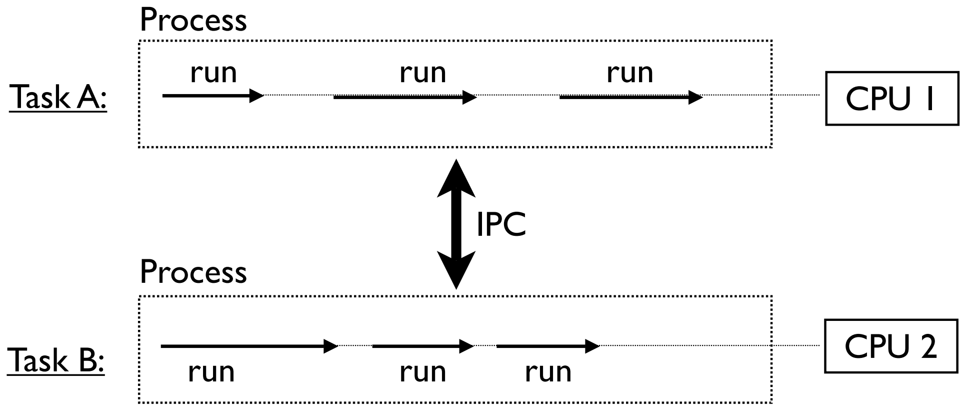
- Tasks may run in the same memory space



- Simultaneous access to objects
- Often a source of unspeakable peril

Processes

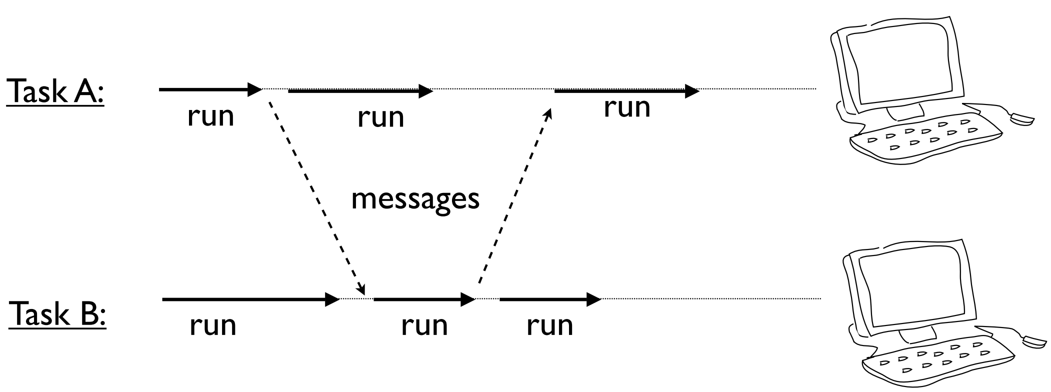
- Tasks might run in separate processes



- Processes coordinate using IPC
- Pipes, FIFOs, memory mapped regions, etc.

Distributed Computing

- Tasks may be running on distributed systems



- For example, a cluster of workstations
- Communication via sockets

Part 2

Why Concurrency and Python?

Some Issues

- Python is interpreted

  "What the hardware giveth, the software taketh away."

- Frankly, it doesn't seem like a natural match for any sort of concurrent programming
- Isn't concurrent programming all about high performance anyways???

Why Use Python at All?

- Python is a very high level language
- And it comes with a large library
    - Useful data types (dictionaries, lists,etc.)
    - Network protocols
    - Text parsing (regexs, XML, HTML, etc.)
    - Files and the file system
    - Databases
- Programmers like using this stuff...

Python as a Framework

- Python is often used as a high-level framework
- The various components might be a mix of languages (Python, C, C++, etc.)
- Concurrency may be a core part of the framework's overall architecture
- Python has to deal with it even if a lot of the underlying processing is going on in C

Programmer Performance

- Programmers are often able to get complex systems to "work" in much less time using a high-level language like Python than if they're spending all of their time hacking C code.

"*The best performance improvement is the transition from the nonworking to the working state.*"
- John Ousterhout

"*Premature optimization is the root of all evil.*"
- Donald Knuth

"*You can always optimize it later.*"
- Unknown

Performance is Irrelevant

- Many concurrent programs are "I/O bound"
- They spend virtually all of their time sitting around waiting
- Python can "wait" just as fast as C (maybe even faster--although I haven't measured it).
- If there's not much processing, who cares if it's being done in an interpreter? (One exception : if you need an extremely rapid response time as in real-time systems)

You Can Go Faster

- Python can be extended with C code
- Look at ctypes, Cython, Swig, etc.
- If you need really high-performance, you're not coding Python--you're using C extensions
- This is what most of the big scientific computing hackers are doing
- It's called "using the right tool for the job"

Commentary

- Concurrency is usually a really bad option if you're merely trying to make an inefficient Python script run faster
- Because its interpreted, you can often make huge gains by focusing on better algorithms or offloading work into C extensions
- For example, a C extension might make a script run 20x faster vs. the marginal improvement of parallelizing a slow script to run on a couple of CPU cores

Part 3

Python Thread Programming

Concept: Threads

- What most programmers think of when they hear about "concurrent programming"
- An independent task running inside a program
- Shares resources with the main program (memory, files, network connections, etc.)
- Has its own independent flow of execution (stack, current instruction, etc.)

Thread Basics

```
% python program.py
        statement
        statement
           ...
```

"main thread"

Program launch. Python loads a program and starts executing statements

Thread Basics

```
% python program.py
        statement
        statement
           ...
    create thread(foo)        def foo():
```

Creation of a thread. Launches a function.

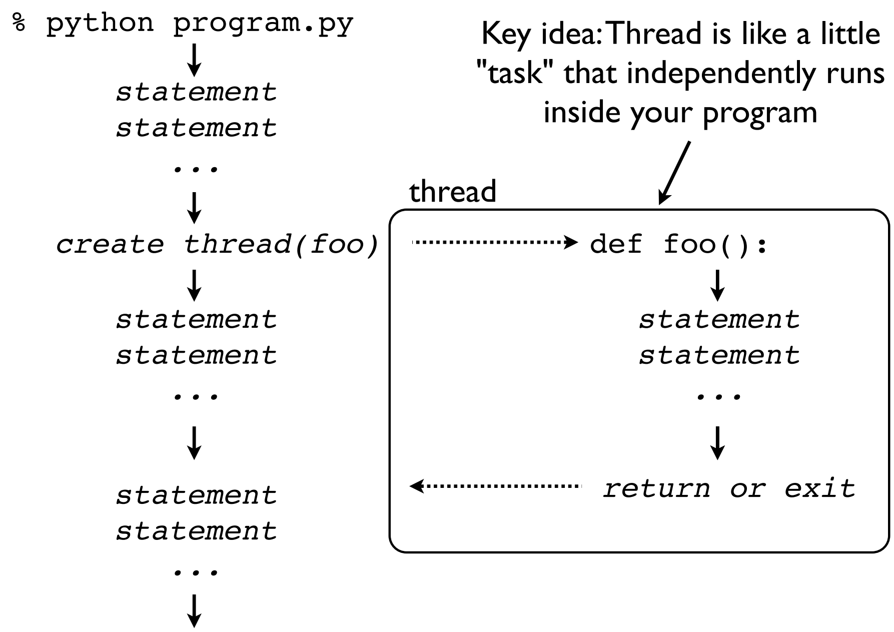
Thread Basics

```
% python program.py
          ↓
      statement
      statement
         ...
          ↓
  create thread(foo)  ·········→ def foo():
          ↓                             ↓
      statement      Concurrent     statement
      statement      execution      statement
         ...       of statements       ...
          ↓                             ↓
```


Thread Basics

```
% python program.py
          ↓
      statement
      statement
         ...
          ↓
  create thread(foo)  ·········→ def foo():
          ↓                             ↓
      statement                     statement
      statement                     statement
         ...       thread terminates   ...
          ↓        on return or exit    ↓
      statement     ←·········   return or exit
      statement
         ...
          ↓
```

Thread Basics



threading module

- Python threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

- You inherit from Thread and redefine run()

threading module

- Python threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

This code executes in the thread

- You inherit from Thread and redefine run()

threading module

- To launch, create thread objects and call start()

```
t1 = CountdownThread(10)  # Create the thread object
t1.start()                # Launch the thread

t2 = CountdownThread(20)  # Create another thread
t2.start()                # Launch
```

- Threads execute until the run() method stops

Functions as threads

- Alternative method of launching threads

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
```

- Creates a Thread object, but its run() method just calls the given function

Joining a Thread

- Once you start a thread, it runs independently
- Use t.join() to wait for a thread to exit

```
t.start()           # Launch a thread
...
# Do other work
...
# Wait for thread to finish
t.join()            # Waits for thread t to exit
```

- This only works from *other* threads
- A thread can't join itself

Daemonic Threads

- If a thread runs forever, make it "daemonic"

```
t.daemon = True
t.setDaemon(True)
```

- If you don't do this, the interpreter will lock when the main thread exits---waiting for the thread to terminate (which never happens)
- Normally you use this for background tasks

Interlude

- Creating threads is really easy
- You can create thousands of them if you want
- Programming with threads is hard
- Really hard

***Q**: Why did the multithreaded chicken cross the road?*

***A:** to To other side. get the*

-- Jason Whittington

Access to Shared Data

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic (which is a really good way to have your head explode)

Accessing Shared Data

- Consider a shared object

```
x = 0
```

- And two threads that modify it

```
Thread-1                        Thread-2
--------                        --------
...                             ...
x = x + 1                       x = x - 1
...                             ...
```

- It's possible that the resulting value will be unpredictably corrupted

Accessing Shared Data

- The two threads

```
Thread-1                    Thread-2
--------                    --------
...                         ...
x = x + 1                   x = x - 1
...                         ...
```

- Low level interpreter execution

```
Thread-1                             Thread-2
--------                             --------
   ↓
LOAD_GLOBAL   1 (x)
LOAD_CONST    2 (1)
                     thread switch →
                                     LOAD_GLOBAL   1 (x)
                                     LOAD_CONST    2 (1)
                                     BINARY_SUB
                                     STORE_GLOBAL  1 (x)
                     ← thread switch
BINARY_ADD
STORE_GLOBAL 1 (x)
```


Accessing Shared Data

- Low level interpreter code

```
Thread-1                             Thread-2
--------                             --------
   ↓
LOAD_GLOBAL   1 (x)
LOAD_CONST    2 (1)
                     thread switch →
                                     LOAD_GLOBAL   1 (x)
                                     LOAD_CONST    2 (1)
                                     BINARY_SUB
                                     STORE_GLOBAL  1 (x)
                     ← thread switch
BINARY_ADD
STORE_GLOBAL 1 (x)
```

These operations get performed with a "stale" value of x. The computation in Thread-2 is lost.

Accessing Shared Data

- Is this actually a real concern?

```
x = 0           # A shared value
def foo():
     global x
     for i in xrange(100000000): x += 1

def bar():
     global x
     for i in xrange(100000000): x -= 1

t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join(); t2.join()   # Wait for completion
print x                # Expected result is 0
```

- Yes, the print produces a random nonsensical value each time (e.g., -83412 or 1627732)

Race Conditions

- The corruption of shared data due to thread scheduling is often known as a "race condition."
- It's often quite diabolical--a program may produce slightly different results each time it runs (even though you aren't using any random numbers)
- Or it may just flake out mysteriously once every two weeks

Thread Synchronization

- Identifying and fixing a race condition will make you a better programmer (e.g., it "builds character")
- However, you'll probably never get that month of your life back...
- To fix : You have to synchronize threads

Part 4

Thread Synchronization Primitives

Synchronization Options

- The threading library defines the following objects for synchronizing threads
  - Lock
  - RLock
  - Semaphore
  - BoundedSemaphore
  - Event
  - Condition

Synchronization Options

- In my experience, there is often a lot of confusion concerning the intended use of the various synchronization objects
- Maybe because this is where most students "space out" in their operating system course (well, yes actually)
- Anyways, let's take a little tour

Mutex Locks

- Mutual Exclusion Lock

```
m = threading.Lock()
```

- Probably the most commonly used synchronization primitive
- Primarily used to synchronize threads so that only one thread can make modifications to shared data at any given time

Mutex Locks

- There are two basic operations

```
m.acquire()                 # Acquire the lock
m.release()                 # Release the lock
```

- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

Use of Mutex Locks

- Commonly used to enclose critical sections

```
x = 0
x_lock = threading.Lock()

Thread-1                              Thread-2
--------                              --------
...                                   ...
x_lock.acquire()                      x_lock.acquire()

x = x + 1                             x = x - 1

x_lock.release()                      x_lock.release()
...                                   ...
```

Critical Section

- Only one thread can execute in critical section at a time (lock gives exclusive access)

Using a Mutex Lock

- It is <u>your</u> responsibility to identify and lock <u>all</u> "critical sections"

```
x = 0
x_lock = threading.Lock()

Thread-1                              Thread-2
--------                              --------
...                                   ...
x_lock.acquire()                      x = x - 1
x = x + 1                             ...
x_lock.release()
...
```

If you use a lock in one place, but not another, then you're missing the whole point. <u>All modifications to shared state must be enclosed by lock acquire()/release()</u>.

Locking Perils

- Locking looks straightforward
- Until you start adding it to your code
- Managing locks is a lot harder than it looks

Lock Management

- Acquired locks must always be released
- However, it gets evil with exceptions and other non-linear forms of control-flow
- Always try to follow this prototype:

```
x = 0
x_lock = threading.Lock()

# Example critical section
x_lock.acquire()
try:
    statements using x
finally:
    x_lock.release()
```

Lock Management

- Python 2.6/3.0 has an improved mechanism for dealing with locks and critical sections

```
x = 0
x_lock = threading.Lock()

# Critical section
with x_lock:
    statements using x
...
```

- This automatically acquires the lock and releases it when control enters/exits the associated block of statements

Locks and Deadlock

- Don't write code that acquires more than one mutex lock at a time

```
x = 0
y = 0
x_lock = threading.Lock()
y_lock = threading.Lock()

with x_lock:
    statements using x
    ...
    with y_lock:
        statements using x and y
        ...
```

- This almost invariably ends up creating a program that mysteriously deadlocks (even more fun to debug than a race condition)

RLock

- Reentrant Mutex Lock

```
m = threading.RLock()      # Create a lock
m.acquire()                # Acquire the lock
m.release()                # Release the lock
```

- Similar to a normal lock except that it can be reacquired multiple times by the same thread
- However, each acquire() must have a release()
- Common use : Code-based locking (where you're locking function/method execution as opposed to data access)

RLock Example

- Implementing a kind of "monitor" object

```
class Foo(object):
    lock = threading.RLock()
    def bar(self):
        with Foo.lock:
            ...
    def spam(self):
        with Foo.lock:
             ...
             self.bar()
             ...
```

- Only one thread is allowed to execute methods in the class at any given time
- However, methods can call other methods that are holding the lock (in the same thread)

Semaphores

- A counter-based synchronization primitive

```
m = threading.Semaphore(n) # Create a semaphore
m.acquire()                # Acquire
m.release()                # Release
```

- acquire() - Waits if the count is 0, otherwise decrements the count and continues
- release() - Increments the count and signals waiting threads (if any)
- Unlike locks, acquire()/release() can be called in any order and by any thread

Semaphore Uses

- Resource control. You can limit the number of threads performing certain operations. For example, performing database queries, making network connections, etc.
- Signaling. Semaphores can be used to send "signals" between threads. For example, having one thread wake up another thread.

Resource Control

- Using a semaphore to limit resources

```
sema = threading.Semaphore(5)    # Max: 5-threads

def fetch_page(url):
    sema.acquire()
    try:
        u = urllib.urlopen(url)
        return u.read()
    finally:
        sema.release()
```

- In this example, only 5 threads can be executing the function at once (if there are more, they will have to wait)

Thread Signaling

- Using a semaphore to signal

```
done = threading.Semaphore(0)
```

<u>Thread 1</u>

```
...
statements
statements
statements
done.release()
```

<u>Thread 2</u>

```
done.acquire()
statements
statements
statements
...
```

- Here, acquire() and release() occur in <u>different</u> threads and in a different order
- Often used with producer-consumer problems

Events

- Event Objects

```
e = threading.Event()
e.isSet()          # Return True if event set
e.set()            # Set event
e.clear()          # Clear event
e.wait()           # Wait for event
```

- This can be used to have one or more threads wait for something to occur
- Setting an event will unblock all waiting threads simultaneously (if any)
- Common use : barriers, notification

Event Example

- Using an event to ensure proper initialization

```
init = threading.Event()

def worker():
    init.wait()      # Wait until initialized
    statements
    ...

def initialize():
    statements       # Setting up
    statements       # ...
    ...
    init.set()       # Done initializing

Thread(target=worker).start()    # Launch workers
Thread(target=worker).start()
Thread(target=worker).start()
initialize()                     # Initialize
```

Event Example

- Using an event to signal "completion"

```
def master():
    ...
    item = create_item()
    evt = Event()
    worker.send((item,evt))
    ...
    # Other processing
    ...
    ...
    ...
    ...
    ...
    # Wait for worker
    evt.wait()
```

Worker Thread

```
item, evt = get_work()
processing
processing
...
...
# Done
evt.set()
```

- Might use for asynchronous processing, etc.

Condition Variables

- Condition Objects

```
cv = threading.Condition([lock])
cv.acquire()     # Acquire the underlying lock
cv.release()     # Release the underlying lock
cv.wait()        # Wait for condition
cv.notify()      # Signal that a condition holds
cv.notifyAll()   # Signal all threads waiting
```

- A combination of locking/signaling
- Lock is used to protect code that establishes some sort of "condition" (e.g., data available)
- Signal is used to notify other threads that a "condition" has changed state

Condition Variables

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
```

Consumer Thread

```
with items_cv:
    ...
    x = items.pop(0)

# Do something with x
...
```

- First, you use the locking part of a CV synchronize access to shared data (items)

Condition Variables

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
    items_cv.notify()
```

Consumer Thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)

# Do something with x
...
```

- Next you add signaling and waiting
- Here, the producer signals the consumer that it put data into the shared list

Condition Variables

- Some tricky bits involving wait()
- Before waiting, you have to acquire the lock
- wait() releases the lock when waiting and reacquires when woken

Consumer Thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)

# Do something with x
...
```

- Conditions are often transient and may not hold by the time wait() returns. So, you must always double-check (hence, the while loop)

Interlude

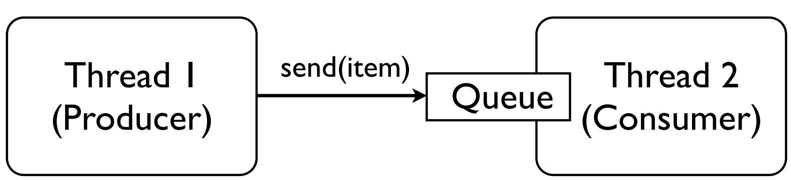
- Working with all of the synchronization primitives is a lot trickier than it looks
- There are a lot of nasty corner cases and horrible things that can go wrong
- Bad performance, deadlock, livelock, starvation, bizarre CPU scheduling, etc...
- All are valid reasons to <u>not</u> use threads

Part 5

Threads and Queues

Threads and Queues

- Threaded programs are often easier to manage if they can be organized into producer/consumer components connected by queues



- Instead of "sharing" data, threads only coordinate by sending data to each other
- Think Unix "pipes" if you will...

Queue Library Module

- Python has a thread-safe queuing module
- Basic operations

```
from Queue import Queue

q = Queue([maxsize])    # Create a queue
q.put(item)             # Put an item on the queue
q.get()                 # Get an item from the queue
q.empty()               # Check if empty
q.full()                # Check if full
```

- Usage : You try to strictly adhere to get/put operations. If you do this, you don't need to use other synchronization primitives.

Queue Usage

- Most commonly used to set up various forms of producer/consumer problems

```
from Queue import Queue
q = Queue()
```

Producer Thread

```
for item in produce_items():
    q.put(item)
```

Consumer Thread

```
while True:
    item = q.get()
    consume_item(item)
```

- Critical point : You don't need locks here

Queue Signaling

- Queues also have a signaling mechanism

```
q.task_done()      # Signal that work is done
q.join()           # Wait for all work to be done
```

- Many Python programmers don't know about this (since it's relatively new)
- Used to determine when processing is done

Producer Thread

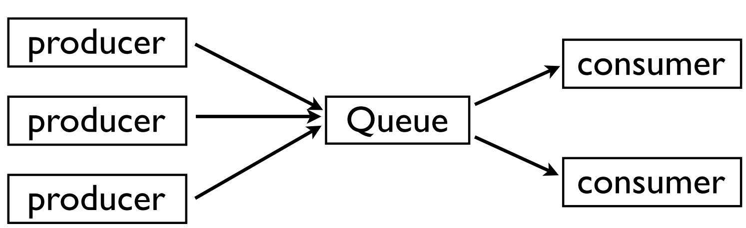
```
for item in produce_items():
    q.put(item)
# Wait for consumer
q.join()
```

Consumer Thread

```
while True:
    item = q.get()
    consume_item(item)
    q.task_done()
```


Queue Programming

- There are many ways to use queues
- You can have as many consumers/producers as you want hooked up to the same queue



- In practice, try to keep it simple

Part 6

The Problem with Threads

An Inconvenient Truth

- Thread programming quickly gets hairy
- End up with a huge mess of shared data, locks, queues, and other synchronization primitives
- Which is really unfortunate because Python threads have some major limitations
- Namely, they have pathological performance!

A Performance Test

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000)
```

- Threaded execution

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
```

- Now, you might expect two threads to run twice as fast on multiple CPU cores

Bizarre Results

- Performance comparison (Dual-Core 2Ghz Macbook, OS-X 10.5.6)

Sequential : 24.6s
Threaded : 45.5s (1.8X slower!)

- If you disable one of the CPU cores...

Threaded : 38.0s

- Insanely horrible performance. Better performance with fewer CPU cores? It makes no sense.

Interlude

- It's at this point that programmers often decide to abandon threads altogether
- Or write a blog rant that vaguely describes how Python threads "suck" because of their failed attempt at Python supercomputing
- Well, yes there is definitely some "suck" going on, but let's dig a little deeper...

Part 7

The Inside Story on Python Threads

"*The horror! The horror!*" - Col. Kurtz

What is a Thread?

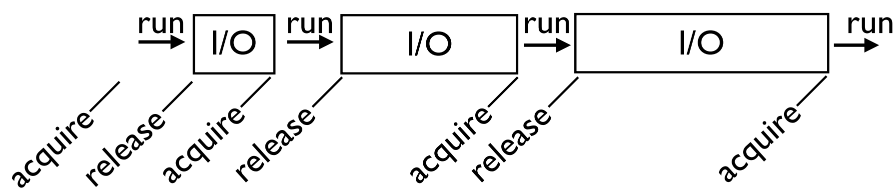
- Python threads are <u>real</u> system threads
    - POSIX threads (pthreads)
    - Windows threads
- Fully managed by the host operating system
    - All scheduling/thread switching
- Represent threaded execution of the Python interpreter process (written in C)

The Infamous GIL

- Here's the rub...
- Only one Python thread can execute in the interpreter at once
- There is a "global interpreter lock" that carefully controls thread execution
- The GIL ensures that sure each thread gets <u>exclusive</u> access to the entire interpreter internals when it's running

GIL Behavior

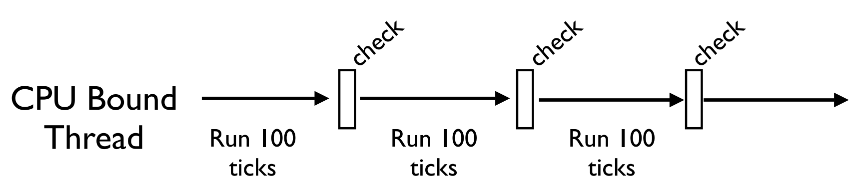
- Whenever a thread runs, it holds the GIL
- However, the GIL is released on blocking I/O



- So, any time a thread is forced to wait, other "ready" threads get their chance to run
- Basically a kind of "cooperative" multitasking

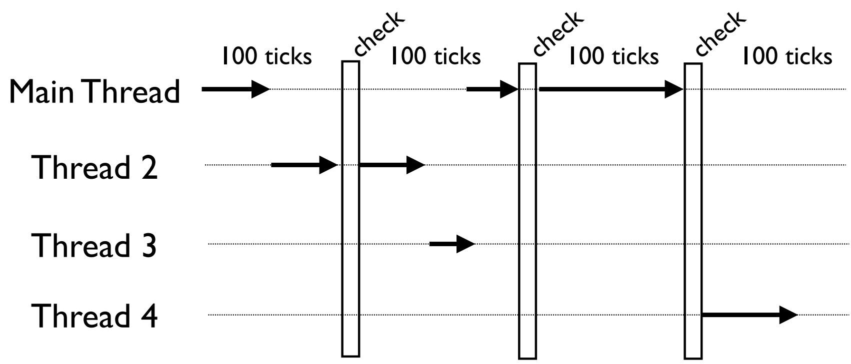
CPU Bound Processing

- To deal with CPU-bound threads, the interpreter periodically performs a "check"
- By default, every 100 interpreter "ticks"



The Check Interval

- The check interval is a global counter that is completely independent of thread scheduling



- A "check" is simply made every 100 "ticks"

The Periodic Check

- What happens during the periodic check?
    - In the main thread only, signal handlers will execute if there are any pending signals
    - Release and reacquisition of the GIL
- That last bullet describes how multiple CPU-bound threads get to run (by briefly releasing the GIL, other threads get a chance to run).

What is a "Tick?"

- Ticks loosely map to interpreter instructions

```
def countdown(n):
    while n > 0:
        print n
        n -= 1
```

- Instructions in the Python VM

```
        >>> import dis
        >>> dis.dis(countdown)
        0 SETUP_LOOP              33 (to 36)
        3 LOAD_FAST                0 (n)
        6 LOAD_CONST               1 (0)
        9 COMPARE_OP               4 (>)
Tick 1  12 JUMP_IF_FALSE          19 (to 34)
        15 POP_TOP
        16 LOAD_FAST               0 (n)
        19 PRINT_ITEM
Tick 2  20 PRINT_NEWLINE
        21 LOAD_FAST               0 (n)
Tick 3  24 LOAD_CONST              2 (1)
        27 INPLACE_SUBTRACT
        28 STORE_FAST              0 (n)
Tick 4  31 JUMP_ABSOLUTE           3
        ...
```


Tick Execution

- Interpreter ticks are not time-based
- Ticks don't have consistent execution times
- Long operations can block everything

```
>>> nums = xrange(100000000)
>>> -1 in nums                 ——→ 1 tick (~ 6.6 seconds)
False
>>>
```

- Try hitting Ctrl-C (ticks are uninterruptible)

```
>>> nums = xrange(100000000)
>>> -1 in nums
^C^C^C   (nothing happens, long pause)
...
KeyboardInterrupt
>>>
```

Thread Scheduling

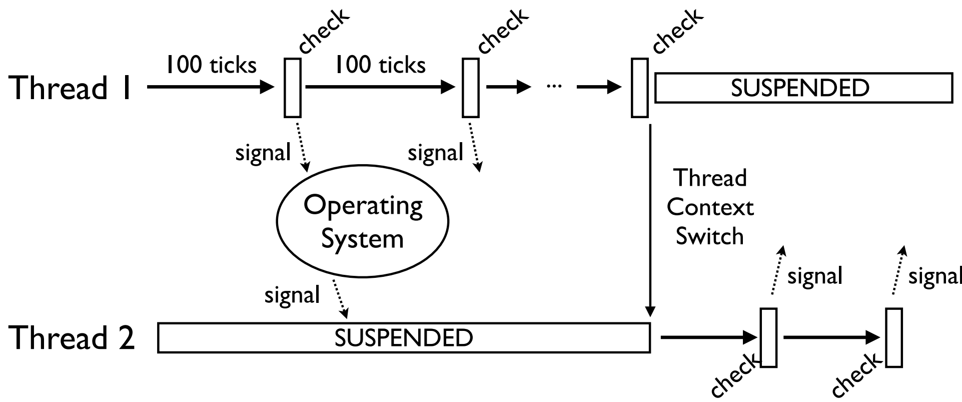
- Python does not have a thread scheduler
- There is no notion of thread priorities, preemption, round-robin scheduling, etc.
- For example, the list of threads in the interpreter isn't used for anything related to thread execution
- All thread scheduling is left to the host operating system (e.g., Linux, Windows, etc.)

GIL Implementation

- The GIL is not a simple mutex lock
- The implementation (Unix) is either...
  - A POSIX unnamed semaphore
  - Or a pthreads condition variable
- All interpreter locking is based on signaling
  - To acquire the GIL, check if it's free. If not, go to sleep and wait for a signal
  - To release the GIL, free it and signal

Thread Scheduling

- Thread switching is far more subtle than most programmers realize (it's tied up in the OS)



- The lag between signaling and scheduling may be significant (depends on the OS)

CPU-Bound Threads

- As we saw earlier, CPU-bound threads have horrible performance properties
- Far worse than simple sequential execution
  - 24.6 seconds (sequential)
  - 45.5 seconds (2 threads)
- A big question : Why?
  - What is the source of that overhead?

Signaling Overhead

- GIL thread signaling is the source of that
- After every 100 ticks, the interpreter
  - Locks a mutex
  - Signals on a condition variable/semaphore where another thread is always waiting
  - Because another thread is waiting, extra pthreads processing and system calls get triggered to deliver the signal

A Rough Measurement

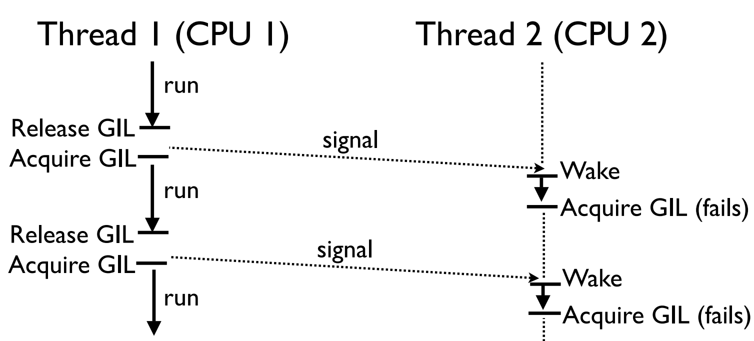
- Sequential Execution (OS-X, 1 CPU)
  - 736 Unix system calls
  - 117 Mach System Calls
- Two threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~ 3.3 Million Mach System Calls
- Yow! Look at that last figure.

Multiple CPU Cores

- The penalty gets far worse on multiple cores
- Two threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~3.3 Million Mach System Calls
- Two threads (OS-X, 2 CPUs)
  - 1149 Unix system calls
  - ~9.5 Million Mach System calls

Multicore GIL Contention

- With multiple cores, CPU-bound threads get scheduled simultaneously (on different processors) and then have a GIL battle



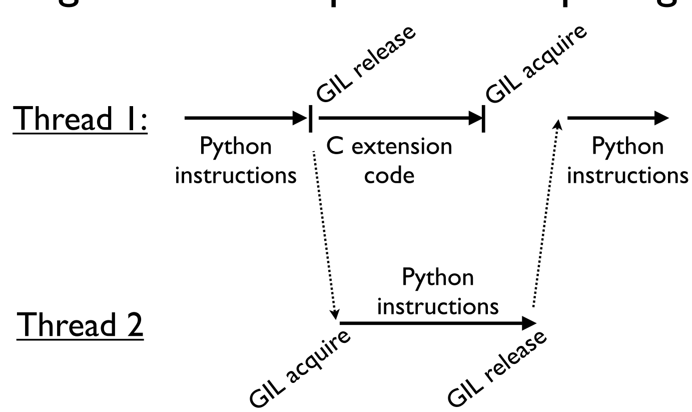
- The waiting thread (T2) may make 100s of failed GIL acquisitions before any success

The GIL and C Code

- As mentioned, Python can talk to C/C++
- C/C++ extensions can release the interpreter lock and run independently
- Caveat : Once released, C code shouldn't do any processing related to the Python interpreter or Python objects
- The C code itself must be thread-safe

The GIL and C Extensions

- Having C extensions release the GIL is how you get into true "parallel computing"



How to Release the GIL

- The ctypes module already releases the GIL when calling out to C code
- In hand-written C extensions, you have to insert some special macros

```
PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code
   ...
   Py_END_ALLOW_THREADS
   ...
}
```


The GIL and C Extensions

- The trouble with C extensions is that you have to make sure they do enough work
- A dumb example (mindless spinning)

```
void churn(int n) {
   while (n > 0) {
      n--;
   }
}
```

- How big do you have to make n to actually see any kind of speedup on multiple cores?

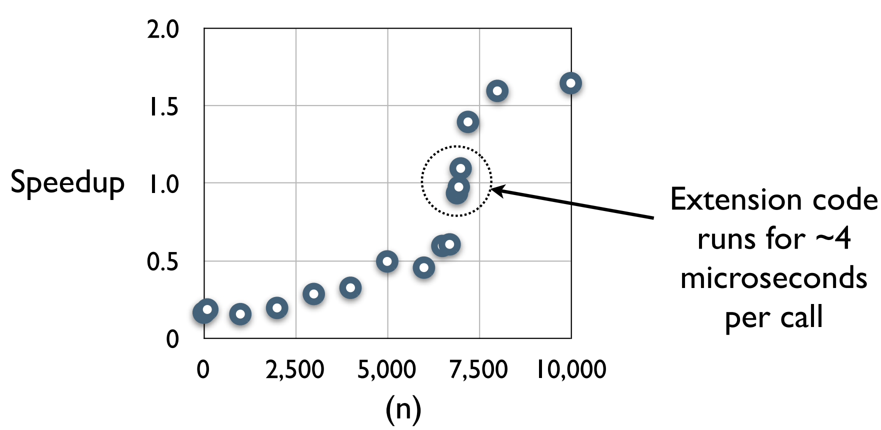
The GIL and C Extensions

- Here's some Python test code

```
def churner(n):
   count = 1000000
   while count > 0:
       churn(n)       # C extension function
       count -= 1

# Sequential execution
churner(n)
churner(n)

# Threaded execution
t1 = threading.Thread(target=churner, args=(n,))
t2 = threading.Thread(target=churner, args=(n,))
t1.start()
t2.start()
```


The GIL and C Extensions

- Speedup of running two threads versus sequential execution



- Note: 2 Ghz Intel Core Duo, OS-X 10.5.6

Why is the GIL there?

- Simplifies the implementation of the Python interpreter (okay, sort of a lame excuse)
- Better suited for reference counting (Python's memory management scheme)
- Simplifies the use of C/C++ extensions. Extension functions do not need to worry about thread synchronization
- And for now, it's here to stay... (although people continue to try and eliminate it)

Part 8

Final Words on Threads

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Using Threads

- Despite some "issues," there are situations where threads are appropriate and where they perform well
- There are also some tuning parameters

I/O Bound Processing

- Threads are still useful for I/O-bound apps
- For example : A network server that needs to maintain several thousand long-lived TCP connections, but is not doing tons of heavy CPU processing
- Here, you're really only limited by the host operating system's ability to manage and schedule a lot of threads
- Most systems don't have much of a problem-- even with thousands of threads

Why Threads?

- If everything is I/O-bound, you will get a very quick response time to any I/O activity
- Python isn't doing the scheduling
- So, Python is going to have a similar response behavior as a C program with a lot of I/O bound threads
- Caveat: You have to stay I/O bound!

Final Comments

- Python threads are a useful tool, but you have to know how and when to use them
    - I/O bound processing only
    - Limit CPU-bound processing to C extensions (that release the GIL)
- Threads are not the only way...

Part 9

Processes and Messages

Concept: Message Passing

- An alternative to threads is to run multiple independent copies of the Python interpreter
- In separate processes
- Possibly on different machines
- Get the different interpreters to cooperate by having them send messages to each other

Message Passing

- On the surface, it's simple
- Each instance of Python is independent
- Programs just send and receive messages
- Two main issues
  - What is a message?
  - What is the transport mechanism?

Messages

- A message is just a bunch of bytes (a buffer)
- A "serialized" representation of some data
- Creating serialized data in Python is easy

pickle Module

- A module for serializing objects
- Serializing an object onto a "file"

```
import pickle
...
pickle.dump(someobj,f)
```

- Unserializing an object from a file

```
someobj = pickle.load(f)
```

- Here, a file might be a file, a pipe, a wrapper around a socket, etc.

pickle Module

- Pickle can also turn objects into byte strings

```
import pickle
# Convert to a string
s = pickle.dumps(someobj)
...
# Load from a string
someobj = pickle.loads(s)
```

- You might use this embed a Python object into a message payload

cPickle vs pickle

- There is an alternative implementation of pickle called cPickle (written in C)
- Use it whenever possible--it is much faster

```
import cPickle as pickle
...
pickle.dump(someobj,f)
```

- There is some history involved. There are a few things that cPickle can't do, but they are somewhat obscure (so don't worry about it)

Pickle Commentary

- Using pickle is almost too easy
- Almost any Python object works
    - Builtins (lists, dicts, tuples, etc.)
    - Instances of user-defined classes
    - Recursive data structures
- Exceptions
    - Files and network connections
    - Running generators, etc.

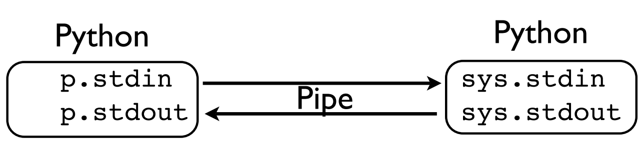
Message Transport

- Python has various low-level mechanisms
  - Pipes
  - Sockets
  - FIFOs
- Libraries provide access to other systems
  - MPI
  - XML-RPC (and many others)

An Example

- Launching a subprocess and hooking up the child process via a pipe
- Use the subprocess module

```
import subprocess

p = subprocess.Popen(['python','child.py'],
                     stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE)

p.stdin.write(data)    # Send data to subprocess
p.stdout.read(size)    # Read data from subprocess
```



Pipes and Pickle

- Most programmers would use the subprocess module to run separate programs and collect their output (e.g., system commands)
- However, if you put a pickling layer around the files, it becomes much more interesting
- Becomes a communication channel where you can send just about any Python object

A Message Channel

- A class that wraps a pair of files

```
# channel.py
import pickle

class Channel(object):
    def __init__(self,out_f,in_f):
        self.out_f = out_f
        self.in_f  = in_f
    def send(self,item):
        pickle.dump(item,self.out_f)
        self.out_f.flush()
    def recv(self):
        return pickle.load(self.in_f)
```

- Send/Receive implemented using pickle

Some Sample Code

- A sample child process

```
# child.py
import channel
import sys

ch = channel.Channel(sys.stdout,sys.stdin)
while True:
    item = ch.recv()
    ch.send(("child",item))
```

- Parent process setup

```
# parent.py
import channel
import subprocess

p = subprocess.Popen(['python','child.py'],
                     stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE)
ch = channel.Channel(p.stdin,p.stdout)
```


Some Sample Code

- Using the child worker

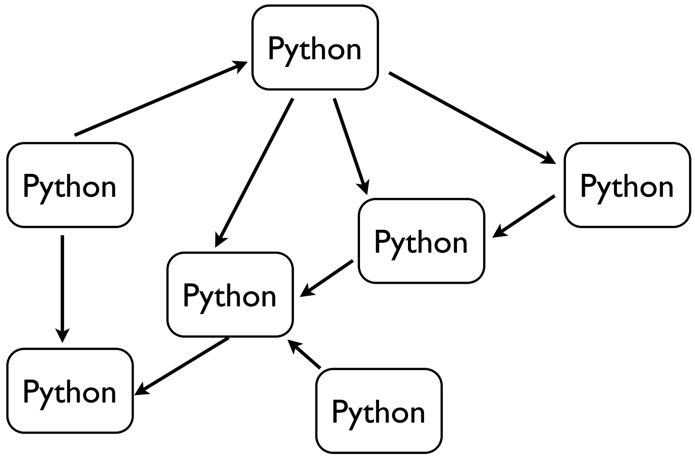
```
>>> ch.send("Hello World")
Hello World
>>> ch.send(42)
42
>>> ch.send([1,2,3,4])
[1, 2, 3, 4]
>>> ch.send({'host':'python.org','port':80})
{'host': 'python.org', 'port': 80}
>>>
```

This output is being produced by the child

- You can send almost any Python object (numbers, lists, dictionaries, instances, etc.)

Big Picture

- Can easily have 10s-1000s of communicating Python interpreters



Interlude

- Message passing is a fairly general concept
- However, it's also kind of nebulous in Python
- No agreed upon programming interface
- Vast number of implementation options
- Intersects with distributed objects, RPC, cross-language messaging, etc.

Part 10

The Multiprocessing Module

multiprocessing Module

- A new library module added in Python 2.6
- Originally known as pyprocessing (a third-party extension module)
- This is a module for writing concurrent Python programs based on communicating processes
- A module that is especially useful for concurrent CPU-bound processing

Using multiprocessing

- Here's the cool part...
- You already know how to use multiprocessing
- At a very high-level, it simply mirrors the thread programming interface
- Instead of "Thread" objects, you now work with "Process" objects.

multiprocessing Example

- Define tasks using a Process class

```
import time
import multiprocessing

class CountdownProcess(multiprocessing.Process):
    def __init__(self,count):
        multiprocessing. Process.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

- You inherit from Process and redefine run()

Launching Processes

- To launch, same idea as with threads

```
if __name__ == '__main__':
   p1 = CountdownProcess(10)  # Create the process object
   p1.start()                 # Launch the process

   p2 = CountdownProcess(20)   # Create another process
   p2.start()                  # Launch
```

- Processes execute until run() stops
- A critical detail : Always launch in main as shown (required for Windows)

Functions as Processes

- Alternative method of launching processes

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

if __name__ == '__main__':
    p1 = multiprocessing.Process(target=countdown,
                                 args=(10,))
p1.start()
```

- Creates a Process object, but its run() method just calls the given function

Does it Work?

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000)
```

→ 24.6s

- Multiprocessing Execution

```
p1 = Process(target=count,args=(100000000,))
p1.start()
p2 = Process(target=count,args=(100000000,))
p2.start()
```

→ 12.5s

- Yes, it seems to work

Other Process Features

- Joining a process (waits for termination)

```
p = Process(target=somefunc)
p.start()
...
p.join()
```

- Making a daemonic process

```
p = Process(target=somefunc)
p.daemon = True
p.start()
```

- Terminating a process

```
p = Process(target=somefunc)
...
p.terminate()
```

- These mirror similar thread functions

Distributed Memory

- With multiprocessing, there are no shared data structures
- Every process is completely isolated
- Since there are no shared structures, forget about all of that locking business
- Everything is focused on messaging

Pipes

- A channel for sending/receiving objects

```
(c1, c2) = multiprocessing.Pipe()
```

- Returns a pair of connection objects (one for each end-point of the pipe)
- Here are methods for communication

```
c.send(obj)           # Send an object
c.recv()              # Receive an object

c.send_bytes(buffer)  # Send a buffer of bytes
c.recv_bytes([max])   # Receive a buffer of bytes

c.poll([timeout])     # Check for data
```

Using Pipes

- The Pipe() function largely mimics the behavior of Unix pipes
- However, it operates at a higher level
- It's not a low-level byte stream
- You send discrete messages which are either Python objects (pickled) or buffers

Pipe Example

- A simple data consumer

```
def consumer(p1, p2):
    p1.close()   # Close producer's end (not used)
    while True:
        try:
            item = p2.recv()
        except EOFError:
            break
        print item      # Do other useful work here
```

- A simple data producer

```
def producer(sequence, output_p):
    for item in sequence:
        output_p.send(item)
```

Pipe Example

```
if __name__ == '__main__':
    p1, p2 = multiprocessing.Pipe()

    cons = multiprocessing.Process(
               target=consumer,
               args=(p1,p2))
    cons.start()

    # Close the input end in the producer
    p2.close()

    # Go produce some data
    sequence = xrange(100)  # Replace with useful data
    producer(sequence, p1)

    # Close the pipe
    p1.close()
```


Message Queues

- multiprocessing also provides a queue
- The programming interface is the same

```
from multiprocessing import Queue

q = Queue()
q.put(item)      # Put an item on the queue
item = q.get()   # Get an item from the queue
```

- There is also a joinable Queue

```
from multiprocessing import JoinableQueue

q = JoinableQueue()
q.task_done()    # Signal task completion
q.join()         # Wait for completion
```

Queue Implementation

- Queues are implemented on top of pipes
- A subtle feature of queues is that they have a "feeder thread" behind the scenes
- Putting an item on a queue returns immediately (allowing the producer to keep working)
- The feeder thread works on its own to transmit data to consumers

Queue Example

- A consumer process

```
def consumer(input_q):
    while True:
        # Get an item from the queue
        item = input_q.get()
        # Process item
        print item
        # Signal completion
        input_q.task_done()
```

- A producer process

```
def producer(sequence,output_q):
    for item in sequence:
        # Put the item on the queue
        output_q.put(item)
```

Queue Example

- Running the two processes

```
if __name__ == '__main__':
    from multiprocessing import Process, JoinableQueue
    q = JoinableQueue()

    # Launch the consumer process
    cons_p = Process(target=consumer,args=(q,))
    cons_p.daemon = True
    cons_p.start()

    # Run the producer function on some data
    sequence = range(100)    # Replace with useful data
    producer(sequence,q)

    # Wait for the consumer to finish
    q.join()
```


Commentary

- If you have written threaded programs that strictly stick to the queuing model, they can probably be ported to multiprocessing
- The following restrictions apply
  - Only objects compatible with pickle can be queued
  - Tasks can not rely on any shared data other than a reference to the queue

Other Features

- multiprocessing has many other features
  - Process Pools
  - Shared objects and arrays
  - Synchronization primitives
  - Managed objects
  - Connections
- Will briefly look at one of them

Process Pools

- Creating a process pool

```
p = multiprocessing.Pool([numprocesses])
```

- Pools provide a high-level interface for executing functions in worker processes
- Let's look at an example...

Pool Example

- Define a function that does some work
- Example : Compute a SHA-512 digest of a file

```
import hashlib

def compute_digest(filename):
    digest = hashlib.sha512()
    f = open(filename,'rb')
    while True:
        chunk = f.read(8192)
        if not chunk: break
        digest.update(chunk)
    f.close()
    return digest.digest()
```

- This is just a normal function (no magic)

Pool Example

- Here is some code that uses our function
- Make a dict mapping filenames to digests

```
import os
TOPDIR = "/Users/beazley/Software/Python-3.0"

digest_map = {}
for path, dirs, files in os.walk(TOPDIR):
    for name in files:
        fullname = os.path.join(path,name)
        digest_map[fullname] = compute_digest(fullname)
```

- Running this takes about 10s on my machine

Pool Example

- With a pool, you can farm out work
- Here's a small sample

```
p = multiprocessing.Pool(2)    # 2 processes

result = p.apply_async(compute_digest,('README.txt',))
...
... various other processing
...
digest = result.get()     # Get the result
```

- This executes a function in a worker process and retrieves the result at a later time
- The worker churns in the background allowing the main program to do other things

Pool Example

- Make a dictionary mapping names to digests

```
import multiprocessing
import os
TOPDIR = "/Users/beazley/Software/Python-3.0"

p = multiprocessing.Pool(2)     # Make a process pool
digest_map = {}
for path, dirs, files in os.walk(TOPDIR):
    for name in files:
        fullname = os.path.join(path,name)
        digest_map[fullname] = p.apply_async(
              compute_digest, (fullname,)
        )

# Go through the final dictionary and collect results
for filename, result in digest_map.items():
    digest_map[filename] = result.get()
```

- This runs in about 5.6 seconds

Part 11

Alternatives to Threads and Processes

Alternatives

- In certain kinds of applications, programmers have turned to alternative approaches that don't rely on threads or processes
- Primarily this centers around asynchronous I/O and I/O multiplexing
- You try to make a single Python process run as fast as possible without any thread/process overhead (e.g., context switching, stack space, and so forth)

Two Approaches

- There seems to be two schools of thought...
- Event-driven programming
  - Turn all I/O handling into events
  - Do everything through event handlers
  - asyncore, Twisted, etc.
- Coroutines
  - Cooperative multitasking all in Python
  - Tasklets, green threads, etc.

Events and Asyncore

- asyncore library module
- Implements a wrapper around sockets that turn all blocking I/O operations into events

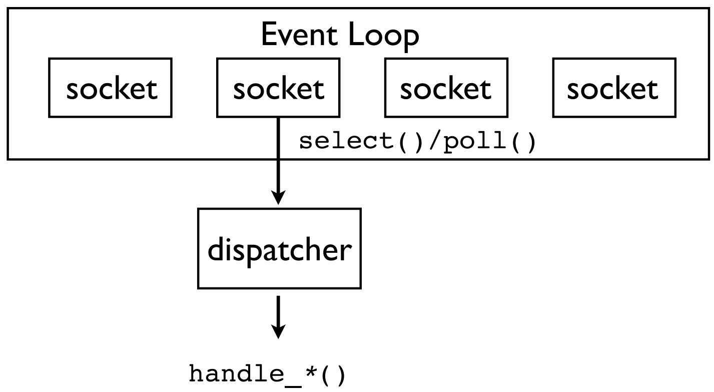
```
s = socket(...)

s.accept()
s.connect(addr)
s.recv(maxbytes)
s.send(msg)
...
```

```
from asyncore import dispatcher
class MyApp(dispatcher):
    def handle_accept(self):
        ...
    def handle_connect(self):
        ...
    def handle_read(self):
        ...
    def handle_write(self):
        ...
# Create a socket and wrap it
s = MyApp(socket())
```

Events and Asyncore

- To run, asyncore provides a central event loop based on I/O multiplexing (select/poll)

```
import asyncore
asyncore.loop()     # Run the event loop
```



Asyncore Commentary

- Frankly, asyncore is one of the ugliest, most annoying, mind-boggling modules in the entire Python library
- Combines all of the "fun" of network programming with the "elegance" of GUI programming (sic)
- However, if you use this module, you can technically create programs that have "concurrency" without any threads/processes

Coroutines

- An alternative concurrency approach is possible using Python generator functions (coroutines)
- This is a little subtle, but I'll give you the gist
- First, a quick refresher on generators

Generator Refresher

- Generator functions are commonly used to feed values to for-loops (iteration)

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for x in countdown(10):
    print x
```

- Under the covers, the countdown function executes on successive next() calls

```
>>> c = countdown(10)
>>> c.next()
10
>>> c.next()
9
>>>
```

An Insight

- Whenever a generator function hits the yield statement, it suspends execution

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

- Here's the idea : Instead of yielding a value, a generator can yield control
- You can write a little scheduler that cycles between generators, running each one until it explicitly yields

Scheduling Example

- First, you set up a set of "tasks"

```
def countdown_task(n):
    while n > 0:
        print n
        yield
        n -= 1

# A list of tasks to run
from collections import deque
tasks = deque([
    countdown_task(5),
    countdown_task(10),
    countdown_task(15)
])
```

- Each task is a generator function

Scheduling Example

- Now, run a task scheduler

```
def scheduler(tasks):
    while tasks:
        task = tasks.popleft()
        try:
            next(task)         # Run to the next yield
            tasks.append(task) # Reschedule
        except StopIteration:
            pass

# Run it
scheduler(tasks)
```

- This loop is what drives the application

Scheduling Example

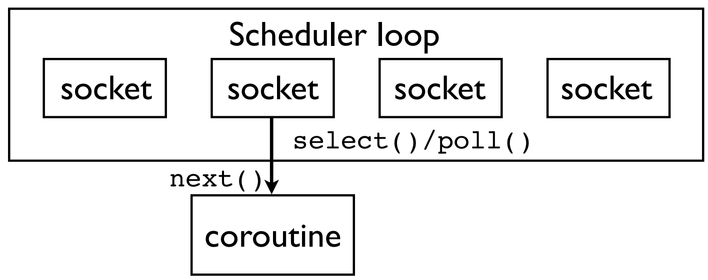
- Output

```
5
10
15
4
9
14
3
8
13
...
```

- You'll see the different tasks cycling

Coroutines and I/O

- It is also possible to tie coroutines to I/O
- You take an event loop (like asyncore), but instead of firing callback functions, you schedule coroutines in response to I/O activity



- Unfortunately, this requires its own tutorial...

Coroutine Commentary

- Usage of coroutines is somewhat exotic
- Mainly due to poor documentation and the "newness" of the feature itself
- There are also some grungy aspects of programming with generators

Coroutine Info

- I gave a tutorial that goes into more detail
- "A Curious Course on Coroutines and Concurrency" at PyCON'09
- http://www.dabeaz.com/coroutines

Part 12

Final Words and Wrap up

Quick Summary

- Covered various options for Python concurrency
  - Threads
  - Multiprocessing
  - Event handling
  - Coroutines/generators
- Hopefully have expanded awareness of how Python works under the covers as well as some of the pitfalls and tradeoffs

Thanks!

- I hope you got some new ideas from this class
- Please feel free to contact me

http://www.dabeaz.com

- Also, I teach Python classes (shameless plug)