A Curious Course on Coroutines and Concurrency

David Beazley
http://www.dabeaz.com

Presented at PyCon'2009, Chicago, Illinois

This Tutorial

- A mondo exploration of Python coroutines

> mondo:
> 1. **Extreme** in degree or nature. (http://www.urbandictionary.com)
>
> 2. An instructional technique of Zen Buddhism consisting of **rapid** dialogue of questions and answers between master and pupil. (Oxford English Dictionary, 2nd Ed)

- You might want to brace yourself...

Requirements

- You need Python 2.5 or newer
- No third party extensions
- We're going to be looking at a lot of code

http://www.dabeaz.com/coroutines/

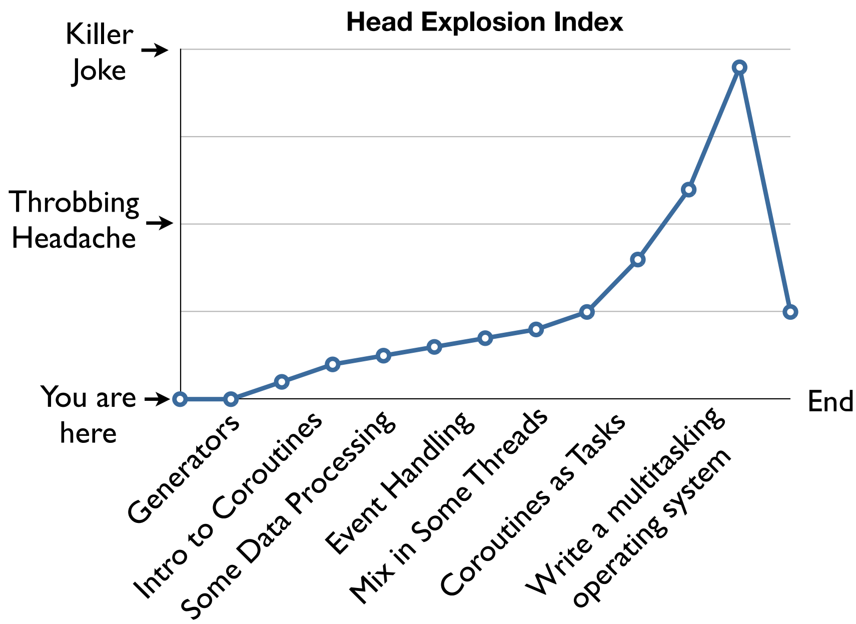
- Go there and follow along with the examples
- I will indicate file names as appropriate

sample.py

High Level Overview

- What in the heck is a coroutine?
- What can you use them for?
- Should you care?
- Is using them even a good idea?

A Pictorial Overview



About Me

- I'm a long-time Pythonista
- Author of the Python Essential Reference (look for the 4th edition--shameless plug)
- Created several packages (Swig, PLY, etc.)
- Currently a full-time Python trainer

Some Background

- I'm an unabashed fan of generators and generator expressions (Generators Rock!)
- See "Generator Tricks for Systems Programmers" from PyCon'08
- http://www.dabeaz.com/generators

Coroutines and Generators

- In Python 2.5, generators picked up some new features to allow "coroutines" (PEP-342).
- Most notably: a new send() method
- If Python books are any guide, this is the most poorly documented, obscure, and apparently useless feature of Python.
- "Oooh. You can now *send* values into generators producing fibonacci numbers!"

Uses of Coroutines

- Coroutines apparently might be possibly useful in various libraries and frameworks

"It's all really quite simple. The toelet is connected to the footlet, and the footlet is connected to the anklelet, and the anklelet is connected to the leglet, and the is leglet connected to the is thighlet, and the thighlet is connected to the hiplet, and the is hiplet connected to the backlet, and the backlet is connected to the necklet, and the necklet is connected to the headlet, and ?????? ..... profit!"

- Uh, I think my brain is just too small...

Disclaimers

- Coroutines - The most obscure Python feature?
- Concurrency - One of the most difficult topics in computer science (usually best avoided)
- This tutorial mixes them together
- It might create a toxic cloud

More Disclaimers

- As a programmer of the 80s/90s, I've never used a programming language that had coroutines--until they showed up in Python
- Most of the groundwork for coroutines occurred in the 60s/70s and then stopped in favor of alternatives (e.g., threads, continuations)
- I want to know if there is any substance to the renewed interest in coroutines that has been occurring in Python and other languages

Even More Disclaimers

- I'm a neutral party
- I didn't have anything to do with PEP-342
- I'm not promoting any libraries or frameworks
- I have no religious attachment to the subject
- If anything, I'm a little skeptical

Final Disclaimers

- This tutorial is not an academic presentation
- No overview of prior art
- No theory of programming languages
- No proofs about locking
- No Fibonacci numbers
- <u>Practical application</u> is the main focus

Performance Details

- There are some later performance numbers
- Python 2.6.1 on OS X 10.4.11
- All tests were conducted on the following:
  - Mac Pro 2x2.66 Ghz Dual-Core Xeon
  - 3 Gbytes RAM
- Timings are 3-run average of 'time' command

Part I

Introduction to Generators and Coroutines

Generators

- A generator is a function that produces a sequence of results instead of a single value

countdown.py

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

>>> for i in countdown(5):
...     print i,
...
5 4 3 2 1
>>>
```

- Instead of returning a value, you generate a series of values (using the yield statement)
- Typically, you hook it up to a for-loop

Generators

- Behavior is quite different than normal func
- Calling a generator function creates an generator object. However, it does <u>not</u> start running the function.

```
def countdown(n):
    print "Counting down from", n
    while n > 0:
        yield n
        n -= 1

>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>>
```

Notice that no output was produced

Generator Functions

- The function only executes on next()

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> x.next()
Counting down from 10
10
>>>
```

Function starts executing here

- yield produces a value, but suspends the function
- Function resumes on next call to next()

```
>>> x.next()
9
>>> x.next()
8
>>>
```

Generator Functions

- When the generator returns, iteration stops

```
>>> x.next()
1
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```


A Practical Example

- A Python version of Unix 'tail -f'

follow.py

```
import time
def follow(thefile):
    thefile.seek(0,2)      # Go to the end of the file
    while True:
         line = thefile.readline()
         if not line:
             time.sleep(0.1)    # Sleep briefly
             continue
         yield line
```

- Example use : Watch a web-server log file

```
logfile  = open("access-log")
for line in follow(logfile):
    print line,
```

Generators as Pipelines

- One of the most powerful applications of generators is setting up processing pipelines
- Similar to shell pipes in Unix

```
input    → generator → generator → generator → for x in s:
sequence
```

- Idea: You can stack a series of generator functions together into a pipe and pull items through it with a for-loop

A Pipeline Example

- Print all server log entries containing 'python'

pipeline.py

```
def grep(pattern,lines):
    for line in lines:
        if pattern in line:
             yield line

# Set up a processing pipe : tail -f | grep python
logfile  = open("access-log")
loglines = follow(logfile)
pylines  = grep("python",loglines)

# Pull results out of the processing pipeline
for line in pylines:
    print line,
```

- This is just a small taste

Yield as an Expression

- In Python 2.5, a slight modification to the yield statement was introduced (PEP-342)
- You could now use yield as an *expression*
- For example, on the right side of an assignment

grep.py

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line,
```

- Question : What is its value?

Coroutines

- If you use yield more generally, you get a coroutine
- These do more than just generate values
- Instead, functions can consume values sent to it.

```
>>> g = grep("python")
>>> g.next()              # Prime it (explained shortly)
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>>
```

- Sent values are returned by (yield)

Coroutine Execution

- Execution is the same as for a generator
- When you call a coroutine, nothing happens
- They only run in response to next() and send() methods

```
>>> g = grep("python")
>>> g.next()
Looking for python
>>>
```

Notice that no output was produced

On first operation, coroutine starts running

Coroutine Priming

- All coroutines must be "primed" by first calling .next() (or send(None))
- This advances execution to the location of the first yield expression.

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line,
```

.next() advances the coroutine to the first yield expression

- At this point, it's ready to receive a value

Using a Decorator

- Remembering to call .next() is easy to forget
- Solved by wrapping coroutines with a decorator

coroutine.py

```
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        cr.next()
        return cr
    return start

@coroutine
def grep(pattern):
    ...
```

- I will use this in most of the future examples

Closing a Coroutine

- A coroutine might run indefinitely
- Use .close() to shut it down

```
>>> g = grep("python")
>>> g.next()              # Prime it
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>> g.close()
```

- Note: Garbage collection also calls close()

Catching close()

- close() can be caught (GeneratorExit)

grepclose.py

```
@coroutine
def grep(pattern):
    print "Looking for %s" % pattern
    try:
        while True:
            line = (yield)
            if pattern in line:
                print line,
    except GeneratorExit:
        print "Going away.  Goodbye"
```

- You cannot ignore this exception
- Only legal action is to clean up and return

Throwing an Exception

- Exceptions can be thrown inside a coroutine

```
>>> g = grep("python")
>>> g.next()                # Prime it
Looking for python
>>> g.send("python generators rock!")
python generators rock!
>>> g.throw(RuntimeError,"You're hosed")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in grep
RuntimeError: You're hosed
>>>
```

- Exception originates at the yield expression
- Can be caught/handled in the usual ways

Interlude

- Despite some similarities, Generators and coroutines are basically two different concepts
- Generators produce values
- Coroutines tend to consume values
- It is easy to get sidetracked because methods meant for coroutines are sometimes described as a way to tweak generators that are in the process of producing an iteration pattern (i.e., resetting its value). This is mostly bogus.

A Bogus Example

- A "generator" that produces <u>and</u> receives values

bogus.py

```
def countdown(n):
    print "Counting down from", n
    while n >= 0:
        newvalue = (yield n)
        # If a new value got sent in, reset n with it
        if newvalue is not None:
            n = newvalue
        else:
            n -= 1
```

- It runs, but it's "flaky" and hard to understand

```
c = countdown(5)
for n in c:
    print n
    if n == 5:
        c.send(3)
```

output →

```
5
2
1
0
```

Notice how a value got "lost" in the iteration protocol

Keeping it Straight

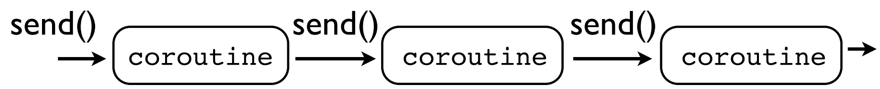
- Generators produce data for iteration
- Coroutines are consumers of data
- To keep your brain from exploding, you don't mix the two concepts together
- <u>Coroutines are not related to iteration</u>
- Note : There is a use of having yield produce a value in a coroutine, but it's not tied to iteration.

Part 2

Coroutines, Pipelines, and Dataflow

Processing Pipelines

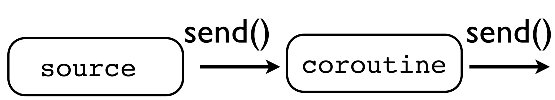
- Coroutines can be used to set up pipes



- You just chain coroutines together and <u>push</u> data through the pipe with send() operations

Pipeline Sources

- The pipeline needs an initial source (a producer)



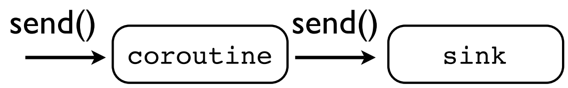
- The source drives the entire pipeline

```
def source(target):
    while not done:
        item = produce_an_item()
        ...
        target.send(item)
        ...
    target.close()
```

- It is typically <u>not</u> a coroutine

Pipeline Sinks

- The pipeline must have an end-point (sink)



- Collects all data sent to it and processes it

```
@coroutine
def sink():
    try:
        while True:
            item = (yield)    # Receive an item
            ...
    except GeneratorExit:     # Handle .close()
        # Done
        ...
```


An Example

- A source that mimics Unix 'tail -f'

cofollow.py

```
import time
def follow(thefile, target):
    thefile.seek(0,2)      # Go to the end of the file
    while True:
        line = thefile.readline()
        if not line:
            time.sleep(0.1)    # Sleep briefly
            continue
        target.send(line)
```

- A sink that just prints the lines

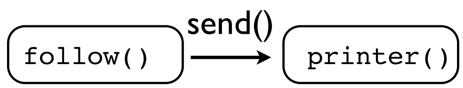
```
@coroutine
def printer():
    while True:
        line = (yield)
        print line,
```

An Example

- Hooking it together

```
f = open("access-log")
follow(f, printer())
```

- A picture



- Critical point : follow() is driving the entire computation by reading lines and pushing them into the printer() coroutine

Pipeline Filters

- Intermediate stages both receive and send

- Typically perform some kind of data transformation, filtering, routing, etc.

```
@coroutine
def filter(target):
    while True:
        item = (yield)            # Receive an item
        # Transform/filter item
        ...
        # Send it along to the next stage
        target.send(item)
```

A Filter Example

- A grep filter coroutine

copipe.py

```
@coroutine
def grep(pattern,target):
    while True:
        line = (yield)           # Receive a line
        if pattern in line:
            target.send(line)    # Send to next stage
```

- Hooking it up

```
f = open("access-log")
follow(f,
       grep('python',
       printer()))
```

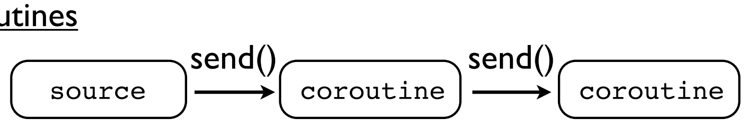
- A picture

Interlude

- Coroutines flip generators around

generators/iteration

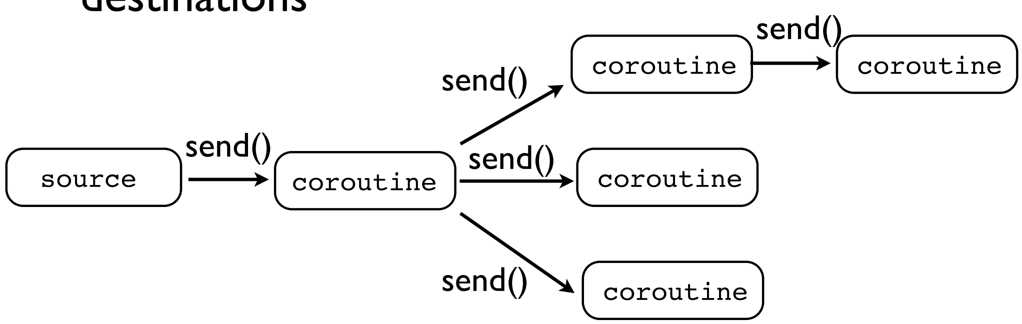
coroutines



- Key difference. Generators pull data through the pipe with iteration. Coroutines push data into the pipeline with send().

Being Branchy

- With coroutines, you can send data to multiple destinations



- The source simply "sends" data. Further routing of that data can be arbitrarily complex

Example : Broadcasting

- Broadcast to multiple targets

cobroadcast.py

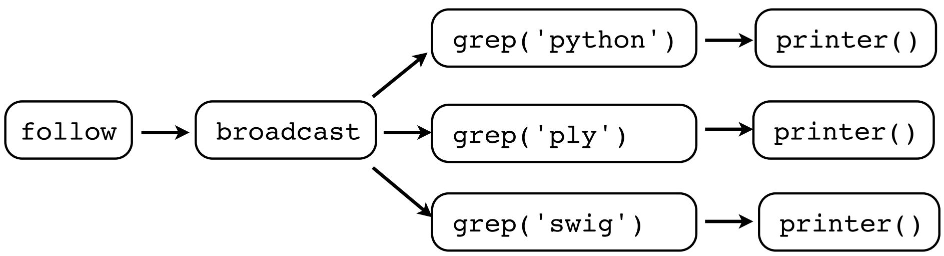
```
@coroutine
def broadcast(targets):
    while True:
        item = (yield)
        for target in targets:
            target.send(item)
```

- This takes a sequence of coroutines (targets) and sends received items to all of them.

Example : Broadcasting

- Example use:

```
f = open("access-log")
follow(f,
       broadcast([grep('python',printer()),
                  grep('ply',printer()),
                  grep('swig',printer())])
  )
```

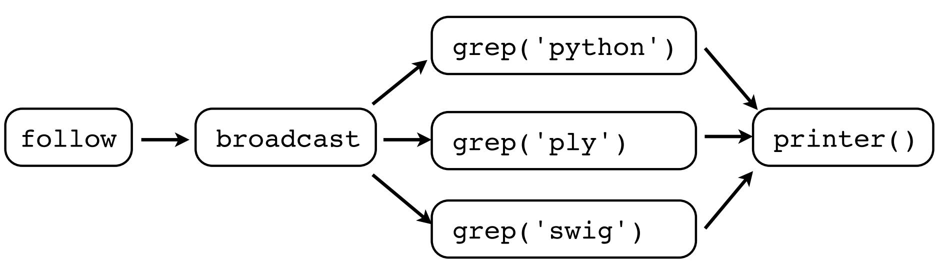


Example : Broadcasting

- A more disturbing variation...

cobroadcast2.py

```
f = open("access-log")
p = printer()
follow(f,
       broadcast([grep('python',p),
                  grep('ply',p),
                  grep('swig',p)])
  )
```



Interlude

- Coroutines provide more powerful data routing possibilities than simple iterators
- If you built a collection of simple data processing components, you can glue them together into complex arrangements of pipes, branches, merging, etc.
- Although there are some limitations (later)

A Digression

- In preparing this tutorial, I found myself wishing that variable assignment was an expression

```
@coroutine
def printer():
    while True:
          line = (yield)
          print line,
```

vs.

```
@coroutine
def printer():
    while (line = yield):
          print line,
```

- However, I'm not holding my breath on that...
- Actually, I'm expecting to be flogged with a rubber chicken for even suggesting it.

Coroutines vs. Objects

- Coroutines are somewhat similar to OO design patterns involving simple handler objects

```
class GrepHandler(object):
    def __init__(self,pattern, target):
        self.pattern = pattern
        self.target = target
    def send(self,line):
        if self.pattern in line:
            self.target.send(line)
```

- The coroutine version

```
@coroutine
def grep(pattern,target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)
```


Coroutines vs. Objects

- There is a certain "conceptual simplicity"
  - A coroutine is one function definition
- If you define a handler class...
  - You need a class definition
  - Two method definitions
  - Probably a base class and a library import
- Essentially you're stripping the idea down to the bare essentials (like a generator vs. iterator)

Coroutines vs. Objects

- Coroutines are <u>faster</u>
- A micro benchmark

benchmark.py

```
@coroutine
def null():
    while True: item = (yield)

line = 'python is nice'
p1   = grep('python',null())          # Coroutine
p2   = GrepHandler('python',null())   # Object
```

- Send in 1,000,000 lines

```
timeit("p1.send(line)",
       "from __main__ import line,p1")
```

⟶ 0.60 s

```
timeit("p2.send(line)",
       "from __main__ import line,p2")
```

⟶ 0.92 s

Coroutines & Objects

- Understanding the performance difference

```
class GrepHandler(object):
    ...
    def send(self,line):
        if self.pattern in line:
            self.target.send(line)
```

Look at these self lookups!

- Look at the coroutine

```
@coroutine
def grep(pattern, target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(d)
```

"self" free

Part 3

Coroutines and Event Dispatching

Event Handling

- Coroutines can be used to write various components that process event streams
- Let's look at an example...

Problem

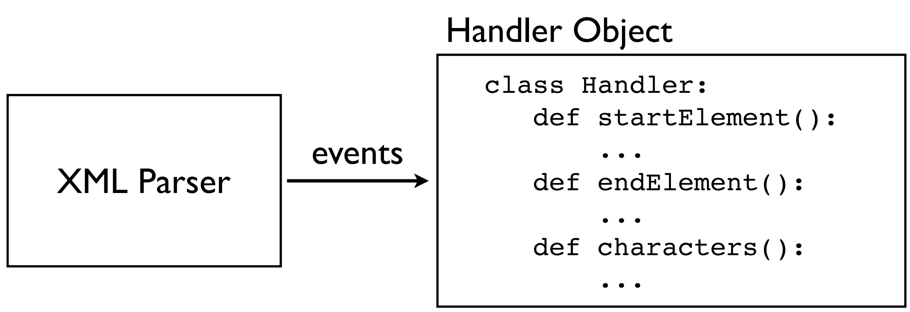
- Where is my ^&#&@* bus?
- Chicago Transit Authority (CTA) equips most of its buses with real-time GPS tracking
- You can get current data on *every* bus on the street as a big XML document
- Use "The Google" to search for details...

Some XML

```
<?xml version="1.0"?>
  <buses>
    <bus>
      <id>7574</id>
      <route>147</route>
      <color>#3300ff</color>
      <revenue>true</revenue>
      <direction>North Bound</direction>
      <latitude>41.925682067871094</latitude>
      <longitude>-87.63092803955078</longitude>
      <pattern>2499</pattern>
      <patternDirection>North Bound</patternDirection>
      <run>P675</run>
      <finalStop><![CDATA[Paulina & Howard Terminal]]></finalStop>
      <operator>42493</operator>
   </bus>
   <bus>
    ...
   </bus>
  </buses>
```

XML Parsing

- There are many possible ways to parse XML
- An old-school approach: SAX
- SAX is an event driven interface



Minimal SAX Example

basicsax.py

```
import xml.sax

class MyHandler(xml.sax.ContentHandler):
    def startElement(self,name,attrs):
        print "startElement", name
    def endElement(self,name):
        print "endElement", name
    def characters(self,text):
        print "characters", repr(text)[:40]

xml.sax.parse("somefile.xml",MyHandler())
```

- You see this same programming pattern in other settings (e.g., HTMLParser module)

Some Issues

- SAX is often used because it can be used to incrementally process huge XML files without a large memory footprint
- However, the event-driven nature of SAX parsing makes it rather awkward and low-level to deal with

From SAX to Coroutines

- You can dispatch SAX events into coroutines
- Consider this SAX handler

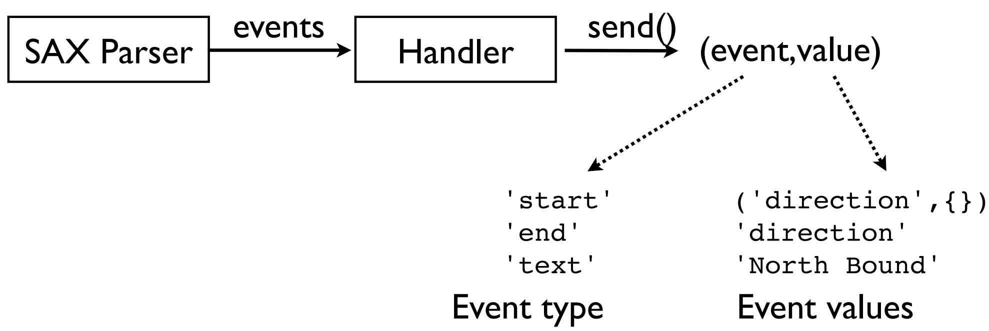
cosax.py

```
import xml.sax

class EventHandler(xml.sax.ContentHandler):
    def __init__(self,target):
        self.target = target
    def startElement(self,name,attrs):
        self.target.send(('start',(name,attrs._attrs)))
    def characters(self,text):
        self.target.send(('text',text))
    def endElement(self,name):
        self.target.send(('end',name))
```

- It does nothing, but send events to a target

An Event Stream

- The big picture



- Observe : Coding this was straightforward

Event Processing

- To do anything interesting, you have to process the event stream
- Example: Convert bus elements into dictionaries (XML sucks, dictionaries rock)

```
<bus>
  <id>7574</id>
  <route>147</route>
  <revenue>true</revenue>
  <direction>North Bound</direction>
  ...
 </bus>
```

```
{
  'id' : '7574',
  'route' : '147',
  'revenue' : 'true',
  'direction' : 'North Bou
  ...
}
```

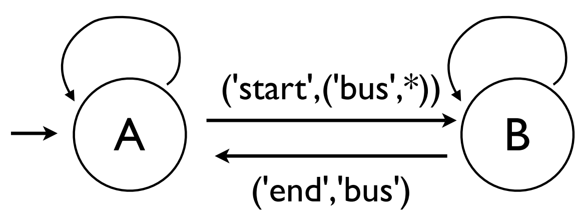
Buses to Dictionaries

buses.py

```
@coroutine
def buses_to_dicts(target):
    while True:
        event, value = (yield)
        # Look for the start of a <bus> element
        if event == 'start' and value[0] == 'bus':
            busdict = { }
            fragments = []
            # Capture text of inner elements in a dict
            while True:
                event, value = (yield)
                if event == 'start':   fragments = []
                elif event == 'text':  fragments.append(value)
                elif event == 'end':
                    if value != 'bus':
                        busdict[value] = "".join(fragments)
                    else:
                        target.send(busdict)
                        break
```


State Machines

- The previous code works by implementing a simple state machine



- State A: Looking for a bus
- State B: Collecting bus attributes
- Comment : Coroutines are perfect for this

Buses to Dictionaries

A

```
@coroutine
def buses_to_dicts(target):
    while True:
        event, value = (yield)
        # Look for the start of a <bus> element
        if event == 'start' and value[0] == 'bus':
            busdict = { }
            fragments = []
            # Capture text of inner elements in a dict
            while True:
                event, value = (yield)
                if event == 'start':   fragments = []
                elif event == 'text':  fragments.append(value)
                elif event == 'end':
                    if value != 'bus':
                        busdict[value] = "".join(fragments)
                    else:
                        target.send(busdict)
                        break
```

B

Filtering Elements

- Let's filter on dictionary fields

```
@coroutine
def filter_on_field(fieldname,value,target):
    while True:
        d = (yield)
        if d.get(fieldname) == value:
            target.send(d)
```

- Examples:

```
filter_on_field("route","22",target)
filter_on_field("direction","North Bound",target)
```

Processing Elements

- Where's my bus?

```
@coroutine
def bus_locations():
    while True:
        bus = (yield)
        print "%(route)s,%(id)s,\"%(direction)s\","\
              "%(latitude)s,%(longitude)s" % bus
```

- This receives dictionaries and prints a table

```
22,1485,"North Bound",41.880481123924255,-87.62948191165924
22,1629,"North Bound",42.01851969751819,-87.6730209876751
...
```


Hooking it Together

- Find all locations of the North Bound #22 bus (the slowest moving object in the universe)

```
xml.sax.parse("allroutes.xml",
              EventHandler(
                   buses_to_dicts(
                   filter_on_field("route","22",
                   filter_on_field("direction","North Bound",
                   bus_locations())))
              ))
```

- This final step involves a bit of plumbing, but each of the parts is relatively simple

How Low Can You Go?

- I've picked this XML example for reason
- One interesting thing about coroutines is that you can push the initial data source as low-level as you want to make it without rewriting all of the processing stages
- Let's say SAX just isn't quite fast enough...

XML Parsing with Expat

- Let's strip it down....

coexpat.py

```
import xml.parsers.expat

def expat_parse(f,target):
    parser = xml.parsers.expat.ParserCreate()
    parser.buffer_size = 65536
    parser.buffer_text = True
    parser.returns_unicode = False
    parser.StartElementHandler = \
       lambda name,attrs: target.send(('start',(name,attrs)))
    parser.EndElementHandler = \
       lambda name: target.send(('end',name))
    parser.CharacterDataHandler = \
       lambda data: target.send(('text',data))
    parser.ParseFile(f)
```

- expat is low-level (a C extension module)

Performance Contest

- SAX version (on a 30MB XML input)

```
xml.sax.parse("allroutes.xml",EventHandler(
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations()))))))
```

8.37s

- Expat version

```
expat_parse(open("allroutes.xml"),
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

4.51s

(83% speedup)

- No changes to the processing stages

Going Lower

- You can even drop send() operations into C
- A skeleton of how this works...

cxml/cxmlparse.c

```
PyObject *
py_parse(PyObject *self, PyObject *args) {
    PyObject *filename;
    PyObject *target;
    PyObject *send_method;
    if (!PyArg_ParseArgs(args,"sO",&filename,&target)) {
        return NULL;
    }
    send_method = PyObject_GetAttrString(target,"send");
    ...

    /* Invoke target.send(item) */
    args   = Py_BuildValue("(O)",item);
    result = PyEval_CallObject(send_meth,args);
    ...
```

Performance Contest

- Expat version

```
expat_parse(open("allroutes.xml"),
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

4.51s

- A custom C extension written directly on top of the expat C library (code not shown)

```
cxmlparse.parse("allroutes.xml",
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

2.95s

(55% speedup)

Interlude

- ElementTree has fast incremental XML parsing

iterbus.py

```
from xml.etree.cElementTree import iterparse

for event,elem in iterparse("allroutes.xml",('start','end')):
    if event == 'start' and elem.tag == 'buses':
        buses = elem
    elif event == 'end' and elem.tag == 'bus':
        busdict = dict((child.tag,child.text)
                        for child in elem)
        if (busdict['route'] == '22' and
            busdict['direction'] == 'North Bound'):
            print "%(id)s,%(route)s,\"%(direction)s\","\
                  "%(latitude)s,%(longitude)s" % busdict
        buses.remove(elem)
```

3.04s

- Observe: Coroutines are in the same range

Part 4

From Data Processing to Concurrent Programming

The Story So Far

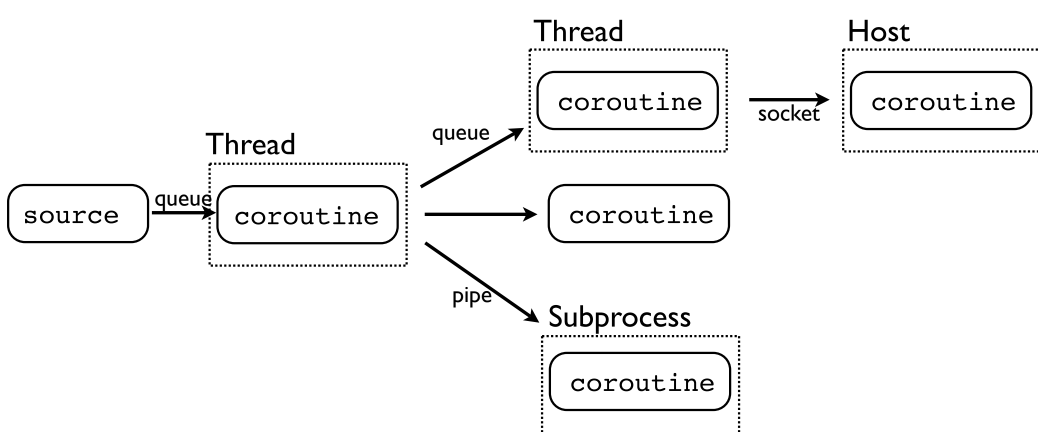
- Coroutines are similar to generators
- You can create collections of small processing components and connect them together
- You can process data by setting up pipelines, dataflow graphs, etc.
- You can use coroutines with code that has tricky execution (e.g., event driven systems)
- However, there is so much more going on...

A Common Theme

- You send data to coroutines
- You send data to threads (via queues)
- You send data to processes (via messages)
- Coroutines naturally tie into problems involving threads and distributed systems.

Basic Concurrency

- You can package coroutines inside threads or subprocesses by adding extra layers



- Will sketch out some basic ideas...

A Threaded Target

cothread.py

```
@coroutine
def threaded(target):
    messages = Queue()
    def run_target():
        while True:
            item = messages.get()
            if item is GeneratorExit:
                target.close()
                return
            else:
                target.send(item)
    Thread(target=run_target).start()
    try:
        while True:
            item = (yield)
            messages.put(item)
    except GeneratorExit:
        messages.put(GeneratorExit)
```


A Threaded Target

```
@coroutine
def threaded(target):
    messages = Queue()
    def run_target():
        while True:
            item = messages.get()
            if item is GeneratorExit:
                target.close()
                return
            else:
                target.send(item)
    Thread(target=run_target).start()
    try:
        while True:
            item = (yield)
            messages.put(item)
    except GeneratorExit:
        messages.put(GeneratorExit)
```

A message queue

A Threaded Target

```
@coroutine
def threaded(target):
    messages = Queue()
    def run_target():
        while True:
            item = messages.get()
            if item is GeneratorExit:
                target.close()
                return
            else:
                target.send(item)
    Thread(target=run_target).start()
    try:
        while True:
            item = (yield)
            messages.put(item)
    except GeneratorExit:
        messages.put(GeneratorExit)
```

A thread. Loop forever, pulling items out of the message queue and sending them to the target

A Threaded Target

```
@coroutine
def threaded(target):
    messages = Queue()
    def run_target():
        while True:
            item = messages.get()
            if item is GeneratorExit:
                target.close()
                return
            else:
                target.send(item)
    Thread(target=run_target).start()
    try:
        while True:
            item = (yield)
            messages.put(item)
    except GeneratorExit:
        messages.put(GeneratorExit)
```

Receive items and pass them into the thread (via the queue)

A Threaded Target

```
@coroutine
def threaded(target):
    messages = Queue()
    def run_target():
        while True:
            item = messages.get()
            if item is GeneratorExit:
                target.close()
                return
            else:
                target.send(item)
    Thread(target=run_target).start()
    try:
        while True:
            item = (yield)
            messages.put(item)
    except GeneratorExit:
        messages.put(GeneratorExit)
```

Handle close() so that the thread shuts down correctly

A Thread Example

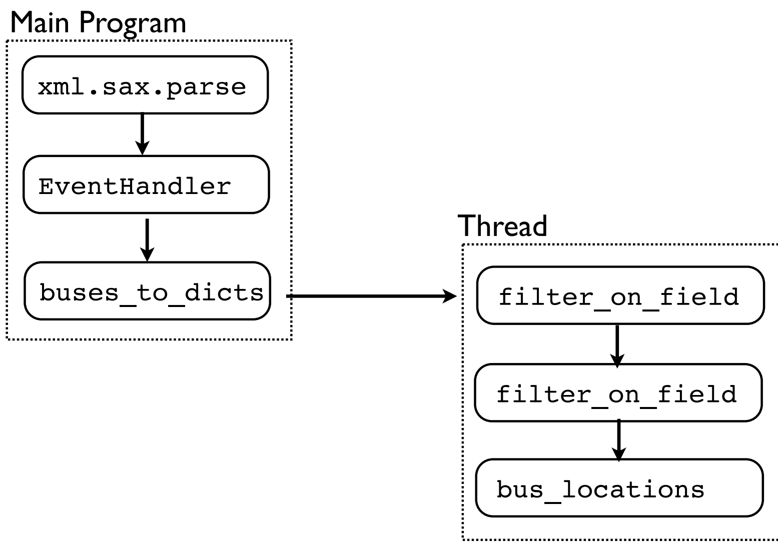
- Example of hooking things up

```
xml.sax.parse("allroutes.xml", EventHandler(
     buses_to_dicts(
     threaded(
        filter_on_field("route","22",
        filter_on_field("direction","North Bound",
        bus_locations()))
     ))))
```

- A caution: adding threads makes this example run about 50% slower.

A Picture

- Here is an overview of the last example



A Subprocess Target

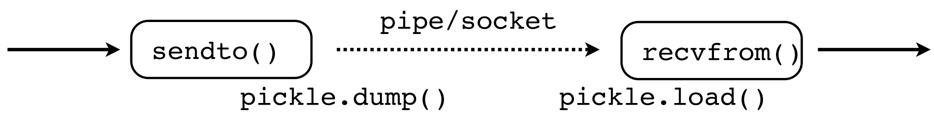
- Can also bridge two coroutines over a file/pipe

coprocess.py

```
@coroutine
def sendto(f):
    try:
        while True:
            item = (yield)
            pickle.dump(item,f)
            f.flush()
    except StopIteration:
        f.close()

def recvfrom(f,target):
    try:
        while True:
            item = pickle.load(f)
            target.send(item)
    except EOFError:
        target.close()
```

A Subprocess Target

- High Level Picture



- Of course, the devil is in the details...
- You would not do this unless you can recover the cost of the underlying communication (e.g., you have multiple CPUs and there's enough processing to make it worthwhile)

Implementation vs. Environ

- With coroutines, you can separate the implementation of a task from its execution environment
- The coroutine is the implementation
- The environment is whatever you choose (threads, subprocesses, network, etc.)

A Caution

- Creating huge collections of coroutines, threads, and processes might be a good way to create an unmaintainable application (although it might increase your job security)
- And it might make your program run slower!
- You need to carefully study the problem to know if any of this is a good idea

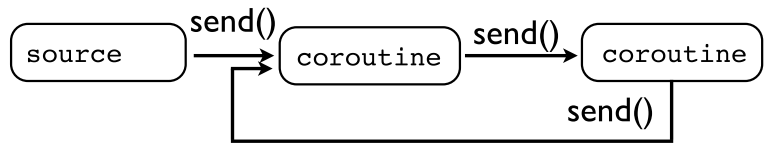
Some Hidden Dangers

- The send() method on a coroutine must be properly synchronized
- If you call send() on an already-executing coroutine, your program will crash
- Example : Multiple threads sending data into the same target coroutine

cocrash.py

Limitations

- You also can't create loops or cycles



- Stacked sends are building up a kind of call-stack (send() doesn't return until the target yields)
- If you call a coroutine that's already in the process of sending, you'll get an error
- send() doesn't suspend coroutine execution

Part 5

Coroutines as Tasks

The Task Concept

- In concurrent programming, one typically subdivides problems into "tasks"
- Tasks have a few essential features
  - Independent control flow
  - Internal state
  - Can be scheduled (suspended/resumed)
  - Can communicate with other tasks
- Claim : Coroutines are tasks

Are Coroutines Tasks?

- Let's look at the essential parts
- Coroutines have their own control flow.

```
@coroutine
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line,
```

statements

- A coroutine is just a sequence of statements like any other Python function

Are Coroutines Tasks?

- Coroutines have their internal own state
- For example : local variables

```
        @coroutine
        def grep(pattern):
            print "Looking for %s" % pattern
            while True:
locals          line = (yield)
                if pattern in line:
                    print line,
```

- The locals live as long as the coroutine is active
- They establish an execution environment

Are Coroutines Tasks?

- Coroutines can communicate
- The .send() method sends data to a coroutine

```
@coroutine
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)    send(msg)
        if pattern in line:
            print line,
```

- yield expressions receive input

Are Coroutines Tasks?

- Coroutines can be suspended and resumed
- yield suspends execution
- send() resumes execution
- close() terminates execution

I'm Convinced

- Very clearly, coroutines look like tasks
- But they're not tied to threads
- Or subprocesses
- A question : Can you perform multitasking without using either of those concepts?
- Multitasking using nothing but coroutines?

Part 6

A Crash Course in Operating Systems

Program Execution

- On a CPU, a program is a series of instructions

```
int main() {
  int i, total = 0;
  for (i = 0; i < 10; i++)
  {
    total += i;
  }
}
```

cc

```
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $0, -12(%ebp)
        movl    $0, -16(%ebp)
        jmp     L2
L3:
        movl    -16(%ebp), %eax
        leal    -12(%ebp), %edx
        addl    %eax, (%edx)
        leal    -16(%ebp), %eax
        incl    (%eax)
L2:
        cmpl    $9, -16(%ebp)
        jle     L3
        leave
        ret
```

- When running, there is no notion of doing more than one thing at a time (or any kind of task switching)

The Multitasking Problem

- CPUs don't know anything about multitasking
- Nor do application programs
- Well, surely *something* has to know about it!
- Hint: It's the operating system

Operating Systems

- As you hopefully know, the operating system (e.g., Linux, Windows) is responsible for running programs on your machine
- And as you have observed, the operating system <u>does</u> allow more than one process to execute at once (e.g., multitasking)
- It does this by rapidly switching between tasks
- Question : How does it do that?

A Conundrum

- When a CPU is running your program, it is <u>not</u> running the operating system
- Question: How does the operating system (which is <u>not</u> running) make an application (which <u>is</u> running) switch to another task?
- The "context-switching" problem...

Interrupts and Traps

- There are usually only two mechanisms that an operating system uses to gain control
  - Interrupts - Some kind of hardware related signal (data received, timer, keypress, etc.)
  - Traps - A software generated signal
- In both cases, the CPU briefly suspends what it is doing, and runs code that's part of the OS
- It is at this time the OS might switch tasks

Traps and System Calls

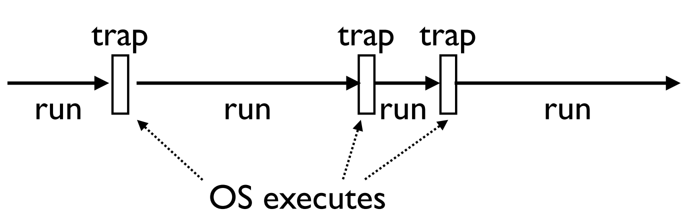
- Low-level system calls are actually traps
- It is a special CPU instruction

```
read(fd,buf,nbytes)  ------>  read:
                                  push  %ebx
                                  mov 0x10(%esp),%edx
                                  mov 0xc(%esp),%ecx
                                  mov 0x8(%esp),%ebx
                                  mov $0x3,%eax
                                  int $0x80     <----- trap
                                  pop %ebx
                                  ...
```

- When a trap instruction executes, the program <u>suspends execution</u> at that point
- And the OS takes over

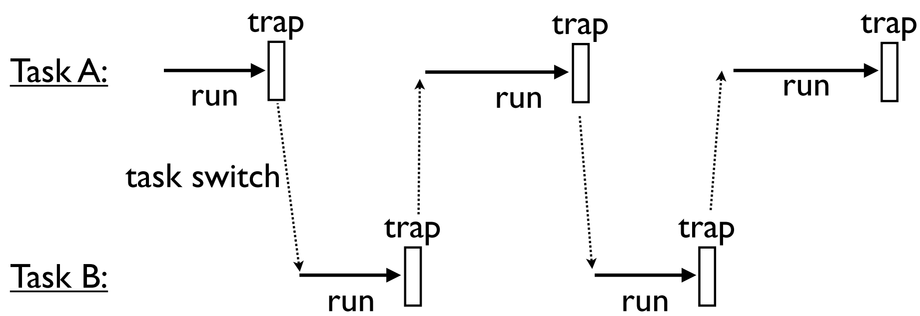
High Level Overview

- Traps are what make an OS work
- The OS drops your program on the CPU
- It runs until it hits a trap (system call)
- The program suspends and the OS runs
- Repeat



Task Switching

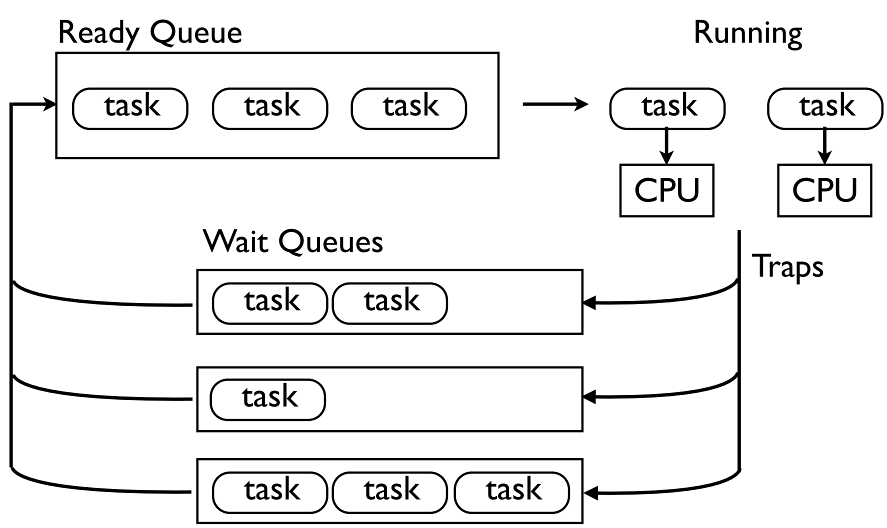
- Here's what typically happens when an OS runs multiple tasks.



- On each trap, the system switches to a different task (cycling between them)

Task Scheduling

- To run many tasks, add a bunch of queues



An Insight

- The yield statement is a kind of "trap"
- No really!
- When a generator function hits a "yield" statement, it immediately suspends execution
- Control is passed back to whatever code made the generator function run (unseen)
- If you treat yield as a trap, you can build a multitasking "operating system"--all in Python!

Part 7

Let's Build an Operating System

(You may want to put on your 5-point safety harness)

Our Challenge

- Build a multitasking "operating system"
- Use nothing but pure Python code
- No threads
- No subprocesses
- Use generators/coroutines

Some Motivation

- There has been a lot of recent interest in alternatives to threads (especially due to the GIL)
- Non-blocking and asynchronous I/O
- Example: servers capable of supporting thousands of simultaneous client connections
- A lot of work has focused on event-driven systems or the "Reactor Model" (e.g., Twisted)
- Coroutines are a whole different twist...

Step 1: Define Tasks

- A task object

pyos1.py

```
class Task(object):
    taskid = 0
    def __init__(self,target):
        Task.taskid += 1
        self.tid     = Task.taskid   # Task ID
        self.target  = target        # Target coroutine
        self.sendval = None          # Value to send
    def run(self):
        return self.target.send(self.sendval)
```

- A task is a wrapper around a coroutine
- There is only one operation : run()

Task Example

- Here is how this wrapper behaves

```
# A very simple generator
def foo():
    print "Part 1"
    yield
    print "Part 2"
    yield

>>> t1 = Task(foo())            # Wrap in a Task
>>> t1.run()
Part 1
>>> t1.run()
Part 2
>>>
```

- run() executes the task to the next yield (a trap)

Step 2: The Scheduler

```
class Scheduler(object):
    def __init__(self):
        self.ready   = Queue()
        self.taskmap = {}

    def new(self,target):
        newtask = Task(target)
        self.taskmap[newtask.tid] = newtask
        self.schedule(newtask)
        return newtask.tid

    def schedule(self,task):
        self.ready.put(task)

    def mainloop(self):
        while self.taskmap:
            task = self.ready.get()
            result = task.run()
            self.schedule(task)
```

pyos2.py

Step 2: The Scheduler

```
class Scheduler(object):
    def __init__(self):
        self.ready   = Queue()
        self.taskmap = {}

    def new(self,target):
        newtask = Task(target)
        self.taskmap[newtask.tid] = newtask
        self.schedule(newtask)
        return newtask.tid

    def schedule(self,task):
        self.ready.put(task)

    def mainloop(self):
        while self.taskmap:
            task = self.ready.get()
            result = task.run()
            self.schedule(task)
```

A queue of tasks that are ready to run

Step 2: The Scheduler

```
class Scheduler(object):
    def __init__(self):
        self.ready   = Queue()
        self.taskmap = {}

    def new(self,target):
        newtask = Task(target)
        self.taskmap[newtask.tid] = newtask
        self.schedule(newtask)
        return newtask.tid

    def schedule(self,task):
        self.ready.put(task)

    def mainloop(self):
        while self.taskmap:
            task = self.ready.get()
            result = task.run()
            self.schedule(task)
```

Introduces a new task to the scheduler

Step 2: The Scheduler

```
class Scheduler(object):
    def __init__(self):
        self.ready   = Queue()
        self.taskmap = {}

    def new(self,target):
        newtask = Task(target)
        self.taskmap[newtask.tid] = newtask
        self.schedule(newtask)
        return newtask.tid

    def schedule(self,task):
        self.ready.put(task)

    def mainloop(self):
        while self.taskmap:
            task = self.ready.get()
            result = task.run()
            self.schedule(task)
```

A dictionary that keeps track of all active tasks (each task has a unique integer task ID)

(more later)

Step 2: The Scheduler

```
class Scheduler(object):
    def __init__(self):
        self.ready   = Queue()
        self.taskmap = {}

    def new(self,target):
        newtask = Task(target)
        self.taskmap[newtask.tid] = newtask
        self.schedule(newtask)
        return newtask.tid

    def schedule(self,task):
        self.ready.put(task)

    def mainloop(self):
        while self.taskmap:
            task = self.ready.get()
            result = task.run()
            self.schedule(task)
```

Put a task onto the ready queue. This makes it available to run.

Step 2: The Scheduler


```
class Scheduler(object):
    def __init__(self):
        self.ready   = Queue()
        self.taskmap = {}

    def new(self,target):
        newtask = Task(target)
        self.taskmap[newtask.tid] = newtask
        self.schedule(newtask)
        return newtask.tid

    def schedule(self,task):
        self.ready.put(task)

    def mainloop(self):
        while self.taskmap:
            task = self.ready.get()
            result = task.run()
            self.schedule(task)
```


The main scheduler loop. It pulls tasks off the queue and runs them to the next yield.

First Multitasking

- Two tasks:

```
def foo():
    while True:
        print "I'm foo"
        yield

def bar():
    while True:
        print "I'm bar"
        yield
```

- Running them into the scheduler

```
sched = Scheduler()
sched.new(foo())
sched.new(bar())
sched.mainloop()
```


First Multitasking

- Example output:

```
I'm foo
I'm bar
I'm foo
I'm bar
I'm foo
I'm bar
```

- Emphasize: yield is a trap
- Each task runs until it hits the yield
- At this point, the scheduler regains control and switches to the other task

Problem : Task Termination

- The scheduler crashes if a task returns

taskcrash.py

```
def foo():
    for i in xrange(10):
        print "I'm foo"
        yield
...
I'm foo
I'm bar
I'm foo
I'm bar
Traceback (most recent call last):
  File "crash.py", line 20, in <module>
    sched.mainloop()
  File "scheduler.py", line 26, in mainloop
    result = task.run()
  File "task.py", line 13, in run
    return self.target.send(self.sendval)
StopIteration
```


Step 3: Task Exit

pyos3.py

```
class Scheduler(object):
    ...
    def exit(self,task):
        print "Task %d terminated" % task.tid
        del self.taskmap[task.tid]
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

Step 3: Task Exit

```
class Scheduler(object):
    ...
    def exit(self,task):
        print "Task %d terminated" % task.tid
        del self.taskmap[task.tid]
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

Remove the task from the scheduler's task map

Step 3: Task Exit

```
class Scheduler(object):
    ...
    def exit(self,task):
        print "Task %d terminated" % task.tid
        del self.taskmap[task.tid]
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

Catch task exit and cleanup

Second Multitasking

- Two tasks:

```
def foo():
    for i in xrange(10):
        print "I'm foo"
        yield

def bar():
    for i in xrange(5):
        print "I'm bar"
        yield

sched = Scheduler()
sched.new(foo())
sched.new(bar())
sched.mainloop()
```


Second Multitasking

- Sample output

```
I'm foo
I'm bar
I'm foo
I'm bar
I'm foo
I'm bar
I'm foo
I'm bar
I'm foo
I'm bar
I'm foo
Task 2 terminated
I'm foo
I'm foo
I'm foo
I'm foo
Task 1 terminated
```

System Calls

- In a real operating system, traps are how application programs request the services of the operating system (syscalls)
- In our code, the scheduler is the operating system and the yield statement is a trap
- To request the service of the scheduler, tasks will use the yield statement with a value

Step 4: System Calls

pyos4.py

```
class SystemCall(object):
    def handle(self):
        pass

class Scheduler(object):
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
                if isinstance(result,SystemCall):
                    result.task  = task
                    result.sched = self
                    result.handle()
                    continue
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

Step 4: System Calls

```
class SystemCall(object):
    def handle(self):
        pass

class Scheduler(object):
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
                if isinstance(result,SystemCall):
                    result.task  = task
                    result.sched = self
                    result.handle()
                    continue
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

System Call base class. All system operations will be implemented by inheriting from this class.

Step 4: System Calls

```
class SystemCall(object):
    def handle(self):
        pass

class Scheduler(object):
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
                if isinstance(result,SystemCall):
                    result.task  = task
                    result.sched = self
                    result.handle()
                    continue
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

Look at the result yielded by the task. If it's a SystemCall, do some setup and run the system call on behalf of the task.

Step 4: System Calls

```
class SystemCall(object):
    def handle(self):
        pass

class Scheduler(object):
    ...
    def mainloop(self):
         while self.taskmap:
            task = self.ready.get()
            try:
                result = task.run()
                if isinstance(result,SystemCall):
                    result.task  = task
                    result.sched = self
                    result.handle()
                    continue
            except StopIteration:
                self.exit(task)
                continue
            self.schedule(task)
```

These attributes hold information about the environment (current task and scheduler)

A First System Call

- Return a task's ID number

```
class GetTid(SystemCall):
    def handle(self):
        self.task.sendval = self.task.tid
        self.sched.schedule(self.task)
```

- The operation of this is little subtle

```
class Task(object):
    ...
    def run(self):
        return self.target.send(self.sendval)
```

- The sendval attribute of a task is like a return value from a system call. It's value is sent into the task when it runs again.

A First System Call

- Example of using a system call

```
def foo():
    mytid = yield GetTid()
    for i in xrange(5):
        print "I'm foo", mytid
        yield

def bar():
    mytid = yield GetTid()
    for i in xrange(10):
        print "I'm bar", mytid
        yield

sched = Scheduler()
sched.new(foo())
sched.new(bar())
sched.mainloop()
```


A First System Call

- Example output

```
I'm foo 1
I'm bar 2
I'm foo 1
I'm bar 2
I'm foo 1
I'm bar 2
I'm foo 1
I'm bar 2
I'm foo 1
I'm bar 2
Task 1 terminated
I'm bar 2
I'm bar 2
I'm bar 2
I'm bar 2
I'm bar 2
Task 2 terminated
```

Notice each task has a different task id

Design Discussion

- Real operating systems have a strong notion of "protection" (e.g., memory protection)
- Application programs are not strongly linked to the OS kernel (traps are only interface)
- For sanity, we are going to emulate this
  - Tasks do not see the scheduler
  - Tasks do not see other tasks
  - yield is the only external interface

Step 5: Task Management

- Let's make more some system calls
- Some task management functions
  - Create a new task
  - Kill an existing task
  - Wait for a task to exit
- These mimic common operations with threads or processes

Creating New Tasks

- Create a another system call

pyos5.py

```
class NewTask(SystemCall):
    def __init__(self,target):
        self.target = target
    def handle(self):
        tid = self.sched.new(self.target)
        self.task.sendval = tid
        self.sched.schedule(self.task)
```

- Example use:

```
def bar():
    while True:
         print "I'm bar"
         yield

def sometask():
   ...
   t1 = yield NewTask(bar())
```


Killing Tasks

- More system calls

```
class KillTask(SystemCall):
    def __init__(self,tid):
        self.tid = tid
    def handle(self):
        task = self.sched.taskmap.get(self.tid,None)
        if task:
            task.target.close()
            self.task.sendval = True
        else:
            self.task.sendval = False
        self.sched.schedule(self.task)
```

- Example use:

```
def sometask():
    t1 = yield NewTask(foo())
    ...
    yield KillTask(t1)
```

An Example

- An example of basic task control

```
def foo():
    mytid = yield GetTid()
    while True:
        print "I'm foo", mytid
        yield

def main():
    child = yield NewTask(foo())    # Launch new task
    for i in xrange(5):
        yield
    yield KillTask(child)           # Kill the task
    print "main done"

sched = Scheduler()
sched.new(main())
sched.mainloop()
```


An Example

- Sample output

```
I'm foo 2
I'm foo 2
I'm foo 2
I'm foo 2
I'm foo 2
Task 2 terminated
main done
Task 1 terminated
```

Waiting for Tasks

- This is a more tricky problem...

```
def foo():
    for i in xrange(5):
        print "I'm foo"
        yield

def main():
    child = yield NewTask(foo())
    print "Waiting for child"
    yield WaitTask(child)
    print "Child done"
```

- The task that waits has to remove itself from the run queue--it sleeps until child exits
- This requires some scheduler changes

Task Waiting

pyos6.py

```
class Scheduler(object):
    def __init__(self):
        ...
        self.exit_waiting = {}
        ...

    def exit(self,task):
        print "Task %d terminated" % task.tid
        del self.taskmap[task.tid]
        # Notify other tasks waiting for exit
        for task in self.exit_waiting.pop(task.tid,[]):
            self.schedule(task)

    def waitforexit(self,task,waittid):
        if waittid in self.taskmap:
            self.exit_waiting.setdefault(waittid,[]).append(task)
            return True
        else:
            return False
```

Task Waiting

```
class Scheduler(object):
    def __init__(self):
        ...
        self.exit_waiting = {}
        ...

    def exit(self,task):
        print "Task %d terminated" % task.tid
        del self.taskmap[task.tid]
        # Notify other tasks waiting for exit
        for task in self.exit_waiting.pop(task.tid,[]):
            self.schedule(task)

    def waitforexit(self,task,waittid):
        if waittid in self.taskmap:
            self.exit_waiting.setdefault(waittid,[]).append(task)
            return True
        else:
            return False
```

This is a holding area for tasks that are waiting. A dict mapping task ID to tasks waiting for exit.

Task Waiting

```
class Scheduler(object):
    def __init__(self):
        ...
        self.exit_waiting = {}
        ...

    def exit(self,task):
        print "Task %d terminated" % task.tid
        del self.taskmap[task.tid]
        # Notify other tasks waiting for exit
        for task in self.exit_waiting.pop(task.tid,[]):
            self.schedule(task)

    def waitforexit(self,task,waittid):
        if waittid in self.taskmap:
            self.exit_waiting.setdefault(waittid,[]).append(task)
            return True
        else:
            return False
```

When a task exits, we pop a list of all waiting tasks off out of the waiting area and reschedule them.

Task Waiting

```
class Scheduler(object):
    def __init__(self):
        ...
        self.exit_waiting = {}
        ...

    def exit(self,task):
        print "Task %d terminated"
        del self.taskmap[task.tid]
        # Notify other tasks waiti
        for task in self.exit_wait
            self.schedule(task)

    def waitforexit(self,task,waittid):
        if waittid in self.taskmap:
            self.exit_waiting.setdefault(waittid,[]).append(task)
            return True
        else:
            return False
```

A utility method that makes a task wait for another task. It puts the task in the waiting area.

Task Waiting

- Here is the system call

```
class WaitTask(SystemCall):
    def __init__(self,tid):
        self.tid = tid
    def handle(self):
        result = self.sched.waitforexit(self.task,self.tid)
        self.task.sendval = result
        # If waiting for a non-existent task,
        # return immediately without waiting
        if not result:
            self.sched.schedule(self.task)
```

- Note: Have to be careful with error handling.
- The last bit immediately reschedules if the task being waited for doesn't exist

Task Waiting Example

- Here is some example code:

```
def foo():
    for i in xrange(5):
        print "I'm foo"
        yield

def main():
    child = yield NewTask(foo())
    print "Waiting for child"
    yield WaitTask(child)
    print "Child done"
```


Task Waiting Example

- Sample output:

```
Waiting for child
I'm foo 2
I'm foo 2
I'm foo 2
I'm foo 2
I'm foo 2
Task 2 terminated
Child done
Task 1 terminated
```

Design Discussion

- The only way for tasks to refer to other tasks is using the integer task ID assigned by the the scheduler
- This is an encapsulation and safety strategy
- It keeps tasks separated (no linking to internals)
- It places all task management in the scheduler (which is where it properly belongs)

Interlude

- Running multiple tasks. Check.
- Launching new tasks. Check.
- Some basic task management. Check.
- The next step is obvious
- We must implement a web framework...
- ... or maybe just an echo sever to start.

An Echo Server Attempt

echobad.py

```
def handle_client(client,addr):
    print "Connection from", addr
    while True:
        data = client.recv(65536)
        if not data:
            break
        client.send(data)
    client.close()
    print "Client closed"
    yield           # Make the function a generator/coroutine

def server(port):
    print "Server starting"
    sock = socket(AF_INET,SOCK_STREAM)
    sock.bind(("",port))
    sock.listen(5)
    while True:
        client,addr = sock.accept()
        yield NewTask(handle_client(client,addr))
```


An Echo Server Attempt

```
def handle_client(client,addr):
    print "Connection from", addr
    while True:
        data = client.recv(65536)
        if not data:
            break
        client.send(data)
    client.close()
    print "Client closed"
    yield           # Make the function a generator/coroutine

def server(port):
    print "Server starting"
    sock = socket(AF_INET,SOCK_STREAM)
    sock.bind(("",port))
    sock.listen(5)
    while True:
        client,addr = sock.accept()
        yield NewTask(handle_client(client,addr))
```

The main server loop. Wait for a connection, launch a new task to handle each client.

An Echo Server Attempt

```
def handle_client(client,addr):
    print "Connection from", addr
    while True:
        data = client.recv(65536)
        if not data:
            break
        client.send(data)
    client.close()
    print "Client closed"
    yield           # Make the function a generator/coroutine

def server(port):
    print "Server starting"
    sock = socket(AF_INET,SOCK_STREAM)
    sock.bind(("",port))
    sock.listen(5)
    while True:
        client,addr = sock.accept()
        yield NewTask(handle_client(client,addr))
```

Client handling. Each client will be executing this task (in theory)

Echo Server Example

- Execution test

```
def alive():
        while True:
            print "I'm alive!"
            yield
sched = Scheduler()
sched.new(alive())
sched.new(server(45000))
sched.mainloop()
```

- Output

```
I'm alive!
Server starting
... (freezes) ...
```

- The scheduler locks up and never runs any more tasks (bummer)

Blocking Operations

- In the example various I/O operations block

```
client,addr = sock.accept()
data = client.recv(65536)
client.send(data)
```

- The real operating system (e.g., Linux) suspends the entire Python interpreter until the I/O operation completes
- Clearly this is pretty undesirable for our multitasking operating system (any blocking operation freezes the whole program)

Non-blocking I/O

- The select module can be used to monitor a collection of sockets (or files) for activity

```
reading = []    # List of sockets waiting for read
writing = []    # List of sockets waiting for write

# Poll for I/O activity
r,w,e = select.select(reading,writing,[],timeout)

# r is list of sockets with incoming data
# w is list of sockets ready to accept outgoing data
# e is list of sockets with an error state
```

- This can be used to add I/O support to our OS
- This is going to be similar to task waiting

Step 6 : I/O Waiting

pyos7.py

```
class Scheduler(object):
    def __init__(self):
        ...
        self.read_waiting = {}
        self.write_waiting = {}
        ...

    def waitforread(self,task,fd):
        self.read_waiting[fd] = task
    def waitforwrite(self,task,fd):
        self.write_waiting[fd] = task

    def iopoll(self,timeout):
        if self.read_waiting or self.write_waiting:
           r,w,e = select.select(self.read_waiting,
                                 self.write_waiting,[],timeout)
           for fd in r: self.schedule(self.read_waiting.pop(fd))
           for fd in w: self.schedule(self.write_waiting.pop(fd))
    ...
```


Step 6 : I/O Waiting

```
class Scheduler(object):
    def __init__(self):
        ...
        self.read_waiting = {}
        self.write_waiting = {}
        ...

    def waitforread(self,task,fd):
        self.read_waiting[fd] = task
    def waitforwrite(self,task,fd):
        self.write_waiting[fd] = task

    def iopoll(self,timeout):
        if self.read_waiting or self.write_waiting:
           r,w,e = select.select(self.read_waiting,
                                 self.write_waiting,[],timeout)
           for fd in r: self.schedule(self.read_waiting.pop(fd))
           for fd in w: self.schedule(self.write_waiting.pop(fd))
    ...
```

Holding areas for tasks blocking on I/O. These are dictionaries mapping file descriptors to tasks

Step 6 : I/O Waiting

```
class Scheduler(object):
    def __init__(self):
        ...
        self.read_waiting = {}
        self.write_waiting = {}
        ...

    def waitforread(self,task,fd):
        self.read_waiting[fd] = task
    def waitforwrite(self,task,fd):
        self.write_waiting[fd] = task

    def iopoll(self,timeout):
        if self.read_waiting or self.write_waiting:
           r,w,e = select.select(self.read_waiting,
                                 self.write_waiting,[],timeout)
           for fd in r: self.schedule(self.read_waiting.pop(fd))
           for fd in w: self.schedule(self.write_waiting.pop(fd))
    ...
```

Functions that simply put a task into one of the above dictionaries

Step 6 : I/O Waiting

```
class Scheduler(object):
    def __init__(self):
        ...
        self.read_waiting = {}
        self.write_waiting = {}
        ...

    def waitforread(self,task,fd):
        self.read_waiting[fd] = task
    def waitforwrite(self,task,fd):
        self.write_waiting[fd] = task

    def iopoll(self,timeout):
        if self.read_waiting or self.write_waiting:
           r,w,e = select.select(self.read_waiting,
                                 self.write_waiting,[],timeout)
           for fd in r: self.schedule(self.read_waiting.pop(fd))
           for fd in w: self.schedule(self.write_waiting.pop(fd))
    ...
```

I/O Polling. Use select() to determine which file descriptors can be used. Unblock any associated task.

When to Poll?

- Polling is actually somewhat tricky.
- You could put it in the main event loop

```
class Scheduler(object):
    ...
    def mainloop(self):
         while self.taskmap:
             self.iopoll(0)
             task = self.ready.get()
             try:
                 result = task.run()
```

- Problem : This might cause excessive polling
- Especially if there are a lot of pending tasks already on the ready queue

A Polling Task

- An alternative: put I/O polling in its own task

```
class Scheduler(object):
    ...
    def iotask(self):
        while True:
            if self.ready.empty():
                self.iopoll(None)
            else:
                self.iopoll(0)
            yield

    def mainloop(self):
        self.new(self.iotask())   # Launch I/O polls
        while self.taskmap:
              task = self.ready.get()
              ...
```

- This just runs with every other task (neat)

Read/Write Syscalls

- Two new system calls

pyos7.py

```
class ReadWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self):
        fd = self.f.fileno()
        self.sched.waitforread(self.task,fd)

class WriteWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self):
        fd = self.f.fileno()
        self.sched.waitforwrite(self.task,fd)
```

- These merely wait for I/O events, but do not actually perform any I/O

A New Echo Server

echogood.py

```
def handle_client(client,addr):
    print "Connection from", addr
    while True:
        yield ReadWait(client)
        data = client.recv(65536)
        if not data:
            break
        yield WriteWait(client)
        client.send(data)
    client.close()
    print "Client closed"

def server(port):
    print "Server starting"
    sock = socket(AF_INET,SOCK_STREAM)
    sock.bind(("",port))
    sock.listen(5)
    while True:
        yield ReadWait(sock)
        client,addr = sock.accept()
        yield NewTask(handle_client(client,addr))
```

All I/O operations are now preceded by a waiting system call

Echo Server Example

- Execution test

```
def alive():
        while True:
            print "I'm alive!"
            yield
sched = Scheduler()
sched.new(alive())
sched.new(server(45000))
sched.mainloop()
```

- You will find that it now works (will see alive messages printing and you can connect)
- Remove the alive() task to get rid of messages

echogood2.py

Congratulations!

- You have just created a multitasking OS
- Tasks can run concurrently
- Tasks can create, destroy, and wait for tasks
- Tasks can perform I/O operations
- You can even write a concurrent server
- Excellent!

Part 8

The Problem with the Stack

A Limitation

- When working with coroutines, you can't write subroutine functions that yield (suspend)
- For example:

```
def Accept(sock):
    yield ReadWait(sock)
    return sock.accept()

def server(port):
    ...
    while True:
        client,addr = Accept(sock)
        yield NewTask(handle_client(client,addr))
```

- The control flow just doesn't work right

A Problem

- The yield statement can only be used to suspend a coroutine at the top-most level
- You can't push yield inside library functions

```
def bar():
    yield

def foo():
    bar()
```

This yield does not suspend the "task" that called the bar() function (i.e., it does not suspend foo)

- Digression: This limitation is one of the things that is addressed by Stackless Python

A Solution

- There is a way to create suspendable subroutines and functions
- However, it can only be done with the assistance of the task scheduler itself
- You have to strictly stick to yield statements
- Involves a trick known as "trampolining"

Coroutine Trampolining

- Here is a very simple example:

trampoline.py

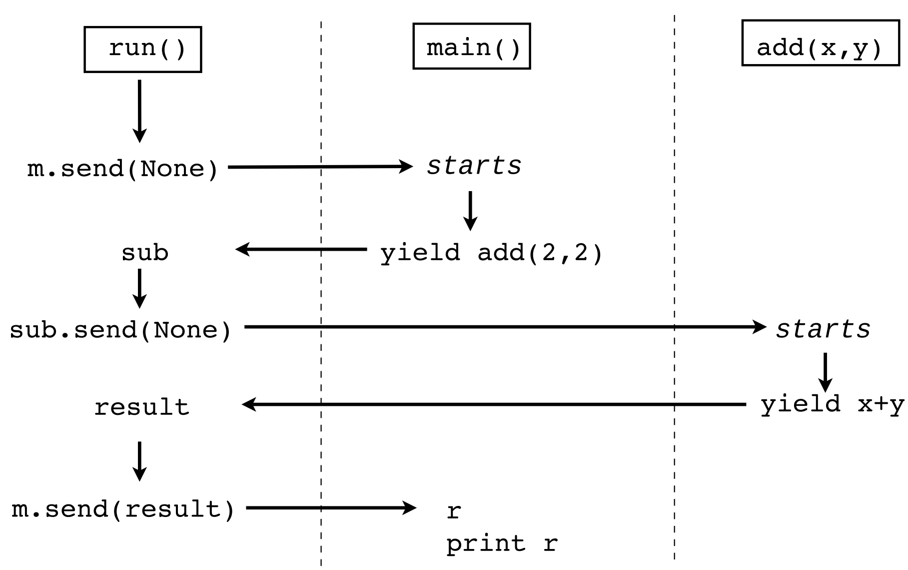
```
# A subroutine
def add(x,y):
    yield x+y

# A function that calls a subroutine
def main():
    r = yield add(2,2)
    print r
    yield
```

- Here is very simpler scheduler code

```
def run():
    m      = main()
    # An example of a "trampoline"
    sub    = m.send(None)
    result = sub.send(None)
    m.send(result)
```

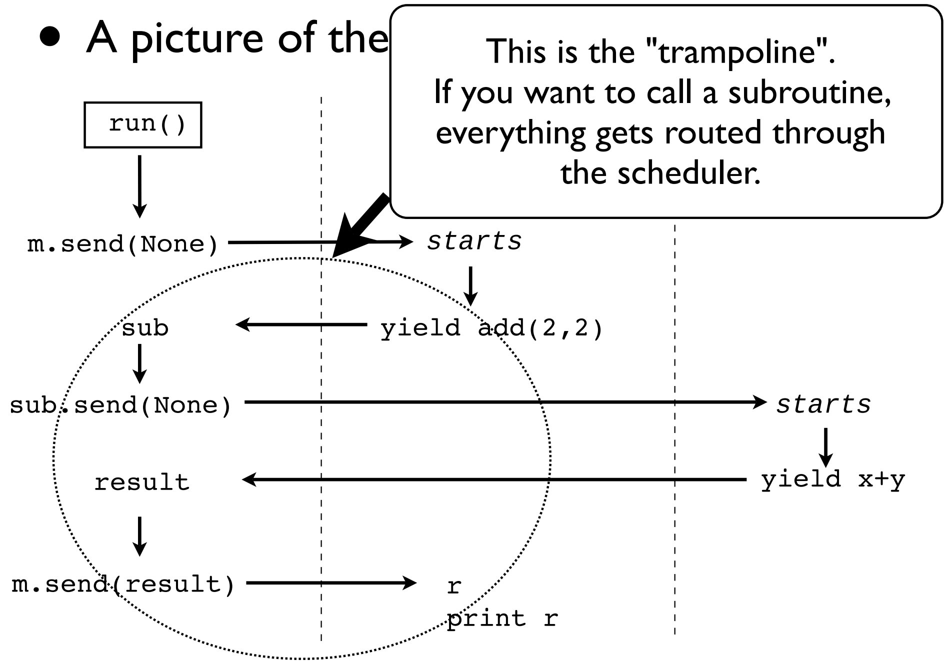

Coroutine Trampolining

- A picture of the control flow



Coroutine Trampolining

- A picture of the



An Implementation

pyos8.py

```
class Task(object):
    def __init__(self,target):
        ...
        self.stack = []
    def run(self):
        while True:
            try:
                result = self.target.send(self.sendval)
                if isinstance(result,SystemCall): return result
                if isinstance(result,types.GeneratorType):
                    self.stack.append(self.target)
                    self.sendval = None
                    self.target  = result
                else:
                    if not self.stack: return
                    self.sendval = result
                    self.target  = self.stack.pop()
            except StopIteration:
                if not self.stack: raise
                self.sendval = None
                self.target = self.stack.pop()
```

An Implementation

```
class Task(object):
    def __init__(self,target):
        ...
        self.stack = []
    def run(self):
        while True:
            try:
                result = self.t
                if isinstance(result,SystemCall): return result
                if isinstance(result,types.GeneratorType):
                    self.stack.append(self.target)
                    self.sendval = None
                    self.target  = result
                else:
                    if not self.stack: return
                    self.sendval = result
                    self.target  = self.stack.pop()
            except StopIteration:
                if not self.stack: raise
                self.sendval = None
                self.target = self.stack.pop()
```

If you're going to have subroutines, you first need a "call stack."

An Implementation

```
class Task(object):
    def __init__(self,target):
        ...
        self.stack = []
    def run(self):
        while True:
            try:
                result = self.target.send(self.sendval)
                if isinstance(result,SystemCall): return result
                if isinstance(result,types.GeneratorType):
                    self.stack.append(self.target)
                    self.sendval = None
                    self.target  = result
                else:
                    if not self.stack: return
                    self.sendval = result
                    self.target  = self.stack.pop()
            except StopIteration:
                if not self.stack: raise
                self.sendval = None
                self.target = self.stack.pop()
```

Here we run the task. If it returns a "System Call", just return (this is handled by the scheduler)

An Implementation

```
class Task(object):
    def __init__(self,target):
        ...
        self.stack = []
    def run(self):
        while True:
            try:
                result = self.target.send(self.sendval)
                if isinstance(result,SystemCall): return result
                if isinstance(result,types.GeneratorType):
                    self.stack.append(self.target)
                    self.sendval = None
                    self.target  = result
                else:
                    if not self
                    self.sendva
                    self.target
            except StopIteratio
                if not self.stack: raise
                self.sendval = None
                self.target = self.stack.pop()
```

If a generator is returned, it means we're going to "trampoline"

Push the current coroutine on the stack, loop back to the top, and call the new coroutine.

An Implementation

```
class Task(object):
    def __init__(self,target):
        ...
        self.stack = []
    def run(self):
        while True:
            try:
                result = sel
                if isinstance
                if isinstance
                    self.stac
                    self.senc
                    self.target  = result
                else:
                    if not self.stack: return
                    self.sendval = result
                    self.target  = self.stack.pop()
            except StopIteration:
                if not self.stack: raise
                self.sendval = None
                self.target = self.stack.pop()
```

If some other value is coming back, assume it's a return value from a subroutine. Pop the last coroutine off of the stack and arrange to have the return value sent into it.

An Implementation

```
class Task(object):
    def __init__(self,target):
        ...
        self.stack = []
    def run(self):
        while True:
            try:
                result = self.target.send(self.sendval)
                if isinstanc
                if isinstanc
                    self.sta
                    self.sen
                    self.tar
                else:
                    if not s
                    self.sendval = result
                    self.target  = self.stack.pop()
            except StopIteration:
                if not self.stack: raise
                self.sendval = None
                self.target = self.stack.pop()
```

Special handling to deal with subroutines that terminate. Pop the last coroutine off the stack and continue (instead of killing the whole task)

Some Subroutines

- Blocking I/O can be put inside library functions

pyos8.py

```
def Accept(sock):
    yield ReadWait(sock)
    yield sock.accept()

def Send(sock,buffer):
    while buffer:
        yield WriteWait(sock)
        len = sock.send(buffer)
        buffer = buffer[len:]

def Recv(sock,maxbytes):
    yield ReadWait(sock)
    yield sock.recv(maxbytes)
```

- These hide all of the low-level details.

A Better Echo Server

```
def handle_client(client,addr):
    print "Connection from", addr
    while True:
        data = yield Recv(client,65536)
        if not data:
            break
        yield Send(client,data)
    print "Client closed"
    client.close()

def server(port):
    print "Server starting"
    sock = socket(AF_INET,SOCK_STREAM)
    sock.bind(("",port))
    sock.listen(5)
    while True:
        client,addr = yield Accept(sock)
        yield NewTask(handle_client(client,addr))
```

echoserver.py

Notice how all I/O operations are now subroutines.

Some Comments

- This is insane!
- You now have two types of callables
    - Normal Python functions/methods
    - Suspendable coroutines
- For the latter, you always have to use yield for both calling and returning values
- The code looks really weird at first glance

Coroutines and Methods

- You can take this further and implement wrapper objects with non-blocking I/O

sockwrap.py

```
class Socket(object):
    def __init__(self,sock):
        self.sock = sock
    def accept(self):
        yield ReadWait(self.sock)
        client,addr = self.sock.accept()
        yield Socket(client),addr
    def send(self,buffer):
        while buffer:
            yield WriteWait(self.sock)
            len = self.sock.send(buffer)
            buffer = buffer[len:]
    def recv(self, maxbytes):
        yield ReadWait(self.sock)
        yield self.sock.recv(maxbytes)
    def close(self):
        yield self.sock.close()
```


A Final Echo Server

echoserver2.py

```
def handle_client(client,addr):
    print "Connection from", addr
    while True:
        data = yield client.recv(65536)
        if not data:
            break
        yield client.send(data)
    print "Client closed"
    yield client.close()

def server(port):
    print "Server starting"
    rawsock = socket(AF_INET,SOCK_STREAM)
    rawsock.bind(("",port))
    rawsock.listen(5)
    sock = Socket(rawsock)
    while True:
        client,addr = yield sock.accept()
        yield NewTask(handle_client(client,addr))
```

Notice how all I/O operations now mimic the socket API except for the extra yield.

An Interesting Twist

- If you only read the application code, it has normal looking control flow!

Coroutine Multitasking

```
while True:
  data = yield client.recv(8192)
  if not data:
     break
  yield client.send(data)
yield client.close()
```

Traditional Socket Code

```
while True:
  data = client.recv(8192)
  if not data:
     break
  client.send(data)
client.close()
```

- As a comparison, you might look at code that you would write using the asyncore module (or anything else that uses event callbacks)

Example : Twisted

- Here is an echo server in Twisted (straight from the manual)

```
from twisted.internet.protocol import Protocol, Factory
from twisted.internet import reactor

class Echo(Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

def main():
    f = Factory()
    f.protocol = Echo
    reactor.listenTCP(45000, f)
    reactor.run()

if __name__ == '__main__':
    main()
```

An event callback

Part 9

Some Final Words

Further Topics

- There are many other topics that one could explore with our task scheduler
    - Intertask communication
    - Handling of blocking operations (e.g., accessing databases, etc.)
    - Coroutine multitasking and threads
    - Error handling
- But time does not allow it here

A Little Respect

- Python generators are far more powerful than most people realize
  - Customized iteration patterns
  - Processing pipelines and data flow
  - Event handling
  - Cooperative multitasking
- It's too bad a lot of documentation gives little insight to applications (death to Fibonacci!)

Performance

- Coroutines have decent performance
- We saw this in the data processing section
- For networking, you might put our coroutine server up against a framework like Twisted
- A simple test : Launch 3 subprocesses, have each open 300 socket connections and randomly blast the echo server with 1024 byte messages.

Twisted	420.7s
Coroutines	326.3s
Threads	42.8s

Note : This is only one test. A more detailed study is definitely in order.

Coroutines vs. Threads

- I'm not convinced that using coroutines is actually worth it for general multitasking
- Thread programming is already a well established paradigm
- Python threads often get a bad rap (because of the GIL), but it is not clear to me that writing your own multitasker is actually better than just letting the OS do the task switching

A Risk

- Coroutines were initially developed in the 1960's and then just sort of died quietly
- Maybe they died for a good reason
- I think a reasonable programmer could claim that programming with coroutines is just too diabolical to use in production software
- Bring my multitasking OS (or anything else involving coroutines) into a code review and report back to me... ("You're FIRED!")

Keeping it Straight

- If you are going to use coroutines, it is critically important to not mix programming paradigms together
- There are three main uses of yield
    - Iteration (a producer of data)
    - Receiving messages (a consumer)
    - A trap (cooperative multitasking)
- Do <u>NOT</u> write generator functions that try to do more than one of these at once

Handle with Care

- I think coroutines are like high explosives
- Try to keep them carefully contained
- Creating a ad-hoc tangled mess of coroutines, objects, threads, and subprocesses is probably going to end in disaster
- For example, in our OS, coroutines have no access to any internals of the scheduler, tasks, etc. <u>This is good</u>.

Some Links

- Some related projects (not an exhaustive list)
  - Stackless Python, PyPy
  - Cogen
  - Multitask
  - Greenlet
  - Eventlet
  - Kamaelia
- Do a search on http://pypi.python.org

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)