

Iteration

- As you know, Python has a "for" statement
- You use it to loop over a collection of items

```
>>> for x in [1,4,5,10]:  
...     print x,  
...  
1 4 5 10  
>>>
```

- And, as you have probably noticed, you can iterate over many different kinds of objects (not just lists)

Iterating over a Dict

- If you loop over a dictionary you get keys

```
>>> prices = { 'GOOG' : 490.10,  
...            'AAPL' : 145.23,  
...            'YHOO' : 21.71 }  
...  
>>> for key in prices:  
...     print key  
...  
YHOO  
GOOG  
AAPL  
>>>
```

Iterating over a String

- If you loop over a string, you get characters

```
>>> s = "Yow!"
>>> for c in s:
...     print c
...
Y
O
W
!
>>>
```

Iterating over a File

- If you loop over a file you get lines

```
>>> for line in open("real.txt"):
...     print line,
... 
```

Real Programmers write in FORTRAN

Maybe they do now,
in this decadent era of
Lite beer, hand calculators, and "user-friendly" software
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes
Real Programmers wrote in machine code.
Not FORTRAN. Not RATFOR. Not, even, assembly language
Machine Code.
Raw, unadorned, inscrutable hexadecimal numbers.
Directly.

Consuming Iterables

- Many functions consume an "iterable" object
- Reductions:

`sum(s), min(s), max(s)`

- Constructors

`list(s), tuple(s), set(s), dict(s)`

- in operator

`item in s`

- Many others in the library

Iteration Protocol

- The reason why you can iterate over different objects is that there is a specific protocol

```
>>> items = [1, 4, 5]
>>> it = iter(items)
>>> it.next()
1
>>> it.next()
4
>>> it.next()
5
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Iteration Protocol

- An inside look at the for statement

```
for x in obj:  
    # statements
```

- Underneath the covers

```
_iter = iter(obj)                # Get iterator object  
while 1:  
    try:  
        x = _iter.next()         # Get next item  
    except StopIteration:        # No more items  
        break  
    # statements  
    ...
```

- Any object that supports `iter()` and `next()` is said to be "iterable."

Supporting Iteration

- User-defined objects can support iteration
- Example: Counting down...

```
>>> for x in countdown(10):  
...     print x,  
...  
10 9 8 7 6 5 4 3 2 1  
>>>
```

- To do this, you just have to make the object implement `__iter__()` and `next()`

Supporting Iteration

- Sample implementation

```
class countdown(object):
    def __init__(self, start):
        self.count = start
    def __iter__(self):
        return self
    def next(self):
        if self.count <= 0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r
```

Iteration Example

- Example use:

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

Iteration Commentary

- There are many subtle details involving the design of iterators for various objects
- However, we're not going to cover that
- This isn't a tutorial on "iterators"
- We're talking about generators...

Generators

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

```
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)

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  
>>>
```

Generator Functions

- A generator function is mainly a more convenient way of writing an iterator
- You don't have to worry about the iterator protocol (.next, `__iter__`, etc.)
- It just works

Generators vs. Iterators

- A generator function is slightly different than an object that supports iteration
- A generator is a one-time operation. You can iterate over the generated data once, but if you want to do it again, you have to call the generator function again.
- This is different than a list (which you can iterate over as many times as you want)

Generator Expressions

- A generated version of a list comprehension

```
>>> a = [1,2,3,4]
>>> b = (2*x for x in a)
>>> b
<generator object at 0x58760>
>>> for i in b: print b,
...
2 4 6 8
>>>
```

- This loops over a sequence of items and applies an operation to each item
- However, results are produced one at a time using a generator

Generator Expressions

- Important differences from a list comp.
 - Does not construct a list.
 - Only useful purpose is iteration
 - Once consumed, can't be reused
- Example:

```
>>> a = [1,2,3,4]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8]
>>> c = (2*x for x in a)
<generator object at 0x58760>
>>>
```

Generator Expressions

- General syntax

```
(expression for i in s if cond1
    for j in t if cond2
    ...
    if condfinal)
```

- What it means

```
for i in s:
    if cond1:
        for j in t:
            if cond2:
                ...
            if condfinal: yield expression
```

A Note on Syntax

- The parens on a generator expression can be dropped if used as a single function argument
- Example:

```
sum(x*x for x in s)
```



Generator expression

Interlude

- We now have two basic building blocks
- Generator functions:

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

- Generator expressions

```
squares = (x*x for x in s)
```

- In both cases, we get an object that generates values (which are typically consumed in a for loop)

Programming Problem

Find out how many bytes of data were transferred by summing up the last column of data in this Apache web server log

```
81.107.39.38 - ... "GET /ply/ HTTP/1.1" 200 7587
81.107.39.38 - ... "GET /favicon.ico HTTP/1.1" 404 133
81.107.39.38 - ... "GET /ply/bookplug.gif HTTP/1.1" 200 23903
81.107.39.38 - ... "GET /ply/ply.html HTTP/1.1" 200 97238
81.107.39.38 - ... "GET /ply/example.html HTTP/1.1" 200 2359
66.249.72.134 - ... "GET /index.html HTTP/1.1" 200 4447
```

Oh yeah, and the log file might be huge (Gbytes)

The Log File

- Each line of the log looks like this:

```
81.107.39.38 - ... "GET /ply/ply.html HTTP/1.1" 200 97238
```

- The number of bytes is the last column

```
bytestr = line.rsplit(None,1)[1]
```

- It's either a number or a missing value (-)

```
81.107.39.38 - ... "GET /ply/ HTTP/1.1" 304 -
```

- Converting the value

```
if bytestr != '-':  
    bytes = int(bytestr)
```

A Non-Generator Soln

- Just do a simple for-loop

```
wwwlog = open("access-log")
total = 0
for line in wwwlog:
    bytestr = line.rsplit(None,1)[1]
    if bytestr != '-':
        total += int(bytestr)

print "Total", total
```

- We read line-by-line and just update a sum
- However, that's so 90s...

A Generator Solution

- Let's use some generator expressions

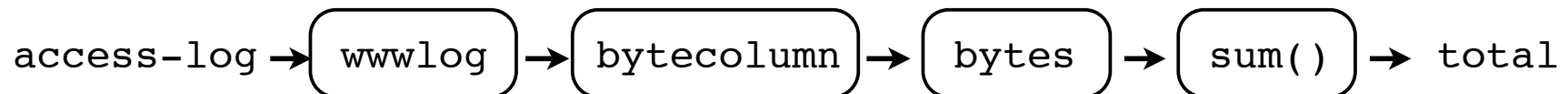
```
wwwlog      = open("access-log")
bytecolumn  = (line.rsplit(None,1)[1] for line in wwwlog)
bytes       = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

- Whoa! That's different!
 - Less code
 - A completely different programming style

Generators as a Pipeline

- To understand the solution, think of it as a data processing pipeline



- Each step is defined by iteration/generation

```
wwwlog      = open("access-log")
bytecolumn  = (line.rsplit(None,1)[1] for line in wwwlog)
bytes       = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Being Declarative

- At each step of the pipeline, we declare an operation that will be applied to the entire input stream

access-log → **wwwlog** → **bytecolumn** → **bytes** → **sum()** → total

`bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)`

This operation gets applied to
every line of the log file

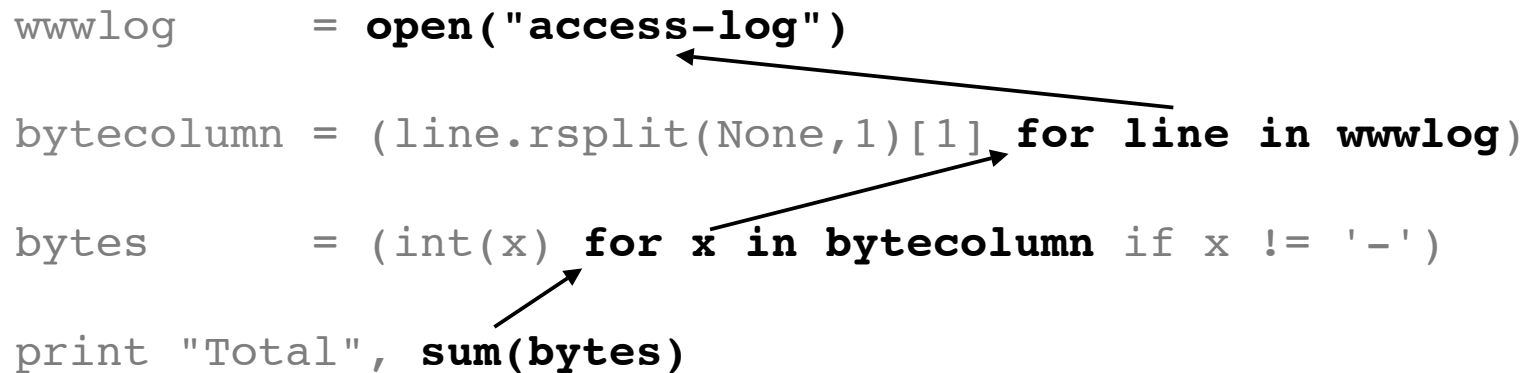
Being Declarative

- Instead of focusing on the problem at a line-by-line level, you just break it down into big operations that operate on the whole file
- This is very much a "declarative" style
- The key :Think big...

Iteration is the Glue

- The glue that holds the pipeline together is the iteration that occurs in each step

```
wwwlog      = open("access-log")  
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)  
bytes       = (int(x) for x in bytecolumn if x != '-')  
print "Total", sum(bytes)
```



- The calculation is being driven by the last step
- The sum() function is consuming values being pushed through the pipeline (via .next() calls)

Performance

- Surely, this generator approach has all sorts of fancy-dancy magic that is slow.
- Let's check it out on a 1.3Gb log file...

```
% ls -l big-access-log  
-rw-r--r-- beazley 1303238000 Feb 29 08:06 big-access-log
```

Performance Contest

```
wwwlog = open("big-access-log")
total = 0
for line in wwwlog:
    bytestr = line.rsplit(None,1)[1]
    if bytestr != '-':
        total += int(bytestr)

print "Total", total
```

Time

27.20

```
wwwlog      = open("big-access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes       = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Time

25.96

Commentary

- Not only was it not slow, it was 5% faster
- And it was less code
- And it was relatively easy to read
- And frankly, I like it a whole better...

"Back in the old days, we used AWK for this and we liked it. Oh, yeah, and get off my lawn!"

Performance Contest

```
wwwlog      = open("access-log")
bytecolumn  = (line.rsplit(None,1)[1] for line in wwwlog)
bytes       = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Time

25.96

```
% awk '{ total += $NF } END { print total }' big-access-log
```

Note:extracting the last
column may not be
awk's strong point

Time

37.33

Food for Thought

- At no point in our generator solution did we ever create large temporary lists
- Thus, not only is that solution faster, it can be applied to enormous data files
- It's competitive with traditional tools

More Thoughts

- The generator solution was based on the concept of pipelining data between different components
- What if you had more advanced kinds of components to work with?
- Perhaps you could perform different kinds of processing by just plugging various pipeline components together

This Sounds Familiar

- The Unix philosophy
- Have a collection of useful system utils
- Can hook these up to files or each other
- Perform complex tasks by piping data

Programming Problem

You have hundreds of web server logs scattered across various directories. In addition, some of the logs are compressed. Modify the last program so that you can easily read all of these logs

```
foo/  
    access-log-012007.gz  
    access-log-022007.gz  
    access-log-032007.gz  
    ...  
    access-log-012008  
bar/  
    access-log-092007.bz2  
    ...  
    access-log-022008
```

os.walk()

- A very useful function for searching the file system

```
import os
```

```
for path, dirlist, filelist in os.walk(topdir):  
    # path      : Current directory  
    # dirlist   : List of subdirectories  
    # filelist  : List of files  
    ...
```

- This utilizes generators to recursively walk through the file system

find

- Generate all filenames in a directory tree that match a given filename pattern

```
import os
import fnmatch

def gen_find(filepat,top):
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist,filepat):
            yield os.path.join(path,name)
```

- Examples

```
pyfiles = gen_find("*.py","/")
logs    = gen_find("access-log*", "/usr/www/")
```

A File Opener

- Open a sequence of filenames

```
import gzip, bz2
def gen_open(filenames):
    for name in filenames:
        if name.endswith(".gz"):
            yield gzip.open(name)
        elif name.endswith(".bz2"):
            yield bz2.BZ2File(name)
        else:
            yield open(name)
```

- This is interesting.... it takes a sequence of filenames as input and yields a sequence of open file objects

cat

- Concatenate items from one or more source into a single sequence of items

```
def gen_cat(sources):  
    for s in sources:  
        for item in s:  
            yield item
```

- Example:

```
lognames = gen_find("access-log*", "/usr/www")  
logfiles = gen_open(lognames)  
loglines = gen_cat(logfiles)
```

grep

- Generate a sequence of lines that contain a given regular expression

```
import re

def gen_grep(pat, lines):
    patc = re.compile(pat)
    for line in lines:
        if patc.search(line): yield line
```

- Example:

```
lognames = gen_find("access-log*", "/usr/www")
logfiles = gen_open(lognames)
loglines = gen_cat(logfiles)
patlines = gen_grep(pat, loglines)
```

Example

- Find out how many bytes transferred for a specific pattern in a whole directory of logs

```
pat          = r"somepattern"
logdir       = "/some/dir/"

filenames    = gen_find("access-log*", logdir)
logfiles     = gen_open(filenames)
loglines     = gen_cat(logfiles)
patlines     = gen_grep(pat, loglines)
bytecolumn   = (line.rsplit(None, 1)[1] for line in patlines)
bytes        = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Important Concept

- Generators decouple iteration from the code that uses the results of the iteration
- In the last example, we're performing a calculation on a sequence of lines
- It doesn't matter where or how those lines are generated
- Thus, we can plug any number of components together up front as long as they eventually produce a line sequence

Part 2

Coroutines, Pipelines, and Dataflow

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

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

grep.py

- 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

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

`coroutine.py`

```
@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)

```
@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"
```

grepclose.py

- 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 and receives values

```
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
```

bogus.py

- 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
- Coroutines are not related to iteration
- Note : There is a use of having yield produce a value in a coroutine, but it's not tied to iteration.

Processing Pipelines

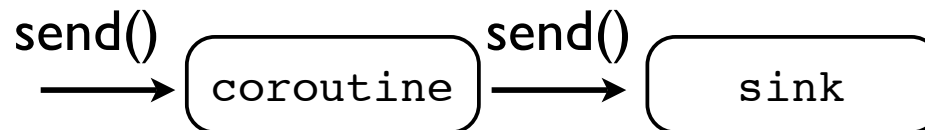
- Coroutines can be used to set up pipes



- You just chain coroutines together and push data through the pipe with `send()` operations

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'

```
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)
```

cofollow.py

- A sink that just prints the lines

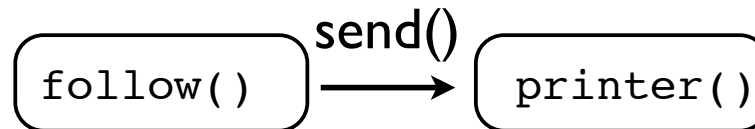
```
@coroutine
def printer():
    while True:
        line = (yield)
        print line,
```

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

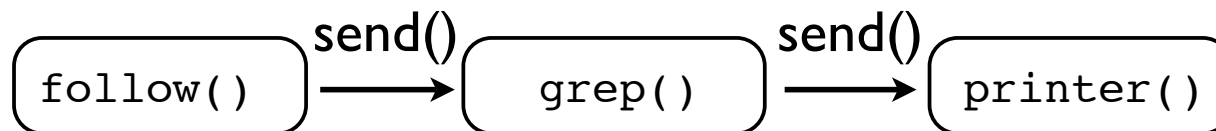
copipe.py

```
@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()))
```

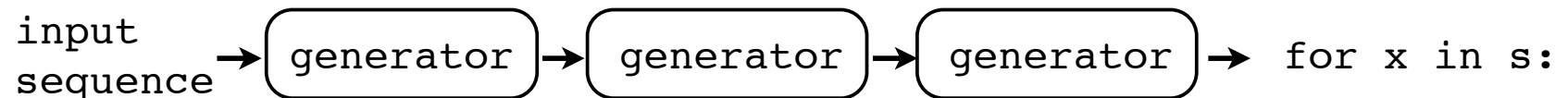
- A picture



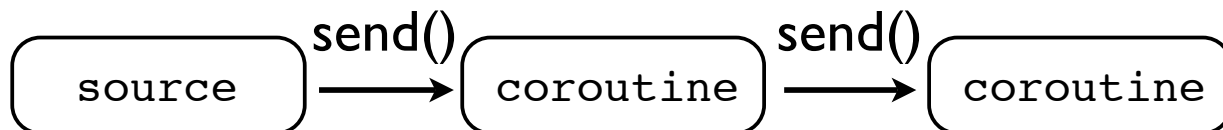
Interlude

- Coroutines flip generators around

generators/iteration



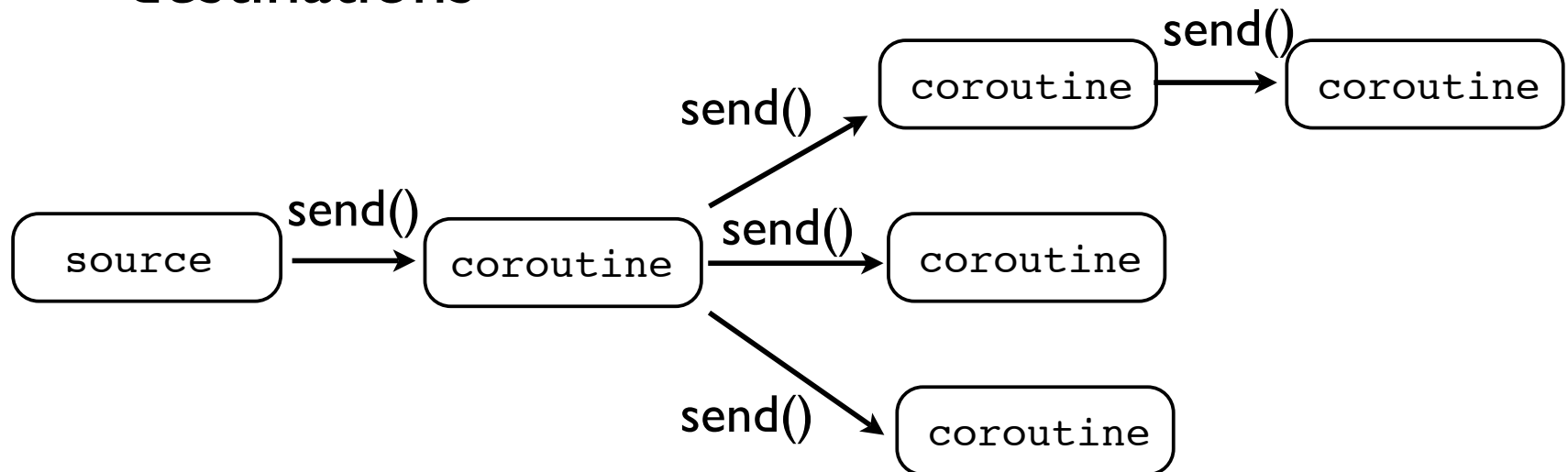
coroutines



- 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

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

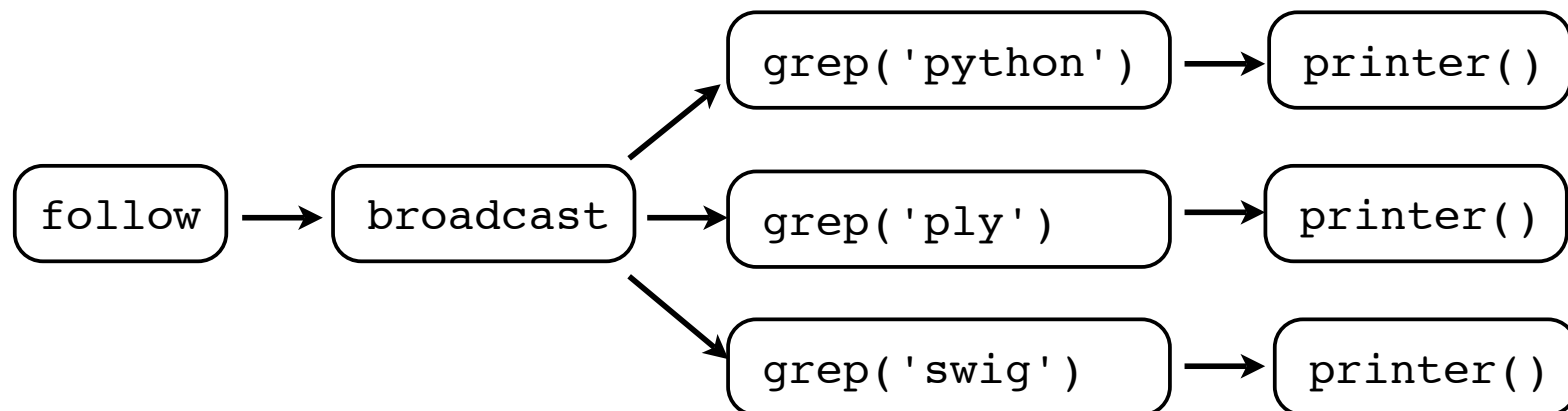
cobroadcast.py

- 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())])
)
```

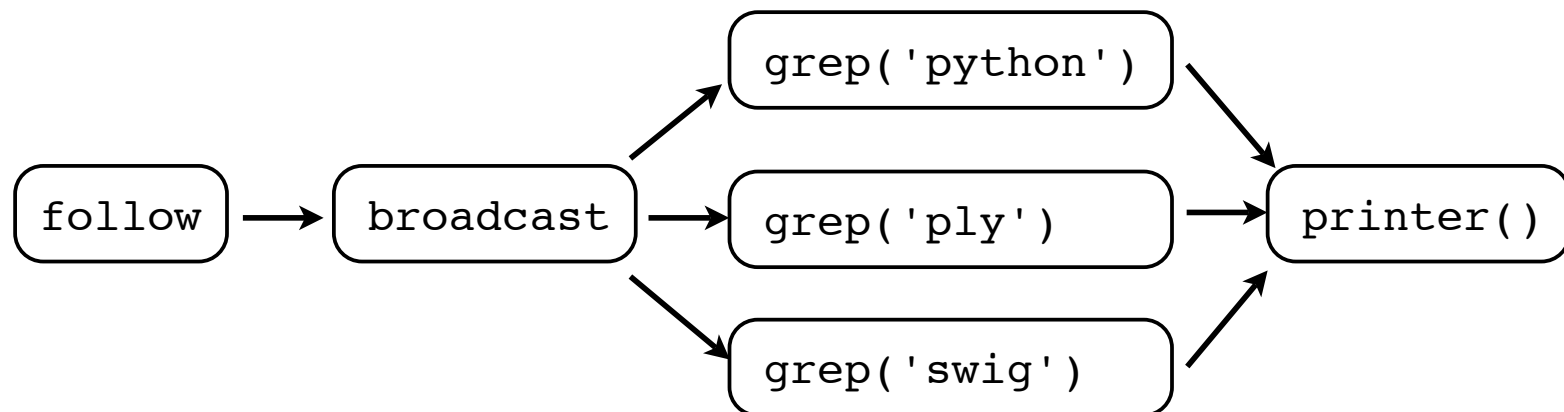


Example : Broadcasting

- A more disturbing variation...

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

cobroadcast2.py



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)

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:
        line = (yield)
        if pattern in line:
            print line,
```

locals →

- 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 does 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 not running the operating system
- Question: How does the operating system (which is not running) make an application (which is 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)` $\longrightarrow$ `read:`

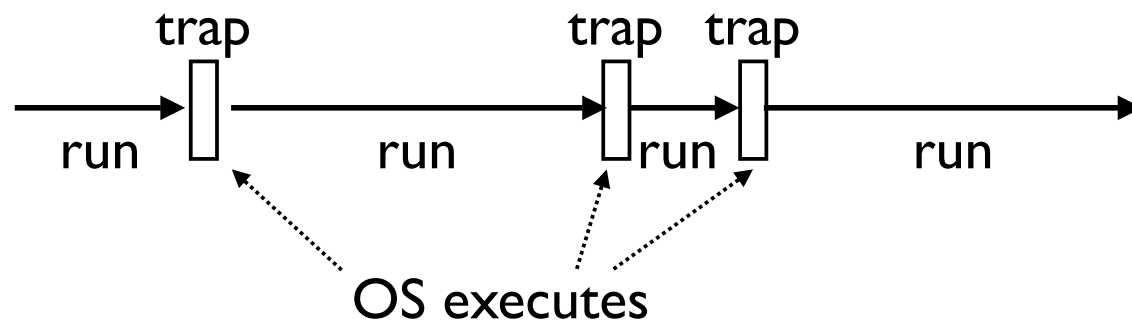
- When a trap instruction executes, the program suspends execution at that point
- And the OS takes over

```
push    %ebx
mov     0x10(%esp),%edx
mov     0xc(%esp),%ecx
mov     0x8(%esp),%ebx
mov     $0x3,%eax
int    $0x80
pop     %ebx
...
```

← trap

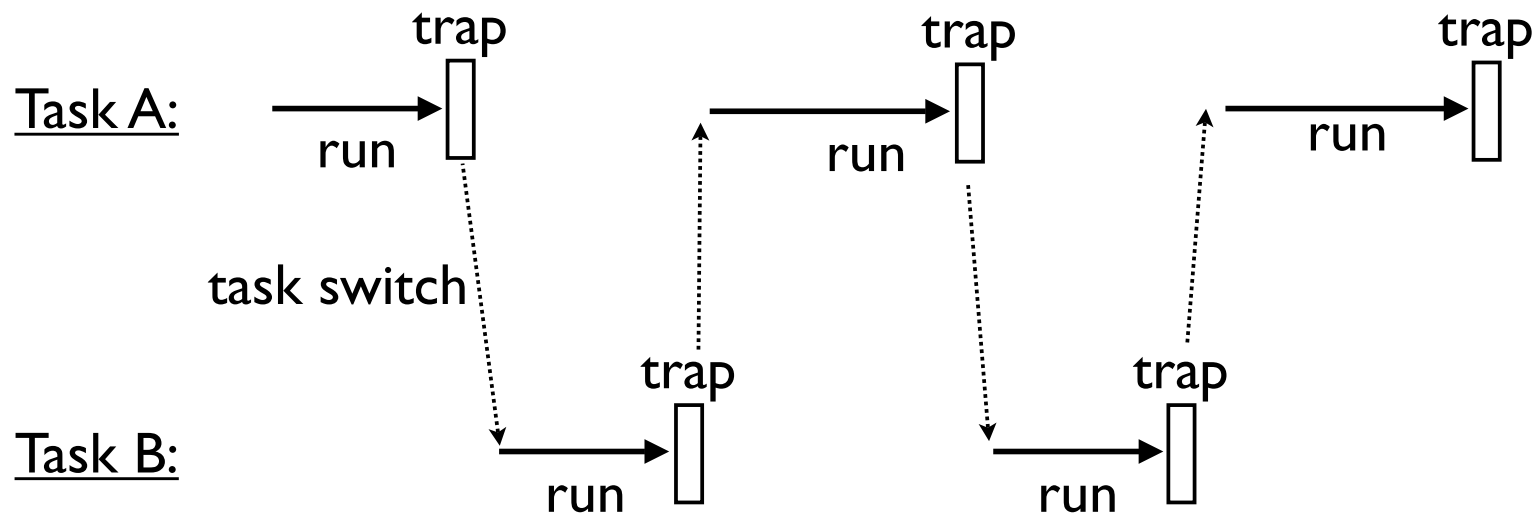
High Level Overview

- 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

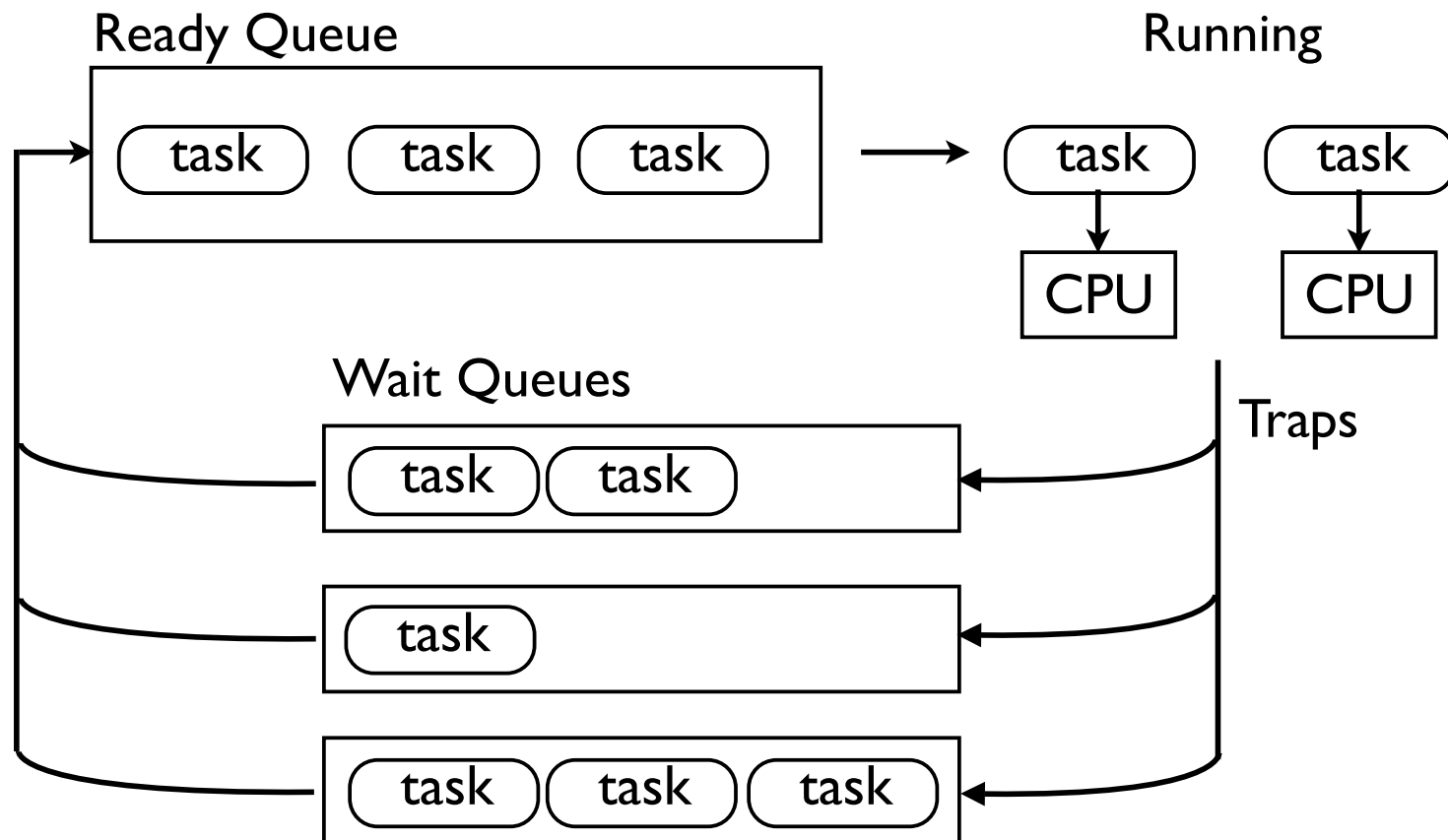
- Here's what typically happens when an OS runs multiple tasks.



- 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

```
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)
```

pyos1.py

- 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() ←
```

A queue of tasks that
are ready to run

```
        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)
```

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

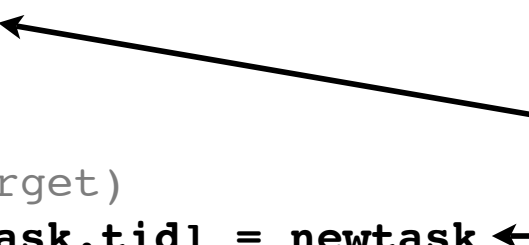
Step 2: The Scheduler

```
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

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

taskcrash.py

```
...
```

```
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

```
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)
```

pyos3.py

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)
```

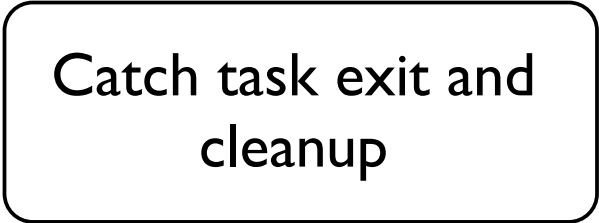
Remove the task
from the scheduler's
task map



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

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

pyos4.py

```
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)

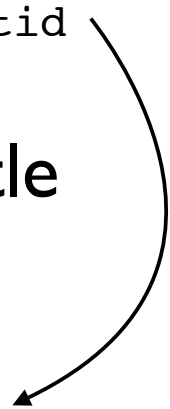
A First System Call

- 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
```

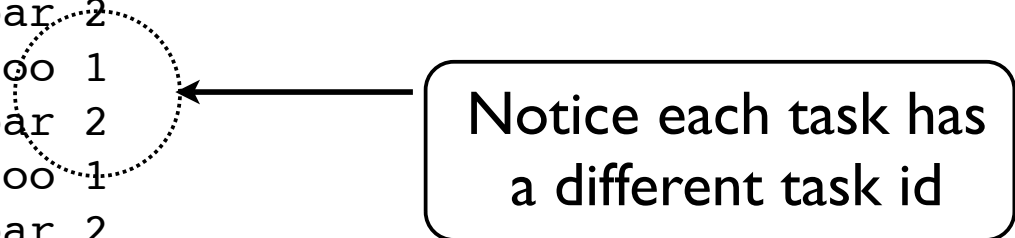
```
def bar():  
    mytid = yield GetTid()  
    for i in xrange(10):  
        print "I'm bar", mytid  
        yield
```

```
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

```
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)
```

pyos5.py

- 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
```

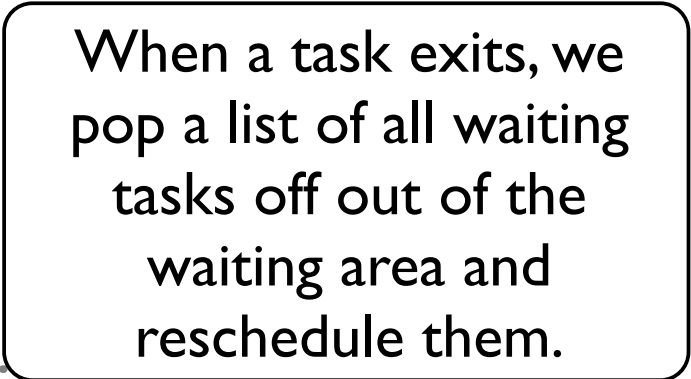
This is a holding area for tasks that are waiting. A dict mapping task ID to tasks waiting for exit.

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" % task.tid
        del self.taskmap[task.tid]
        # Notify other tasks waiting for task
        for task in self.exit_waiting[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
```

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 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
```

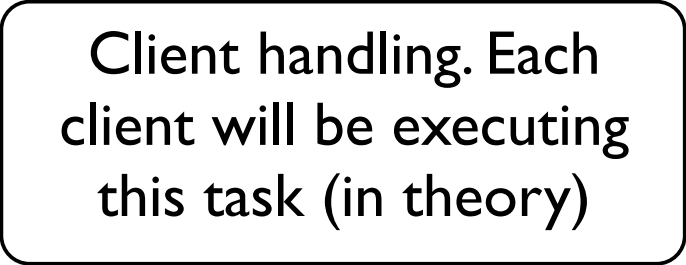
```
def server(port):  
    print "Server starting"  
    sock = socket(AF_INET,SOCK_STREAM)  
    sock.bind(("",port))  
    sock.listen(5)  
    while True:  
        client,addr = sock.accept()  
        yield NewTask(handle_client(client,addr))
```

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
```



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

```
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))
```

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

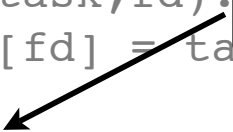
Step 6 : I/O Waiting

```
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

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

pyos7.py

```
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!