Section 11

Concurrency and Distributed Computing

Get This ➔ http://www.dabeaz.com/python/pythonmasterconcurrent.zip

Overview

- Survey of programming techniques related to concurrency and distributed computing
- A huge topic area
- Focus is on common idioms and some Python specific issues
- Not a focus: Third party libraries

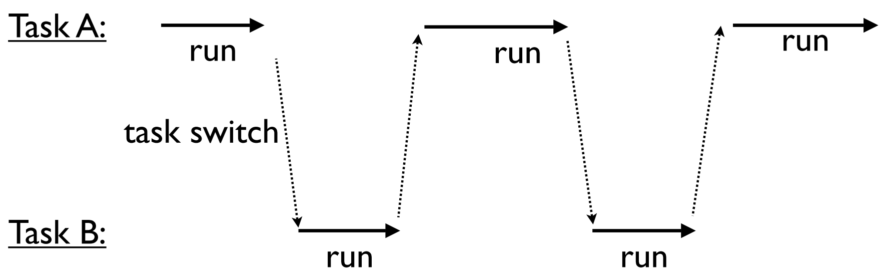
Basic Concepts

Concurrent Programming

- Applications that work on more than one thing at a time--possibly spread out over a whole cluster of machines
- Example : A network server that communicates with several hundred clients all connected at once
- Example :A big number crunching job that spreads its work across hundreds of CPUs

Multitasking

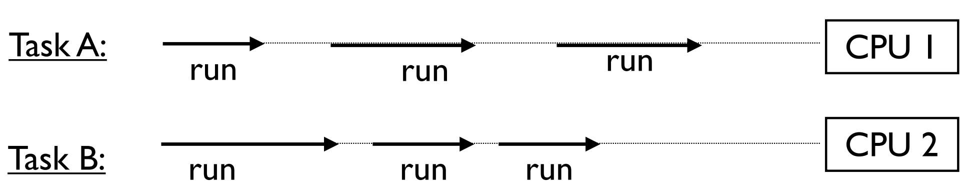
- On a single CPU, concurrency typically implies "multitasking"



- Periodic task switching

Parallel Processing

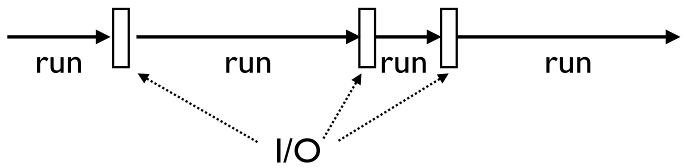
- You may have parallelism (many CPUs)



- Simultaneous task execution

Task Execution

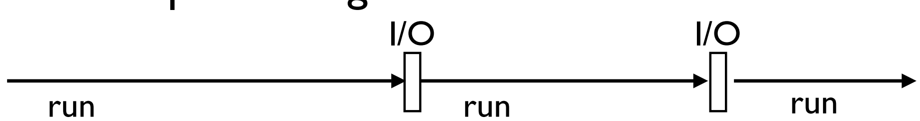
- All tasks execute by alternating between CPU processing and I/O handling



- For I/O, tasks must wait (sleep)
- Behind the scenes, the system carries out the I/O operation and wakes the task when done

CPU Bound Tasks

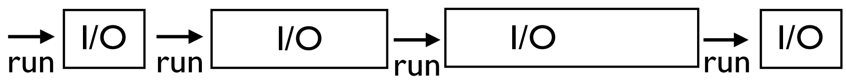
- A task is "CPU Bound" if it spends most of its time processing with little I/O



- Examples:
  - Crunching big matrices
  - Image processing

I/O Bound Tasks

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



- Examples:
  - Reading input from the user
  - Networking
  - File processing

Blocking I/O

- If an I/O operation (e.g., read or write) does not return until the operation actually completes, the operation is said to "block."
- Example : s.recv() on a network socket
- Causes a task to suspend until data available

Non-blocking I/O

- Instead of waiting, I/O operations that are going to block return immediately with an exception instead of suspending a task
- Allows a task to switch its attention to something else while waiting

Non-blocking Example

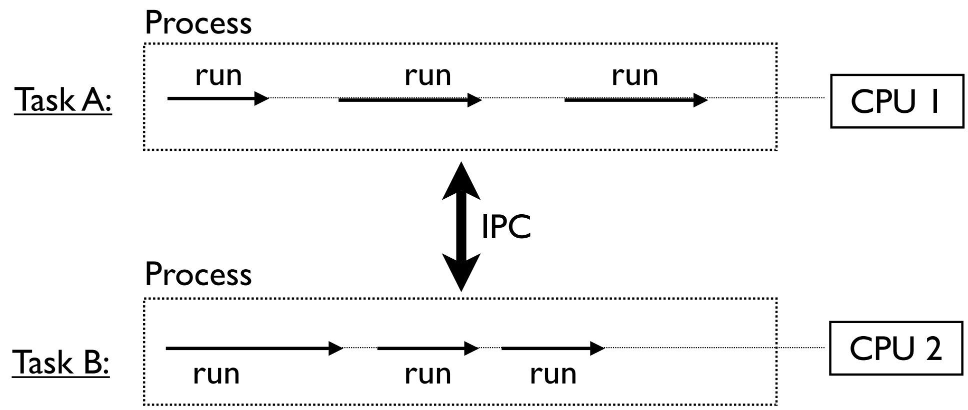
- Non-blocking socket read

```
import errno

s.setblocking(False)
try:
    data = s.recv(8192)
    ...
except socket.error as e:
    if e.errno == errno.EWOULDBLOCK:
        # Would have blocked.  Do something else
        ...
    else:
        # Some other socket error
        ...
```

- Note: Can quickly get messy (more later)

Processes

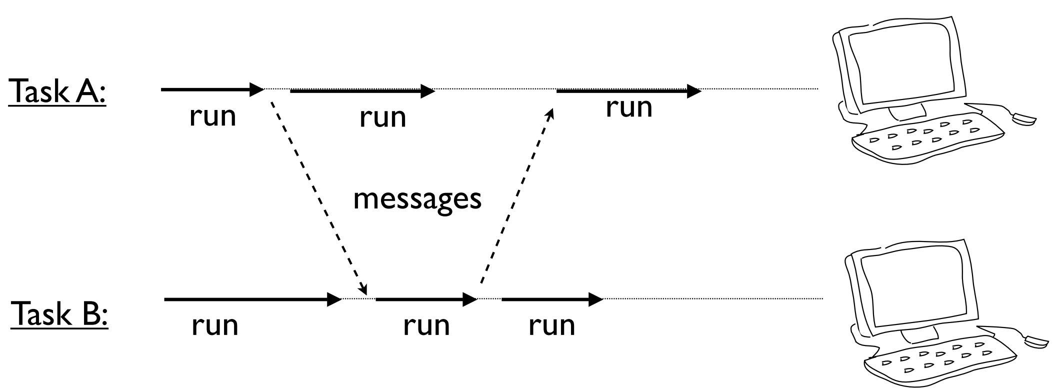
- Tasks might run in separate processes



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

Distributed Computing

- Tasks may be running on distributed systems



- For example, a cluster of workstations
- Or servers out in the "cloud."

The Landscape

- For I/O processing
  - Threads
  - Event-loops
  - Coroutines
- For CPU processing
  - Communicating processes (message passing)
  - C Extensions + Threads

Threads

Concept: Threads

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

Thread Basics

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

"main thread"

Program launch. Python loads a program and starts executing statements

Thread Basics

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

Creation of a thread.
Launches a function.

Thread Basics

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

Concurrent
execution
of statements

Thread Basics

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

thread terminates on return or exit

Thread Basics

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

thread

Key idea: Thread is like a little "task" that independently runs inside your program

Functions as threads

- How to launch a function in a thread

```
import threading

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

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

- Starts the supplied callable (target) which runs in a thread until returns

threading module

- Alternative: Define threads as a class

```
import time
import threading

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

- You inherit from Thread and redefine run()

threading module

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

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

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

- Threads execute until the run() method stops

Joining a Thread

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

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

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

Daemonic Threads

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

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

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

Exercise 11.1

What is a Thread?

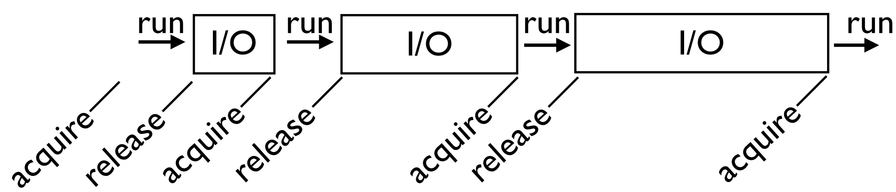
- Python threads are <u>real</u> system threads
  - POSIX threads (pthreads)
  - Windows threads
- Fully managed by the host operating system
  - All scheduling/thread switching
- Represent threaded execution of the Python interpreter process (written in C)

The Infamous GIL

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

GIL Behavior

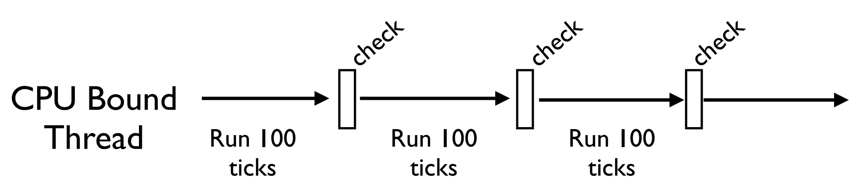
- Whenever a thread runs, it holds the GIL
- However, the GIL is released on I/O



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

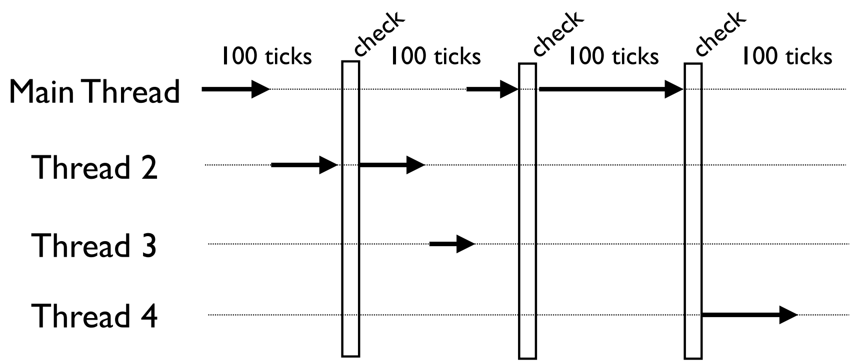
CPU Bound Processing

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



The Check Interval

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



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

The Periodic Check

- What happens during the periodic check?
  - In the main thread only, signal handlers will execute if there are any pending signals
  - Release and reacquisition of the GIL
- Periodic release of GIL allows all threads to run

What is a "Tick?"

- Ticks loosely map to interpreter instructions

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

- Instructions in the Python VM

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


Tick Execution

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

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

- Try hitting Ctrl-C (ticks are uninterruptible)

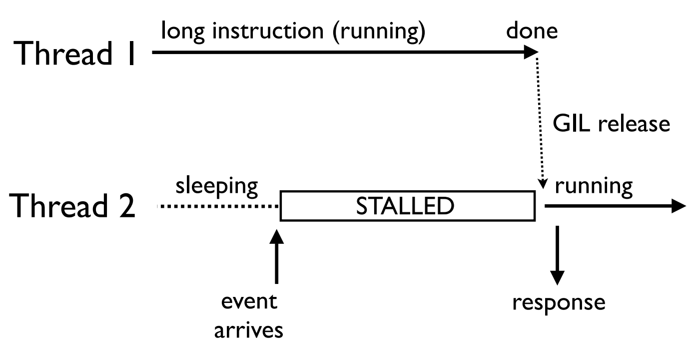
```
>>> nums = xrange(100000000)
>>> -1 in nums
^C^C^C   (nothing happens, long pause)
...
KeyboardInterrupt
>>>
```

Why You Care

- Long running instructions block progress
- Would manifest itself as an annoying "pause" in a GUI, game, or network application
- Example : A request is sent to a server, but it doesn't respond for 10 seconds

Long Running Instructions

- Illustration



- No way for a long instruction to be preempted
- All other threads stall, waiting for completion

Thread Scheduling

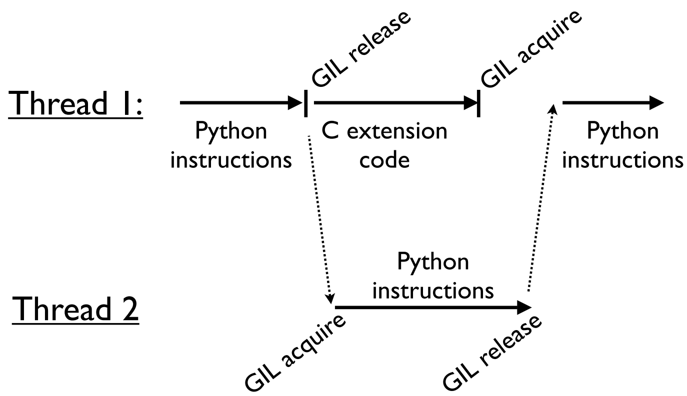
- Python does <u>not</u> have a thread scheduler
- There is no notion of thread priorities, preemption, round-robin scheduling, etc.
- All thread scheduling is left to the host operating system (e.g., Linux, Windows, etc.)

The GIL and C Code

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

The GIL and C Extensions

- Having C extensions release the GIL is one approach for CPU-bound parallel computing



More on the GIL

- I gave some talks in 2009 and 2010 that went into extensive detail on the GIL

  http://www.dabeaz.com/GIL

- There is a new GIL in Python 3.2
- Look at that material on your own

The Reality

- I'm not trying to scare you.
- Threads work great for I/O handling
- Not so much for CPU intensive work

Exercise 11.2

Interlude

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

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

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

-- Jason Whittington

Some Thread Recipes

- Rather than repeat an entire OS course...
- Will look at a few useful thread techniques
  - Thread termination
  - Manipulating shared state
  - Communicating threads
  - Thread worker pools
  - Performing Background Work

Stopping the Show

- Problem:
  - Threads can't be killed
  - Programs using threads can't be killed
- Only solution?
  - kill -9 pid

Thread Termination

- You must implement termination yourself

```
class CountdownTask(threading.Thread):
    def __init__(self, start):
        threading.Thread.__init__(self)
        self.count = start
        self.running = True

    def run(self):
        while self.count > 0 and self.running:
            print("Counting down", self.count)
            self.count -= 1
            time.sleep(5)

    def terminate(self):
        self.running = False
```

- Yes: Periodic Polling

Example

- How it works...

```
counter = CountdownTask(10)

counter.start()
...

counter.terminate()      # Set termination request
counter.join()           # Wait for thread to terminate
```

- There is no other way

Thread Termination

- Another example involving sockets

```
class ClientTask(threading.Thread):
    def __init__(self, sock):
        threading.Thread.__init__(self)
        self.sock = sock
        self.running = True

    def run(self):
        sock.settimeout(5.0)
        while self.running:
             try:
                 data = sock.recv(8192)
             except socket.timeout:
                 continue
             ...

    def terminate(self):
        self.running = False
```

All blocking operations need to rewritten with timeouts, periodic checks

Program Termination

- Problem: How to get a threaded program to stop

```
bash $ python example.py
^C^C
^Z
[4]+  Stopped                  python example.py
bash $ ps | grep python
 8136 ttys002    0:03.52 python example.py
bash $ kill -9 8136
bash $
```

- Frankly, it's super annoying (especially debugging)

Program Termination

- One Solution: Use daemonic threads and have the main thread spin uselessly

```
def main():
    ...

def mainthread():
    while True:
         time.sleep(1)

if __name__ == '__main__':
    t = threading.Thread(target=main)
    t.daemon = True
    t.start()
    mainthread()
```

- Note: All threads must be set daemonic

Program Termination

- Alternate Solution: Have main thread catch termination and attempt clean termination

```
def mainthread():
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        for t in threading.enumerate():
            if hasattr(t, 'terminate'):
                t.terminate()

if __name__ == '__main__':
    # Launch threads
    ...
    # Go spin
    mainthread()
```


Discussion

- Must understand role of the main thread
- Signals and program termination events can only be processed by the main thread
- If main thread gets blocked on a lock or stuck on I/O, there's no way to regain control
- By doing nothing in main thread, it's free to handle a more graceful program shutdown

Exercise 11.3

Accessing Shared State

- Problem: Two or more threads need to access and possibly modify a shared value
- Issue: You don't know the order in which the threads will execute (nondeterministic)

Shared State

"If there's one lesson we've learned from 30+ years of concurrent programming it is: *just don't share state*. It's like two drunkards trying to share a beer. It doesn't matter if they're good buddies. Sooner or later they're going to get into a fight. And the more drunkards you add to the pavement, the more they fight each other over the beer. The tragic majority of multithreaded applications look like drunken bar fights."

- ØMQ (The Guide)

Mutex Locks

- Mutual Exclusion Lock

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

- Used to synchronize threads so that only one thread can make modifications to shared data at any given time
- Think transactions

Mutex Locks

- Using a lock

```
m = threading.Lock()
with m:                 # Acquires the lock
     statements
     statements
                        # Releases the lock
statements
```

- Key feature: Only one thread can execute inside the 'with' statement at once
- If lock is already in use, a thread waits

Use of Mutex Locks

- Commonly used to enclose "critical sections"

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

Thread-1                            Thread-2
--------                            --------
...                                 ...
with x_lock:                        with x_lock:
    x = x + 1                           x = x - 1
...                                 ...
```

Critical Section

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

Using a Mutex Lock

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

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

Thread-1                          Thread-2
--------                          --------
...                               ...
with x_lock:                      x = x - 1
    x = x + 1                     ...

...
```

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

Alternate Interface

- Alternate interface for locks

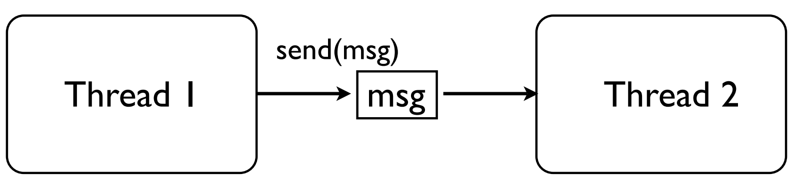
```
x = 0
x_lock = threading.Lock()

x_lock.acquire()
statements using x
...
x_lock.release()
```

- Very tricky to use correctly due to issues with exception handling
- Better to use the 'with' statement

Exercise 11.4

Communicating Threads

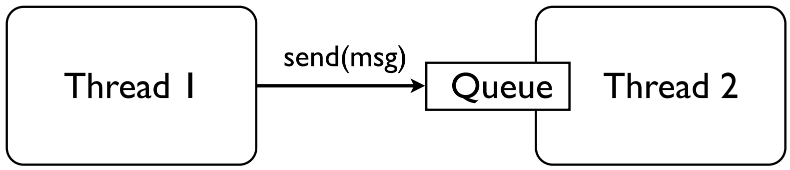
- Threaded programs are often easier to manage if they are designed around messaging



- Example: The Actor Model
- No shared state, only messages

Actors: Using Queues

- Can implement actors using queues



- Only shared state is the queue

Queue Library Module

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

```
from Queue import Queue

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

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

Sample Implementation

```
from threading import Thread
from Queue import Queue

class Actor(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.mailbox = Queue()

    # Send a message to this task (used by other threads)
    def send(self, msg):
        self.mailbox.put(msg)

    # Receive a message (only used by this thread)
    def recv(self):
        return self.mailbox.get()
```


Sample Implementation

- Consumer task

```
class Consumer(Actor):
    def run(self):
        while True:
            msg = self.recv()
            # Process msg
            ...
```

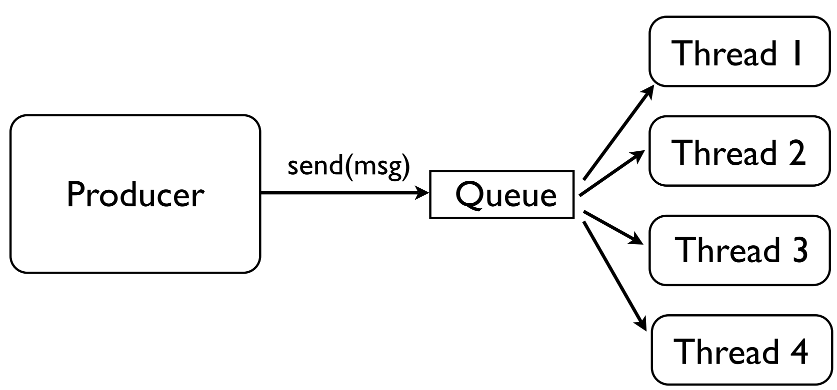
- Producer

```
target = Consumer()    # Start the consumer thread
target.start()

# Produce data and send to consumer
while running:
    msg = produce_data()
    target.send(msg)
```

Exercise 11.5

Thread Worker Pools

- Can have multiple threads per queue



- Queues are already safe, don't need locking

Sample Implementation

```
from threading import Thread
from Queue import Queue

class ThreadPool(object):
    def __init__(self, nworkers):
        self.mailbox = Queue()
        for n in range(nworkers):
            t = Thread(target=self.run)
            t.daemon = True
            t.start()

    def send(self, msg):
        self.mailbox.put(msg)

    def recv(self):
        return self.mailbox.get()

    def run(self):
        raise NotImplementedError()
```


Sample Implementation

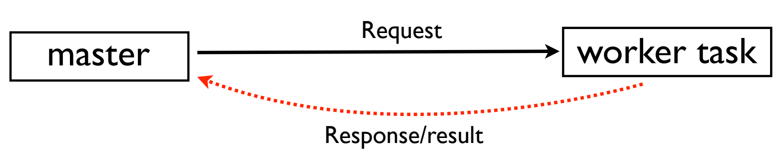
- Sample use of a pool

```
class SamplePool(ThreadPool):
    def run(self):
        while True:
            # Get a message
            msg = self.recv()
            # Process the message
            ...

# Initialize
s = SamplePool()

# Send a message to be processed
s.send(msg)
```

Exercise 11.6

Background Work (Futures)

- Sometimes concurrent tasks/threads are used to perform background work on behalf of other code



- Scenario : Master hands work over to a separate task and continues with other processing. Gets the result at some later time

The Problem

- This is nothing like a normal function call
- The work is finished at some undetermined time in the future
- The master doesn't know when the result will arrive--and it may want to do other things in the meantime
- Comment: This problem also comes up in other settings (distributed computing, etc.)

Futures

- Define an object that represents a future result

```
class FutureResult:
    def set(self,value):
        self._value = value

    def get(self):
        return self._value
```

- Provides set/get methods for managing result
- However, how do you use it?

Worker Task

```
class Worker(Actor):
    def request(self, msg):
        fresult = FutureResult()      # Create FutureResult
        self.send((fresult, msg))     # Send along with msg
        return fresult                # Return result object
    def run(self):
        while True:
            # Get a message
            fresult, msg = self.recv()
            # Work on msg
            ...
            # Set the result
            fresult.set(response)   # Set the response
```

- FutureResult object is created by worker and given back to the requestor
- Worker sets the result when work finished

Requesting Work

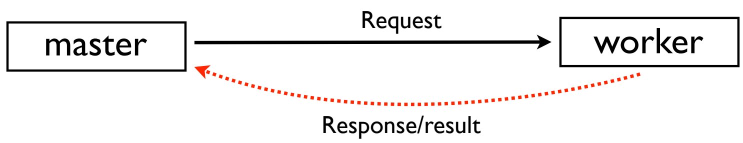
- Example of making a request:

```
fresult = worker.request(msg)  # Make request to worker
...
... do other things while worker works
...
# Get the result (at a later time)
r = fresult.get()
```

- Keep in mind: the worker is operating concurrently in the background
- When it finishes (at unknown time), it will store data in the returned result object

A Coordination Problem

- How does the master thread know when the result has been made available?



- Does it have to constantly poll?
- Does it just wait for awhile?
- This is a timing/synchronization issue

Event Waiting

- Using an event to signal "completion"

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

Worker Thread

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

Results with Waiting

- Using events in our result object

```
from threading import Event

class FutureResult:
    def __init__(self):
        self._evt = Event()

    def set(self, value):
        self._value = value
        self._evt.set()

    def get(self):
        self._evt.wait()
        return self._value
```

- Idea : get() will simply use the event to wait for the result to become available

Advanced Features

- FutureResult can be expanded to support a variety of other very useful features
  - Cancellation
  - Completion callbacks

Work Cancellation

- Allow requestor to cancel

```
class FutureResult:
    def __init__(self):
        self._cancel = False

    def cancel(self):
        self._cancel = True
```

- Example use in worker

```
def run(self):
    while True:
        fresult, msg = self.recv()
        if fresult._cancel:
            continue
        ...
```


Completion Callbacks

- A function that fires when result is ready

```
class FutureResult:
    def __init__(self):
        self._callback = None

    def set_callback(self, cb):
        self._callback = cb

    def set(self, result):
        self._value = result
        if self._callback:  # Invoke callback (if set)
           self._callback(result)
```

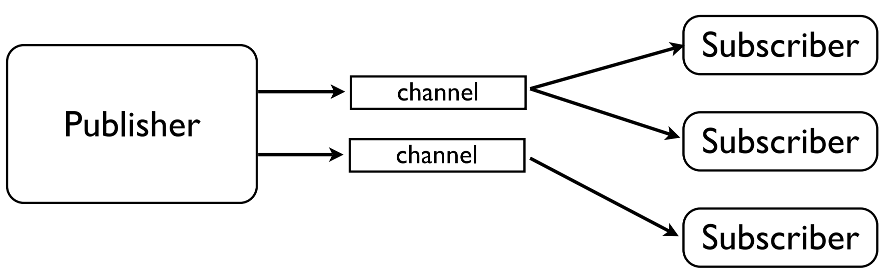
- Example use:

```
def when_done(result):
    print(result)

fresult = worker.request(msg)
fresult.set_callback(when_done)
```

Exercise 11.7

Publish/Subscribe

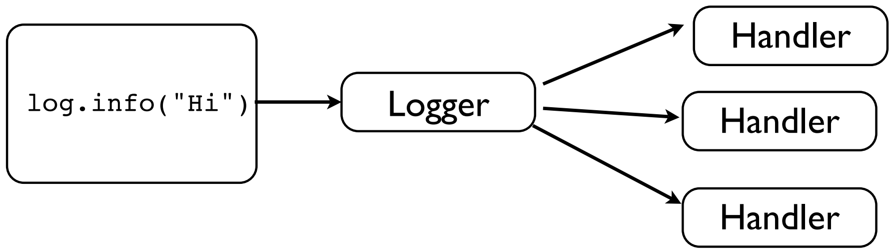
- An alternate thread communication pattern



- Think chat, RSS, XMPP, logging, etc...
- Publishers send message into a channel, subscribers receive the feed

Example: Logging

- You're already familiar with something that works exactly like this: the logging module



- Logger gets logging messages and publishes them to various subscribed handlers
- Can use it as a rough design model

Example Channel

- Simplistic implementation of a channel

```
class Channel(object):
    def __init__(self):
        self._subscribers = set()

    def subscribe(self, task):
        self._subscribers.add(task)

    def unsubscribe(self, task):
        self._subscribers.remove(task)

    def publish(self, msg):
        for task in self._subscribers:
            task.send(msg)
```

- Caution: It might need some added locking

Channel Management

- Management of Channels (by name)

```
from collections import defaultdict

class ChannelManager(object):
    def __init__(self):
        self._channels = defaultdict(Channel)

    def get_channel(self, name):
        return self._channels[name]
```

- Example:

```
mgr = ChannelManager()

ch = mgr.get_channel('spam')
ch.publish(msg)
```


Subscriber Disconnect

- Issue : Tasks come and go.
- Must be careful to manage subscriptions

```
class Task(Actor):
    ...
    def run(self):
        channel = mgr.get_channel('spam')
        channel.subscribe(self)
        try:
           ...
        finally:
           channel.unsubscribe(self)
```

- Example: unsubscribe on exception/return

Exercise 11.8

Processes

Problem: CPU-Bound Work

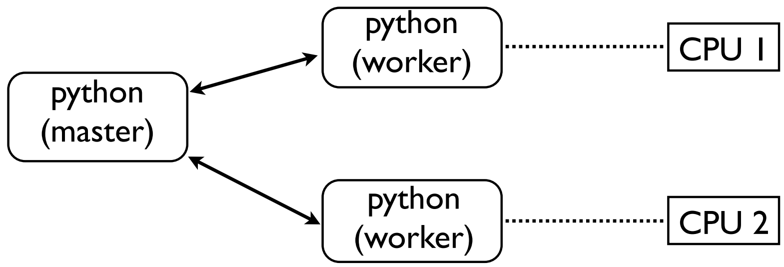
- As noted: Don't use threads for CPU-heavy work
- Can't utilize multiple CPU cores
- Encounter various problems due to GIL
- Need an alternate approach

Concept : Process

- Each running program is a "process"
  - Executes independently
  - Has own memory
  - Has own resources (files, sockets, etc.)
  - Can be scheduled on different CPUs by OS
- Each instance of the Python interpreter that runs on your system is a process

Cooperating Processes

- For CPU-intensive work, a common strategy is to use cooperating processes



- Multiple copies of the python interpreter that run on different CPUs and exchange data
- Not networking (all on the same machine)

multiprocessing Module

- A standard library module for carrying out work in separate processes
- Can be used to distribute work to other CPUs and to take advantage of multiple cores

Process Pools

- A process-based worker pool

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

- It executes functions in a subprocess
- It's very high-level (you don't need to worry a lot about internal details)

Process Pools

- Core pool operations

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

p.apply(func [, args [, kwargs]])
p.apply_async(func [, args [, kwargs [, callback]]])
```

- There are some others, but these are enough to get started
- Let's see some examples

Pool apply()

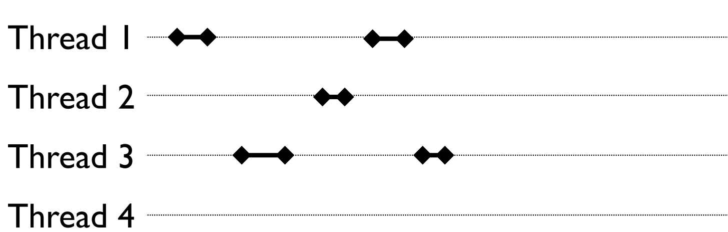
- Running a function in another process

```
def add(x,y):
    return x+y

if __name__ == '__main__':
   p = Pool(2)
   r = p.apply(add,(2,3))
   print(r)
```

- apply() runs a function in <u>one</u> of the worker processes and returns the result
- Note: It waits for the result to come back

apply() Illustrated

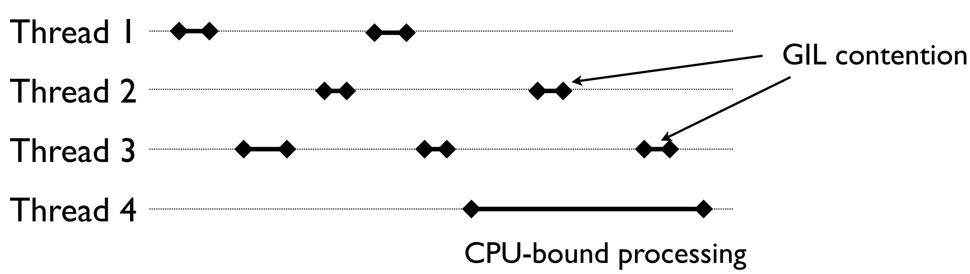
- Suppose you have a lot of threads



- If they're all I/O bound, life is good
- Mostly they sleep, hardly any GIL contention

apply() Illustrated

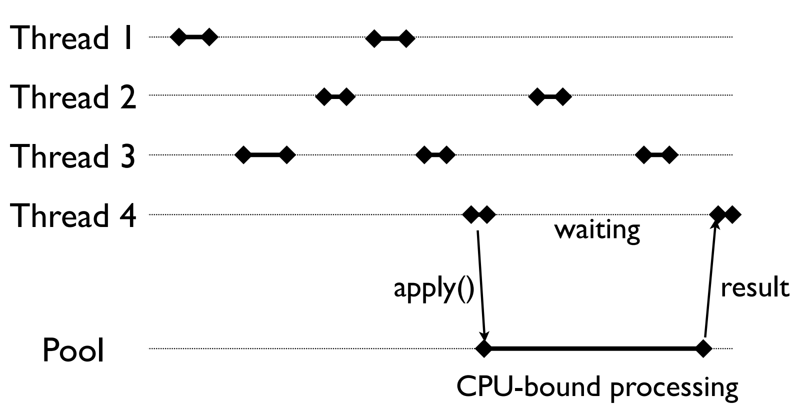
- Now suppose a thread wants to do work



- Thread holds GIL
- Causes contention with other threads

apply() Illustrated

- Delegating work to a pool



- Pool is separate process. No GIL

Pool apply_async()

- Asynchronous execution

```
def add(x,y):
    return x+y

if __name__ == '__main__':
   p = Pool(2)
   r = p.apply_async(add,(2,3))
   # Other work
   ...
   # Collect the result at a later time
   print(r.get())
```

- Here, you get a handle to an object for retrieving the result at some later time (like a future result)

Async Results

- For asynchronous execution, you get a special AsyncResult object
- Here is a mini reference on it

```
a.get([timeout])      # Get the result
a.ready()             # Result ready?
a.successful()        # Completed without errors
a.wait([timeout])     # Wait for result
```

- get() is the most useful method, but there are other operations for polling, querying error status, etc.

apply_async() Callbacks

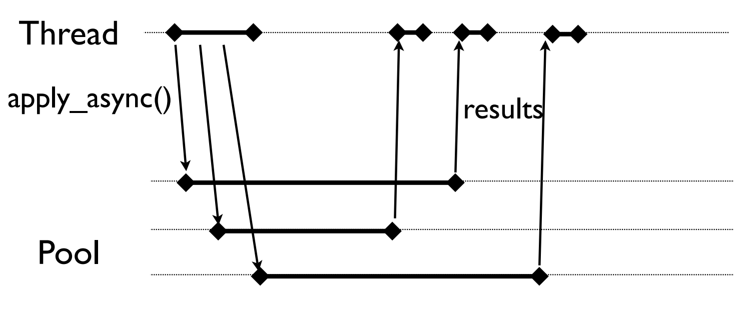
- Asynchronous execution with callback

```
def add(x,y):
    return x+y

def gotresult(result):
    print(result)

if __name__ == '__main__':
   p = Pool(2)
   r = p.apply_async(add,(2,3),callback=gotresult)
```

- Here, a callback function fires when the result is received

apply_async() Illustrated

- Used to initiate parallel computation



- Thread initiates multiple operations
- Collects results later

apply() vs apply_async()

- Usage depends on the context
- Use pool.apply() if you're using a pool to do work on behalf of a thread (and there are a lot of threads)
- Use pool.apply_async() if there's only one execution thread and it's trying to farm out work to multiple workers at once

Exercise 11.9

Event-Driven I/O

(Alternatives to Threads and Processes)

Alternatives

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

I/O Polling/Multiplexing

- Polling - An approach where you manually check for I/O activity and respond to it
- Typically associated with event-loops

```
while True:
    ...
    processing
    ...
    if poll_for_io():
        process I/O
        ...
    ...
    processing
```

- For example, a program might check for I/O activity every few milliseconds

select module

- Used to support polling
- Provides interfaces to the following
    - select() - Unix and Windows
    - poll() - Unix
    - epoll() - Linux
    - kqueue() - BSD
    - kevent() - BSD

select() function

- Used for I/O multiplexing/polling
- Usage : select(rset,wset,eset [,timeout])

```
readers = [...]   # sockets waiting to read
writers = [...]   # sockets waiting to write
exc     = [...]   # sockets to check for exceptions

rset,wset,eset = select(readers,writers,exc)
for r in rset:          # Handle readers
    handle_read(r)

for w in wset:          # Handle writers
    handle_write(w)

for e in eset:          # Handle exceptions
    handle_exception(e)
```


Event Driven I/O

- Can use select() to build event-driven systems
- The underlying idea is actually pretty simple
- You monitor a collection of I/O streams and create a stream of "events" that put pushed into callback or handler functions
- The basis of systems such as Twisted

Event Driven I/O

- Processing is put into event callback methods

```
class IOHandler:
    # Method to return a file descriptor
    def fileno(self):
        pass

    # Reading
    def readable(self):
        return False
    def handle_read(self):
        pass

    # Writing
    def writable(self):
        return False
    def handle_write(self):
        pass
```


Event Driven I/O

- Program driven by an I/O dispatching loop

```
class EventDispatcher:
    def __init__(self):
        self.handlers = set()
    def register(self,handler):
        self.handlers.add(handler)
    def unregister(self,handler):
        self.handlers.remove(handler)
    def run(self,timeout=None):
        while self.handlers:
            readers = [h for h in self.handlers
                          if h.readable()]
            writers = [h for h in self.handlers
                          if h.writable()]
            rset,wset,e = select(readers,writers,[],timeout)
            for r in rset:
                r.handle_read()
            for w in wset:
                w.handle_write()
```

Event Driven I/O

- Event handler registration

```
class EventDispatcher:
    def __init__(self):
        self.handlers = set()
    def register(self,handler):
        self.handlers.add(handler)
    def unregister(self,handler):
        self.handlers.remove(handler)
    def run(self,timeout=None):
        while self.handlers:
            readers = [h for h in self.handlers
                          if h.readable()]
            writers = [h for h in self.handlers
                          if h.writable()]
            rset,wset,e = select(readers,writers,[],timeout)
            for r in rset:
                r.handle_read()
            for w in wset:
                w.handle_write()
```

Registration and management of event handlers

Event Driven I/O

- Collecting handler read/write status

```
class EventDispatcher:
    def __init__(self):
        self.handlers = set()
    def register(self,handler):
        self.handlers.add(handler)
    def unregister(self,handler):
        self.handlers.remove(handler)
    def run(self,timeout=None):
        while self.handlers:
            readers = [h for h in self.handlers
                          if h.readable()]
            writers = [h for h in self.handlers
                          if h.writable()]
            rset,wset,e = select(readers,writers,[],timeout)
            for r in rset:
                r.handle_read()
            for w in wset:
                w.handle_write()
```

Collect all of the sockets that want to read or write

Event Driven I/O

- Polling for activity handlers that want I/O

```
class EventDispatcher:
    def __init__(self):
        self.handlers = set()
    def register(self,handler):
        self.handlers.add(handler)
    def unregister(self,handler):
        self.handlers.remove(handler)
    def run(self,timeout=None):
        while self.handlers:
            readers = [h for h in self.handlers
                          if h.readable()]
            writers = [h for h in self.handlers
                          if h.writable()]
poll -->    rset,wset,e = select(readers,writers,[],timeout)
            for r in rset:
                r.handle_read()
            for w in wset:
                w.handle_write()
```


Event Driven I/O

- Invocation of I/O callback methods

```
class EventDispatcher:
    def __init__(self):
        self.handlers = set()
    def register(self,handler):
        self.handlers.add(handler)
    def unregister(self,handler):
        self.handlers.remove(handler)
    def run(self,timeout=None):
        while self.handlers:
            readers = [h for h in self.handlers
                          if h.readable()]
            writers = [h for h in self.handlers
                          if h.writable()]
            rset,wset,e = select(readers,writers,[],timeout)
            for r in rset:
                r.handle_read()
            for w in wset:
                w.handle_write()
```

invoke handlers for sockets that can read/write

Event Driven "Tasks"

- In this framework, applications get implemented as IOHandler objects wrapped around a specific file or socket object

```
class SomeHandler(IOHandler):
    def __init__(self,sock):
        self.sock = sock
        ...
    def fileno(self):
        return self.sock.fileno()
```

- The internals don't really matter, but there must be a fileno() method to supply a file descriptor to select()/poll() operations

Event Driven "Tasks"

- Tasks must keep internal state that determines if they are interested in reading or writing

```
class SomeHandler(IOHandler):
    def __init__(self,sock):
        ...
        self.wants_to_read = True
        self.wants_to_write = False
        ...
    def readable(self):
        return self.wants_to_read
    def writable(self):
        return self.wants_to_write
    ...
```

- These methods tell the polling loop what events it should be looking for at any given time

Event Driven "Tasks"

- Tasks must define methods to actually handle read/write events

```
class SomeHandler(IOHandler):
        ...
    def handle_read(self):
        ...
        data = self.sock.recv(8192)
        ...
    def handle_write(self):
        ...
        self.sock.send(somedata)
        ...
```

- These methods only get called if the event loop has received some kind of matching event

Multitasking

- To run multiple tasks, you just register multiple handlers with the event loop and run its main event loop

```
dispatcher = EventDispatcher()
dispatcher.register(SomeHandler(s1))   # s1 is a socket
dispatcher.register(SomeHandler(s2))   # s2 is a socket
...
dispatcher.run()
```

- In theory, this set up allows your program to monitor multiple network connections

Exercise 11.10

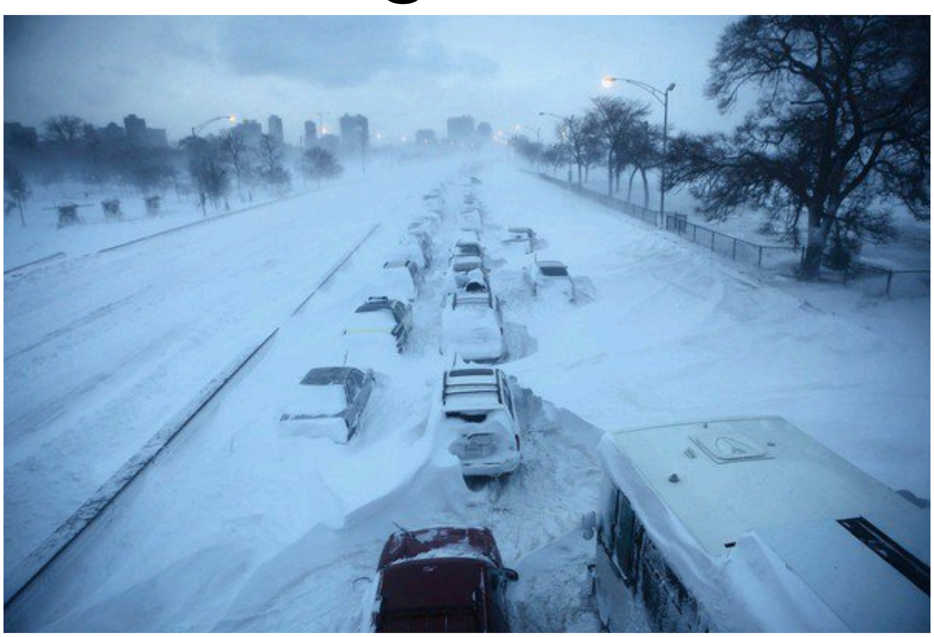
Long-Running Calculations

- If an event handler runs a long calculation, it blocks everything until it completes
- Example : Parsing a large XML message
- Remember, there are no threads or preemption
- This would manifest itself as program "stall". You've probably seen this with GUIs.

Blocking Operations

- Event-driven systems also have a really hard time dealing with blocking operations
  - Reading from the file system
  - Performing database queries
  - Calling out to third-party libraries
- If any of these operations take place in an event handler, the entire server/application stalls until it completes (no threads)

Blocking Illustrated



The Blocking Problem

- Consider this code...

```
class ApplicationHandler(object):
    ...
    def handle_request(self):
        ...
        results = db.execute("select * from table where ...")
        for r in results:
            ...
```

A database query (blocks?)

Processing of the results (CPU-intensive?)

- Everything waits until the callback method finishes its execution
- An issue if it happens to take a long time

Using Threads

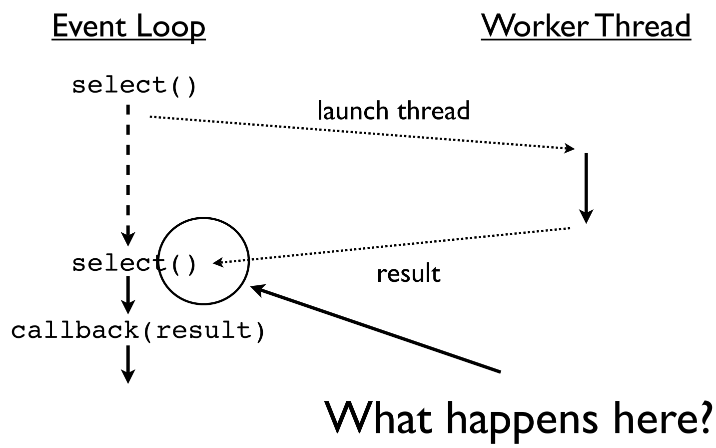
- Blocking operations might be handed to a separate thread/process to avoid stalling

```
class ApplicationHandler(object):
    ...
    def handle_request(self):
        ...
        launch_thread(do_query,
                      "select * from table where ...",
                      callback=self.got_result)
    def got_result(self, result):
        ...
```

- But there is the tricky of problem of coordinating what happens upon completion

Threads and Polling

- Commentary: Coordinating threads and I/O polling is a lot trickier than it looks



Interoperability Problems

- Event-driven programming tends to force an event-driven programming style across your entire application program
- This includes all external libraries and everything else used by your application
- However, most programming libraries are not written in an event-driven style
- For instance, the entire standard library

Exercise 11.11

Coroutines

- An alternative concurrency approach is possible using Python generator functions
- This is a little subtle, but I'll give you the gist

An Insight

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

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

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

Scheduling Example

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

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

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

- Each task is a generator function

Scheduling Example

- Now, run a task scheduler

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

# Run it
scheduler(tasks)
```

- This loop is what drives the application

Scheduling Example

- Output

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

- You'll see the different tasks cycling

Yielding For I/O

- If you are a littler clever, you can have yield integrate with "blocking" I/O requests
- The big idea : set up some kind of operation and then yield to have it carried out in the background by the generator scheduler

Talking to the Scheduler

- Tasks can send values back to the scheduler by yielding an "interesting" value
- Consider the following classes

```
class IOWait:
     def __init__(self,f):
         self.fileno = f.fileno()

class ReadWait(IOWait): pass
class WriteWait(IOWait): pass
```

- These classes represent the concept of "waiting" for a specific kind of I/O event on a given file object

An Example Task

- Now, consider this generator function

```
# Echo data received on s back to the sender
def echo_data(s):
    while True:
        yield ReadWait(s)     # Wait for data
        msg = s.recv(16384)   # Read data
        yield WriteWait(s)    # Wait for writing
        s.send(msg)
```

- This generator yields instances of the classes just defined back to the scheduler
- Now, let's go back to the scheduler code...

Signaling an I/O Request

- Here is the scheduler

```
def scheduler(tasks):
    while tasks:
        task = tasks.popleft()
        try:
            next(task) <-------------------- yield ReadWait(s)
            tasks.append(task)
        except StopIteration:
            pass

# Run it
scheduler(tasks)
```

Task

```
def echo_data(s):
    ...
    yield ReadWait(s)
    ...
```

- When the task yields, an instance of ReadWait or WriteWait is going to be returned by next

Signaling an I/O Request

- A modified scheduler

```
def scheduler(tasks):
    while tasks:
        task = tasks.popleft()
        try:
            r = next(task)
            if isinstance(r,ReadWait):
                handle_read_wait(r,task)
            elif isinstance(r,WriteWait):
                handle_write_wait(r,task)
            else:
                tasks.append(task)
        except StopIteration:
            pass

# Run it
scheduler(tasks)
```

Looking for different I/O wait requests and taking action

Implementing I/O Waits

- We haven't built the I/O yet, but it's easy
- To implement I/O waiting, you need two pieces
    - A holding area for tasks that are waiting for an I/O operation
    - An I/O poller that looks for I/O activity and removes tasks from the holding area when I/O is possible
- Let's look at an example

Building a Scheduler

- A scheduler class

```
class Scheduler:
    def __init__(self):
        self.numtasks       = 0
        self.ready          = deque()
        self.read_waiting  = {}
        self.write_waiting = {}
    def iopoll(self):
        rset,wset,eset = select(self.read_waiting,
                                self.write_waiting,[])
        for r in rset:
            self.ready.append(self.read_waiting.pop(r))
        for w in wset:
            self.ready.append(self.write_waiting.pop(w))
```


Building a Scheduler

- A scheduler class

Total number of tasks being managed

```
class Scheduler:
    def __init__(self):
        self.numtasks       = 0
        self.ready          = deque()
        self.read_waiting  = {}
        self.write_waiting = {}
    def iopoll(self):
        rset,wset,eset = select(self.read_waiting,
                                self.write_waiting,[])
        for r in rset:
            self.ready.append(self.read_waiting.pop(r))
        for w in wset:
            self.ready.append(self.write_waiting.pop(w))
```

Queue of tasks that can run

Building a Scheduler

- A scheduler class

```
class Scheduler:
    def __init__(self):
        self.numtasks      = 0
        self.ready         = deque()
        self.read_waiting  = {}
        self.write_waiting = {}
    def iopoll(self):
                 set = select(self.r
                                self.write_waiting,[])
                 et:
                 ady.append(self.read_waiting.pop(r))
                 et:
            self.ready.append(self.write_waiting.pop(w))
```

file descriptor

task

```
{
   3 : <generator>,
   7 : <generator>,
   6 : <generator>,
}
```

Dictionaries that serve as I/O holding areas

Building a Scheduler

- A scheduler class

```
class Scheduler:
    def __init__(self):
        self.numtasks
        self.ready
        self.read_waiting  = {}
        self.write_waiting = {}
    def iopoll(self):
        rset,wset,eset = select(self.read_waiting,
                                self.write_waiting,[])
        for r in rset:
            self.ready.append(self.read_waiting.pop(r))
        for w in wset:
            self.ready.append(self.write_waiting.pop(w))
```

An I/O polling function. This looks for any I/O activity on suspended tasks.

If there is I/O, move the task back to the ready queue

Building a Scheduler

- Add some scheduling methods (convenience)

```
class Scheduler:
    def __init__(self):
        self.numtasks      = 0
        self.ready         = deque()
        self.read_waiting  = {}
        self.write_waiting = {}
    ...
    def new(self,task):
        self.ready.append(task)
        self.numtasks += 1

    def readwait(self,fileno,task):
        self.read_waiting[fileno] = task

    def writewait(self,fileno,task):
        self.write_waiting[fileno] = task
```


Building a Scheduler

- Implement the main scheduler loop

```
class Scheduler:
    ...
    def run(self):
        while self.numtasks:
            try:
                task = self.ready.popleft()
                try:
                    r = next(task)
                    if isinstance(r,ReadWait):
                        self.readwait(r.fileno,task)
                    elif isinstance(r,WriteWait):
                        self.writewait(r.fileno,task)
                    else:
                        self.ready.append(task)
                except StopIteration:
                    self.numtasks -= 1
            except IndexError:
                self.iopoll()
```

Building a Scheduler

- Implement the main scheduler loop

```
class Scheduler:
    ...
    def run(self):
        while self.numtasks:
            try:
                task = self.ready.popleft()
                try:
                    r = next(task)
                    if isinstance(r,ReadWait):
                        self.readwait(r.fileno,task)
                    elif isinstance(r,WriteWait):
                        self.writewait(r.fileno,task)
                    else:
                        self.ready.append(task)
                except StopIteration:
                    self.numtasks -= 1
            except IndexError:
                self.iopoll()
```

Run a task until it yields, check the return value

Building a Scheduler

- Implement the main scheduler loop

```
class Scheduler:
    ...
    def run(self):
        while self.numtasks:
            try:
                task = self.ready.popleft()
                try:
                    r = next(task)
                    if isinstance(r,ReadWait):
                        self.readwait(r.fileno,task)
                    elif isinstance(r,WriteWait):
                        self.writewait(r.fileno,task)
                    else:
                        self.ready.append(task)
                except StopIteration:
                    self.numtasks -= 1
            except IndexError:
                self.iopoll()
```

Poll for I/O (only runs if no other work to do)

Example : Time Server

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def timeserver(addr):
    s = socket(AF_INET, SOCK_DGRAM)
    s.bind(addr)
    while True:
         yield ReadWait(s)
         msg,addr = s.recvfrom(8192)
         yield WriteWait(s)
         s.sendto((time.ctime()+"\n").encode('ascii'),
                  addr)

sched = Scheduler()
sched.new(timeserver(('',15000)))   # Create three server
sched.new(timeserver(('',16000)))   # instances and add
sched.new(timeserver(('',17000)))   # to the scheduler
sched.run()
```


Example : Echo Server

```
class EchoServer:
    def __init__(self,addr,sched):
        self.sched = sched
        sched.new(self.server_loop(addr))
    def server_loop(self,addr):
        s = socket(AF_INET,SOCK_STREAM)
        s.bind(addr)
        s.listen(5)
        while True:
            yield ReadWait(s)
            c,a = s.accept()
            print("Got connection from", a)
            self.sched.new(self.client_handler(c))
    def client_handler(self,client):
        while True:
            yield ReadWait(client)
            msg = client.recv(8192)
            if not msg: break
            yield WriteWait(client)
            client.send(msg)
        client.close()
        print("Client closed")
```

Example : Echo Server

- Running the echo server

```
sched = Scheduler()
echo  = EchoServer(('',15000),sched)
sched.run()
```

- Test it out with telnet
- Will find that it works fine with multiple clients

Commentary

- One appeal of coroutines is that they have normal-looking control flow (like threads)
- Built using event-driven I/O under the covers
- Still has same problems with blocking, computation, interoperability, etc.

Coroutine Info

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

Exercise 11.12

Message Passing

Concept: Message Passing

- Multiple independent copies of the Python interpreter (or programs in other languages)
- Running in separate processes
- Possibly on different machines
- Sending/receiving messages

Commentary

- Message passing is a well-established technique for concurrent programming
- It has been successfully scaled up to systems involving tens of thousands of processors (e.g., supercomputers, Linux clusters, etc.)
- The foundation of distributed computing
- We've already covered some basic ideas

Message Passing

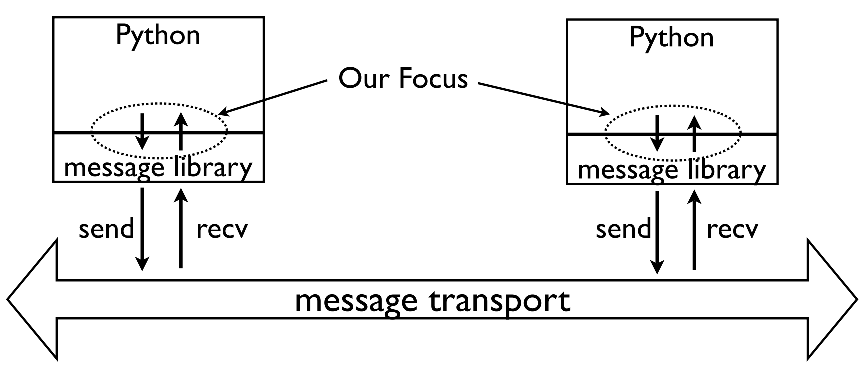
- On the surface, it's really simple
- Processes only send and receive messages
- There are really only two main issues
  - What is a message?
  - How is it transported?

An Issue

- There is no universally accepted programming interface or implementation of messaging
- There are dozens of different packages that offer different features and options
- Covering every possible angle of message passing interfaces is simply impossible here
- And a reference manual would be rather dull

Section Focus

- Our focus is going to be on general programming idioms related to messaging
- This mostly concentrates on the boundary between Python and the messaging layer



What is a Message?

- Usually just a collection of bytes (a buffer)
- A "serialized" representation of some data

Message Encoding

- Messages have to be formatted or encoded in some manner that enables transport
- To send, a message is encoded
- To receive, a message is decoded

Encoding Example

- A minimal encoding (size prefixed bytes)

size	Message (bytes)

- Message is just bytes with a size header
- No interpretation of the bytes (opaque)
- So, payload could be anything at all (any encoding, any programming language, etc.)

Message Transport

- Messages have to be transmitted (somehow) between running processes
- Inter-Process Communication (IPC)
- Some low-level communication primitives
  - Pipes
  - FIFOs
  - Sockets (Network Programming)

Exercise 11.13

Object Serialization

- How to serialize Python objects?
  - Lists, dictionaries, sets, instances, etc.
- An issue here is that Python is extremely flexible with respect to data types
- Containers can also hold mixed data
- There is no easy format for describing Python objects (e.g., a simple array or by fixed binary data structures)

pickle Module

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

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

- Unserializing an object from a file

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

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

Pickling to Strings

- Pickle also creates byte strings

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

- This can be used if you need to embed a Python object into some other messaging protocol or data encoding

pickle Compatibility

- What objects are compatible?
- Nearly any object that consists of data
  - None, numbers, strings
  - Tuples, lists, dicts, sets, etc.
  - Instances of objects
  - Functions and classes (tricky)
- The underlying message encoding is "self-describing" (which hides a lot of details)

pickle Incompatibility

- Objects <u>not</u> compatible with pickle
- Anything involving system or runtime state
  - Open files, sockets, etc.
  - Threads
  - Running generator functions
  - Stack frames
  - Closures

Pickle Security

- It's not secure at all
- Never use pickle with untrusted clients (malformed pickles can be used to execute arbitrary system commands)
- Bottom line : Never receive pickled data on an untrusted or unauthenticated connection

Miscellaneous Comments

- Pickle is really only useful for Python
- Would not use if you need to communicate to other programming languages
- However, you can do some pretty amazing things with it if Python is your environment

Exercise 11.14

High-Level Messaging

- There are many messaging frameworks
  - AMQP
  - ØMQ
  - RabbitMQ
  - Celery
- Common theme : Putting a higher-level interface on top of sockets, pipes, etc.

Messaging Features

- Support for common messaging patterns
  - Push/Pull (Queues)
  - Request/Reply
  - Publish/subscribe
- Reliability/scalability features
  - Load balancing
  - Durable connections

Connection Objects

- multiprocessing provides Listener and Client objects that transmit pickled data
- A Listener (receives connections)

```
from multiprocessing.connection import Listener
serv = Listener(('',15000),authkey='12345')

# Wait for a connection
client = serv.accept()

# Now, wait for messages to arrive
while True:
     msg = client.recv()
     # process the message
```

Connection Objects

- Example Client

```
from multiprocessing.connection import Client
conn = Client(('localhost',15000),authkey='12345')

conn.send(msg)
```

- You will notice a similarity to sockets
- Except that it's much higher level and it sends pickled objects

Connection Objects

- Some important features of connections
  - Authentication (uses HMAC, a technique based on message digests such as SHA)
  - Instead of bytes, you send Python objects
  - Data is encoded using pickle
- Is extremely useful if you're just going to hook two Python interpreters together

Example : ØMQ

- ZeroMQ (http://www.zeromq.org/)
- In a nutshell : Message-based sockets
- In their own words...

"A ØMQ socket is what you get when you take a normal TCP socket, inject it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombard it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex."

- I would cautiously agree

Example : ØMQ

- Here's an example of an echo server:

```
# echoserver.py

import zmq
context = zmq.Context()
sock = context.socket(zmq.REP)
sock.bind("tcp://*:6000")
while True:
    message = sock.recv()       # Get a message
    sock.send(b"Hi:"+message)   # Send a reply
```

- That's it
- And this server can already handle requests from 100s (or 1000s) or connected clients

Example : ØMQ

- Here's an example of a echo client

```
# echoclient.py

import zmq
context = zmq.Context()
sock = context.socket(zmq.REQ)
sock.connect("tcp://localhost:6000")

sock.send(b"Spam")      # Send a request
resp = sock.recv()      # Get response
print(resp)
```

- That's also pretty simple (it just works)

Example : ØMQ

- Some cool features
  - Can start server or client in any order
  - Clients can connect to multiple servers (load balancing, redundancy, etc.)
  - Variety of socket types (Reply, Request, Push, Pull, Publish, Subscribe, etc.)

Example : ØMQ

- Example client connected to multiple servers

```
import zmq
context = zmq.Context()
sock = context.socket(zmq.REQ)
sock.connect("tcp://host1.com:6000")
sock.connect("tcp://host2.com:7000")
sock.connect("tcp://host3.com:7000")

sock.send(b"Spam")      # Send a request
resp = sock.recv()      # Get response
```

- Request gets sent to one of the servers
- Think about scaling, redundancy, etc.

Exercise 11.15

Distributed Computing

Major Topics

- Actors
- Distributed data (key-value stores)
- Remote Procedure Call (RPC)
- Distributed Objects
- Interoperability, foreign systems, etc.

Tasks Revisited

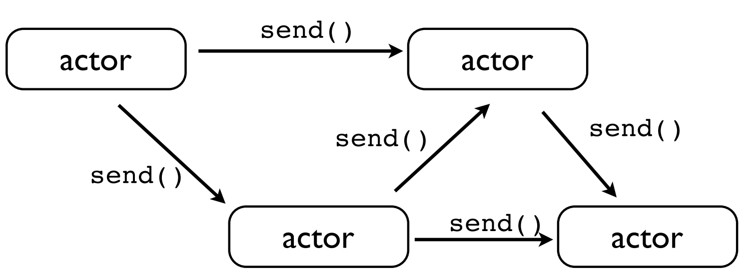
- In the thread section, we defined tasks that received and acted upon messages sent to them

```
class MyTask(Actor):
    def run(self):
        while True:
             msg = self.recv()   # Get a message
             ...
             # Do something with it
             ...

m = MyTask()
m.start()
m.send(msg)        # Send a task a message
```

- This model extends naturally

Actors

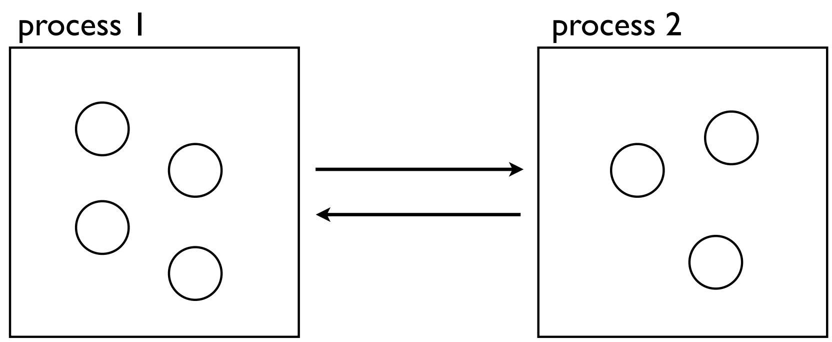
- Many actors might work together



- But, maybe they're on different machines

Distributed Actors

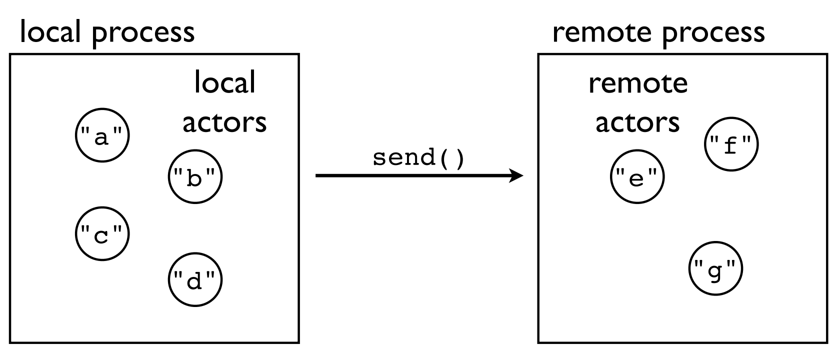
- To distribute actors, you need to have some kind of IPC/networking component



- You can use the earlier messaging techniques
- multiprocessing, ØMQ, etc.

Distributed Send

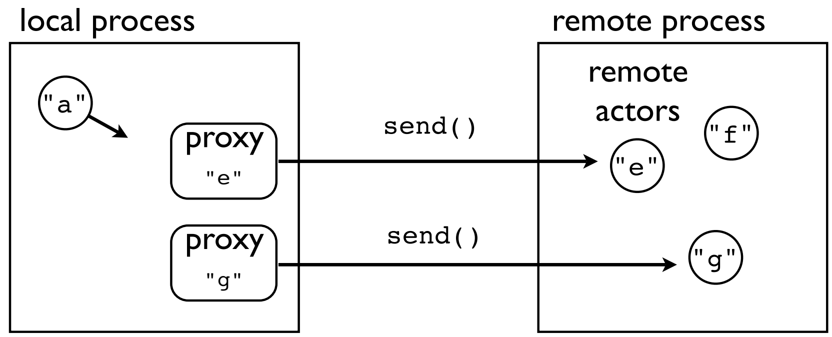
- To implement distributed actors, you need to have some kind of naming/addressing scheme



- Similar to network addresses (of some kind)

Proxies

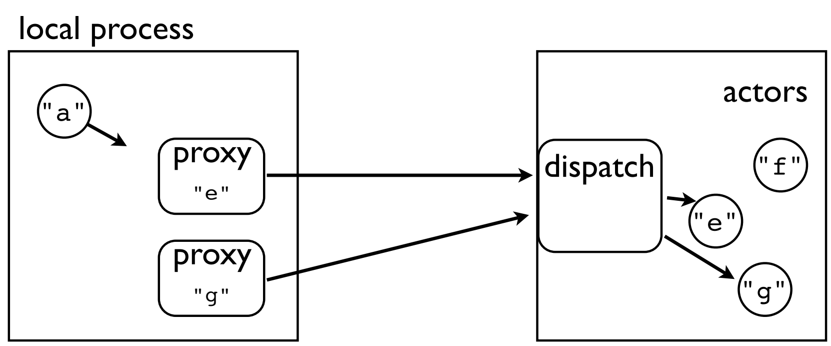
- Messages are directed to a remote system through the use of a proxy actors



- A proxy receives messages for a remote actor and forwards them to a remote process

Message Dispatching

- A dispatcher is needed to receive messages

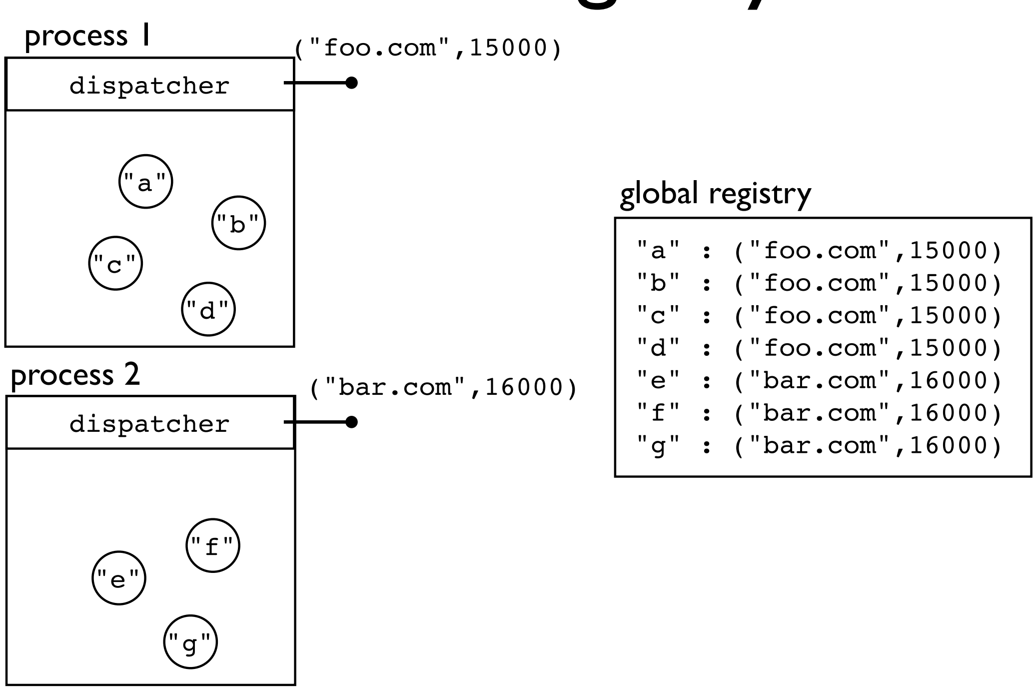


- The dispatcher is a server that accepts connections, receives messages, and forwards them to the local actors

Actor Addressing

- Trying to keep track of actor locations is hard
- Especially if you force programmers to do it manually by hard-coding everything
- Better solution: Create a global registry for mapping actor names to dispatchers (hosts)
- Think DNS

Name Registry

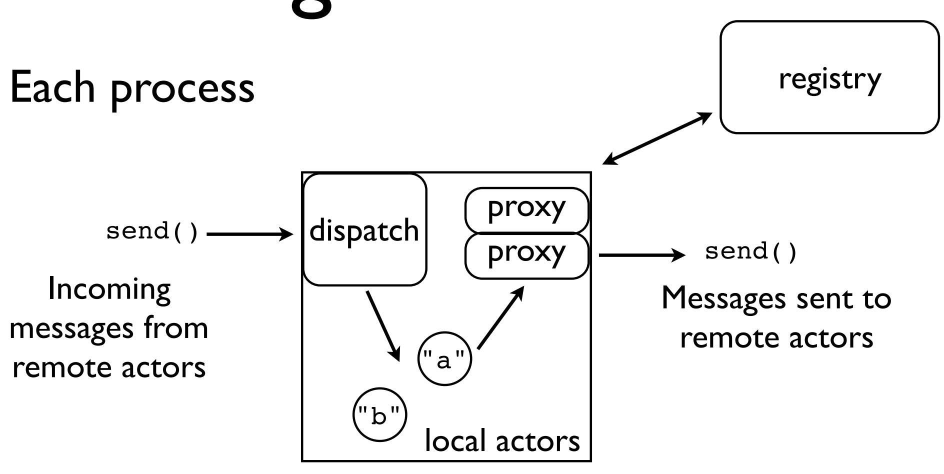


Building a Registry

- Registry is essentially just a centralized table
- A sensible option: Use a key-value store
- It's exactly what it sounds like--a dictionary
- You can easily build your own
- Or use an existing one : memcached, redis, CouchDB, MongoDB, Cassandra,

Big Picture

- Each process



- Many parts working together

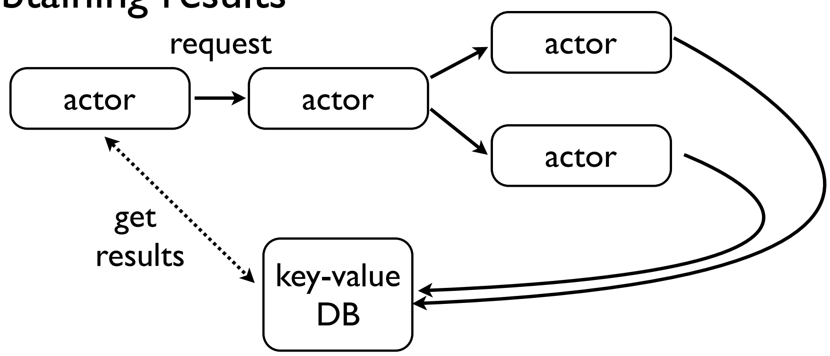
Exercise 11.16

Key-value Stores

- Key-value stores can be used for a variety of other purposes (more than just a registry)
- Maintain system-wide configuration data
- Store results from distributed calculation
- Provide work queues
- Etc.

Example: Results

- Obtaining results

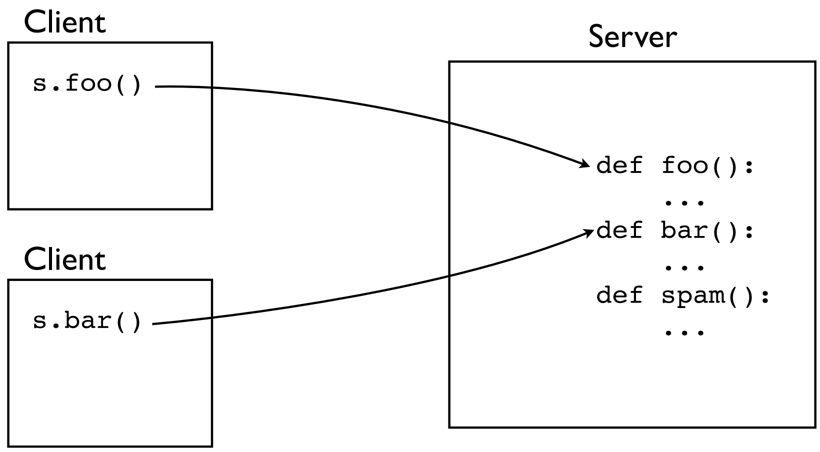


- Example : Actor sends out some message that disappears into a "cloud" of other actors
- Picks up results by watching the DB.

Exercise 11.17

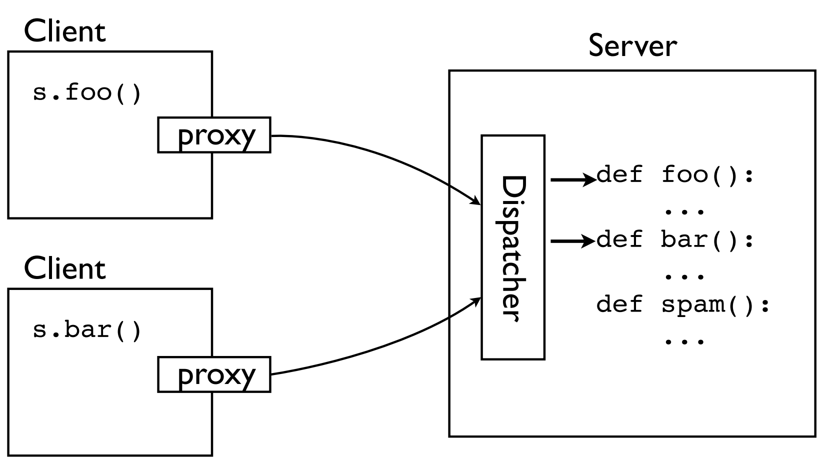
Remote Procedure Call

- Remote invocation of procedures implemented on a server process



Remote Procedure Call

- RPC implementation uses a similar technique as used for distributed actors (dispatcher, proxies)



RPC Details

- RPC messages simply identify a method name and include method arguments

```
# funcname  = name of function
# args      = tuple of positional args
# kwargs    = dict of keyword args

msg = (funcname, args, kwargs)   # Make an RPC message
send(target, msg)                # Send it somewhere
```

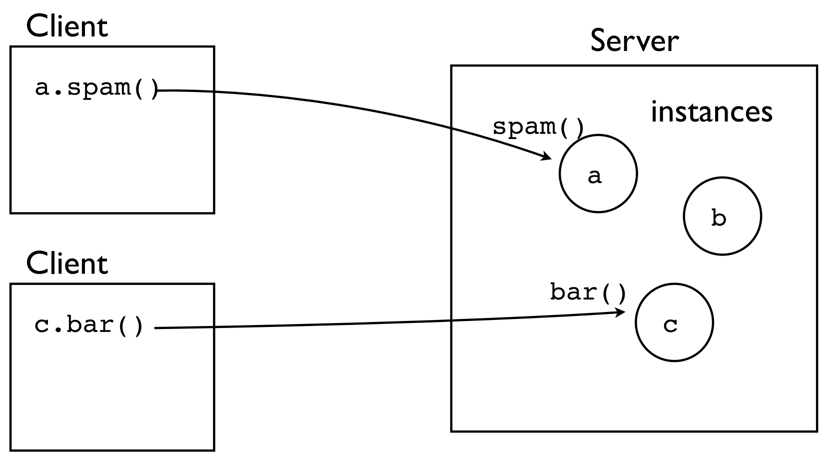
- In the server, just dispatch

```
# Get a message
funcname, args, kwargs = receive()

# Look up the function and dispatch
func = _functions[funcname]
result = func(*args, **kwargs)
send(sender, result)
```


Exercise 11.18

Distributed Objects

- Objects live on a server (where they stay put)
- Clients remotely invoke instance methods



Distributed Objects

- In principle, supporting distributed objects is similar to remote procedure call (RPC)
- But there is one really big difference
- Distributed objects involves the manipulation of state (instances) stored on the server
- With state comes extra complication (memory management, locking, persistence, etc.)

Server Instances

- Objects are defined by a normal class

```
class Foo(object):
    def bar(self):
        ...
    def spam(self):
        ...
```

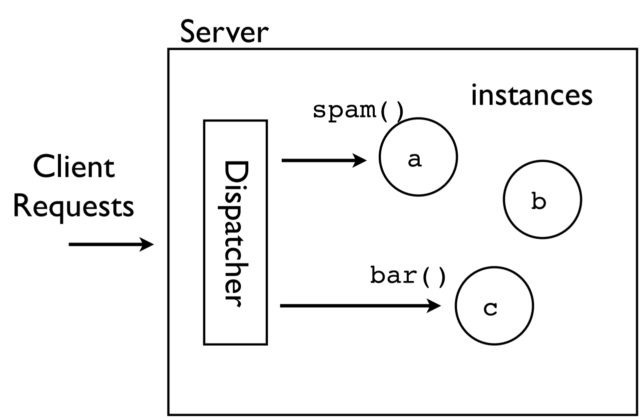
- On the server, various instances are created

```
a = Foo()
b = Foo()
c = Foo()
```

- These are <u>normal</u> Python objects

Server Dispatching

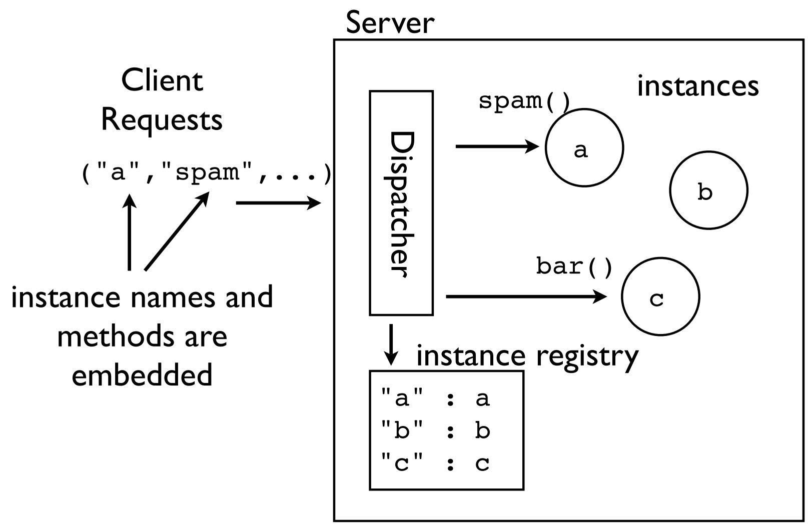
- For remote access, a dispatcher is needed



- Exactly the same idea as with actors, RPC

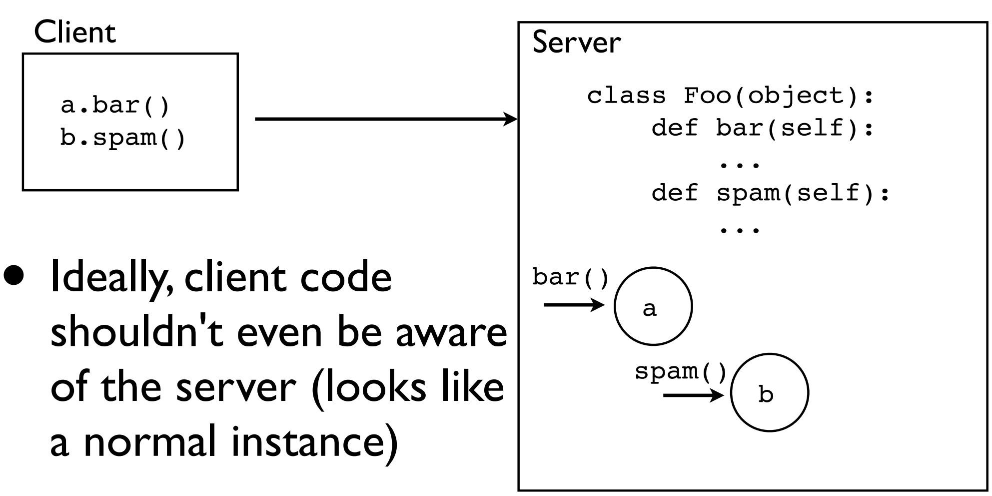
Request Messages

- Incoming requests must identify both the instance and a method to execute



Client Interfaces

- Clients generally want to use the same programming interface as the class



- Ideally, client code shouldn't even be aware of the server (looks like a normal instance)

Client Proxies

- To emulate the API, proxy classes are needed

```
class FooProxy(object):
    def __init__(self,name,serveraddr):
        self.name = name
        self.conn = connect_to(serveraddr)
    def bar(self,*args):
        # send "bar" request to server
        # return result
        ...
    def spam(self,*args):
        # send "spam" request to server
        # return result
        ...
```

- The proxy has the same programming API as the original object (same methods)
- Proxy methods issue RPC requests to server

Problem : Synchronization

- In a distributed environment, many clients may be connected simultaneously
- There might be server threads
- May be concurrent access to the objects
- Thus, you may need locking

Problem : Instance Creation

- What happens if new instances get created on the server in response to requests?
  - How are they referenced by clients?
  - Who is responsible for managing them?
  - How long do they live?
  - Do they persist? (In a database)
- Countless things can go wrong...

Problem : Reliability

- What happens if the server crashes? (objects disappear and clients crash?)
- Can software on the server be fixed/updated?
- Can class definitions be modified?
- API changes?

Comments

- Using distributed objects is often a bad idea
- Massive amounts of added complexity, library dependencies, programming sophistication
- Example: I once had a consulting gig where I was supposed to analyze a one million line distributed C++ application. 95% of the code was related to distributed objects (not fun)

Some Resources

- pyro (Python Remote Objects). A python-centric distributed object framework. Assumes that you're only working in Python. Simplifies many tasks that are harder in other systems.
- CORBA. Distributed object framework designed for multiple languages. Look at: OmniORB, fnorb. Note: as far as I can tell CORBA is not hugely popular in the Python world (excessive complexity?)

Exercise 11.19

Interoperability

- You may want parts of your distributed system to interoperate with other components
- Possibly written in other languages
- Possibly located elsewhere
- Possibly implemented by someone else

Interoperability Tips

- To connect to foreign systems, you really want to focus on well-documented standards
- Use common data encodings (XML, JSON, etc.)
- Use common protocols (HTTP, XML-RPC, etc.)

XML-RPC

- Remote Procedure Call
- Uses HTTP as a transport protocol
- Parameters/Results encoded in XML
- Supported by languages other than Python

Simple XML-RPC

- How to create a stand-alone server

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x,y):
    return x+y

s = SimpleXMLRPCServer(("",8080))
s.register_function(add)
s.serve_forever()
```

- How to test it (xmlrpclib)

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy("http://localhost:8080")
>>> s.add(3,5)
8
>>> s.add("Hello","World")
"HelloWorld"
>>>
```


Simple XML-RPC

- Adding multiple functions

```
from xmlrpc.server import SimpleXMLRPCServer

s = SimpleXMLRPCServer(("",8080))
s.register_function(add)
s.register_function(foo)
s.register_function(bar)
s.serve_forever()
```

- It's fairly straightforward...

XML-RPC Undercover

- Here's what gets sent for a request:

```
s = ServerProxy("...")
s.add(3,5)
```

```
POST /RPC2 HTTP/1.0
Content-Type: text/xml
Content-Length: 187

<?xml version='1.0'?>
<methodCall>
<methodName>add</methodName>
<params>
<param>
<value><int>3</int></value>
</param>
<param>
<value><int>5</int></value>
</param>
</params>
</methodCall>
```


XML-RPC Commentary

- XML-RPC is extremely easy to use
- Almost too easy to be honest
- I have encountered a lot of major projects that are using XML-RPC for distributed control

Other RPC Libraries

- Some RPC libraries of interest.
- Thrift. A cross-language RPC framework developed by Facebook and released as open-source.
- Protocol Buffers. A cross-language RPC framework developed by Google. Also open-source.
- Both use much more efficient data serialization than XML-RPC (and have other features)

RESTful Services

- REST (Representation State Transfer)
- It's a data-centric software architecture where servers host data (resources) and implement methods for remotely interacting with the data
- Strongly tied to HTTP, but think about structured data instead of HTML pages.

REST Resources

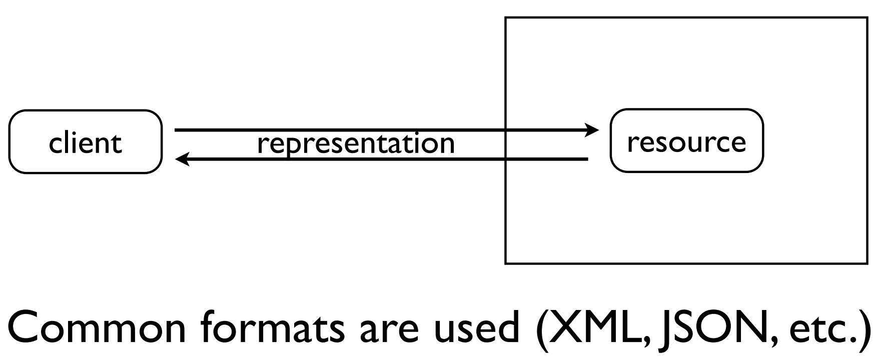
- Core component of REST is a "resource"
- A resource usually represents data
- Resources have an associated identifier (URI)

```
http://somehost.com/someresource
```

- The URI alone contains everything needed to locate and identify the resource (protocol, hostname, path, etc.)

Resource Representation

- Data associated with a resource is typically represented using a standard data encoding



- Common formats are used (XML, JSON, etc.)
- May be multiple representations

REST Actions

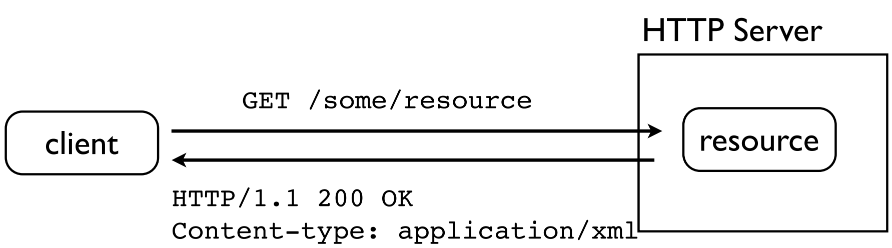
- Clients interact with servers and resources using a preset vocabulary of actions (verbs)

```
GET resource
PUT resource
DELETE resource
POST resource
HEAD resource
```

- These are usually just HTTP methods
- PUT and DELETE are related to creating/updating a resource (not common with browsers)

REST Examples

- Retrieving a resource (GET)




```
HTTP/1.1 200 OK
Content-type: application/xml
...

<?xml ...?>
<root>
    ...
</root>
```

Stateless Implementation

- REST services are stateless
  - Server does not record client state
  - GET, POST, etc. are the only operations
  - May occur in any order and at any time
- It's a critical feature of the architecture
  - May have multiple servers (heavy load)
  - Fault handling (if a server crashes, etc.)

Reuse of HTTP

- REST web services build upon HTTP
  - Authentication/security
  - Caching
  - Proxies
- Integrates well with existing software
  - HTTP servers
  - Middleware libraries
  - Almost anything that speaks HTTP

Implementing REST

- You typically build a REST service using the same techniques for other web programming
  - CGI scripting
  - WSGI
  - Web frameworks (Django, Zope, etc.)
  - Stand-alone HTTP server
- My preference: WSGI

Example with WSGI

- WSGI is a standard programming interface used by a lot of Python web frameworks and libraries
- It's relatively low-level
- Loosely based on CGI scripting
- Advantage: By using WSGI, code will be able to integrate with other packages

Example with WSGI

- WSGI "Hello World" Application

```
def helloapp(environ, start_response):
    resource = environ['PATH_INFO']
    method   = environ['REQUEST_METHOD']
    start_response("200 OK", [
        ('Content-type','text/plain')
    ])
    response = [
       "Request method: %s\r\n" % method,
       "Resource      : %s\r\n" % resource
    ]
    return response
```


Example with WSGI

- HTTP Request information

```
def helloapp(environ, start_response):
    resource = environ['PATH_INFO']
    method   = environ['REQUEST_METHOD']
    start_response("200 OK", [
        ('Content-type', 'text/pl
    ]
    response = [
       "Request method: %s\r\n" %
       "Resource      : %s\r\n" %
    ]
    return response
```

Information about the incoming request is placed into a Python dictionary (environ)

Example with WSGI

- Starting an HTTP Response

```
def helloapp(environ, start_response):
    resource = environ['PATH_INFO']
    method   = environ['REQUEST_METHOD']
    start_response("200 OK", [
        ('Content-type', 'text/plain')
    ]
    response = [
       "Request method:
       "Resource       :
    ]
    return response
```

start_response is a function that you call to initiate an HTTP response. Takes a status code and a list of HTTP headers.

Example with WSGI

- Response body

```
def helloapp(environ, st
    resource = environ['
    method   = environ['
    start_response("200 (
        ('Content-type',
    ]
    response = [
       "Request method: %s\r\n" % method,
       "Resource       : %s\r\n" % resource
    ]
    return response
```

Responses are returned as a sequence of byte strings (e.g., a list, generator, or other iterable)

- Normally, you would return actual content here (e.g., HTML, XML, JSON, etc.)

Running an WSGI App

- WSGI applications usually run inside some other framework (e.g., a web server)
- You can run standalone for testing

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    serv = make_server('',8080, helloapp)
    serv.serve_forever()
```

- Connect with a browser, try it out

```
http://localhost:8080/foo/bar
```


REST Links

- Too many packages to list (all Python 2)
  - restlib
  - restkit
  - restish
  - Many others on PyPI
- Note: Don't confused with packages related to reStructured Text (reST)

Exercise 11.20