Eventlet

Asynchronous I/O with a Synchronous Interface

Donovan Preston

Network Servers

Processes, Threads, or Non-Blocking I/O?

The C10K Problem

- http://www.kegel.com/c10k.html
- “It's time for web servers to handle ten thousand clients simultaneously, don't you think?”

Processes, Threads, Non-Blocking I/O

- Processes
  - Too heavyweight
- Threads
  - Non-determinism sucks
- Non-Blocking I/O
  - Requires callback-style programming
    - Rules out many existing libraries

Solution: Coroutines

- **Callbacks**: Register a callback function and then **Return** to the main loop
- **Coroutines**: Register a callback coroutine and then **Call** the main loop
  - The call stack is preserved
  - Does not require cooperation

    from the caller

Solution: Greenlet

- Greenlet Provides Hard Switching from Stackless in a Regular Python Module
- Stack Slicing is used to implement coroutine switching
    - Portions of the C Stack are copied to the Heap and vice versa

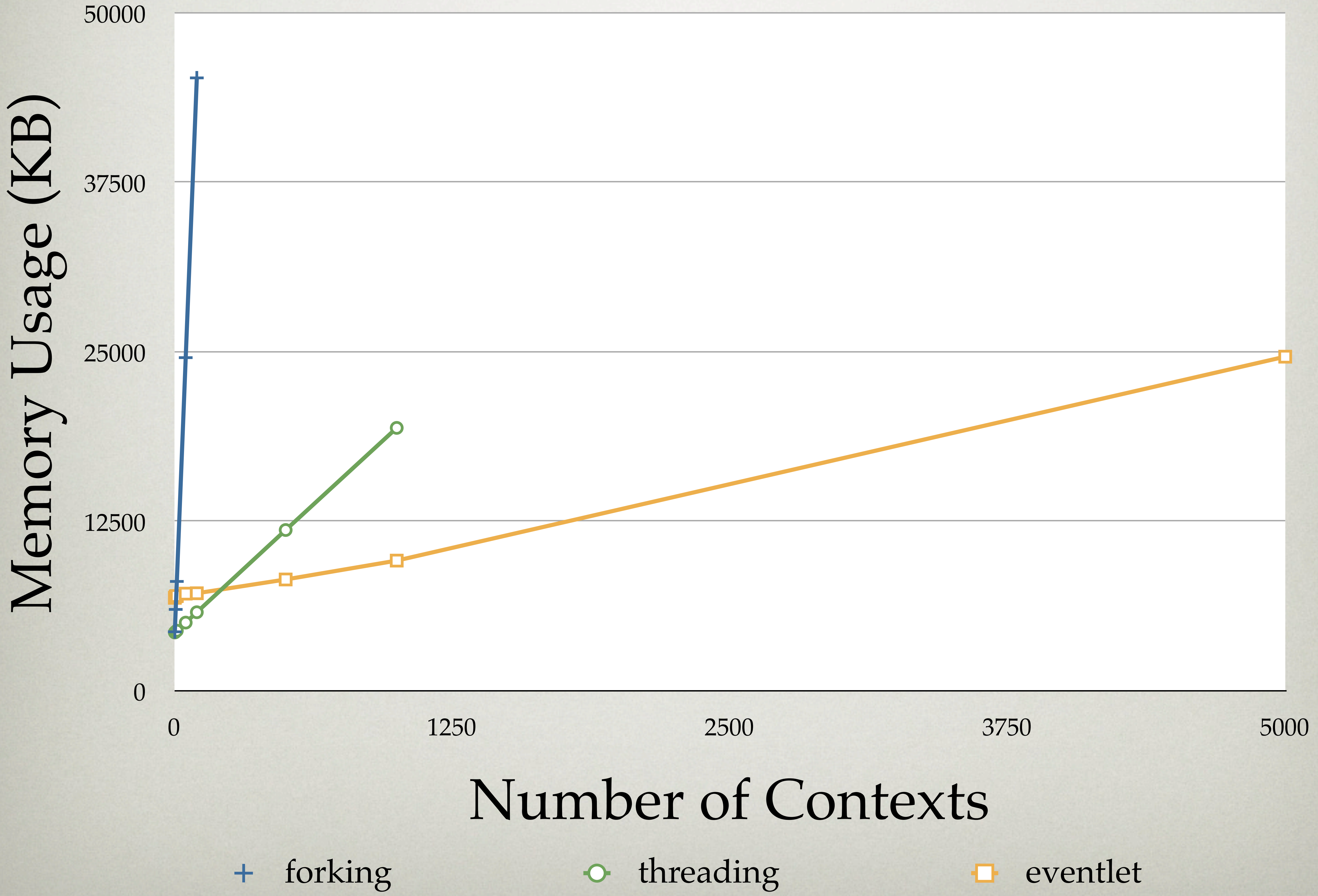
Eventlet

Green Threads on top of Greenlet

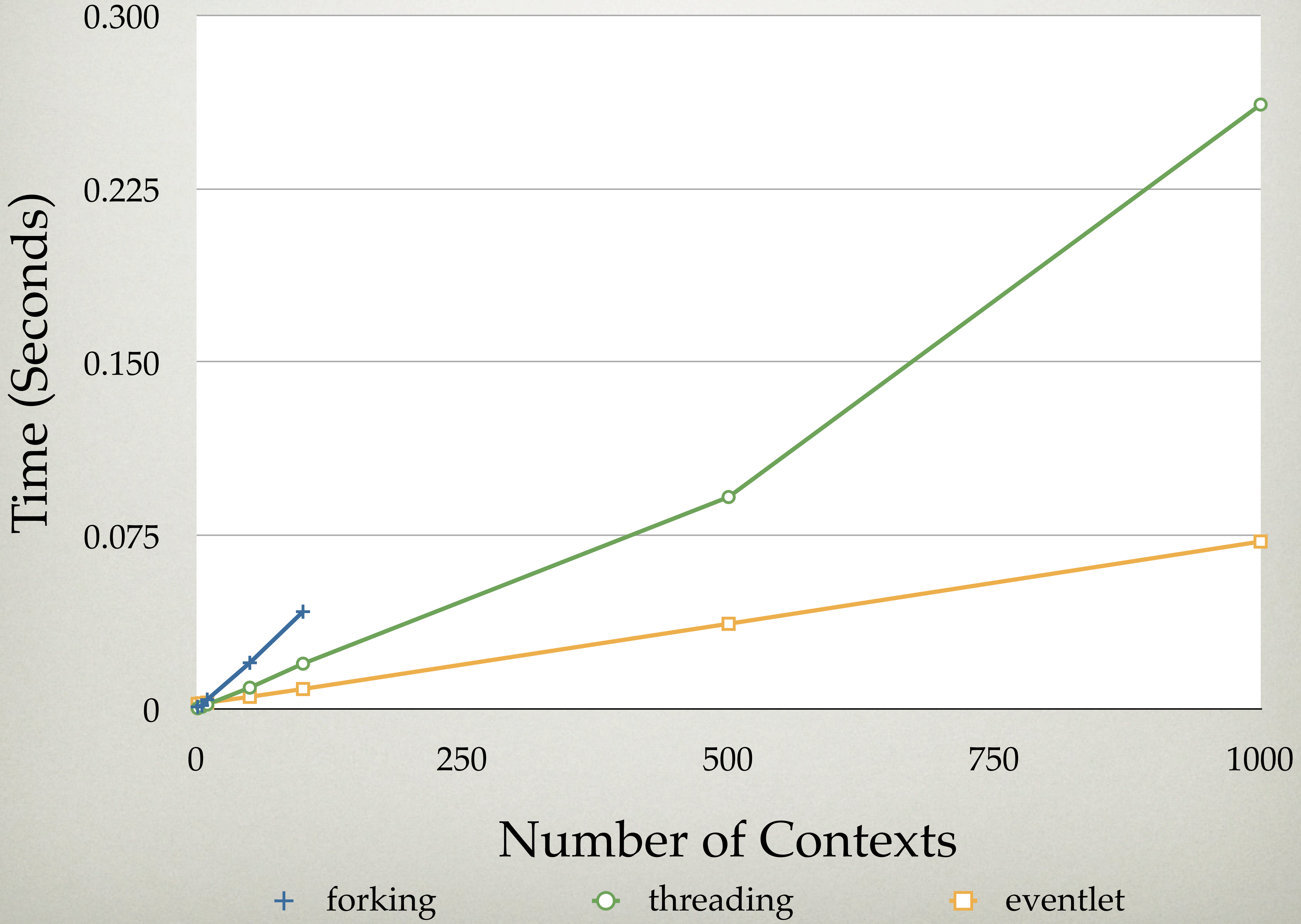
Green Threads: Lightweight Threads

- Regular POSIX threads are **Preemptive**
  - Non-Deterministic
- Green Threads are **Cooperative**
  - Deterministic
- Green Threads use much less memory

Memory Usage



Time


0.300
0.225
0.150
0.075
0
Time (Seconds)
0
250
500
750
1000
Number of Contexts
+ forking
○ threading
□ eventlet

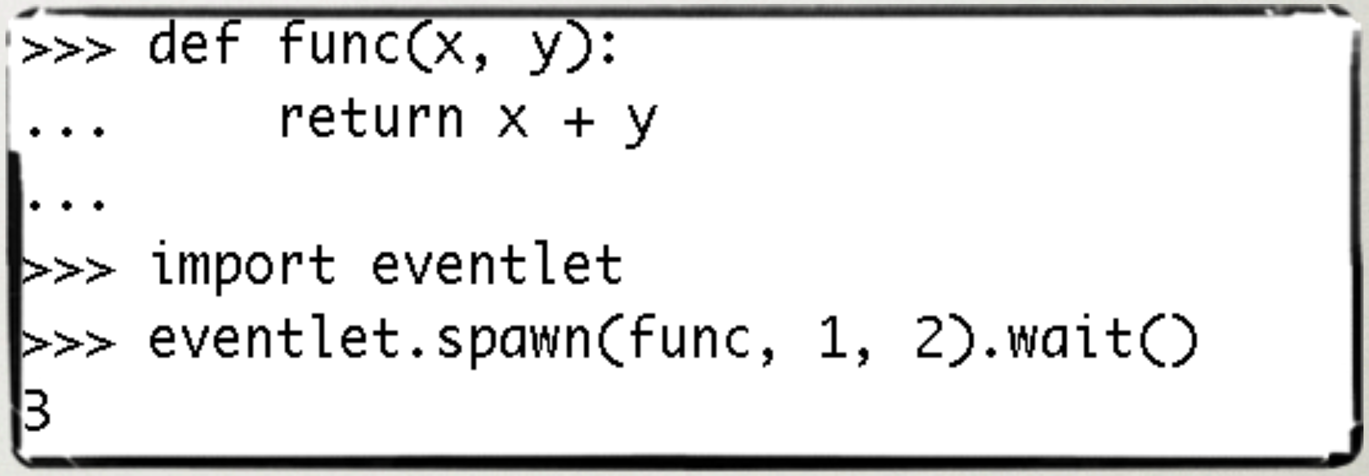
Spawning a Green Thread

- spawn(

  func,

  *args,

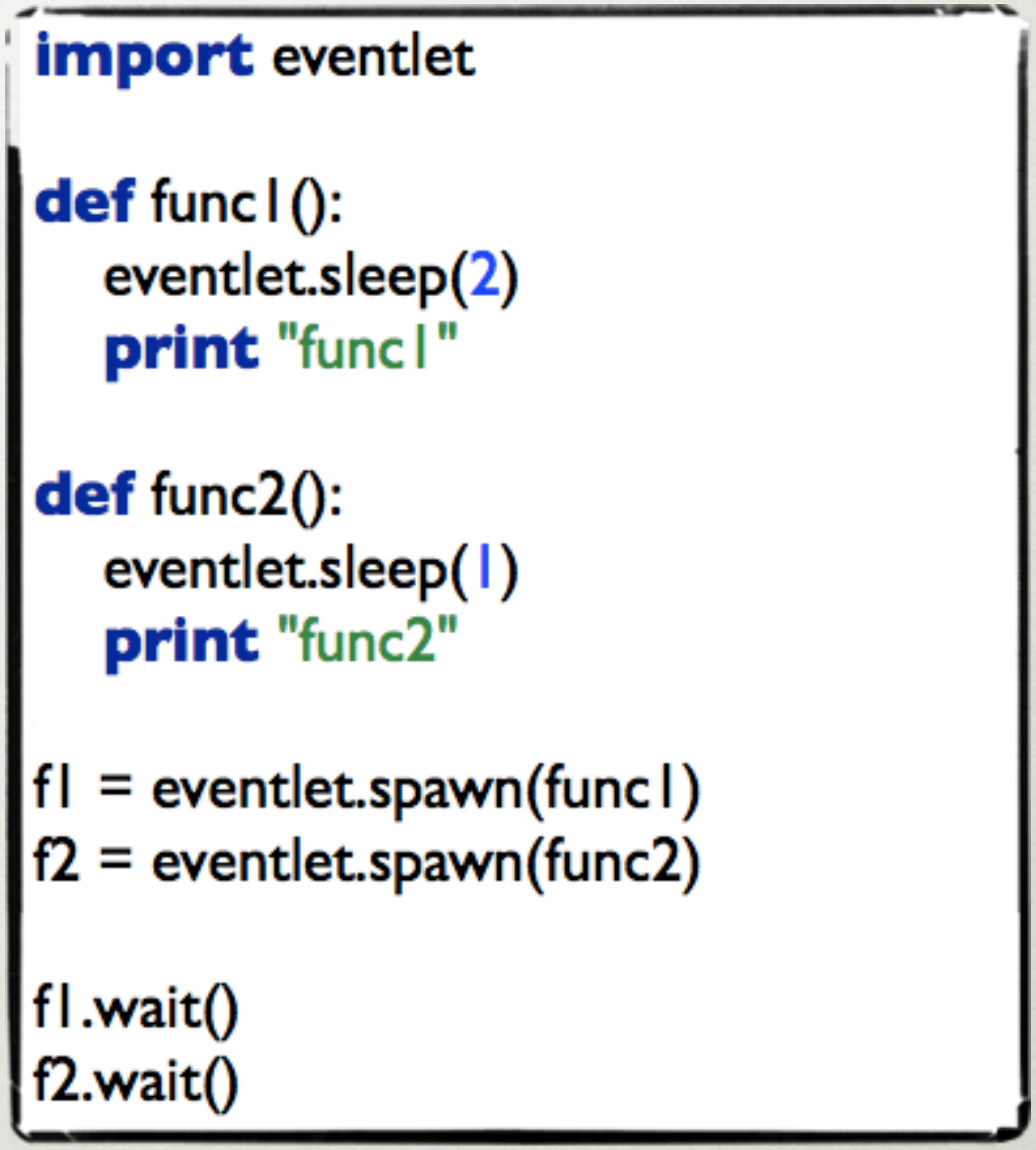
  **kwargs)



```
>>> def func(x, y):
...     return x + y
...
>>> import eventlet
>>> eventlet.spawn(func, 1, 2).wait()
3
```

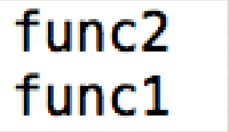
Cooperating: Voluntarily Yielding

- sleep()
  - “Run something else, then switch back to me as soon as possible”
- sleep(1)
  - “Switch to me after 1 second”



```
import eventlet

def func1():
    eventlet.sleep(2)
    print "func1"

def func2():
    eventlet.sleep(1)
    print "func2"

f1 = eventlet.spawn(func1)
f2 = eventlet.spawn(func2)

f1.wait()
f2.wait()
```

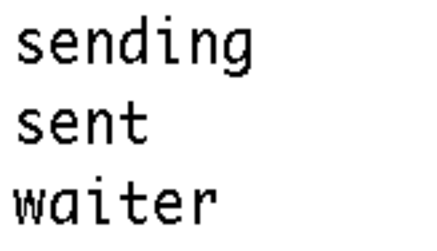
Outputs:



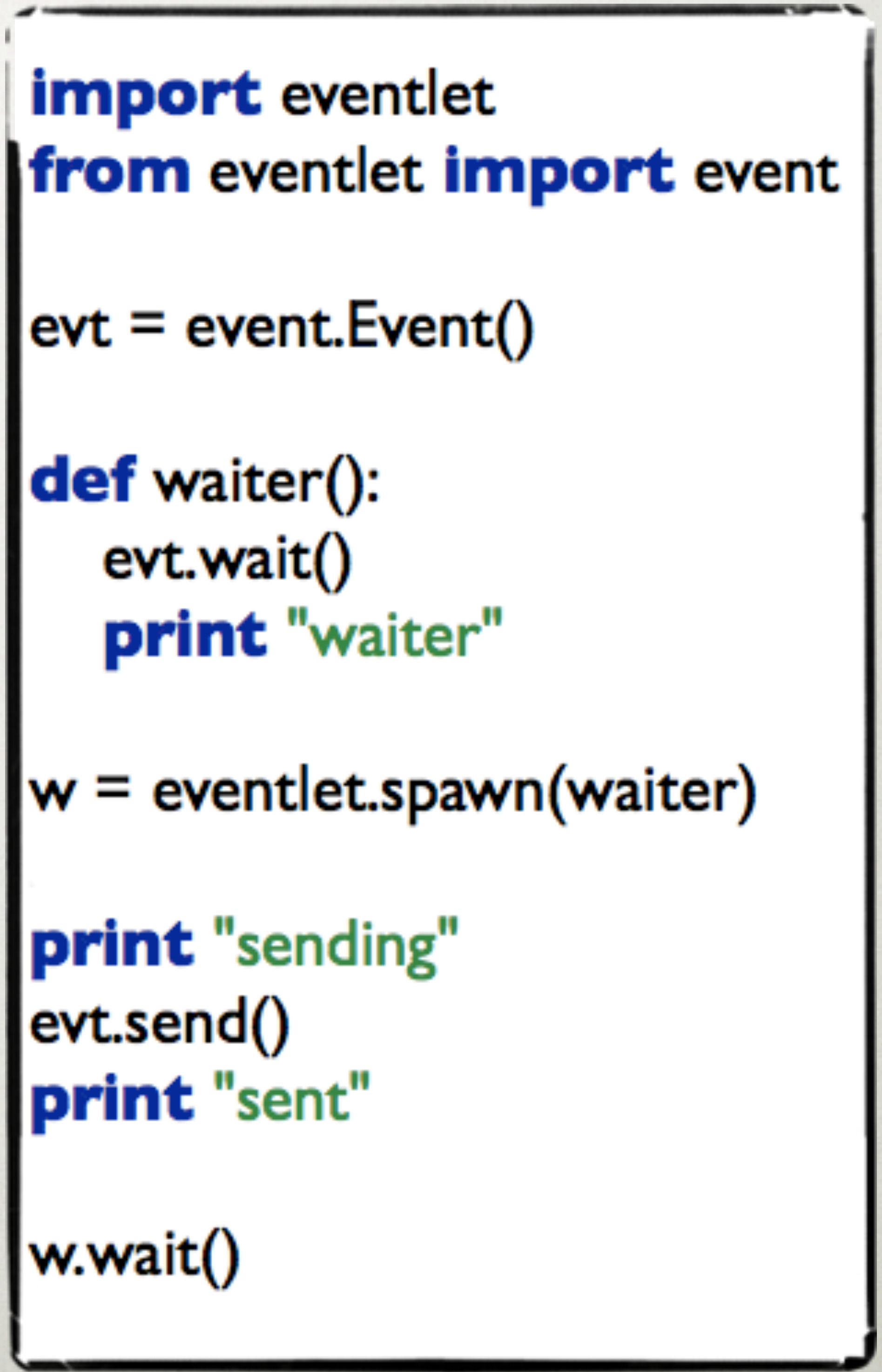
```
func2
func1
```

Synchronization: Event

- One sender, multiple waiters
- One use
- Output:



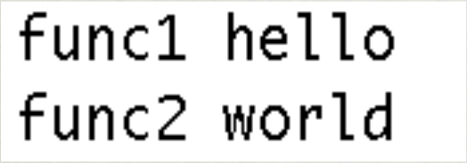
```
sending
sent
waiter
```



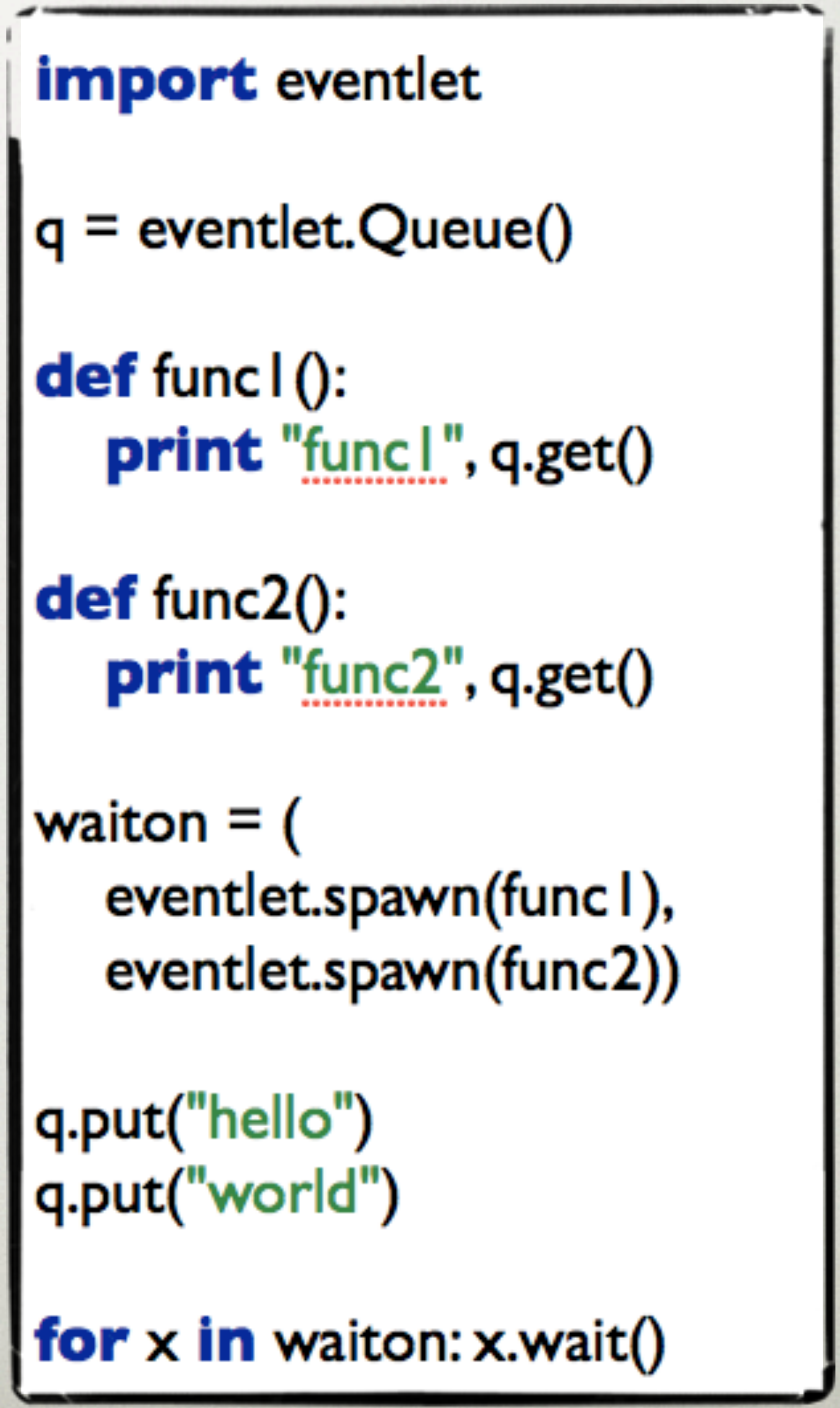
```
import eventlet
from eventlet import event

evt = event.Event()

def waiter():
    evt.wait()
    print "waiter"

w = eventlet.spawn(waiter)

print "sending"
evt.send()
print "sent"

w.wait()
```

Synchronization: Queue

- Multiple senders, multiple waiters
- Multiple use
- Output:



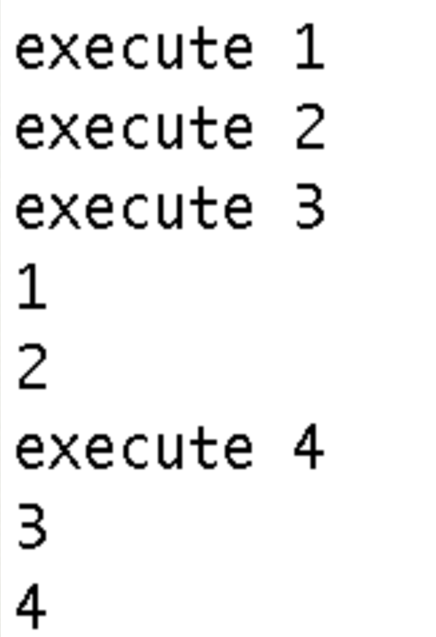
```
func1 hello
func2 world
```



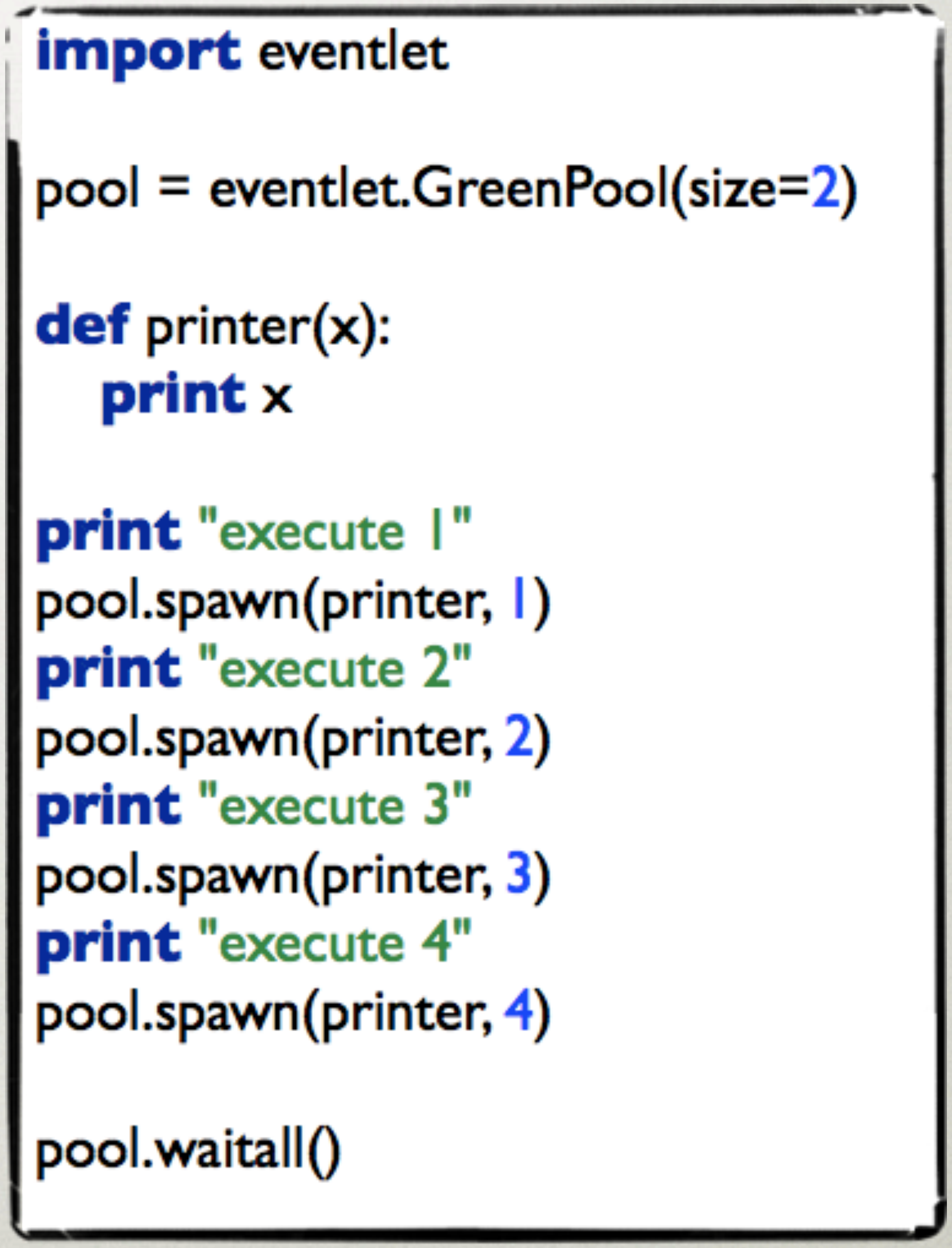
```
import eventlet

q = eventlet.Queue()

def func1():
    print "func1", q.get()

def func2():
    print "func2", q.get()

waiton = (
    eventlet.spawn(func1),
    eventlet.spawn(func2))

q.put("hello")
q.put("world")

for x in waiton: x.wait()
```

Concurrency Control: Pool

- Pools can be used to limit concurrency
- Output:



```
execute 1
execute 2
execute 3
1
2
execute 4
3
4
```



```
import eventlet

pool = eventlet.GreenPool(size=2)

def printer(x):
    print x

print "execute 1"
pool.spawn(printer, 1)
print "execute 2"
pool.spawn(printer, 2)
print "execute 3"
pool.spawn(printer, 3)
print "execute 4"
pool.spawn(printer, 4)

pool.waitall()
```

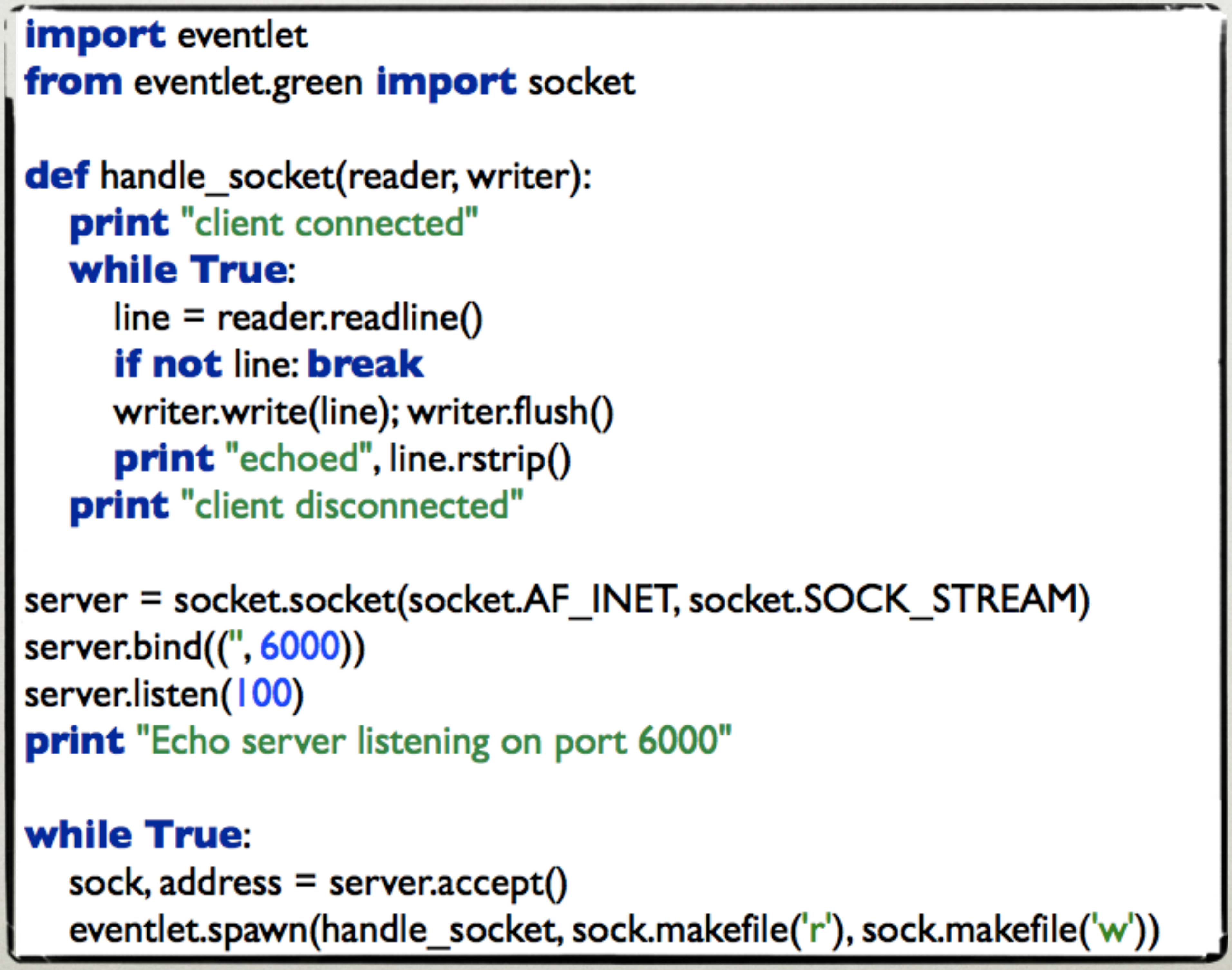
EVENTLET.GREEN

Cooperative Sockets

eventlet.green: Cooperative Sockets

- Same interface as socket.socket
- Instead of blocking, the cooperative socket switches to the main loop
- Main loop runs select (or poll, etc) and switches back to suspended coroutine when I/O is ready

Socket Example



```
import eventlet
from eventlet.green import socket

def handle_socket(reader, writer):
    print "client connected"
    while True:
        line = reader.readline()
        if not line: break
        writer.write(line); writer.flush()
        print "echoed", line.rstrip()
    print "client disconnected"

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('', 6000))
server.listen(100)
print "Echo server listening on port 6000"

while True:
    sock, address = server.accept()
    eventlet.spawn(handle_socket, sock.makefile('r'), sock.makefile('w'))
```

Sockets Have Implicit Cooperation Points

- Any API which would normally block cooperates instead
  - connect
  - read
  - write
  - etc.

Emulated Modules

- asyncore
- BaseHTTPServer
- httplib
- os
- select
- socket
- SocketServer
- ssl
- subprocess
- thread
- threading
- time
- urllib
- urllib2

Patching Other Libraries to Cooperate

- Import one module patched with cooperative sockets
  - patcher.import_patched
- Monkeypatch sys.modules globally
  - patcher.monkey_patch

Release Schedule

- Releasing 0.9.5 today
  - Cleanup release
- Sprinting this week
- 1.0 release soon!

Spawning

WSGI Server Written Using Eventlet

Spawning: Highly Configurable

- Can be configured to use:
  - Multiple OS Processes
  - Multiple POSIX Threads
  - Green Threads
- And various combinations of the three

Spawning: Designed for COMET

- “Real Time” web applications are finally becoming popular
- Servers must keep open one connection per active user
- When Spawning is configured to use eventlet’s green threads it is perfect for COMET

Summary

Eventlet

- High Scalability Non-Blocking I/O
- True Coroutines using Greenlet
- Green Threads with Scheduler
- Cooperative socket Implementation
- Easy to Integrate with Existing Libraries

Eventlet in Production

- In production at Linden Lab (Second Life) since 2006
- Handles a huge amount of traffic

Q&A