Denis Bilenko

Introduction to Gevent

Timeline

- 1999 Stackless Python
- 2004 Greenlet
- 2006 Eventlet
- 2009 Gevent 0.x (libevent)
- 2011 Gevent 1.0dev (libev, c-ares)

Changes in 1.0dev

- Replaced libevent with libev
- Replaced libevent-dns with c-ares
- Event loop is pluggable
- Resolver is pluggable
- Multiple OS threads supported

Fixed annoyances with 0.x

- Python’s signal module now works
- Resolver reads /etc/hosts & /etc/resolv.conf
- Fork no longer breaks DNS resolver

Plan

- Coroutines: why use them
  - Blocking vs. non-blocking sockets
- Gevent
  - Implementation
  - API
- 3rd party packages

Why coroutines

How to make network apps

- Blocking sockets
  - Examples: httplib, Django
- Non-blocking sockets
  - Examples: Twisted, Tornado
- Non-blocking but looks like blocking
  - Examples: gevent, eventlet

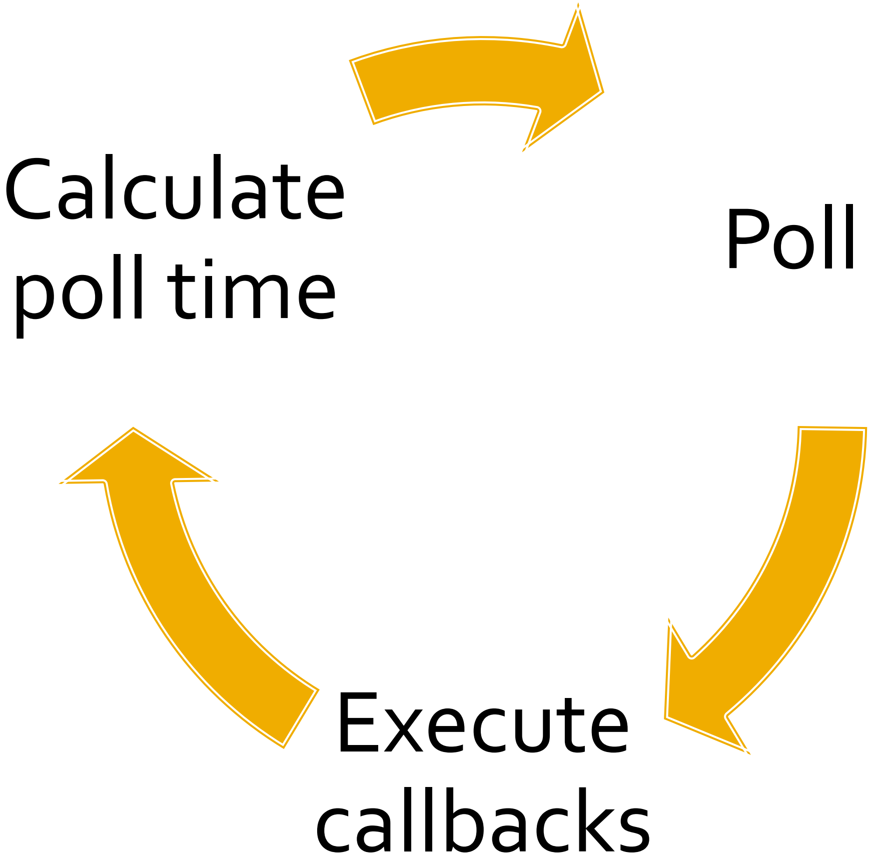
Blocking sockets

- Simple for single connection
- Concurrent via multithreading
  - Portable (+)
  - Need locks and thread-safe libraries (-)
  - Memory hungry (-)
  - Python’s GIL, contention on multicore (-)

Non-blocking sockets

- Scalable (better memory usage)
- Caller must retry when descriptor is ready
- Check readiness with select/poll/epoll/kqueue
- select/poll scales as O(N of total descriptors)
- epoll scales as O(N of active descriptors)

Event loop



Calculate poll time

Poll

Execute callbacks

Callback-based programming

- Otherwise known as callback hell
  - still used a lot
- Incompatible with blocking libraries
  - stdlib
  - most web frameworks

Green threads

- Scalable as callbacks
- Context switch on I/O
  - Locks are rarely needed
- Only use single process (as any non-blocking)
  - No GIL problems
  - To utilize multicore use multiple processes
- Drop-in replacement for multithreading

What is a coroutine

- multi-shot vs. single-shot
- symmetric vs. asymmetric
- stackful vs. non-stackful

- Stackless Python: multi-shot, stackful
- **greenlet**: single-shot, stackful
- yield: single-shot, non-stackful

yield

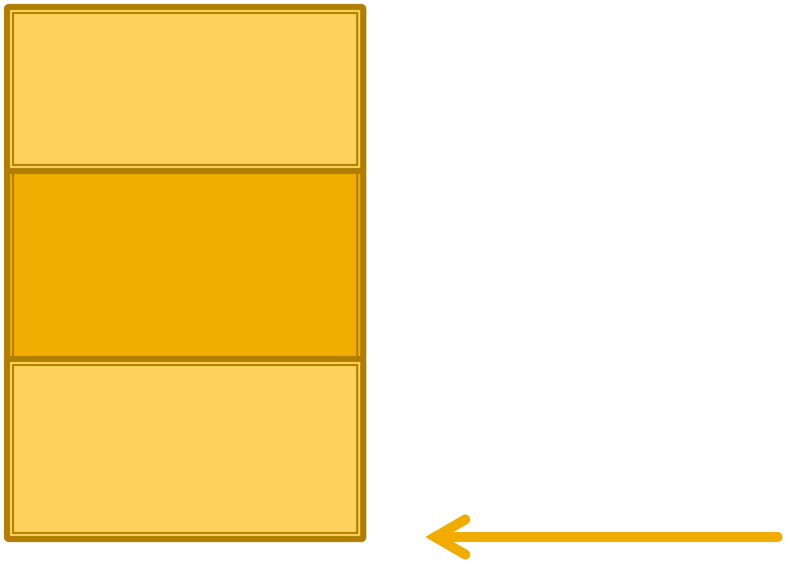
```
def myfunction(sock):
        yield sock.connect(<address>)
        yield sock.sendall(<data>)
        response = yield sock.read()
```

- yield is required at all levels

Can’t do this with yield

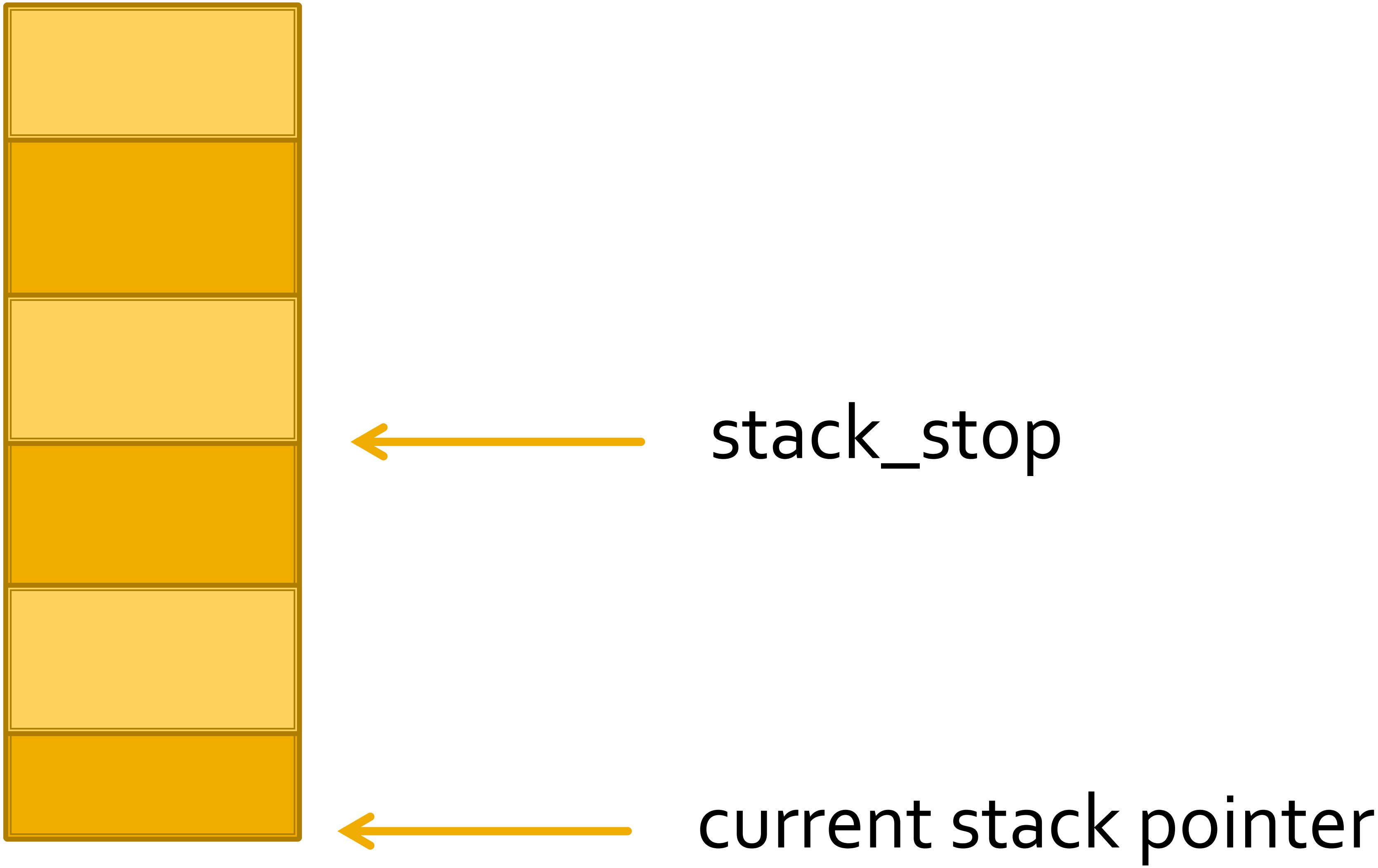
```
MAIN = greenlet.getcurrent()
def function_internal():
        MAIN.switch(10)
def function():
   function_internal()
   return 11

g1 = greenlet(function)
g1.parent   # => MAIN
g1.switch() # => 10
g1.switch() # => 11
g1.dead     # => True
```

Stack switching



first switch into g1: remember stack_stop

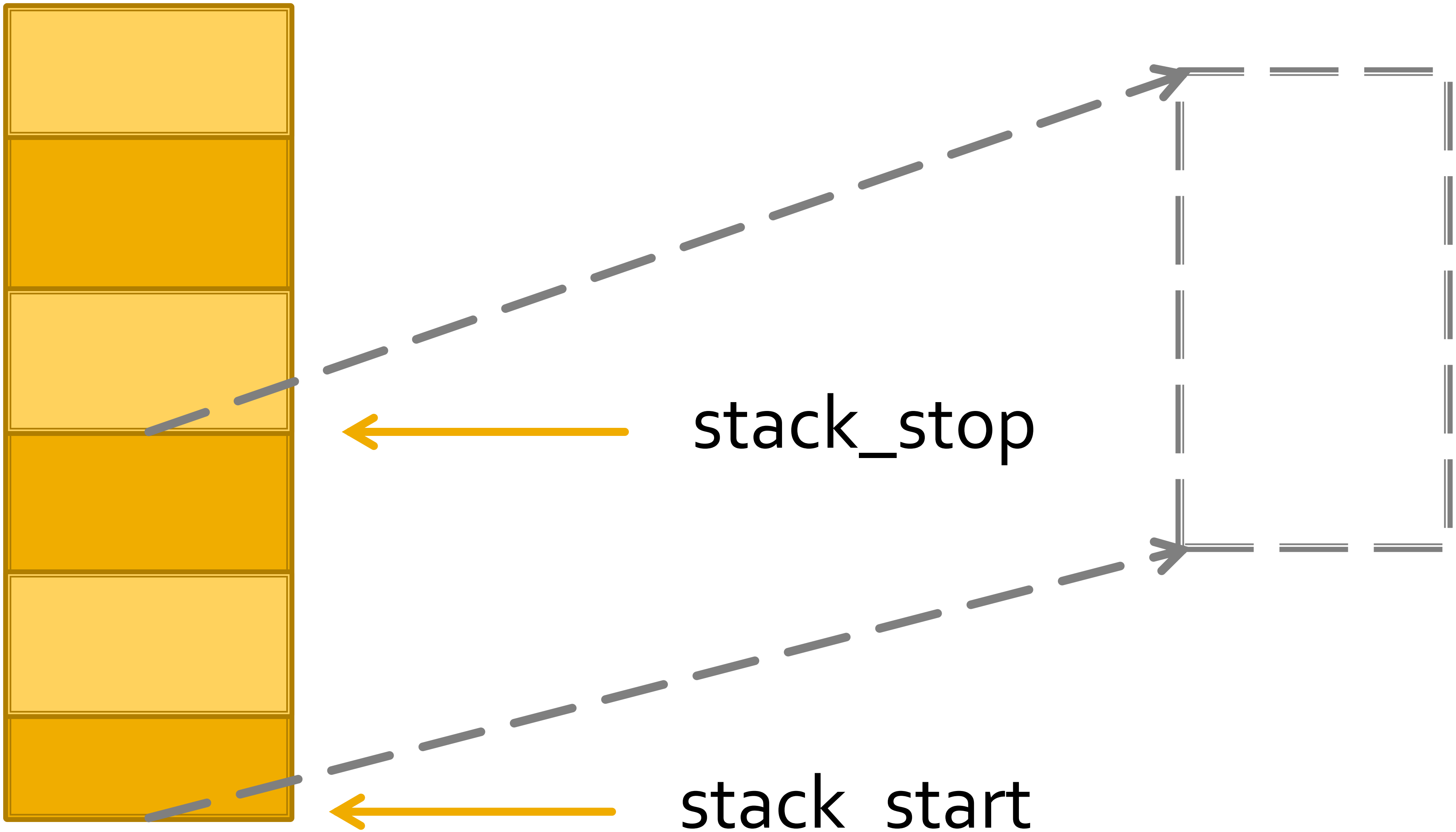
Stack switching



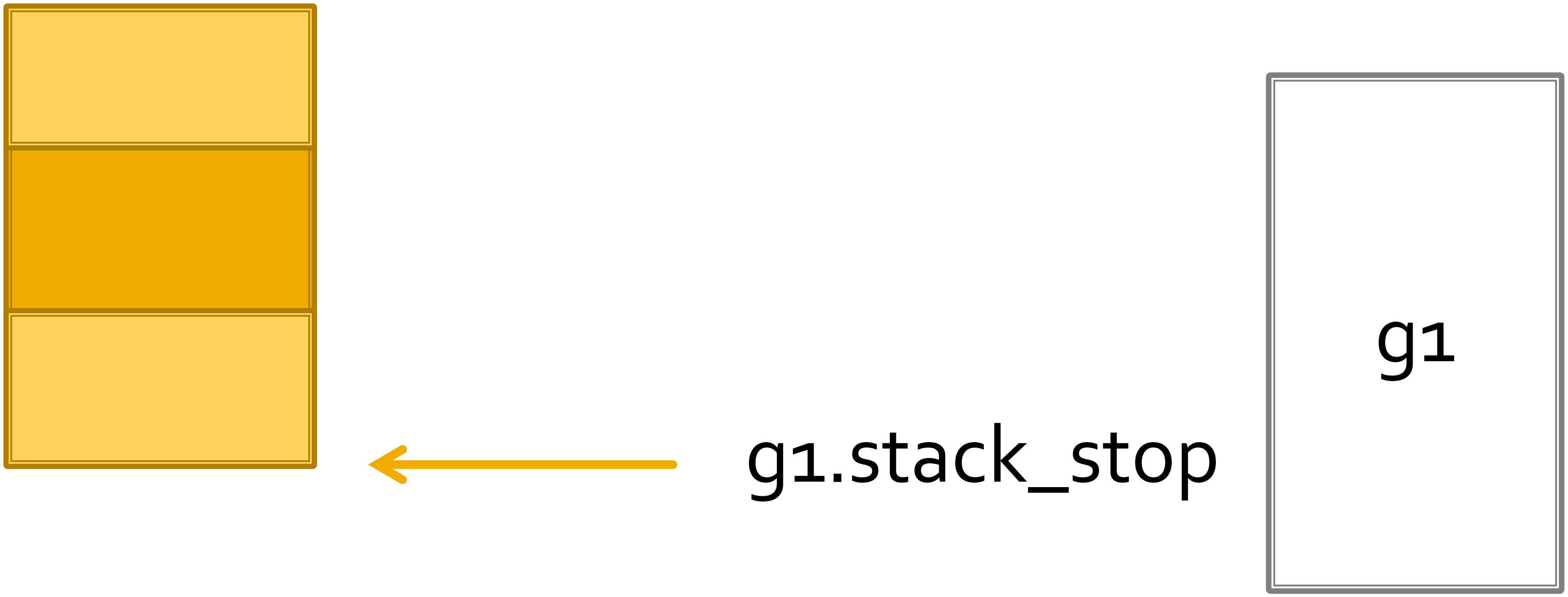
stack_stop

current stack pointer

Stack switching



Stack switching



g1.stack_stop

g1

now g1 is inactive and on the heap

greenlet

Pros:

- It’s quite fast
- It uses memory efficiently

Cons:

- Portability limited
- PyThreadState is shared between greenlets
  - Gevent clears and restores the exception (tb lost)

What about swapcontext

- Possible to implement greenlet API
  - https://github.com/redbo/python-swapcontext
- Memory has to be allocated upfront
  - Similar memory requirements as with threading
- Slower, does at least syscall or two per switch

How Gevent works

gevent.core: event loop

- Wrapper around libev
  - libevent before 1.0

```
loop = gevent.core.loop(optional parameters)

io_watcher = loop.io(<fd>, READ)
io_watcher .start(myhandler[, arg1, ...])
loop.run()
```

Internal API, not needed in applications

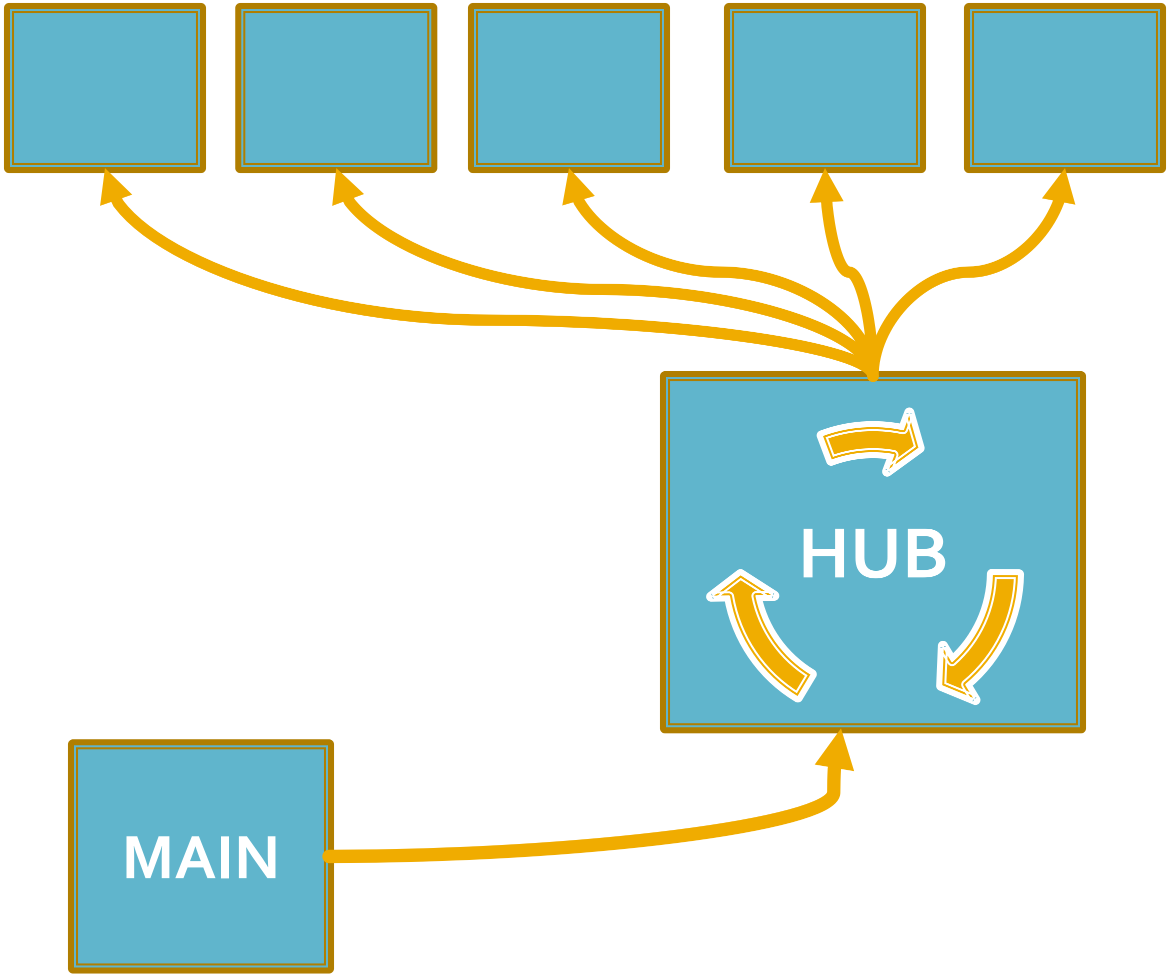
Gevent.core: watchers

- io(<fd>, <event>)
- timer(<at>, <repeat>)
- signal(<signalnum>)
- idle()
- async()
- fork()
- prepare()/check()
- callback()

watcher.start(func, *args)
watcher.stop()

http://cvs.schmorp.de/libev/

Hub: event loop in a greenlet



Hub: event loop in a greenlet

- hub = get_hub() # get or create
- hub.loop   # access the loop
- hub.switch()   # resume the loop
- hub.wait()   # wait for event

```
# put the current greenlet to sleep
def sleep(seconds):
    hub.wait(hub.loop.timer(seconds))
```

Hub: wait for event

```
def wait(self, watcher):
    watcher.start(getcurrent().switch)
    try:
        self.switch()
    finally:
        watcher.stop()
```

Hub: wait for event

```
def wait(self, watcher):
    unique = object()
    watcher.start(getcurrent().switch, unique)
    try:
        result = self.switch()
        assert result is unique, result
    finally:
        watcher.stop()
```

Cooperative socket

```
def recv(self, *args):
    while True:
    try:
        return self._sock.recv(*args)
    except socket.error as ex:
        if ex.args[0] != EWOULDBLOCK:
            raise
        io = hub.loop.io(self.fileno(), READ)
        hub.wait(io)
```

Cooperative networking

- gevent.socket
  - DNS resolution via c-ares (libevent-dns before 1.0)
- gevent.ssl
- gevent.select (only select())

Example

```
from gevent import monkey; monkey.patch_all()
import gevent, urllib2

def download(url):
        print urllib2.urlopen(url).read()

g = gevent.spawn(download, “http://gevent.org”)
download(“http://python.org”)
g.join()
```

Monkey patching

- monkey.patch_all()
  - socket
  - ssl
  - time.sleep, select.select
  - thread
  - threading, incl. local
- monkey.patch_all(thread=False)

Not necessary but highly recommended

Greenlet

Greenlet.spawn creates Greenlet instance and starts it

```
g = Greenlet(function, arg1, arg2=value)
g.start()  # asynchronous

# wait for it to complete
g.join()

# raise an asynchronous exception
g.kill()
```

Greenlet

Greenlet.spawn creates Greenlet instance and starts it

```
g = Greenlet(function, arg1, arg2=value)
g.start()  # asynchronous

# wait for it to complete
g.join(timeout=2)

# raise an async exception, wait for g to die
g.kill(timeout=2)
```

Timeout

```
with gevent.Timeout(5):
 response = urllib2.urlopen(url)
 for line in response:
  print line
# raises Timeout if not done after 5 seconds

with gevent.Timeout(5, False):
 response = urllib2.urlopen(url)
 for line in response:
  print line
# exits block if not done after 5 seconds
```

- Beware of “except:”
- Cannot interrupt non-yielding code (use SIGALRM for that)

Pool

```
pool = gevent.pool.Pool(10000)

while True:
        socket, address = listener.accept()
        pool.spawn(handle, socket, address)
        # spawn blocks if more than 10000 conns
```

join, kill, apply, apply_async, imap, imap_unordered, map

TCP Server

```
def handle(socket, address):
        socket.sendall(“hello”)

server = StreamServer((‘’, 5000), handle)
server.start()
server.stop()
```

Supports SSL, Pools

Greenlet communication

- gevent.event
  - Event
  - AsyncResult
- gevent.queue
  - Queue, PriorityQueue, JoinableQueue
- gevent.coros
  - Semaphore, BoundedSemaphore, Lock, Rlock
- If you know the name, you know the API!

WSGI Server

- 0.x
  - Based on libevent-http: gevent.wsgi
  - Pure Python: gevent.pywsgi
- 1.0
  - gevent.pywsgi
- Gunicorn:
  - Pre-fork workers for any of gevent servers
  - http://gunicorn.org

database drivers

- Psycopg2: generic support for coroutines
- amysql and gevent-mysql
- gevent-memcache
- All pure Python packages, e.g. redis

3rdparty

- WebSocket protocol and Socket.io backend
- Locust – HTTP load testing tool
- tproxy/hroute – TCP/HTTP proxies with logic in Python
- gevent-zeromq
  - kaylee – Distributed MapReduce with 0MQ
  - Miyamoto – fast clusterable task queue inspired by GAE

http://bit.ly/ProjectsUsingGevent

Case study: omegle.com

- half a million visitors / day
- 20000 online users
- 3 servers, 4gb of memory each
  - 10% of memory used
  - 60% cpu used
- ~60 KB/connection
- Switched to gevent from twisted
  - When it had 5000 users in a single process
  - Single process use grew up to 9600 peak users

Future plans

1.0

- Fast WSGI server: gevent.wsgi
- Documentation

Do not block the release:

- Py3k support
- Thread pools
- Process pools

Summary

- coroutines are easy to use threads
- as efficient as async libraries
- works well if app if app is I/O bound
- simple API many things familiar
- works with unsuspecting 3rd party modules

Thank you!

http://gevent.org

@gevent