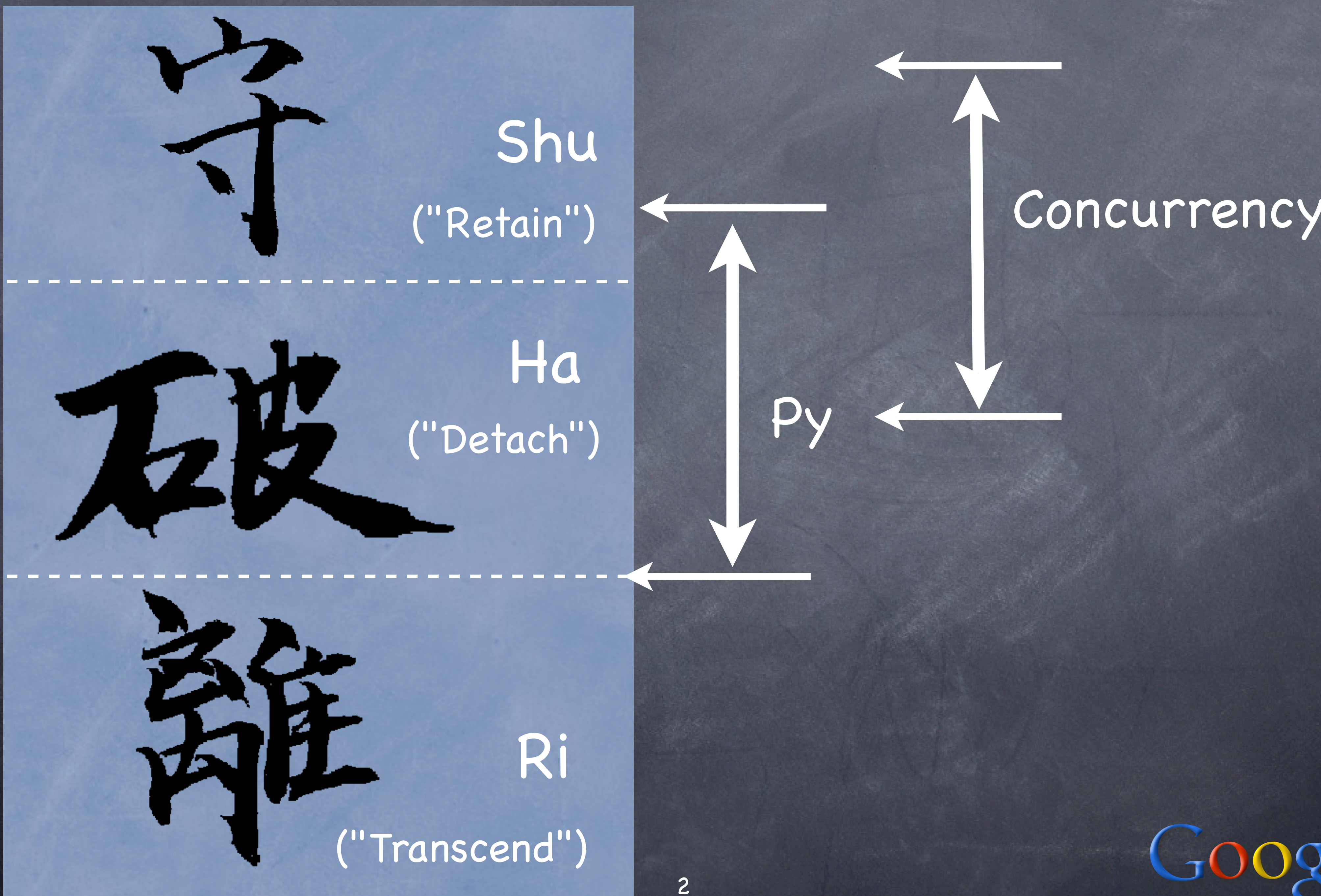
Python Patterns of Concurrent Programming

http://www.aleax.it/accu_pyconc.pdf

Google™

The "levels" of this talk



Concurrency

- we want to do several things "at once"
  - to react to "events" (mostly, but not exclusively, "external" ones)
    - actions by the user[s] (on a UI, or...)
    - network events, I/O completion, OS...
    - parsing, graph-walking, ...
  - to reduce "latency" (vs "throughput")
  - to exploit many available resources (CPUs, cores, computers...) towards a single goal simultaneously
    - "parallel", "distributed", ...

Google

Event-driven: the callback

- the "callback" idea
  - shades of the Observer DP
- pass a callable to "somebody"
  - in Observer: to the Observable target
- the "somebody" stores it "somewhere"
  - a container, an attribute, whatever
- and will call it "when appropriate"
  - in Observer: on state-changes
  - in Event-driven: on meaningful "events"
- also used in "customization" (e.g., sort)

Callback issues

- what arguments are to be used on the call?
  - none: simplest, very rough
  - in ODP: the Observable object whose state just changed
    - lets 1 callable observe several Obs'bles
  - or: "description" of state changes
    - saves "round-trips" to obtain them
  - in EDA: identifier or description of event
- but -- what about other arguments (related to the callable, not to the Obs'ble/Event)...?

Fixed args in callbacks

- functools.partial(callable, *a, **kw)
  - pre-bind any or all arguments
- however...:
  - x.set_cb(functools.partial(f, *a, **kw))
    - vs
  - x.set_cb(f, *a, **kw)
- ...having the set-callback itself accept (and pre-bind) arguments is far neater/handier
- sombunall[1] Python callback systems do that

[1]: Robert Anton Wilson

Callback "dispatching"

- what if more than one callback is set for a single event (or, Observable)?
  - remember and call the latest one only
    - simplest, roughest
  - remember and call them all
    - LIFO? FIFO? or...?
    - how do you _remove_ a callback?
    - can one callback preempt others?
- can events (or state changes) be "grouped"?
  - use object w/methods instead of callable

Callbacks and Errors

- are "errors" events like any others?
- or are they best singled-out?
  http://www.python.org/pycon/papers/deferex/
- the Deferred pattern: one Deferred holds...
  - N "chained" callbacks for "successes" +
  - M "chained" callbacks for "errors"
  - each callback is held WITH opt *a, **kw
  - plus, argument for "event / error identification" (or, result of previous callback along the appropriate "chain")

Scheduled callbacks

- standard library module sched
- s = sched.Sched(timefunc, delayfunc)
  - e.g, Sched(time.time, time.sleep)
- evt = s.enter(delay, priority, callable, arg)
  - or s.enterabs(time, priority, callable, arg)
  - may s.cancel(evt) later
- s.run() runs events until queue is empty (or an exception is raised in callable or delayfunc: it propagates but leaves s in stable state, s.run can be called again later)

Google

Some other Python callbacks

- for system-events:
  - atexit.register(callable, *a, **k)
  - oldhandler = signal.signal(signum, callable)
  - sys.displayhook, sys.excepthook, sys.settrace(callable)
  - readline.set_startup_hook, set_pre_input_hook, set_completer
- parsing, timing, debugging, customization, ...

Event-driven: "the Loop"

- demultiplex external events coming into the program from several distinct channels
  - select.select, select.poll (+ 3rd-party ones: python-epoll, Py-Kqueue, ...)
  - win32all.MsgWaitForMultipleEvents
- "register" event sources (fd's, windows, kernel sync objects, ...)
- call the demux function (maybe w/timeout)
- returns set of events occurred (or, times out, allowing optional further "polling"...)

Dispatching Events

- the "Event Loop" per se ``bottlenecks" events into 1 spot (+ others if ``modal")
- worst solution: if/else tree at that spot
- better: register callbacks for "specific" events (by type, source, or ...)
  - & add to the "Event Loop" the registry and dispatching of callbacks it affords
- Loop + Registry/Dispatching == REACTOR
  - http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

Google

asyncore

- asyncore.loop(timeout, use_poll, map, count)
  - only supports events handled by select (or poll, if use_poll is true)
  - also does dispatching (so: a reactor)
  - items in map (or asyncore.socket_map) must have methods readable, writable, all of socket's methods, plus handle_* ones...
    - normally subclass asyncore.dispatcher
      - also handles add-to-the-map, wraps socket methods
      - to buffer: dispatcher_with_send
      - for files (Unix only): file_dispatcher

asyncore Echo srv (1/2)

```
  import asyncore

class MainServerSocket(asyncore.dispatcher):
    def __init__(self, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,
                                          socket.SOCK_STREAM)
        self.bind(('',port))
        self.listen(5)
    def handle_accept(self):
        newSocket, address = self.accept()
        SecondaryServerSocket(newSocket)
```

asyncore Echo srv (2/2)

```
class SecondaryServerSocket(
                asyncore.dispatcher_with_send):
    def handle_read(self):
        receivedData = self.recv(8192)
        if receivedData:
            self.send(receivedData)
        else:
            self.close()

MainServerSocket(8881)
asyncore.loop()
```

Twisted Core

- twisted.internet.reactor module
  - IReactorCore: "system events", run, stop
  - IReactorTCP: listen/connect w/Factory (ServerFactory, ClientFactory)
    - Factory itf has buildProtocol method
    - IProtocol has dataReceived method
  - many other reactor itf's: FDSet, Process, SSL, Threads, Time, UDP, UNIX
  - many implementations: select, poll, KQueue, Win32 (WFMO/IOCP) + many with GUI event-loops integration

Twisted Echo server

```
  from twisted.internet import protocol,
  reactor

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

factory = protocol.Factory()
factory.protocol = EchoProtocol

reactor.listenTCP(8881, factory)
reactor.run()
```

Interleaving execution

- traditional sequential processing:

```
data = get_all_data()
results = process_all_data(data)
show_all_results(results)
```

- an "interleaved" approach:

```
finished = False
while not finished:
    d, f = get_some_data()
    r, f = process_some_data(d, f)
    finished = show_some_results(r, f)
```

- reduces latency (<u>may</u> worsen throughput)
- AKA "µthreads", "fibers", "tasklets"...

Google

Interleaving issues

- each component must:
  - restore its previous state (if any),
  - do "some" work (not TOO much!),
  - save its current state (if not finished),
  - yield control to "other components"
  - note: MUST yield if risks blocking (I/O...)
- clear pluses: programmer has complete control, works great w/event-driven approaches, no "surprises", ...
- issues: lots of work, some "delicate" points

Python interleaving

- may use classes (to group state & behavior)
- generators are very handy for this
  - automatically "preserve state" including "point of execution" at yield time
  - yield is now "bidirectional" (expression)
- no standard "scheduling" conventions (yet)
  - but, cfr http://mail.python.org/pipermail/python-list/2007-May/438370.html
- don't confuse microthreading, coroutines, and continuations...

Stackless Python

- tasklets (AKA microthreads): wrap a function for interleaving purposes
  - supply critical sections, scheduling (cooperative _or_ even preemptive...)
- channels: let tasklets send/receive data
  - intrinsically cooperate w/scheduling
- serialization (pickling/unpickling) for checkpointing, transfering tasks, ...
- popular in games, online and non (EVE Online, Mythos, Sylphis 3D, ...) for speed and convenience

"Real" Threads

- the threading module (NOT legacy thread)
- uses underlying OS (preemptive) threading facilities (for CPython; underlying _VM_ threading facilities for Jython, IronPython)
  - CPython adds a Global Interpreter Lock (GIL) to ease integrating non-thread-aware/safe external C libraries
  - C-coded extensions may explicitly release the GIL ("allow threading")/re-acquire it
  - semantics constrained by cross-platform needs (priorities, thread-interruption...)

Threads & "Atomicity"

```
# will this work? why, or, why not?

d = {}; fd = open('fil.txt')
def f():
  for L in fd:
    k, v = L.split()[:2]
    d[k] = v
t1 = threading.Thread(target=f)
t2 = threading.Thread(target=f)
t1.start(); t2.start()
t1.join(); t2.join()
```

"Thread-safe" iterator

```
def tsiter(it, L=None):
  if L is None:
    L = threading.Lock()
  it = iter(it)
  while True:
    with L:
      yield it.next()

# NB: in 2.5, this needs adding a:
# from __future__ import with_statement
```

"Thread-safe" mapping

```
class tsdict(dict):
  def __init__(self, *a, **k):
    self.L = threading.Lock()
    with self.L:
      dict.__init__(self, *a, **k)
  def __setitem__(self, k, v):
    with self.L:
      dict.__setitem__(self, k, v)
  def __getitem__(self, k):
    with self.L:
      return dict.__getitem__(self, k)
  ...
```

Big risks with this...

- not covering _all_ "atomicity" needs:
  - e.g.: "for k in tsiter(tsd): <body>" STILL needs "NO ALTERATION" to tsd throughout the loop (what ensures this?)
- the more, finer-grained locks are around, the higher the risk of deadlocks:
  - T1 gets lock A then waits on lock B,
  - T2 gets lock B then waits on lock A...
- race conditions and deadlocks are the worst kinds of bugs (hard to reproduce, hard to test for, very hard to debug, ...)

A more structured approach

- make every "shared resource" either UNCHANGING during multitasking,
- or, have it OWNED by ONE dedicated thread (only one changing OR accessing it)
- every other thread requests operations on the shared resource by sending MESSAGES to the dedicated owner-thread (and may wait for a result-message if applicable)
- Queue.Queue is the intrinsically-threadsafe communication structure for messages from thread to thread (work-request, result)

TS file-read-to-Queue

```
# a "self-activated" owner-thread variant

lines = Queue.Queue(N)
def tsread(fn, linequeue):
  with open(fn) as fd:
    for line in fd: linequeue.put(line)
t = threading.Thread(target=tsread,
        args=('fil.txt', lines))
t.setDaemon(True)
t.start()
```

TS "dict service" (1/2)

```
# a more classic "dedicated work thread"

def tsmapping(wrq, d):
  while True:
    op, k, v, rq = wrq.get()
    if op=='set': d[k] = v
    elif op=='keys': v = d.keys()
    elif op=='in': v = k in d
    else: v = d.get(k, v)
    if rq: rq.put(v)
```

TS "dict service" (2/2)

```
# ...and a little syntax sugar on top...:
class tsdict(object, UserDict.DictMixin):
  def __init__(self, *a, **k):
    self.wrq = Queue.Queue(N)
    t = threading.Thread(target=tsmapping,
          args=(self.wrq, dict(*a, **k)))
    t.setDaemon(True); t.start()
    self.rsq = Queue.Queue()
  def keys(self):
    self.wrq.put(('keys','','',self.rsq))
    return self.rsq.get()
  ...__getitem__, __setitem__,...
```

Granularity & Performance

- each context-switch among threads has a performance cost (potentially in both latency and throughput)
  - so does each locking (and Queue.Queue intrinsically does locking, too)
- consider "batching things up" a bit
  - compromise betw. throughput & latency
- ideal points for context switches: where they would occur anyway (syscalls, I/O)
- avoid polling; consider thread-pools; mix and match threads w/event-driven ops; ...

Processes

- heavier/costlier than threads (but not by much, in Linux, Solaris, Mac OS X, BSD, ...)
- better isolation is a good guard vs bugs (and, can "drop privileges" &c for security)
- no GIL -- use all CPUs/cores implicitly
- "resource sharing" goes a bit against the grain (possible, but "message-style" IPC mechanisms are generally preferable)
  - IPC via sockets affords nearly unbounded potential scalability...
- see http://pypi.python.org/pypi/processing

"Orchestrating" concurrency

- Twisted is good at orchestrating all this...
  - focus on event-driven operations
  - strong support for threads & processes
  - particularly strong at networking
  - some support for interleaving ("Flow" now deprecated, use task.Cooperator class)
- highly structured (chiefly via interfaces)
- lots of "moving parts" (alternatives, details, docs, abstractions...)
- is there something simpler to use...?

NetworkSpaces

- close kin to Linda / tuple spaces
- Python implementation uses Twisted
  - clients also available for R (and Matlab)
  - dual-licensed open source (GPL or Pro)
- 5 primitives: store, fetch{Try}, find{Try}
  - store/fetch like Queue's put/get
  - multiple "slots" w/arbitrary name per ws
  - "find" for non-removing read access
- works with Sleigh (to distribute and load-balance work across cores/CPUs/nodes)

nws basics

- you need to be running an nws server
  - there's only one kind (Python+Twisted)
  - may run as a daemon
  - may monitor it through a web itf
    - optional, helpful "pybabelfish" daemon
- must have nws clients installed on all hosts
  - may be Python, R, or Matlab ones
  - language interop via ASCII strings, only
  - within 1 language, any serializable obj OK

An NWS "server"

```
>>> from nws import client
>>> ws = client.NetWorkSpace('primes')
>>> import gmpy
>>> pr = [gmpy.mpz(2)]
>>> def store_prime(n):
...   while len(primes)<=n:
...     pr.append(pr[-1].next_prime())
...   ws.store('prime', int(primes[n]))
...
>>> while True:
...   store_prime(ws.fetch('n'))
...
```

An NWS "client"

```
>>> from nws import client
>>> ws = client.NetWorkSpace('primes')
>>> def getprime(n):
...     pr.store('n', n)
...     return pr.fetch('prime')
...
>>> print [getprime(n) for n in range(23)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83]
```


Google

Sleigh

- an enhancement of the original Simple Network Of Workstation (SNOW) idea
- a nws.sleigh.Sleigh instance s coordinates a set of "workers"
  - may be local, or on arbitrary nodes
  - uses ssh (or web, or ...) to start nodes
- may send the same task to each worker (s.eachWorker), and/or,
- may shard tasks among them (s.eachElem)
  - also: s.imap, s.starmap, ...
- [use SleighArgs, *not* "SleighArguments"]

Parallel Python

- similar to Sleigh (but more direct, less fancy iteration): message-based, supports both SMP and clusters semi-transparently
- class pp.Server supports ncpus, remote nodes, stats/logging, and:

```
submit(func, args, depfuncs, modules,
  callback, callbackargs, group, globals)
```

- returns callable that wait & returns results
- also, for explicit use:

```
wait(group)
```

Q & A

http://www.aleax.it/accu_pyconc.pdf

? !