PyParallel:

How we removed the GIL and exploited all cores

(and came up with the most sensationalized presentation title we could think of)

(without actually needing to remove the GIL at all!)

PyData NYC 2013, Nov 10th

Trent Nelson
Software Architect
Continuum Analytics
@ContinuumIO, @trentnelson
trent.nelson@continuum.io
http://speakerdeck.com/trent/

CONTINUUM ANALYTICS

Before we begin…

- 153 slides
- 45 minutes
- = 17.64 seconds per slide
- First real “public” presentation about PyParallel
- Compressed as much info as possible about the work into this one presentation (on the basis that the slides and video will be perpetually available online)
- It’s going to be fast
- It’s going to be technical
- It’s going to be controversial
- …
- 50/50 chance of it being coherent

About Me

- Core Python Committer
- Subversion Committer
- Founder of Snakebite

http://www.snakebite.net

HOME | NEWS | NETWORK | VISION | NEEDED | DONATE | CONTACT

snakebite

the open source development network

Why do we develop open source software on closed networks? Why do open source developers only have access to a fraction of platforms that their software will eventually run on? And why are the buildbots always red?!

Snakebite was created out of a desire to try and address problems like these faced by open source projects.

Mission Statement

> *Snakebite is a network that strives to provide developers of open source projects complete and unrestricted access to as many different* ***platforms****,* ***operating systems****,* ***architectures****,* ***compilers****,* ***devices****,* ***databases****,* ***tools*** *and* ***applications*** *that they may need in order to optimally develop their software.*

Background

The idea for Snakebite was conceived by Trent Nelson, a Python committer, after the PyCon 2008 conference in Chicago. Frustrated with some of the limitations of Buildbot and the general difficulties involved in cross-platform development for a platform you don't have direct access to, Trent envisioned a

Progress

Fast-forward to 2012: the science-lab-turned-server-room is now home to three racks, two industrial exhaust fans, six portable air-conditioners and jury-rigged ventilation and plumbing system that would make MacGyver proud:

First Milestone

In September 2012, Snakebite reached its first milestone. Over twenty Snakebite build slaves had been set up for the Python project on a wide variety of platforms (AIX, Solaris, HP-UX, IRIX,

About Me

- Core Python Committer
- Subversion Committer
- Founder of Snakebite
  - One big amorphous mass of heterogeneous UNIX gear
  - AIX RS/6000
  - SGI IRIX/MIPS
  - Alpha/Tru64
  - Solaris/SPARC
  - HP-UX/IA64
  - FreeBSD, NetBSD, OpenBSD, DragonFlyBSD
- Background is 100% UNIX, love it. Romantically.
- But I made my peace with Windows when XP came out

Survey Says...

- How many people use Windows...
  - at work, on the desktop?
  - at work, on the server?
  - at home?
- How many people use Linux...
  - at work, on the desktop?
  - at work, on the server?
  - at home?
- How many people use OS X...
  - at work, on the desktop?
  - at work, on the server?
  - at home?

Survey Says...

- Other UNIX at work on the server?
  - AIX
  - Solaris
  - HP-UX
  - Other?
- New work project; Python 2 or 3?
  - Python 2
  - Python 3
- Knowledge check:
  - Good understanding of Linux I/O primitives? (epoll etc)
  - Good understanding of asynchronous I/O on Windows via IOCP, overlapped I/O and threads?

Controversial Survey Says...

- Pro-Linux; how many people think...
  - Linux kernel is technically superior to Windows?
  - Linux I/O facilities (epoll etc) are technically superior to Windows?
- Pro-Windows; how many people think...
  - Windows kernel/executive is technically superior to Linux?
  - Windows asynchronous I/O facilities (IOCP, overlapped I/O) are technically superior to Linux?
- Apples and oranges; both are good

Thanks!

Moving on…

TL;DR What is PyParallel?

- Set of modifications to CPython interpreter
- Allows multiple interpreter threads to run concurrently
- ….without incurring any additional performance penalties
- Intrinsically paired with Windows asynchronous I/O primitives
- Catalyst was python-ideas async discussion (Sep 2012)
- Working prototype/proof-of-concept after 3 months:
  - Performant HTTP server, written in Python, automatically exploits all cores
  - pyparallel.exe -m async.http.server
- Source code:
  - https://bitbucket.org/tpn/pyparallel
- Coming soon: `conda install pyparallel`!

What's it look like?

...

Minimalistic PyParallel async server

```
0 import async
1
2 class Disconnect:
3     pass
4
5 server = async.server('localhost', 8080)
6 async.register(transport=server, protocol=Disconnect)
7 async.run()
8
```

Protocol-driven…
(protocols are just classes)

```
 0 class Disconnect:
 1     """
 2     Disconnects a connection as soon as it is established.
 3     """
 4     pass
 5
 6 class Discard:
 7     """
 8     Null sink; discards all incoming data, never disconnects.
 9     """
10     def data_received(self, data):
11         pass
12
13 class QOTD:
14     """
15     Prints a quote of the day, then disconnects.
16     """
17     initial_bytes_to_send = b'An apple a day keeps the doctor away.\r\n.'
18
```

You implement completion-oriented methods

```
 0
 1 import time
 2 class Daytime:
 3     """
 4     Send a string representation of the current time, then disconnect.
 5     """
 6     def initial_bytes_to_send(self):
 7         return time.ctime() + '\r\n'
 8
 9
10 import socket
11 from datetime import datetime
12 class Time:
13     """
14     Send a 32-bit unsigned integer in binary format and network byte
15     order, representing the number of seconds since 00:00 (midnight)
16     January 1st, 1900 GMT, then close the connection.
17     """
18     def initial_bytes_to_send(self):
19         delta = datetime.utcnow() - datetime(1900, 1, 1)
20         return socket.htonl(int(delta.total_seconds()))
21
```

Hollywood Principle: Don't call us, we'll call you

```
0
1  class EchoData:
2      """
3      Echo service; return (write) all data received back to the sender.
4      (The data_received() callback will be invoked as soon as any data
5      is received from the sender.)
6      """
7      def data_received(self, data):
8          return data
9
10 class EchoLine:
11     """
12     Echo service; return (write) all lines received back to the sender.
13     (The lines_received() callback will be invoked whenever at least one
14     line is received.  ``lines`` will always be a tuple (and thus, iterable)
15     even if only one line has been received.  This allows the callback to
16     peek ahead at any and all lines.)
17     """
18     line_mode = True
19     def lines_received(self, lines):
20         return os.linesep.join(lines)
21
```

async.server() = transport

async.register() = fuses protocol + transport

```
0 import async
1
2 class Disconnect:
3     pass
4
5 server = async.server('localhost', 8080)
6 async.register(transport=server, protocol=Disconnect)
7 async.run()
8
```

Part 1

The Catalyst

• • •

asyncore: included batteries don't fit

https://mail.python.org/pipermail/python-ideas/2012-October/016311.html

A seemingly innocuous e-mail...

- Late September 2012: “asyncore: batteries not included” discussion on python-ideas
- Whirlwind of discussion relating to new async APIs over October
- Outcome:
  - PEP-3156: Asynchronous I/O Support Rebooted
  - Tulip/asyncio
- Adopted some of Twisted’s (better) paradigms

Things I've Always Liked About Twisted

- Separation of protocol from transport
- Completion-oriented protocol classes:

```
 0 from twisted.internet import protocol
 1
 2 class EchoClient(protocol.Protocol):
 3     def connectionMade(self):
 4         self.transport.write("hello, world!")
 5
 6     def dataReceived(self, data):
 7         "As soon as any data is received, write it back."
 8         print "Server said:", data
 9         self.transport.loseConnection()
10
11     def connectionLost(self, reason):
12         print "connection lost"
```

PEP-3156 & Protocols

```
 0 class SomeProtocol:
 1     def connection_made(self, transport):
 2         # Called exactly once when the connection is established.
 3         ...
 4
 5     def data_received(self, data):
 6         # Called zero or more times whenever data is received; `data` is
 7         # always a non-empty bytes object.
 8         ...
 9
10     def eof_received(self):
11         # Called at most once when the other end has written an EOF.
12         ...
13
14     def connection_lost(self, exc):
15         # Called exactly once when the connection has been closed or aborted;
16         # additional error information may be available in `exc`.
17         ...
18
```

Understanding the catalyst…

- Completion-oriented protocols, great!
- But I didn't like the implementation details
- Why?
- Things we need to cover in order to answer that question:
  - Socket servers: readiness-oriented versus completion-oriented
  - Event loops and I/O multiplexing techniques on UNIX
  - What everyone calls asynchronous I/O but is actually just synchronous non-blocking I/O (UNIX)
  - Actual asynchronous I/O (Windows)
  - I/O Completion Ports
- Goal in 50+ slides: "ahhh, that's why you did it like that!"

Socket Servers: Completion versus Readiness

```
class SomeProtocol:
    def connection_made(self, transport):
        # Called exactly once when the connection is established.
        ...

    def data_received(self, data):
        # Called zero or more times whenever data is received; `data` is
        # always a non-empty bytes object.
        ...

    def eof_received(self):
        # Called at most once when the other end has written an EOF.
        ...

    def connection_lost(self, exc):
        # Called exactly once when the connection has been closed or aborted;
        # additional error information may be available in `exc`.
        ...

```

Socket Servers: Completion versus Readiness

- Protocols are **completion-oriented**
- ….but UNIX is inherently **readiness-oriented**
- read() and write():
  - No data available for reading? Block!
  - No buffer space left for writing? Block!
- Not suitable when serving more than one client
  - (A blocked process is only unblocked when data is available for reading or buffer space is available for writing)
- So how do you serve multiple clients?

Socket Servers Over the Years

(Linux/UNIX/POSIX)

- One process per connection:

  accept() -> fork()
- One thread per connection
- Single-thread + non-blocking I/O + event multiplexing

accept()->fork()

- Single server process sits in an accept() loop
- fork() child process to handle new connections
- One process per connection, doesn't scale well

One thread per connection...

- Popular with Java, late 90s, early 00s
- Simplified programming logic
- Client classes could issue blocking reads/writes
- Only the blocking thread would be suspended
- Still has scaling issues (but better than accept()->fork())
  - Thousands of clients = thousands of threads

Non-blocking I/O + event multiplexing

- Sockets set to non-blocking:
  - read()/write() calls that would block return EAGAIN/EWOULDBLOCK instead
- Event multiplexing method
  - Query readiness of multiple sockets at once
- "Readiness-oriented"; can I do something?
  - Is this socket ready for reading?
  - Is this socket ready for writing?
- (As opposed to "completion-oriented": that thing you asked me to do has been done.)

I/O Multiplexing Over the Years (Linux/UNIX/POSIX)

- select()
- poll()
- /dev/poll
- epoll
- kqueue

I/O Multiplexing Over the Years

select() and poll()

- select()
  - BSD 4.2 (1984)
  - Pass in a set of file descriptors you're interested in (reading/writing/exceptional conditions)
  - Set of file descriptors = bit fields in array of integers
  - Fine for small sets of descriptors, didn't scale well
- poll()
  - AT&T System V (1983)
  - Pass in an array of "pollfds": file descriptor + interested events
  - Scales a bit better than select()

I/O Multiplexing Over the Years

select() and poll()

- Both methods had O(n)* kernel (and user) overhead
- Entire set of fds you're interested in passed to kernel on each invocation
- Kernel has to enumerate all fds – also O(n)
- ….and you have to enumerate all results – also O(n)
- Expensive when you're monitoring tens of thousands of sockets, and only a few are "ready"; you still need to enumerate your entire set to find the ready ones

[*] select() kernel overhead O(n^3)

Late 90s

- Internet explosion
- Web servers having to handle thousands of simultaneous clients
- select()/poll() becoming bottlenecks
- C10K problem (Kegel)
- Lots of seminal papers started coming out
- Notable:
  - Banga et al:
    - “A Scalable and Explicit Event Delivery Mechanism for UNIX”
    - June 1999 USENIX, Monterey, California

Early 00s

- Banga inspired some new multiplexing techniques:
  - FreeBSD: kqueue
  - Linux: epoll
  - Solaris: /dev/poll
- Separate ***declaration of interest*** from ***inquiry about readiness***
  - Register the set of file descriptors you're interested in ahead of time
  - Kernel gives you back an identifier for that set
  - You pass in that identifier when querying readiness
- Benefits:
  - Kernel work when checking readiness is now O(1)
- epoll and kqueue quickly became the preferred methods for I/O multiplexing

Back to the python-ideas async discussions

- Completion-oriented protocols were adopted (great!)

```
0 class SomeProtocol:
1     def connection_made(self, transport):
2         # Called exactly once when the connection is established.
3         ...
4
5     def data_received(self, data):
6         # Called zero or more times whenever data is received; `data` is
7         # always a non-empty bytes object.
8         ...
9
10    def eof_received(self):
11        # Called at most once when the other end has written an EOF.
12        ...
13
14    def connection_lost(self, exc):
15        # Called exactly once when the connection has been closed or aborted;
16        # additional error information may be available in `exc`.
17        ...
18
```

- But how do you drive completion-oriented Python classes when your OS is readiness based?

The Event Loop

- Twisted, Tornado, Tulip, libevent, libuv, ZeroMQ, node.js
- All single-threaded, all use non-blocking sockets
- Event loop ties everything together

The Event Loop (cont.)

- It's literally an endless loop that runs until program termination
- Calls an I/O multiplexing method upon each "run" of the loop
- Enumerate results and determine what needs to be done
  - Data ready for reading without blocking? Great!
    - read() it, then invoke the relevant protocol.data_received()
  - Data can be written without blocking? Great! Write it!
  - Nothing to do? Fine, skip to the next file descriptor.

Recap: Asynchronous I/O (PEP-3156/Tulip)

- Exposed to the user:
  - Completion-oriented protocol classes
- Implementation details:

```
Single-threaded* server +
Non-blocking sockets    +
Event loop              +
I/O multiplexing method = asynchronous I/O!
```

([*] Not entirely true; separate threads are used, but only to encapsulate blocking calls that can't be done in a non-blocking fashion. They're still subject to the GIL.)

The thing that bothers me about all the "async I/O" libraries out there...

- ....is that the implementation
  - Single-threaded
  - Non-blocking sockets
  - Event loop
  - I/O multiplex via kqueue/epoll
- ....is well suited to Linux, BSD, OS X, UNIX
- But:
  - There's nothing asynchronous about it!
  - It's technically synchronous, non-blocking I/O
  - It's inherently single-threaded.
    - (It's 2013 and my servers have 64 cores and 256GB RAM!)
- And it's just awful on Windows...

Ah, Windows

- The bane of open source
- Everyone loves to hate it
- “It’s terrible at networking, it only has select()!”
- “If you want high-performance you should be using Linux!”
- …
- “Windows 8 sucks”
- “Start screen can suck it!”

(If you're not a fan of Windows, try keep an open mind for the next 20-30 slides)

…

Windows NT: 1993+

- Dave Cutler: DEC OS engineer (VMS et al)
- Despised all things UNIX
  - *Quipped on Unix process I/O model:*
    - *"getta byte, getta byte, getta byte byte byte"*
- Got a call from Bill Gates in the late 80s
  - *"Wanna' build a new OS?"*
- Led development of Windows NT
- Vastly different approach to threading, kernel objects, synchronization primitives and I/O mechanisms
- What works well on UNIX isn't performant on Windows
- What works well on Windows isn't possible on UNIX

I/O on Contemporary Windows Kernels (Vista+)

- Fantastic support for asynchronous I/O
- Threads have been first class citizens since day 1 (not bolted on as an afterthought)
- Designed to be programmed in a completion-oriented, multi-threaded fashion
- Overlapped I/O + IOCP + threads + kernel synchronization primitives = excellent combo for achieving high performance

I/O on Windows

If there were a list of things not to do…

- Penultimate place:
  - One thread per connection, blocking I/O calls
- Tied for last place:
  - accept() -> fork()
    - no real equivalent on Windows anyway
  - Single-thread, non-blocking sockets, event loop, I/O multiplex system call

So for the implementation of PEP-3156/Tulip…

…

(or any “asynchronous I/O” library that was developed on UNIX then ported to Windows…)

….let's do the worst one!

- The **best** option on UNIX is the **absolute worst** option on Windows
  - Windows doesn't have a kqueue/epoll equivalent* (nor should it)
  - So you're stuck with select()…

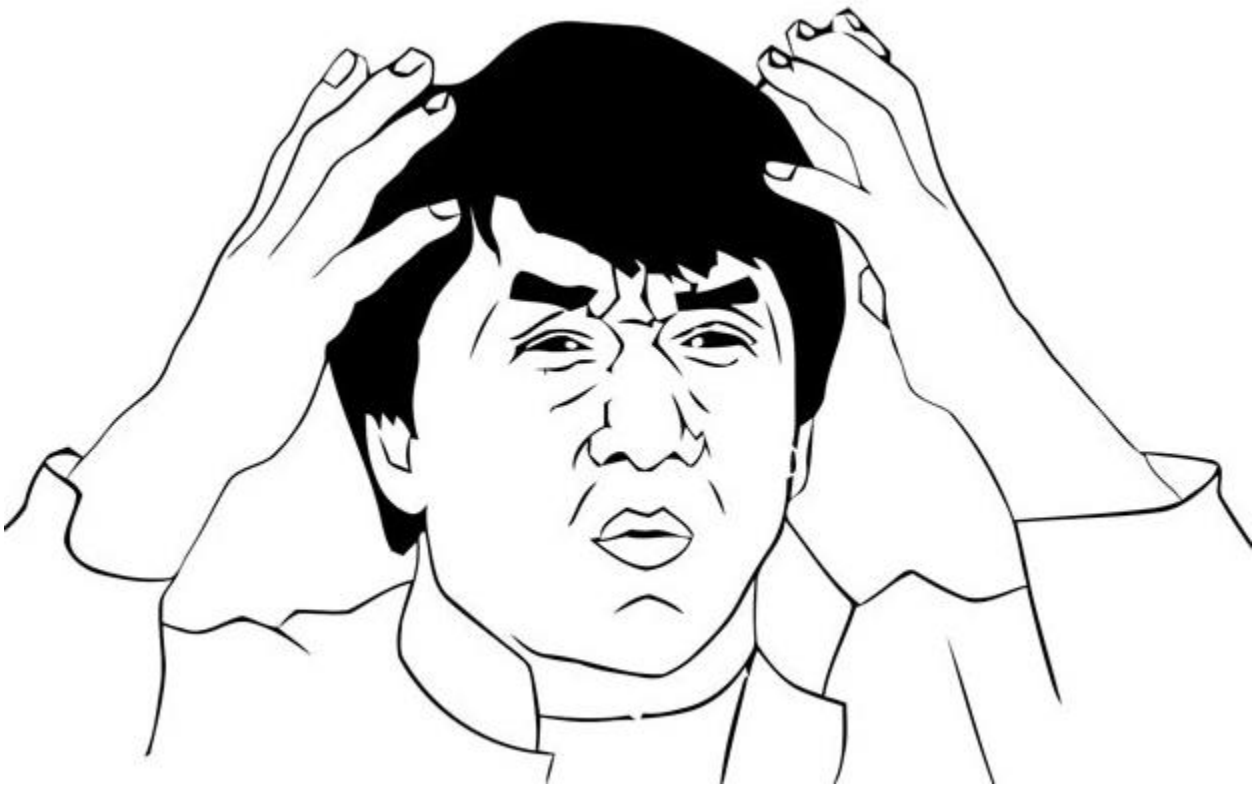
- [*] (Calling GetQueuedCompletionStatus() in a single-threaded event loop doesn't count; you're using IOCP wrong)

….but select() is terrible on Windows!

- And we're using it in a single-thread, with non-blocking sockets, via an event loop, in an entirely readiness-oriented fashion…
- All in an attempt to **simulate** asynchronous I/O…
- So we can drive completion-oriented protocols…

- …instead of using the native Windows facilities?
- Which allow **actual** asynchronous I/O
- And are all completion-oriented?

?!?



Let's dig into the details of asynchronous I/O on Windows

...

I/O Completion Ports

• • •

(It's like AIO, done right.)

IOCP: Introduction

- The best way to grok IOCP is to understand the problem it was designed to solve:
  - Facilitate writing high-performance network/file servers (http, database, file server)
  - Extract maximum performance from multi-processor/multi-core hardware
  - (Which necessitates optimal resource usage)

IOCP: Goals

- **Extract maximum performance** through parallelism
  - Thread running on every core servicing a client request
  - Upon finishing a client request, immediately processes the next request if one is waiting
  - Never block
  - (And if you do block, handle it as optimally as possible)
- **Optimal resource usage**
  - One active thread per core
  - Anything else introduces unnecessary context switches

On not blocking...

- UNIX approach:
  - Set file descriptor to non-blocking
  - Try read or write data
  - Get EAGAIN instead of blocking
  - Try again later
- Windows approach
  - Create an overlapped I/O structure
  - Issue a read or write, passing the overlapped structure and completion port info
  - Call returns immediately
  - Read/write done asynchronously by I/O manager
  - Optional completion packet queued to the completion port a) on error, b) on completion.
  - Thread waiting on completion port de-queues completion packet and processes request

On not blocking...

- UNIX approach:
  - Is this ready to write yet yet?
  - No? How about now?
  - Still no?
  - Now?
  - Yes!? Really? Ok, write it!
  - Hi! Me again. Anything to read?
  - No?
  - How about now?
- Windows approach:
  - Here, do this. Let me know when it's done.

Readiness-oriented
(reactor pattern)

Completion-oriented
(proactor pattern)

On not blocking...

- Windows provides an asynchronous/overlapped way to do just about everything
- Basically, if it *could* block, there's a way to do it asynchronously in Windows
- WSASend and WSARecv
- AcceptEx() vs accept()
- ConnectEx() vs connect()
- DisconnectEx() vs close()
- GetAddrinfoEx() vs getaddrinfo() (Windows 8+)
- (And that's just for sockets; all device I/O can be done asynchronously)

The key to understanding what makes asynchronous I/O in Windows special is…

…

Thread-specific I/O
versus
Thread-agnostic I/O

• • •

The act of getting the data out of nonpaged kernel memory into user memory

Thread-specific I/O

- Thread allocates buffer:
  - char *buf = malloc(8192)
- Thread issues a WSARecv(buf)
- I/O manager creates an I/O request packet, dispatches to NIC via device driver
- Data arrives, NIC copies 8192 bytes of data into nonpaged kernel memory
- NIC passes completed IRP back to I/O manager
  - (Typically involves DMA, then DIRQL -> ISR -> DPC)
- I/O manager needs to copy that data back to thread's buffer

Getting data back to the caller

- Can only be done when caller's address space is active
- The only time a caller's address space is active is when the calling thread is running
- Easy for synchronous I/O: address space is already active, data can be copied directly, WSARecv() returns
  - This is exactly how UNIX does synchronous I/O too
  - Data becomes available; last step before read() returns is for the kernel to transfer data back into the user's buffer
- Getting the data back to the caller when you're doing asynchronous I/O is much more involved...

Getting data back to the caller asynchronously

(when doing thread-specific I/O)

- I/O manager has to delay IRP completion until thread's address space is active
- Does this by queuing a kernel APC (asynchronous procedure call) to thread
  - (which has already entered an altertable wait state via SleepEx, WaitFor(Single|MultipleObjects) etc)
- This awakes the thread from its alertable wait state
- APC executes, copies data from kernel to user buffer
- Execution passes back to the thread
- Detects WSARecv() completed and continues processing

Disadvantages of thread-specific I/O

- IRPs need to be queued to the thread that initiated the I/O request via kernel APCs
- Kernel APCs wake threads in an alertable wait state
  - This requires access to the highly-contented, extremely critical global dispatcher lock
- The number of events a thread can wait for when entering an alertable wait state is limited to 64
- Alertable waits were never intended to be used for I/O

  (At least not high-performance I/O)

Thread-agnostic I/O

- Thread-specific I/O: IRP must be completed by calling thread
  - (IRP completion = copying data from nonpaged kernel memory to user memory)
  - (nonpaged = can't be swapped out; imperative when you've potentially got a device DMA'ing directly to the memory location)
- Thread-agnostic I/O: IRP does not have to be completed by calling thread

Thread-agnostic I/O Two Options:

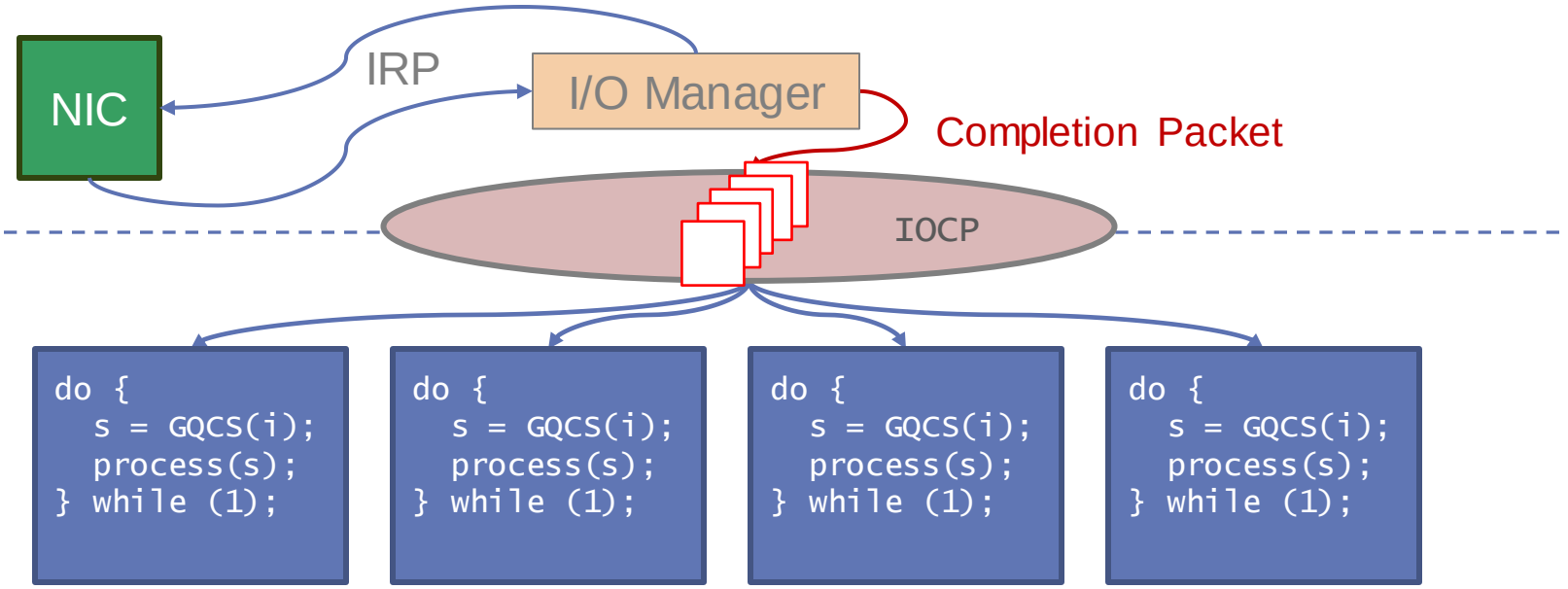
- I/O completion ports
  - IRP can be completed by any thread that has access to the completion port
- Registered I/O (Windows 8+)
  - User allocates large contiguous buffers at startup.
  - Buffer locked during IRP processing (nonpaged; can't be swapped out)
  - Mapped into the kernel address space
  - NIC DMAs data into kernel address space as usual
  - ....which just happens to also be the user's buffer
  - No need for the I/O manager to perform a copy back into user address space
  - (Similar role to SetFileIoOverlappedRange effect when doing overlapped file I/O)

Thread-agnostic I/O with IOCP

- Secret sauce behind asynchronous I/O on Windows
- IOCPs allow IRP completion (copying data from nonpaged kernel memory back to user's buffer) to be deferred to a thread-agnostic queue
- Any thread can wait on this queue (completion port) via GetQueuedCompletionStatus()
- IRP completion done just before that call returns
- Allows I/O manager to rapidly queue IRP completions
- ....and waiting threads to instantly dequeue and process

IOCP and Concurrency

- IOCPs can be thought of as FIFO queues
- I/O manager pushes completion packets asynchronously
- Threads pop completions off and process results:



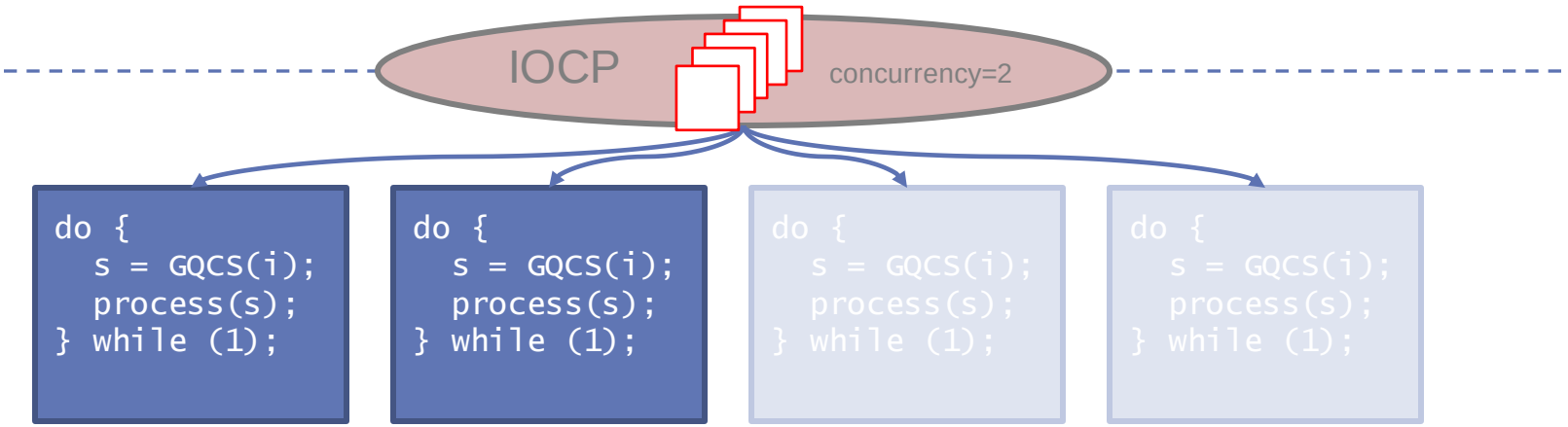
GQCS = GetQueuedCompletionStatus()

IOCP and Concurrency

- Remember IOCP design goals:
  - Maximise performance
  - Optimize resource usage
- Optimal number of active threads running per core: 1
- Optimal number of total threads running: 1 * ncpu
- Windows can't control how many threads you create and then have waiting against the completion port
- But it can control when and how many threads get awoken
- ….via the IOCP's maximum concurrency value
- (Specified when you create the IOCP)

IOCP and Concurrency

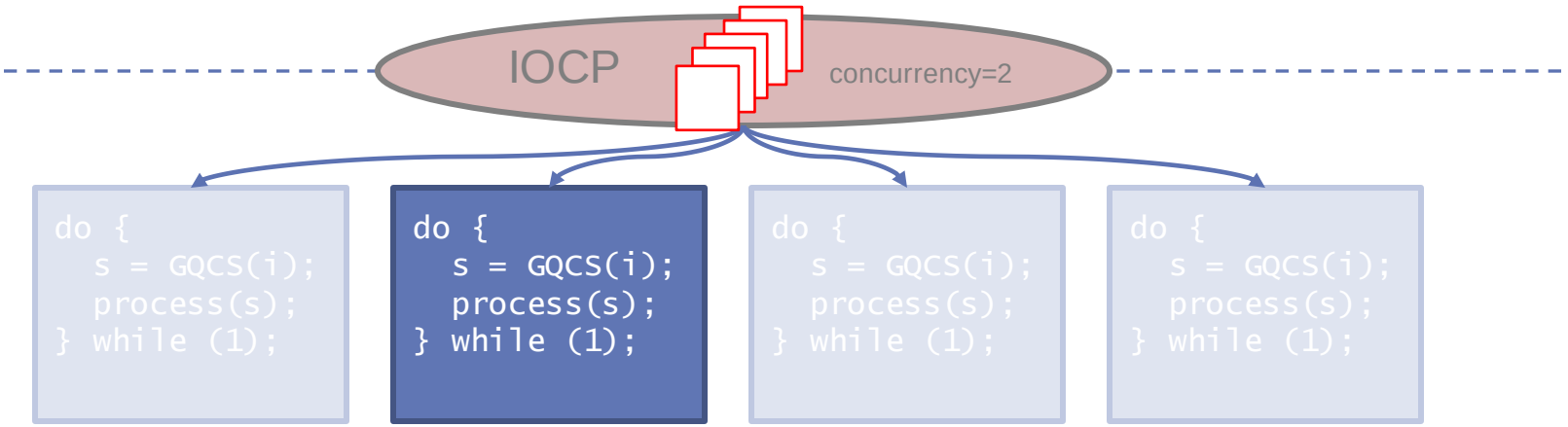
- Set I/O completion port's concurrency to ncpu
- Create ncpu * 2 threads



- An active thread does something that blocks (i.e. file I/O)

IOCP and Concurrency

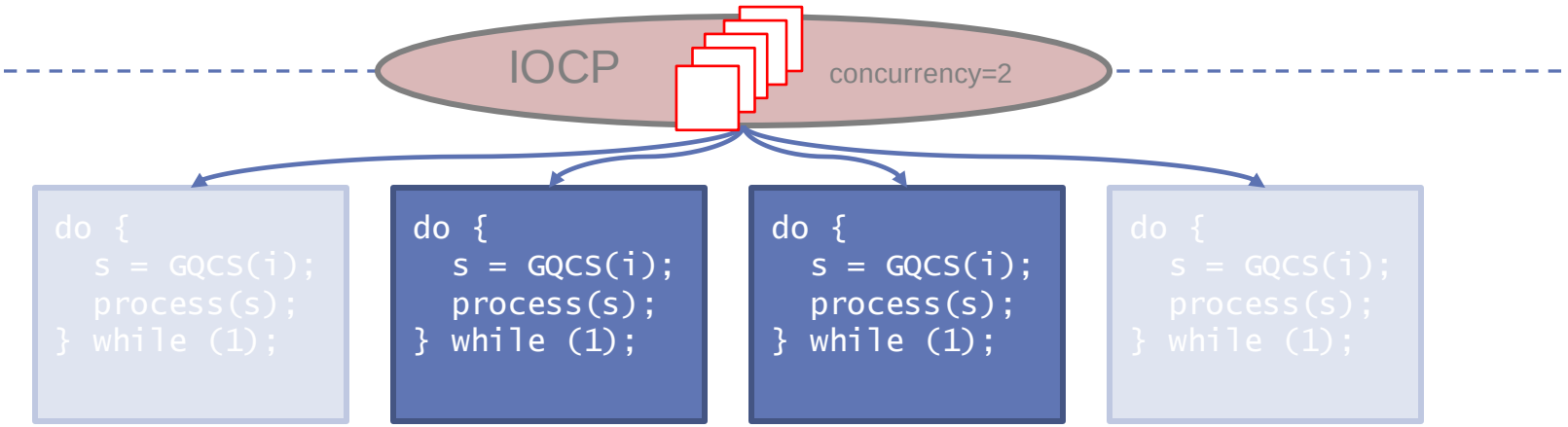
- Set I/O completion port's concurrency to ncpu
- Create ncpu * 2 threads



- An active thread does something that blocks (i.e. file I/O)
- Windows can detect that the active thread count (1) has dropped below max concurrency (2) and that there are still outstanding packets in the completion queue

IOCP and Concurrency

- Set I/O completion port's concurrency to ncpu
- Create ncpu * 2 threads



- An active thread does something that blocks (i.e. file I/O)
- Windows can detect that the active thread count (1) has dropped below max concurrency (2) and that there are still outstanding packets in the completion queue
- ….and schedules another thread to run

IOCP and Concurrency

- ....although just because you can block, doesn't mean you should!
- On Windows, everything can be done asynchronously
- ....so there's no excuse for blocking!
- (Except for a low-latency corner case I'll discuss later)

More cool IOCP stuff: thread affinity

- HTTP Server
  - Short-lived requests
  - Stateless
- Let's say you have 64 cores (thus, 64 active threads), and infinite incoming load
- No thread is going to be better than the other at serving a given request
- Thus, one I/O completion port is sufficient

More cool IOCP stuff: thread affinity

- What about P2P protocols?
- One I/O completion port
  - Tick 1: thread A processes client X, thread B processes client Y
  - Tick 2: thread A processes client Y, thread B processes client X
- Thread A has the benefit of memory/cache locality when processing back-to-back requests from client X
- For protocols where low-latency/high-throughput is paramount, threads should always serve the same clients
- Solution:
  - Create one I/O completion port per core (concurrency = 1)
  - Create 2 threads per completion port
  - Bind threads to core via thread affinity
- Very important in minimizing cache-coherency traffic between CPU cores

Cheating with PyParallel

- Vista introduced new thread pool APIs
- Tightly integrated into IOCP/overlapped ecosystem
- Greatly reduces the amount of scaffolding code I needed to write to prototype the concept

```
void PxSocketClient_Callback();
CreateThreadpoolIo(.., &PxSocketClient_Callback)
..
StartThreadpoolIo(..)
AcceptEx(..)/WSASend(..)/WSARecv(..)
```

- That's it. When the async I/O op completes, your callback gets invoked
- Windows manages everything: optimal thread pool size, NUMA-cognizant dispatching
- Didn't need to create a single thread, no mutexes, none of the normal headaches that come with multithreading

Tying it altogether

...

and leveraging backwards synergy overflow

-Liz Lemon, 2009

Thread waits on completion port …invokes our callback (process(s))

```
do {
  s = GetQueuedCompletionStatus();
  process(s);
} while (1);
```

We do some prep, then call the money maker: PxSocket_IOLoop

```
do {
}
```

```
void
NTAPI
PxSocketClient_Callback(
    PTP_CALLBACK_INSTANCE instance,
    void *context,
    void *overlapped,
    ULONG io_result,
    ULONG_PTR nbytes,
    TP_IO *tp_io
)
{
    Context *c = (Context *)context;
    PxSocket *s = (PxSocket *)c->io_obj;

    EnterCriticalSection(&(s->cs));

    ENTERED_IO_CALLBACK();

    PxSocket_IOLoop(s);

    LeaveCriticalSection(&(s->cs));
}
```

Our thread I/O loop figures out what to do based on a) the protocol we provided, and b) what just happened

```
do {
    void
    NTAPI
    PxSock
        PT
        vo
        vo
        UL
        UL
        TP
    )
    {
        Co
        Px

        En

        EN

        Px

        Le
    }
} while (1);
```

```
PxSocket_IOLoop()
{
    ...
    send_initial_bytes = (
        is_new_connection and
        hasattr(
            protocol,
            'initial_bytes_to_send',
        )
    )
}
```

CONTINUUM ANALYTICS

And then calls into our protocol (via PyObject_CallObject)

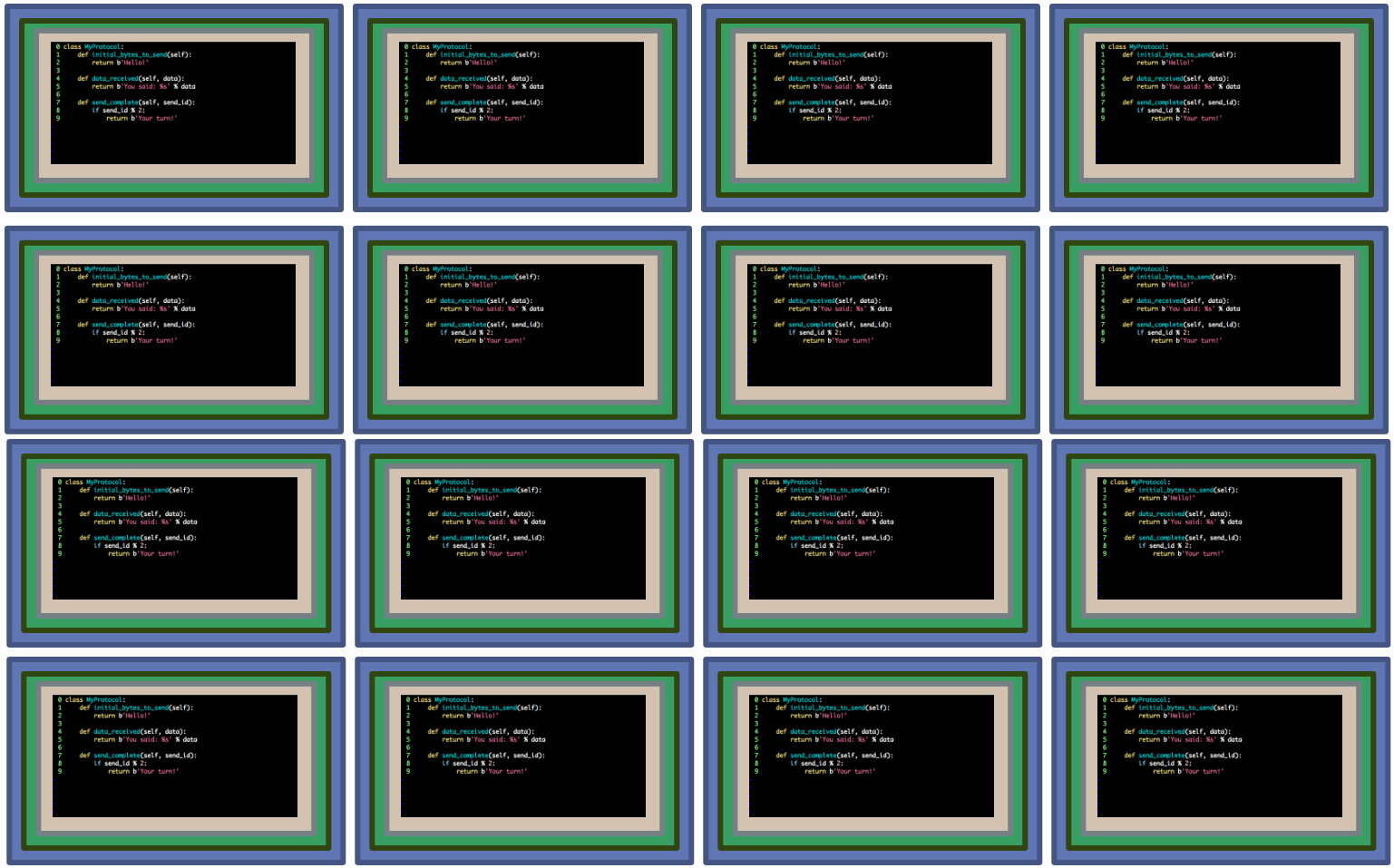
```
do {
    void
    NTAPI
    PxSock
        PT
        vo
        vo
        UL
        UL
        TP
    )
    {
        Co
        Px

        En

        EN

        Px

        Le
    }
} while (0);
```

PxSocket_IOLoop()

```
{
}
```

```
0 class MyProtocol:
1     def initial_bytes_to_send(self):
2         return b'Hello!'
3
4     def data_received(self, data):
5         return b'You said: %s' % data
6
7     def send_complete(self, send_id):
8         if send_id % 2:
9             return b'Your turn!'
```

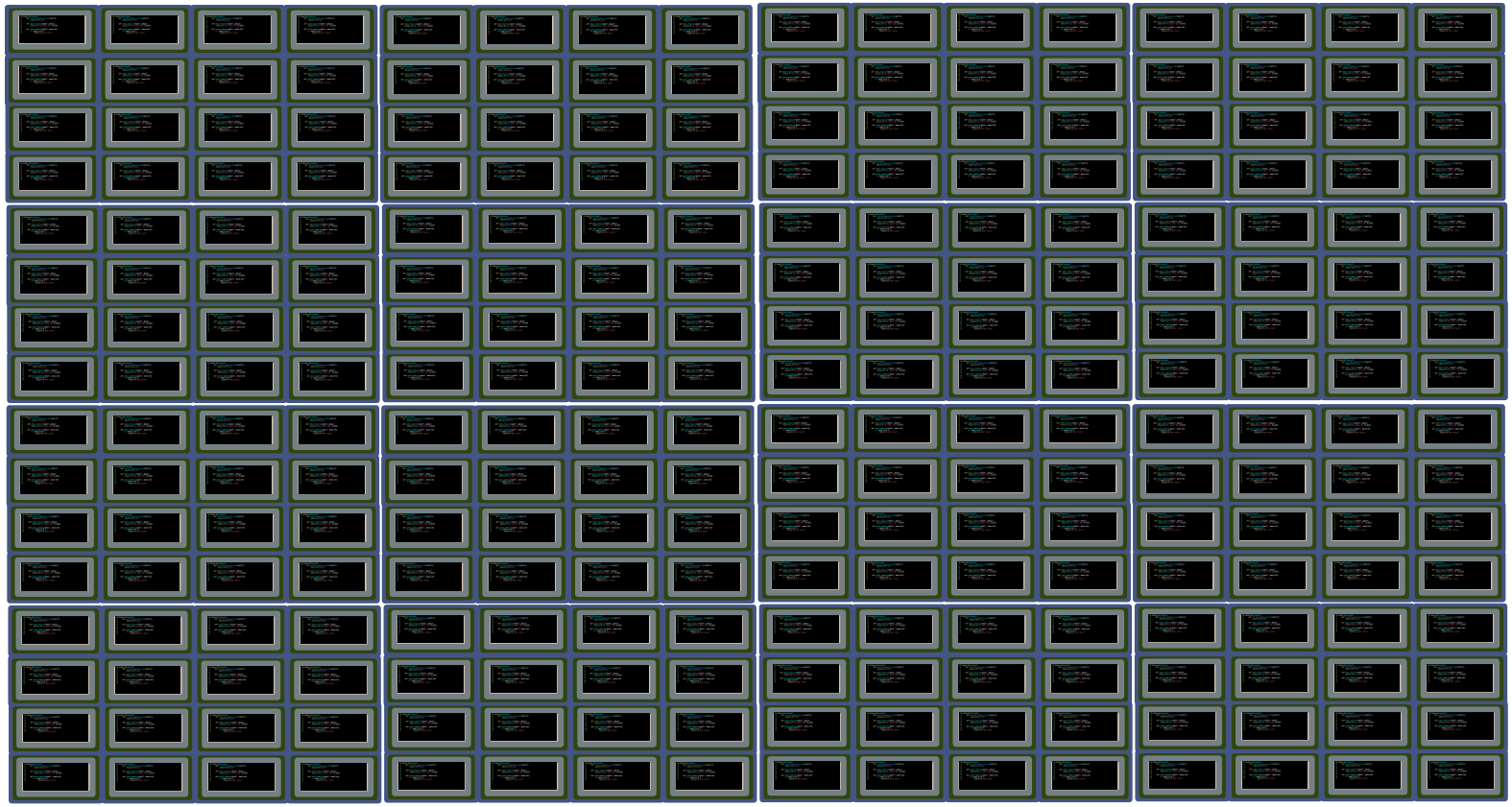
Now times that by ncpu…



….and it should start to become obvious…

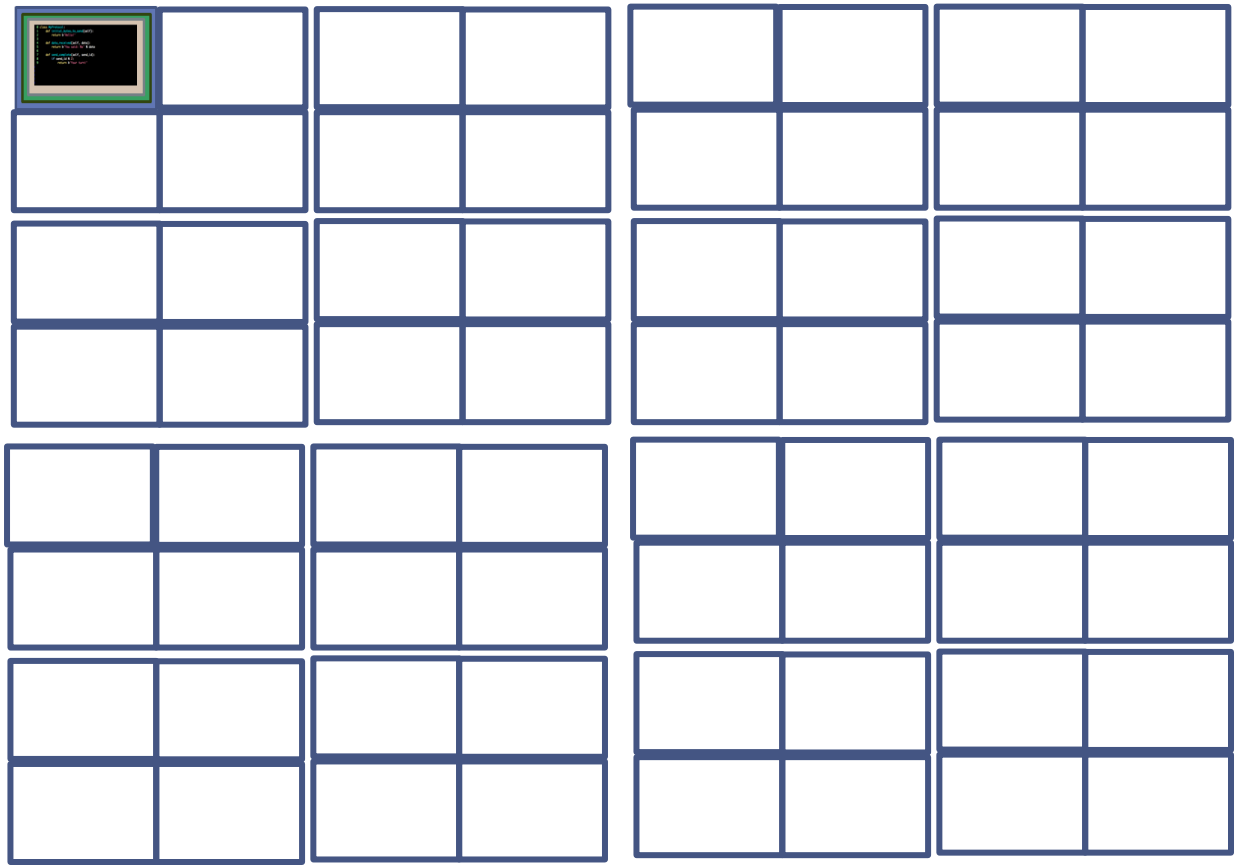
CONTINUUM ANALYTICS

….why it's a better solution…

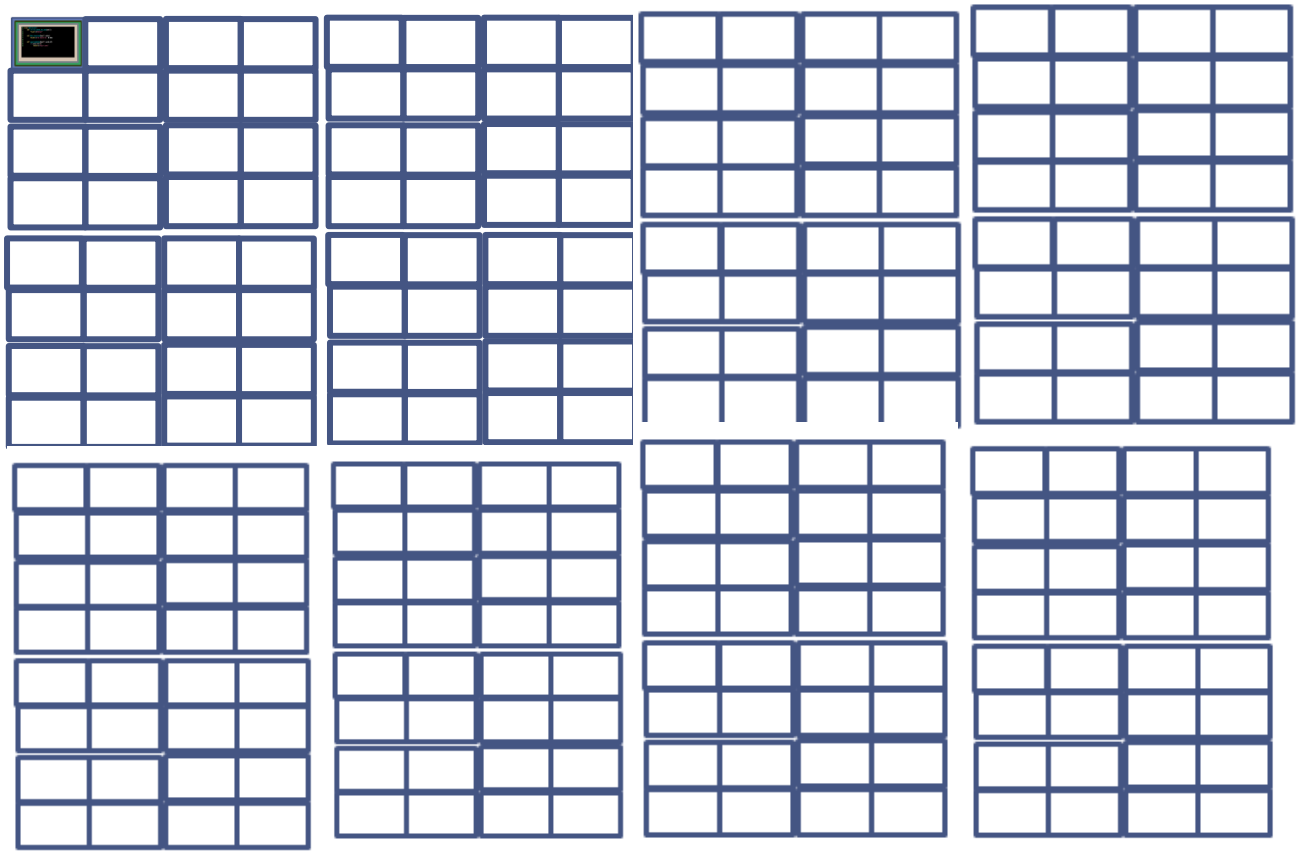


CONTINUUM ANALYTICS

….than the defacto way of doing async I/O in the past…



....via single-threaded, non-blocking, synchronous I/O



"Ahh, so that's why you did it like that!"

But the CPython interpreter isn't thread safe!

The GIL! The GIL!

…

Part 2

...

Removing the GIL

...

(Without needing to remove the GIL.)

So how does it work?

- First, how it doesn't work:
  - No GIL removal
    - This was previously tried and rejected
    - Required fine-grained locking throughout the interpreter
    - Mutexes are expensive
    - Single-threaded execution significantly slower
  - Not using PyPy's approach via Software Transactional Memory (STM)
    - Huge overhead
    - 64 threads trying to write to something, 1 wins, continues
    - 63 keep trying
    - *63 bottles of beer on the wall…*
- Doesn't support "free threading"
  - Existing code using threading.Thread won't magically run on all cores
  - You need to use the new async APIs

PyParallel Key Concepts

- Main-thread
  - Main-thread objects
  - Main-thread execution
  - In comparison to existing Python: the thing that runs when the GIL is held
  - Only runs when parallel contexts aren't executing
- Parallel contexts
  - Created in the main-thread
  - Only run when the main-thread isn't running
  - Read-only visibility to the global namespace established in the main-thread
- Common phrases:
    - *"Is this a main thread object?"*
    - *"Are we running in a parallel context?"*
    - *"Was this object created from a parallel context?"*

I'll explain the purple text later.

Simple Example

```
import async
a = 1
def work():
      b = a * 2
async.submit_work(work)
async.run()
```

- async.submit_work()
  - Creates a new parallel context for the `work` callback
- async.run()
  - Main-thread suspends
  - Parallel contexts allowed to run
  - Automatically executed across all cores (when sufficient work permits)
  - When all parallel contexts complete, main thread resumes, async.run() returns
- 'a' = main thread object
- 'b = a * 2'
  - Executed from a parallel context
  - 'b' = parallel context object

Parallel Contexts

- Parallel contexts are executed by separate threads
- Multiple parallel contexts can run concurrently on separate cores
- Windows takes care of all the thread stuff for us
  - Thread pool creation
  - Dynamically adjust number of threads based on load and physical cores
  - Cache/NUMA-friendly thread scheduling/dispatching
- Parallel threads execute the same interpreter, same ceval loop, same view of memory as the main thread etc
- (No IPC overhead as with multiprocessing)

But the CPython interpreter isn't thread safe!

- Global statics used frequently (free lists)
- Reference counting isn't atomic
- Objects aren't protected by locks
- Garbage collection definitely isn't thread safe
  - You can't have one thread performing a GC run, deallocating objects, whilst another thread attempts to access said objects concurrently
- Creation of interned strings isn't thread safe
- Bucket memory allocator isn't thread safe
- Arena memory allocator isn't thread safe

Concurrent Interpreter Threads

- Basically, every part of the CPython interpreter assumes it's the only thread running (if it has the GIL held)
- The only possible way of allowing multiple threads to run the same interpreter concurrently would be to add fine-grained locking to all of the above
- This is what Greg Stein did ~13 years ago
  - Introduced fine-grained locks in lieu of a Global Interpreter Lock
  - Locking/unlocking introduced huge overhead
  - Single-threaded code 40% slower

PyParallel's Approach

- Don't touch the GIL
  - It's great, serves a very useful purpose
- Instead, intercept all thread-sensitive calls:
  - Reference counting
    - Py_INCREF/DECREF/CLEAR
  - Memory management
    - PyMem_Malloc/Free
    - PyObject_INIT/NEW
  - Free lists
  - Static C globals
  - Interned strings
- If we're the main thread, do what we normally do
- However, *if we're a parallel thread, do a thread-safe alternative*

Main thread or Parallel Thread?

- "If we're a parallel thread, do X, if not, do Y"
  - X = thread-safe alternative
  - Y = what we normally do
- **"If we're a parallel thread"**
  - Thread-sensitive calls are ubiquitous
  - But we want to have a negligible performance impact
  - So the challenge is how quickly can we detect if we're a parallel thread
  - The quicker we can detect it, the less overhead incurred

The Py_PXCTX macro

"Are we running in a parallel context?"

```
#define Py_PXCTX (Py_MainThreadId != _Py_get_current_thread_id())
```

- What's so special about _Py_get_current_thread_id()?
  - On Windows, you could use GetCurrentThreadId()
  - On POSIX, pthread_self()
- Unnecessary overhead (this macro will be everywhere)
- Is there a quicker way?
- Can we determine if we're running in a parallel context without needing a function call?

Windows Solution: Interrogate the TEB

```
#ifdef WITH_INTRINSICS
#   ifdef MS_WINDOWS
#       include <intrin.h>
#       if defined(MS_WIN64)
#           pragma intrinsic(__readgsdword)
#           define _Py_get_current_process_id() (__readgsdword(0x40))
#           define _Py_get_current_thread_id()  (__readgsdword(0x48))
#       elif defined(MS_WIN32)
#           pragma intrinsic(__readfsdword)
#           define _Py_get_current_process_id() __readfsdword(0x20)
#           define _Py_get_current_thread_id()  __readfsdword(0x24)
```

Py_PXCTX Example

```
-#define _Py_ForgetReference(op) _Py_INC_TPFREES(op)
+#define _Py_ForgetReference(op)                         \
+    do {                                                \
+        if (Py_PXCTX)                                   \
+            _Px_ForgetReference(op);                    \
+        else                                            \
+            _Py_INC_TPFREES(op);                        \
+    } while (0)
+
+#endif /* WITH_PARALLEL */
```

- `Py_PXCTX == (Py_MainThreadId == __readfsdword(0x48))`
- Overhead reduced to a couple more instructions and an extra branch (cost of which can be eliminated by branch prediction)
- That's basically free compared to STM or fine-grained locking

PyParallel Advantages

- Initial profiling results: 0.01% overhead incurred by Py_PXCTX for normal single-threaded code
  - GIL removal: 40% overhead
  - PyPy's STM: "200-500% slower"
- Only touches a relatively small amount of code
  - No need for intrusive surgery like re-writing a thread-safe bucket memory allocator or garbage collector
- Keeps GIL semantics
  - Important for legacy code
  - 3rd party libraries, C extension code
- Code executing in parallel context has full visibility to "main thread objects" (in a read-only capacity, thus no need for locks)
- Parallel contexts are intended to be shared-nothing
  - Full isolation from other contexts
  - No need for locking/mutexes

"If we're a parallel thread, do X" X = thread-safe alternatives

- First step was attacking memory allocation
  - Parallel contexts have localized heaps
  - PyMem_MALLOC, PyObject_NEW etc all get returned memory backed by this heap
  - Simple block allocator
    - Blocks of page-sized memory allocated at a time (4k or 2MB)
    - Request for 52 bytes? Current pointer address returned, then advanced 52 bytes
    - Cognizant of alignment requirements
- What about memory deallocation?
  - Didn't want to write a thread-safe garbage collector
  - Or thread-safe reference counting mechanisms
  - And our heap allocator just advances a pointer along in blocks of 4096 bytes
  - Great for fast allocation
  - Pretty useless when you need to deallocate

Memory Deallocation within Parallel Contexts

- The allocations of page-sized blocks are done from a single heap
  - Allocated via HeapAlloc()
- These parallel contexts aren't intended to be long-running bits of code/algorithm
- Let's not free() anything…
- ….and just blow away the entire heap via HeapFree() with one call, once the context has finished

Deferred Memory Deallocation

- Pros:
  - Simple (even more simple than the allocator)
  - Good fit for the intent of parallel context callbacks
    - Execution of stateless Python code
    - No mutation of shared state
    - The lifetime of objects created during the parallel context is limited to the duration of that context
- Cons:
  - You technically couldn't do this:

```
def work():
    for x in xrange(0, 1000000000):
        …
```

  - (Why would you!)

Reference Counting

- Why do we reference count in the first place?
- Because the memory for objects is released when the object's reference count goes to 0
- But we release all parallel context memory in one fell swoop once it's completed
- And objects allocated within a parallel context can't "escape" out to the main-thread
  - i.e. appending a string from a parallel context to a list allocated from the main thread
- So… there's no point referencing counting objects allocated within parallel contexts!

Reference Counting (cont.)

- What about reference counting main thread objects we may interact with?
- Well all main thread objects are read-only
- So we can't mutate them in any way
- And the main thread doesn't run whilst parallel threads run
- So we don't need to be worried about main thread objects being garbage collected when we're referencing them
- So… no need for reference counting of main thread objects when accessed within a parallel context!

Garbage Collection

- If we deallocate everything at the end of the parallel context's life
- And we don't do any reference counting anyway
- Then there's no possibility for circular references
- Which means there's no need for garbage collection!
- ….things just got a whole lot easier!

Python code executing in parallel contexts…

- Memory allocation is incredibly simple
  - Bump a pointer
  - (Occasionally grab another page-sized block when we run out)
- Simple = fast
- Memory deallocation is done via one call: HeapFree()
- No reference counting necessary
- No garbage collection necessary
- Negligible overhead from the Py_PXCTX macro
- End result: Python code actually executes faster within parallel contexts than main-thread code
- ….and can run concurrently across all cores, too!

Asynchronous Socket I/O

- The main catalyst for this work was allow the callbacks for completion-oriented protocols to execute concurrently

```
import async
class Disconnect: pass
server = async.server('localhost', 8080)
async.register(transport=server, protocol=Disconnect)
async.run()
```

- Let's review some actual protocol examples
  - Keep in mind that all callbacks are executed in parallel contexts
  - If you have 8 cores and sufficient load, all 8 cores will be saturated
- We use AcceptEx to pre-allocate sockets ahead of time
  - Reduces initial connection latency
  - Allows use of IOCP and thread pool callbacks to service new connections
  - Not subject to serialization limits of accept() on POSIX
- And WSAAsyncSelect(FD_ACCEPT) to notify us when we need to pre-allocate more sockets

Completion-oriented Protocols

Examples of common TCP/IP services in PyParallel

```
 0 class Disconnect:
 1     """
 2     Disconnects a connection as soon as it is established.
 3     """
 4     pass
 5
 6 class Discard:
 7     """
 8     Null sink; discards all incoming data, never disconnects.
 9     """
10     def data_received(self, data):
11         pass
12
13 class QOTD:
14     """
15     Prints a quote of the day, then disconnects.
16     """
17     initial_bytes_to_send = b'An apple a day keeps the doctor away.\r\n.'
18
```

Completion-oriented Protocols

Examples of common TCP/IP services in PyParallel

```
 0
 1 import time
 2 class Daytime:
 3     """
 4     Send a string representation of the current time, then disconnect.
 5     """
 6     def initial_bytes_to_send(self):
 7         return time.ctime() + '\r\n'
 8
 9
10 import socket
11 from datetime import datetime
12 class Time:
13     """
14     Send a 32-bit unsigned integer in binary format and network byte
15     order, representing the number of seconds since 00:00 (midnight)
16     January 1st, 1900 GMT, then close the connection.
17     """
18     def initial_bytes_to_send(self):
19         delta = datetime.utcnow() - datetime(1900, 1, 1)
20         return socket.htonl(int(delta.total_seconds()))
21
```

Short-lived Protocols

- Previous examples all disconnect shortly after the client connects
- Perfect for our parallel contexts
  - All memory is deallocated when the client disconnects
- What about long-lived protocols?

Long-lived Protocols

```
0
1  class EchoData:
2      """
3      Echo service; return (write) all data received back to the sender.
4      (The data_received() callback will be invoked as soon as any data
5      is received from the sender.)
6      """
7      def data_received(self, data):
8          return data
9
10 class EchoLine:
11     """
12     Echo service; return (write) all lines received back to the sender.
13     (The lines_received() callback will be invoked whenever at least one
14     line is received.  ``lines`` will always be a tuple (and thus, iterable)
15     even if only one line has been received.  This allows the callback to
16     peek ahead at any and all lines.)
17     """
18     line_mode = True
19     def lines_received(self, lines):
20         return os.linesep.join(lines)
21
```

Long-lived Protocols

```
 0 def chargen(lineno, nchars=72):
 1     start = ord(' ')
 2     end = ord('~')
 3     c = lineno + start
 4     while c > end:
 5         c = (c % end) + start
 6     b = bytearray(nchars)
 7     for i in range(0, nchars-2):
 8         if c > end:
 9             c = start
10         b[i] = c
11         c += 1
12
13     b[nchars-1] = ord('\n')
14
15     return b
```

```
 0 class Chargen:
 1     """
 2     Character generator protocol (RFC 864).
 3     """
 4     def initial_bytes_to_send(self):
 5         return chargen(0)
 6
 7     def send_complete(self, transport, send_id):
 8         return chargen(send_id)
 9
10
11
12
13
14
15
```

Long-lived Protocols

```
 0 def chargen(lineno, nchars=72):
 1     start = ord(' ')
 2     end = ord('~')
 3     c = lineno + start
 4     while c > end:
 5         c = (c % end) + start
 6     b = bytearray(nchars)
 7     for i in range(0, nchars-2):
 8         if c > end:
 9             c = start
10         b[i] = c
11         c += 1
12
13     b[nchars-1] = ord('\n')
14
15     return b
```

```
Terminal — zsh — 80×24
~
~
(trent@xenon:ttys000) (Sun/11:39) ..                    (~)
% python3.3 chargen.py
b' !"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcde\x00\n
\n!"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdef\x00\n
\n"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefg\x00\n
\n#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefgh\x00\n
\n$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghi\x00\n
\n%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghij\x00\n
\n&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijk\x00\n
\n\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijkl\x00\n
\n()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklm\x00\n\
n)*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmn\x00\n\n
*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmno\x00\n\n+
,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnop\x00\n\n,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopq\x00\n\n-./
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqr\x00\n\n./01
23456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrs\x00\n\n/0123
456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrst\x00\n\n012345
6789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstu\x00\n\n1234567
89:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuv\x00\n'
(trent@xenon:ttys000) (Sun/11:39) ..                    (~)
%
```

CONTINUUM ANALYTICS

Long-lived Protocols

- Clients could stay connected indefinitely
- Each time a callback is run, memory is allocated
- Memory is only freed when the context is finished
- Contexts are considered finished when the client disconnects
- ….that's not a great combo

Tweaking the memory allocator

- The simple block allocator had served us so well until this point!
- Long-running contexts looked to unravel everything
- The solution: heap snapshots

Heap Snapshots

- Before PyParallel invokes the callback
  - (Via PyObject_CallObject)
- It takes a “heap snapshot”
- Each snapshot is paired with a corresponding “heap rollback”
- Can be nested (up to 64 times):

```
snapshot1 = heap_snapshot()
    snapshot2 = heap_snapshot()
    # do work
    heap_rollback(snapshot2)
heap_rollback(snapshot1)
```

Heap Snapshots

- Tightly integrated with PyParallel's async I/O socket machinery
- A rollback simply rolls the pointers back in the heap to where they were before the callback was invoked
- Side effect: very cache and TLB friendly
  - Two invocations of data_received(), back to back, essentially get identical memory addresses
  - All memory addresses will already be in the cache
  - And if not, they'll at least be in the TLB (a TLB miss can be just as expensive as a cache miss)

Latency vs Concurrency vs Throughput

- Different applications have different performance requirements/preferences:
  - Low latency preferred
  - High concurrency preferred
  - High throughput preferred
- What control do we have over latency, concurrency and throughput?
- Asynchronous versus synchronous:
  - An async call has higher overhead compared to a synchronous call
    - IOCP involved
    - Thread dispatching upon completion
  - If you can perform a synchronous send/recv at the time, without blocking, that will be faster
- How do you decide when to do sync versus async?

Dynamically switching between synchronous and asynchronous I/O

• • •

Chargen: a case study

Chargen: the I/O hog

```
0 class Chargen:
1     """
2     Character generator protocol (RFC 864).
3     """
4     def initial_bytes_to_send(self):
5         return chargen(0)
6
7     def send_complete(self, transport, send_id):
8         return chargen(send_id)
9
10
11
12
13
14
15
```

- Sends a line as soon as a connection is made
- Sends a line as soon as that line has sent
- ….sends a line as soon as that next line has sent
- ….and so on
- Always wants to send something
- PyParallel term for this: I/O hog

PyParallel's Dynamic I/O Loop

- Initially, separate methods were implemented for PxSocket_Send, PxSocket_Recv
- Chargen forced a rethink
- If we have four cores, but only one client connected, there's no need to do async sends
  - A synchronous send is more efficient
  - Affords lower latency, higher throughput
- But chargen always wants to do another send when the last send completed
- If we're doing a synchronous send from within PxSocket_Send… doing another send will result in a recursive call to PxSocket_Send again
- Won't take long before we exhaust our stack

PxSocket_IOLoop

- Similar idea to the ceval loop
- A single method that has all possible socket functionality inlined
- Single function = single stack = no stack exhaustion
- Allows us to dynamically choose optimal I/O method (sync vs async) at runtime

PxSocket_IOLoop

- If active client count < available CPU cores-1: try sync first, fallback to async after X sync EWOULDBLOCKs
  - Reduced latency
  - Higher throughput
  - Reduced concurrency
- If active client count >= available CPU cores-1: immediately do async
  - Increased latency
  - Lower throughput
  - Better concurrency

- (I'm using "better concurrency" here to mean "more able to provide a balanced level of service to a greater number of clients simultaneously")

PxSocket_IOLoop

- We also detect how many active I/O hogs there are (globally), and whether this protocol is an I/O hog, and factor that into the decision
- Protocols can also provide a hint:

```
class HttpServer:
    concurrency = True
class FtpServer:
    throughput = True
```

A note on sending…

```
class Chargen:
    """
    Character generator protocol (RFC 864).
    """
    def initial_bytes_to_send(self):
        return chargen(0)

    def send_complete(self, transport, send_id):
        return chargen(send_id)
```

```
class MyProtocol:
    def initial_bytes_to_send(self):
        return b'Hello!'

    def data_received(self, data):
        return b'You said: %s' % data

    def send_complete(self, transport, send_id):
        if send_id % 2:
            return b'Your turn!'
```

- Note the absence of an explicit send/write, i.e.
  - No transport.write(data) like with Tulip/Twisted
- You “send” by returning a “sendable” Python object from the callback
  - PyBytesObject
  - PyByteArray
  - PyUnicode
- Supporting only these types allow for a cheeky optimisation:
  - The WSABUF’s len and buf members are pointed to the relevant fields of the above types; no copying into a separate buffer needs to take place

No explicit transport.send(data)?

- Forces you to construct all your data at once (not a bad thing), not trickle it out through multiple write()/flush() calls
- Forces you to leverage send_complete() if you want to send data back-to-back (like chargen)
- send_complete() clarification:
  - What it **doesn't** mean: other side got it
  - What it **does** mean: send buffer is empty (became bytes on a wire)
  - What it **implies**: you're free to send more data if you've got it, it won't block

Nice side-effects of no explicit transport.send()

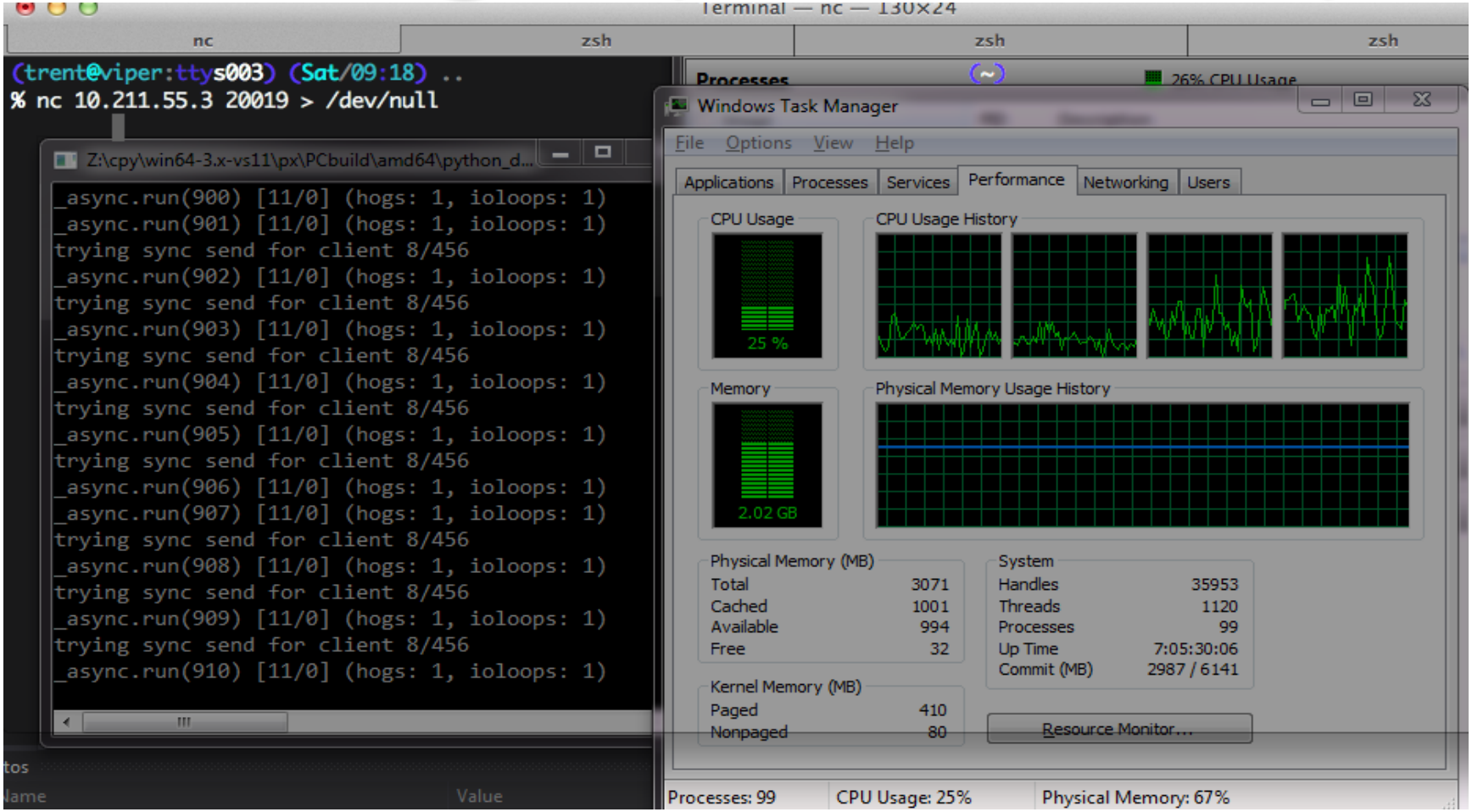
- No need to buffer anything internally
- No need for producer/consumer relationships like in Twisted/Tulip
  - pause_producing()/stop_consuming()
- No need to deal with buffer overflows when you're trying to send lots of data to a slow client – the protocol essentially buffers itself automatically
- Keeps a tight rein on memory use
- Will automatically trickle bytes over a link, to completely saturating it

PyParallel In Action

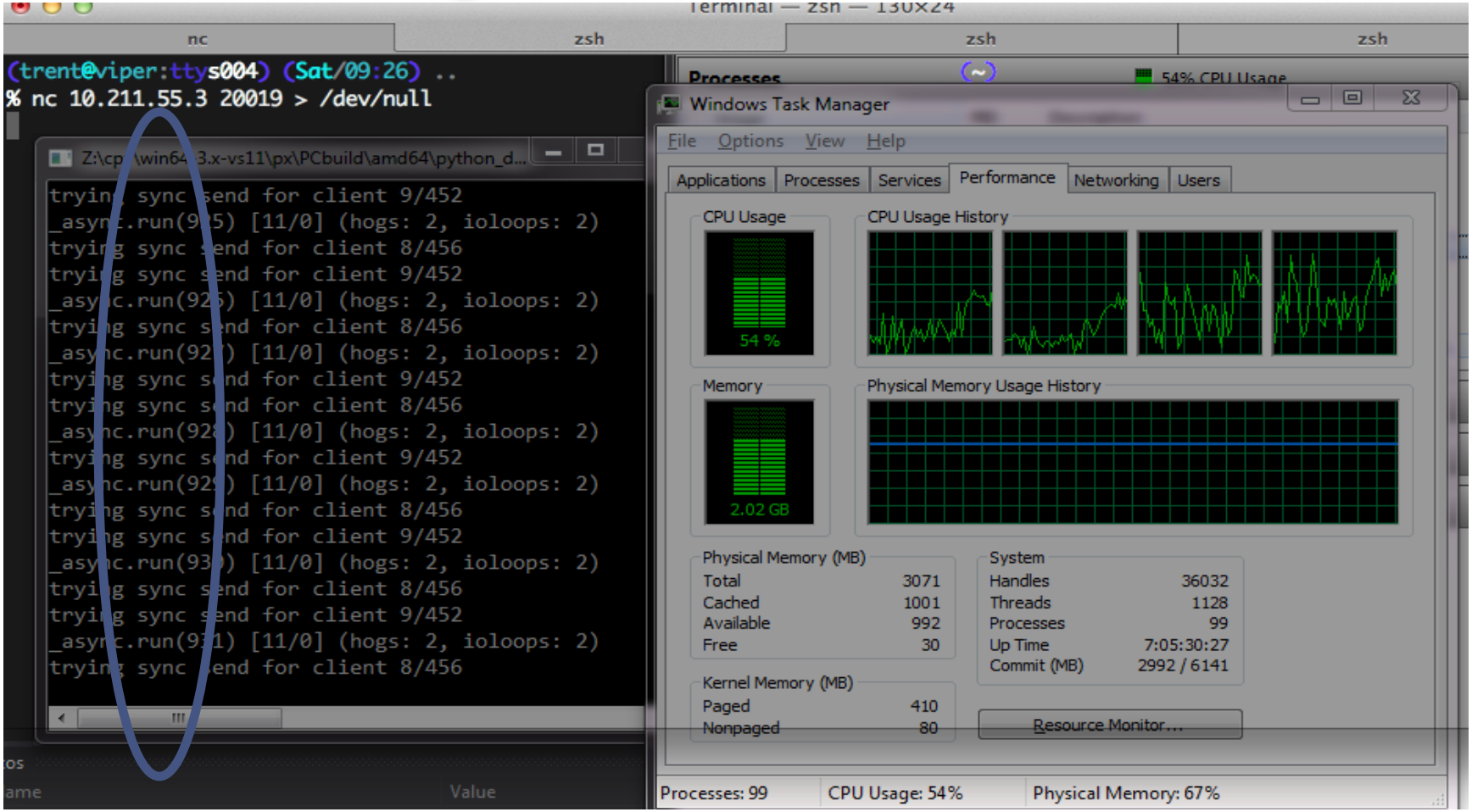
- Things to note with the chargen demo coming up:
  - One python_d.exe process
  - Constant memory use
  - CPU use proportional to concurrent client count (1 client = 25% CPU use)
  - Every 10,000 sends, a status message is printed
    - Depicts dynamically switching from synchronous sends to async sends
    - Illustrates awareness of active I/O hogs
- Environment:
  - Macbook Pro, 8 core i7 2.2GHz, 8GB RAM
  - 1-5 netcat instances on OS X
  - Windows 7 instance running in Parallels, 4 cores, 3GB

Num. Processes CPU% Mem%

1 Chargen (99/25%/67%)

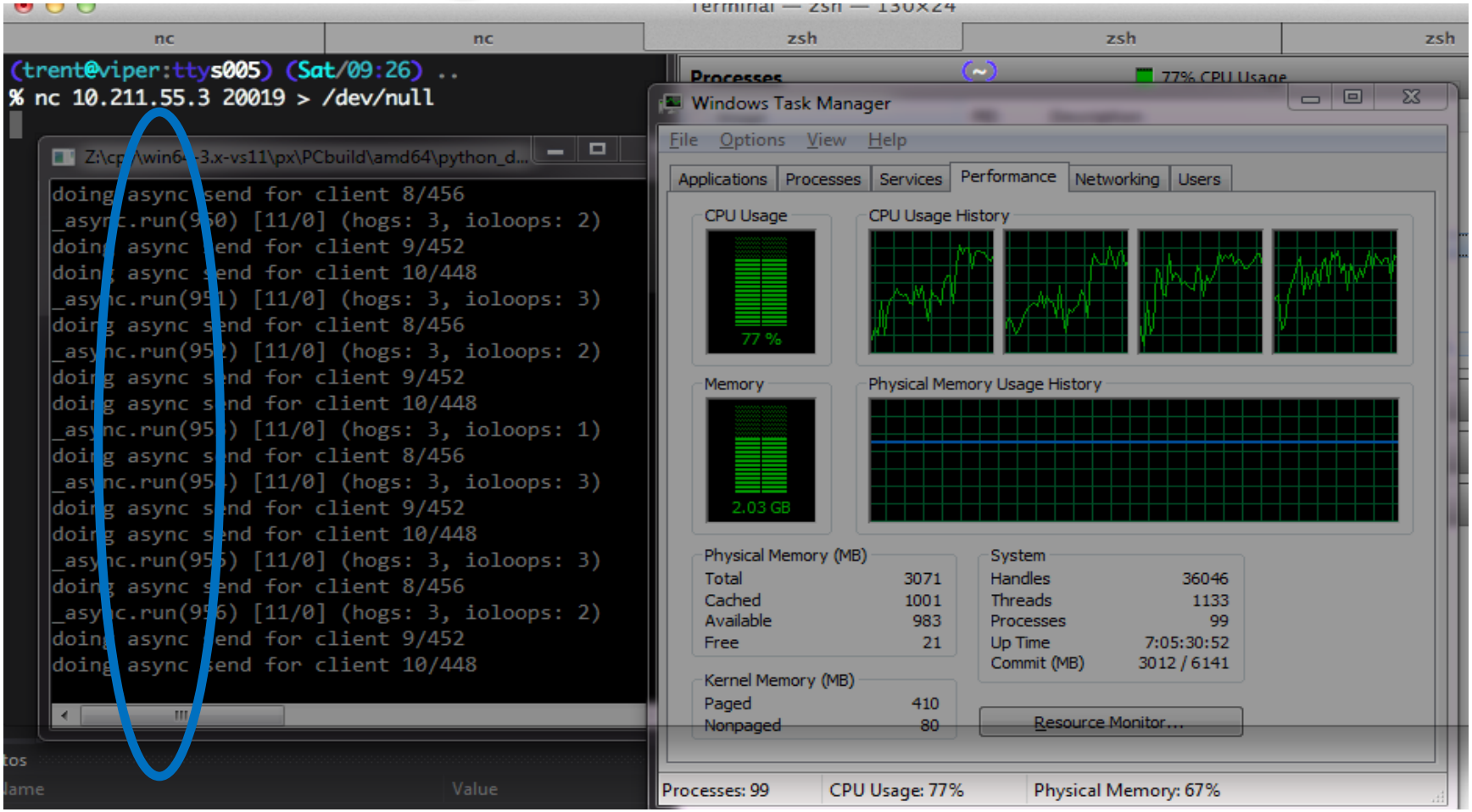


2 Chargen (99/54%/67%)



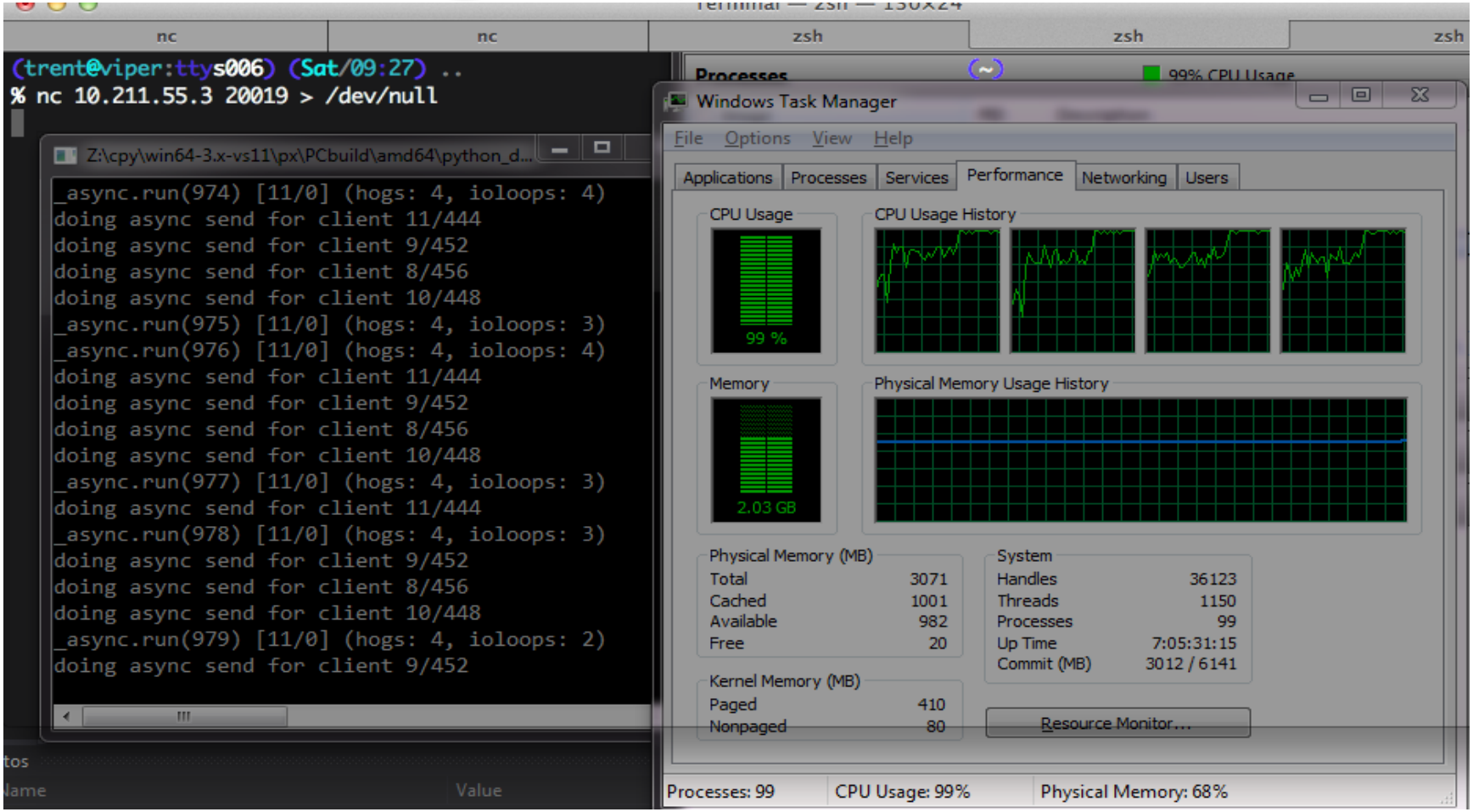
CONTINUUM ANALYTICS

3 Chargen (99/77%/67%)



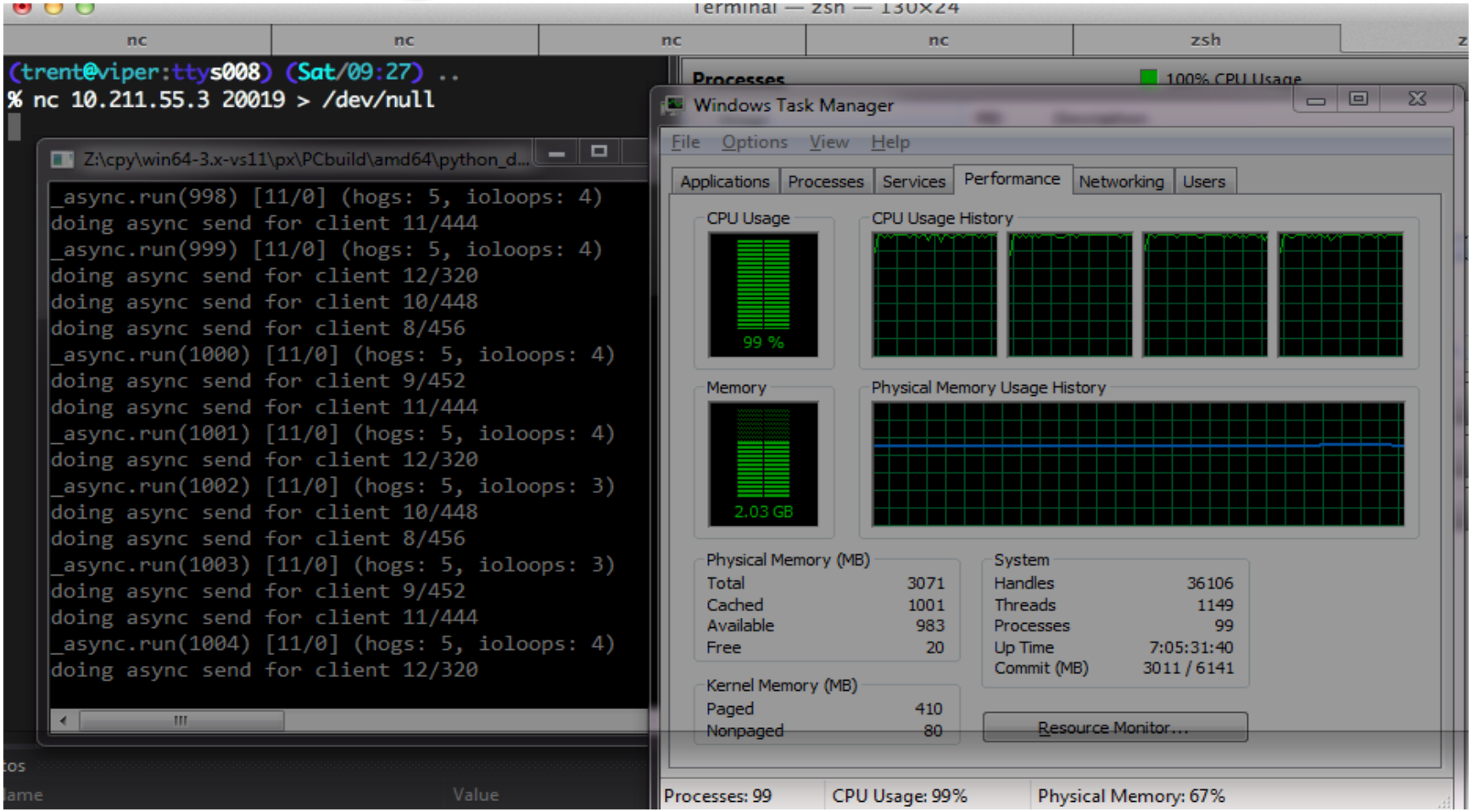
CONTINUUM ANALYTICS

4 Chargen (99/99%/68%)



CONTINUUM ANALYTICS

5 Chargen?! (99/99%/67%)



CONTINUUM ANALYTICS

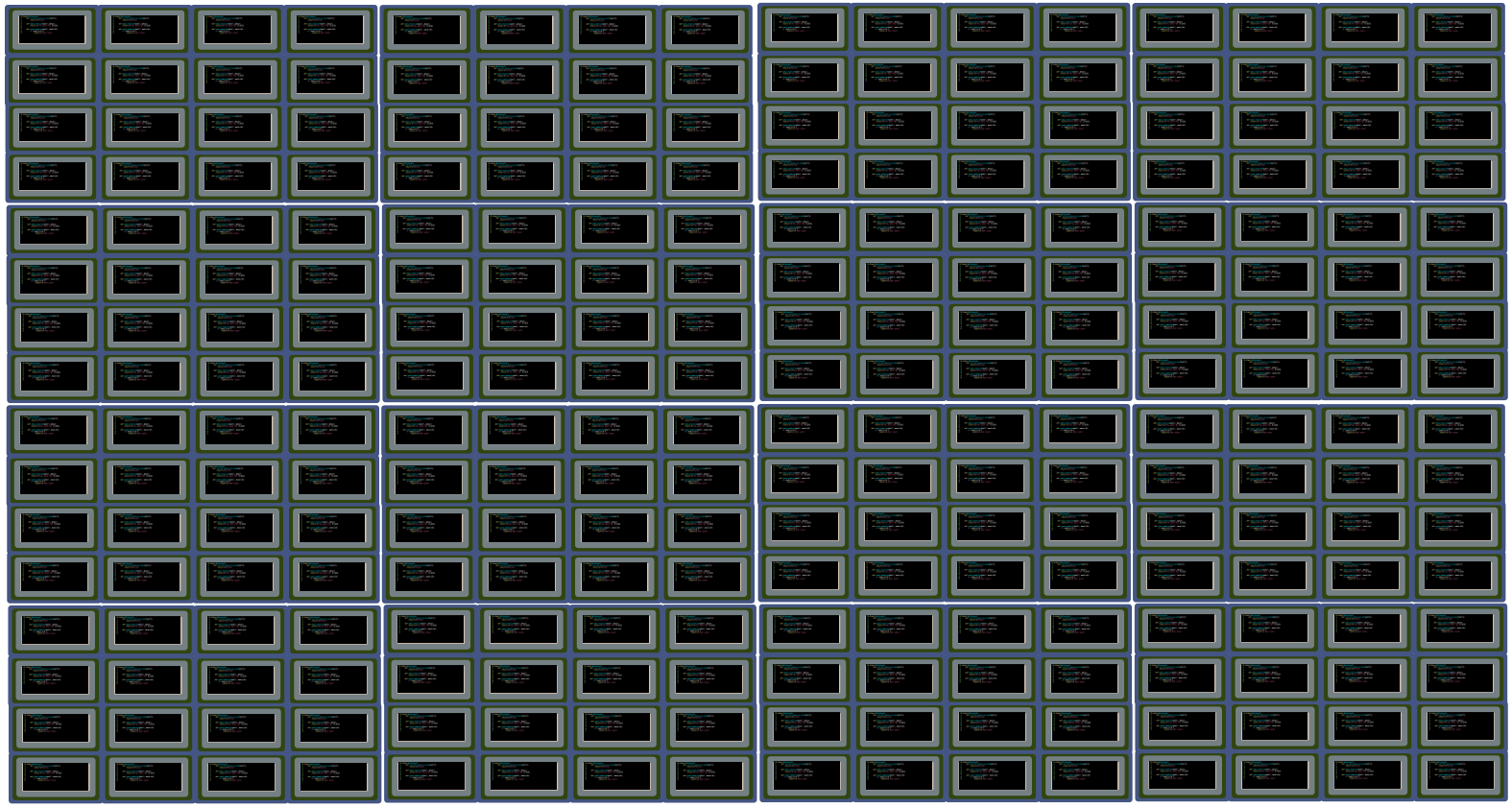
Why chargen turned out to be so instrumental in shaping PyParallel…

- You're only sending 73 bytes at a time
- The CPU time required to generate those 73 bytes is **not** negligible (compared to the cost of sending 73 bytes)
  - Good simulator of real world conditions, where the CPU time to process a client request would dwarf the IO overhead communicating the result back to the client
- With a default send socket buffer size of 8192 bytes and a local netcat client, you're never going to block during send()
- Thus, processing a single request will immediately throw you into a tight back-to-back send/callback loop, with no opportunity to service other clients (when doing synchronous sends)
- Highlighted all sorts of problems I needed to solve before moving on to something more useful: the async HTTP server

PyParallel's async HTTP Server

- async.http.server.HttpServer version of stdlib's SimpleHttpServer.
  http://hg.python.org/sandbox/trent/file/0e70a0caa1c0/Lib/async/http/server.py
- Final piece of the async "proof-of-concept"
- PxSocket_IOLoop modified to optimally support TransmitFile
  - Windows equivalent to POSIX sendfile()
  - Serves file content directly from file system cache, very efficient
  - Tight integration with existing IOCP/threadpool support

So we've now got an async HTTP server, in Python, that scales to however many cores you have



CONTINUUM ANALYTICS

(On Windows. Heh.)

• • •

Thread-local interned strings and heap snapshots

- Async HTTP server work highlighted a flaw in the thread-local redirection of interned strings and heap snapshot/rollback logic
- I had already ensured the static global string intern stuff was being intercepted and redirected to a thread-local equivalent when in a parallel context
- However, string interning involves memory allocation, which was being fulfilled from the heap associated with the active parallel context
- Interned strings persist for the life of the thread, though, parallel context heap allocations got blown away when the client disconnected

Thread-local Heap Overrides

- Luckily, I was able to re-use previously implemented-then-abandoned support for a thread-local heap:

```
PyAPI_FUNC(int)  _PyParallel_IsTLSHeapActive(void);
PyAPI_FUNC(int)  _PyParallel_GetTLSHeapDepth(void);
PyAPI_FUNC(void) _PyParallel_EnableTLSHeap(void);
PyAPI_FUNC(void) _PyParallel_DisableTLSHeap(void);
```

- Prior to interning a string, we check to see if we're a parallel context, if we are, we enable the TLS heap, proceed with string interning, then disable it.
- The parallel context _PyHeap_Malloc() method would divert to a thread-local equivalent if the TLS heap was active
- Ensured that interned strings were always backed by memory that wasn't going to get blown away when a context disappears

A few notes on non-socket I/O related aspects of PyParallel

...

Memory Protection

- How do you handle this:

```
foo = []
def work():
    timestamp = async.rdtsc()
    foo.append(timestamp)
async.submit_work(work)
async.run()
```

- That is, how do you handle either:
  - Mutating a main-thread object from a parallel context
  - Persisting a parallel context object outside the life of the context
- That was a big showstopper for the entire three months
- Came up with numerous solutions that all eventually turned out to have flaws

Memory Protection

- Prior to the current solution, I had all sorts of things in place all over the code base to try and detect/intercept the previous two occurrences
- Had an epiphany shortly after PyCon 2013 (when this work was first presented)
- The solution is deceptively simple:
  - Suspend the main thread before any parallel threads run.
  - Just prior to suspension, write-protect all main thread pages
  - After all the parallel contexts have finished, return the protection to normal, then resume the main thread
- Seems so obvious in retrospect!
- All the previous purple code refers to this work – it's not present in the earlier builds

Memory Protection

- If a parallel context attempts to mutate (write) to a main-thread allocated object, a general protection fault will be issued
- We can trap that via Structured Exception Handlers
  - (Equivalent to a SIGSEV trap on POSIX)
- By placing the SEH trap's __try/__except around the main ceval loop, we can instantly convert the trap into a Python exception, and continue normal execution
  - Normal execution in this case being propagation of the exception back up through the parallel context's stack frames, like any other exception
- Instant protection against all main-thread mutations without needing to instrument *any* of the existing code

Enabling Memory Protection

- Required a few tweaks in obmalloc.c (which essentially calls malloc() for everything)
- For VirtualProtect() calls to work efficiently, we'd need to know the base address ranges of main thread memory allocations
  - This doesn't fit well with using malloc() for everything
  - Every pointer + size would have to be separately tracked and then fed into VirtualProtect() every time we wanted to protect pages
- Memory protection is a non-trivial expense
  - For each address passed in (base + range), OS has to walk all affected page tables and alter protection bits
- I employed two strategies to mitigate overhead:
  - Separate memory allocation into two phases: reservation and commit.
  - Use large pages.

Reserve, then Commit

- Windows allows you to reserve memory separate to committing it
  - (As does UNIX)
- Reserved memory is free; no actual memory is used until you subsequently commit a range (from within the reserved range)
- This allows you to reserve, say, 1GB, which gives you a single base address pointer that covers the entire 1GB range
- ….and only commit a fraction of that initially, say, 256KB
- This allows you to toggle write-protection on all main thread pages via a single call to VirtualProtect() via the base address call
- Added benefit: easily test origin of an object by masking its address against known base addresses

Large Pages

- 2MB for amd64, 4MB for x86 (standard page size for both is 4KB)
- Large pages provide significant performance benefits by minimizing the number of TLB entries required for a process's virtual address space
- Fewer TLB entries per address range = TLB can cover greater address range = better TLB hit ratios = direct impact on performace (TLB misses are very costly)
- Large pages also means the OS has to walk significantly fewer page table entries in response to our VirtualProtect() call

Memory Protection Summary

- Very last change I made to PyParallel just before getting hired by Continuum after PyCon earlier this year
  - I haven't had time to hack on PyParallel since then
- Was made in a proof-of-concept fashion
  - Read: "I butchered the crap out of everything to test it out"
- Lots of potential for future expansion in this area
  - Read: "Like unbutchering everything"

Part 3
The Future

• • •

Various ideas for PyParallel going forward

The Future…

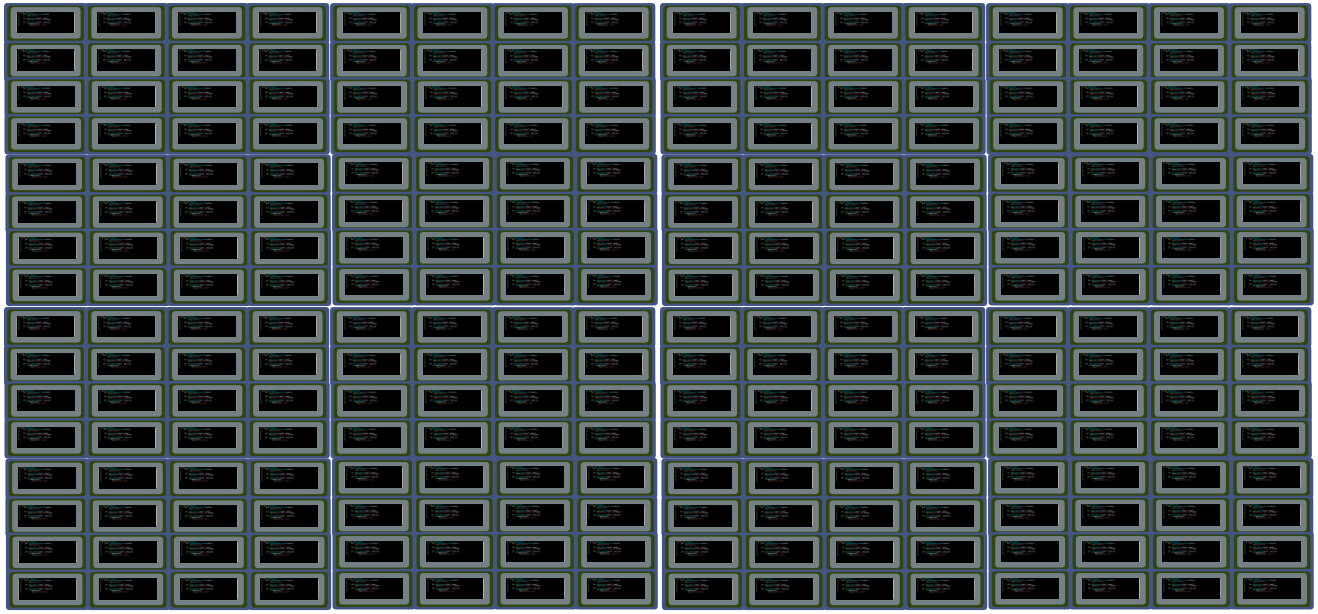
- PyParallel for parallel task decomposition
  - Limitations of the current memory model
  - Ideas for new set of interlocked data types
- Continued work on memory management enhancements
  - Use context managers to switch memory allocation protocols within parallel contexts
  - Rust does something similar in this area
- Integration with Numba
  - Parallel callbacks passed off to Numba asynchronously
  - Numba uses LLVM to generate optimized version
  - PyParallel atomically switches the CPython version with the Numba version when ready

The Future…

- Dynamic PxSocket_IOLoop endpoints
  - Socket source, file destination
  - One socket source, multiple socket destinations (1:m)
  - Provide similar ZeroMQ bridge/fan-out/router functionality
- This would provide a nice short-term option for leveraging PyParallel for computation/parallel task decomposition
  - Bridge different protocols together
  - Each protocol represents a stage in a parallel pipeline
  - Use pipes instead of socket I/O to ensure zero copy where possible
  - No need for synchronization primitives
  - This is how ZeroMQ does “parallel computation”

The Future

- ….lends itself quite nicely to pipeline composition:



- Think of all the ways you could compose things based on your problem domain

The Future…

- PyParallel for UI apps
  - Providing a way for parallel callbacks to efficiently queue UI actions (performed by a single UI thread)
- NUMA-aware memory allocators
- CPU/core-aware thread affinity
- Integrating Windows 8's registered I/O support
- Multiplatform support:
  - MegaPipe for Linux looks promising
  - GCD on OS X/FreeBSD
  - IOCP on AIX
  - Event ports for Solaris

The Future…

- Ideally we'd like to see PyParallel merged back into the CPython tree
  - Although started as a proof-of-concept, I believe it is Python's best option for exploiting multiple cores
  - So it'll probably live as pyparallel.exe for a while (like Stackless)
- I'm going to cry if Python 4.x rolls out in 5 years and I'm still stuck in single-threaded, non-blocking, synchronous I/O land
- David Beazley: "the GIL is something all Python committers should be concerned about"

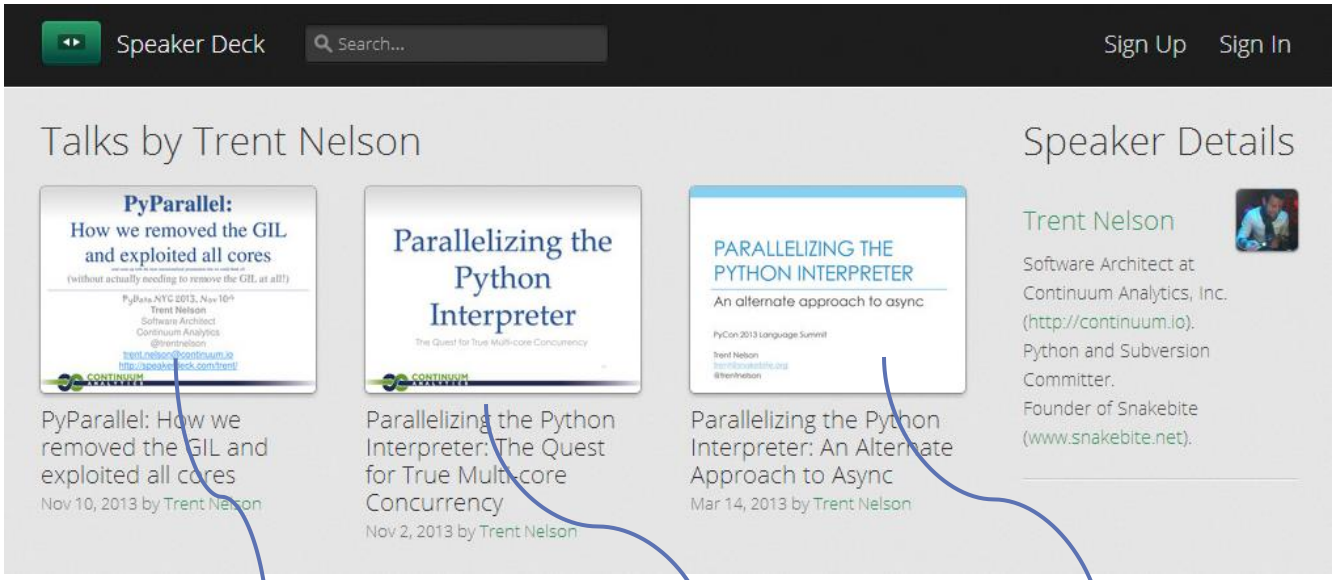
Survey Says…

- If there were a kickstarter to fund PyParallel
  - Including performant options for parallel compute, not just async socket I/O
  - And equal platform support between Linux, OS X and Windows
    - (Even if we have to hire kernel developers to implement thread-agnostic I/O support and something completion-port-esque)
- Would you:
  A. Not care.
  B. Throw your own money at it.
  C. Get your company to throw money at it.
  D. Throw your own money, throw your company's money, throw your kids' college fund, sell your grandmother and generally do everything you can to get it funded because damnit it's 2018 and my servers have 1024 cores and 4TB of RAM and I want to be able to easily exploit that in Python!

Slides are available online

(except for this one, which just has a placeholder right now so I could take this screenshot)

- http://speakerdeck.com/trent/



Longest, newest (this presentation)

Long, new

Short, old

Thanks!

• • •

Follow us on Twitter for more PyParallel announcements!

@ContinuumIO

@trentnelson

http://continuum.io/