Python Performance

Bob Hancock

hancock.robert@gmail.com

Premise

- To make an informed decision you must understand how something works under the hood.

- The most important quality in a programmer is patience. (See Guy Steele.)

High Level Overview

- How Posix operating systems service requests
- Generators
- Coroutines
- Futures
- Blocking/Non-blocking I/O
- Greenlets, libev, and gevent
- Why do I care about all this low level stuff?
- Design considerations
- What are the other options?

Let's Agree on Vocabulary

- Concurrency - processes executing in parallel that potentially needs to communicate with each other.
  - Keyboard, mouse, and GUI

- Parallelism - is about executing simultaneous independent deterministic processes. i.e., Shoving as much data as possible down the pipelines at the same time.
  - The map part of map-reduce.

Part 1

How Does the Operating System Service Requests ?

What is the OS?

- Set of programs that manage hardware and provide services.
- The kernel is the core program.
- You can download the source and build your own kernel.
- Your applications access the kernel via system calls.
- Only the kernel can access privileged instructions like accessing devices directly.

Kernel Mode

- The kernel executes system operations.
- It executes only in protected mode.
- Only the kernel can execute privileged commands.
- When the OS is in kernel mode, your programs must wait.
- It talks to hardware via interrupts.

User Mode

- This where your code executes.
- User code makes system calls to the kernel.
- Asks the kernel to execute a privileged instruction and gives you a result.
- Requests are in the form of a system call, also known as a trap.
- While in user space the kernel is free to service other requests.

Long Ago in the 1960s

- There was only kernel space and all programs executed sequentially.
- If you made a mistake, you could easily crash the whole operating system.
- Each vendor had their own operating system.
- IBM C360 added a hardware register for protected mode.
- MIT, Bell Labs, and General Electric started to develop Mutiplexed Information and Computing System (MULTICS).

Death of Multics

- Bell Labs pulled out of the project.
- Ken Thomson was bored and there was PDP-7.
- No protected mode, so it was just one mode.
- Brian Kernighan dubbed in Uniplexed Information and Computing Service – UNICS.
- Dennis Ritchie became interested and they found a PDP-11 with protected mode.
- C and UNIX were born.

Trivia Question

- In the UNIX password file there is a field called “gecos”.
- What does gecos signify and why is it there?

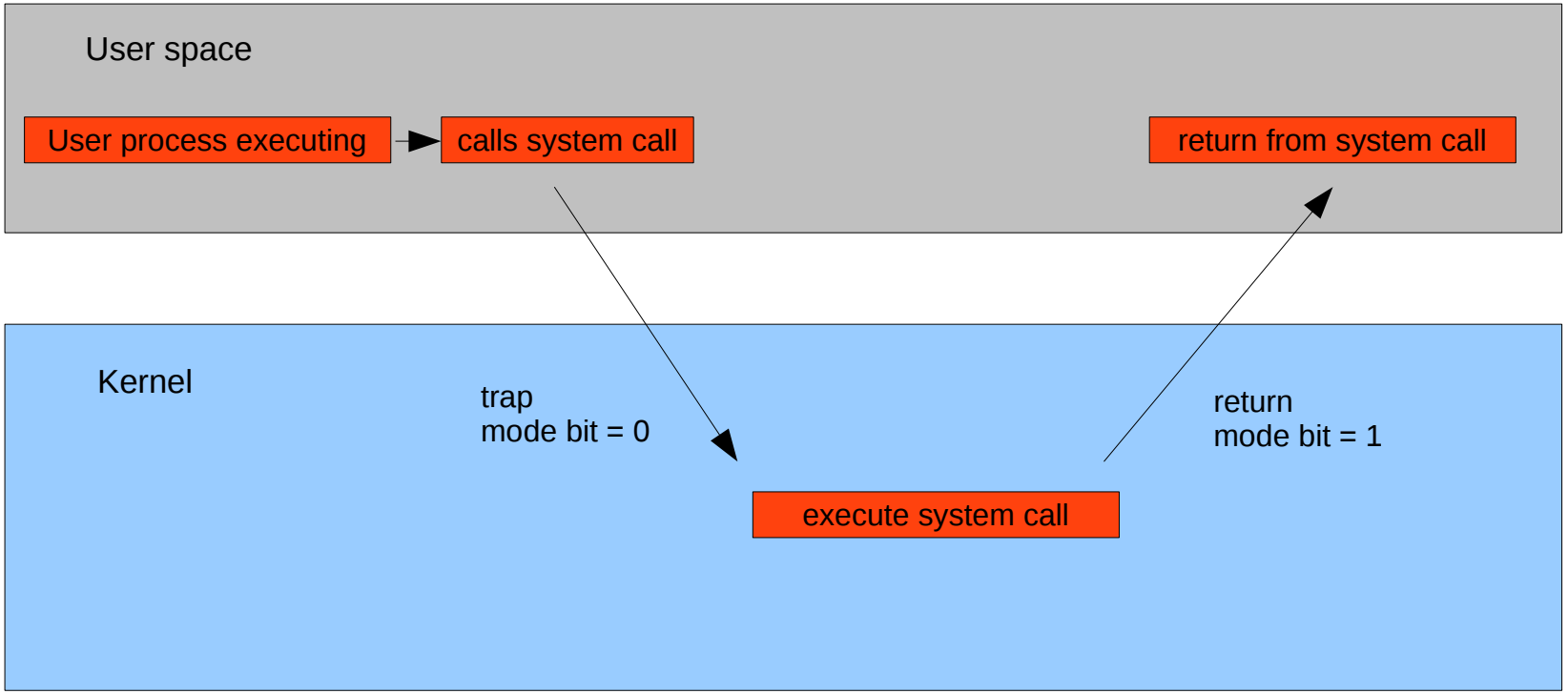
The Answer Is...

- General Electric Comprehensive Operating System
- Bell Labs used GECOS systems for print spooling.
- This field was added to keep track of a user's GECOS identity.
- Yes, the same General Electric that makes light bulbs used to be a major OS vendor.

Mode Switch

- Protected mode meant that now there were two modes – user and kernel.
- You can only operate in user or kernel mode.
- Moving from one to the other is a mode switch.
- A physical register is set to indicate either user or kernel mode.

Transition User to Kernel



Context Switch

- The context of a process is the union of user-level, register-level, and system-level context.
- When you switch processes, the OS must copy all this so it can return to the correct state.
- Steps
    1. Decide whether to do a context switch.
    2. Save context of “old” process.
    3. Find “best” process to execute.
    4. Restore context.

What Do Things Cost?

- These are non-blocking operations.
- Read CPU register = thought in your head.
- Read main memory = find a file in a drawer.
- Context switch = make an inventory of everything in your cube and its position relative to everything else, move out of your cube, move in a colleague, set up the cube for him, he works, make an inventory of everything in your cube and its position relative to everything else, colleague moves out, you move back, replace everything, continue work.

What Do Things Cost?

- These are potentially blocking operations.
- Disk seek = requesting a physical file from a branch office.
  - While waiting you work until the file arrives.
- Network request = Put someone on a plane, go to Tokyo via Frankfurt, find the office, find the person responsible, ask for the file, return flight via Sao Paolo, check to see if it is the correct file, process the file.
  - While waiting you work until the file arrives.

One At A Time

- The OS can only execute one operation at a time!
- It does it very quickly, but it is still sequential execution.
- So what is multi-processing?
- We need to understand the difference between processes and threads.

Processes

- An instance of a computer program that is executing.
- It is brought into to being with the fork command.
- Actually, in Linux, it is the clone command, but that is another talk.
- Each process has a process id.

rhancock 2068 17:08 00:00:00 /usr/bin/python /usr/bin/zeitgeist-daemon

rhancock 2313 17:08 00:00:00 /usr/bin/python /usr/share/system-config-printer/applet.py

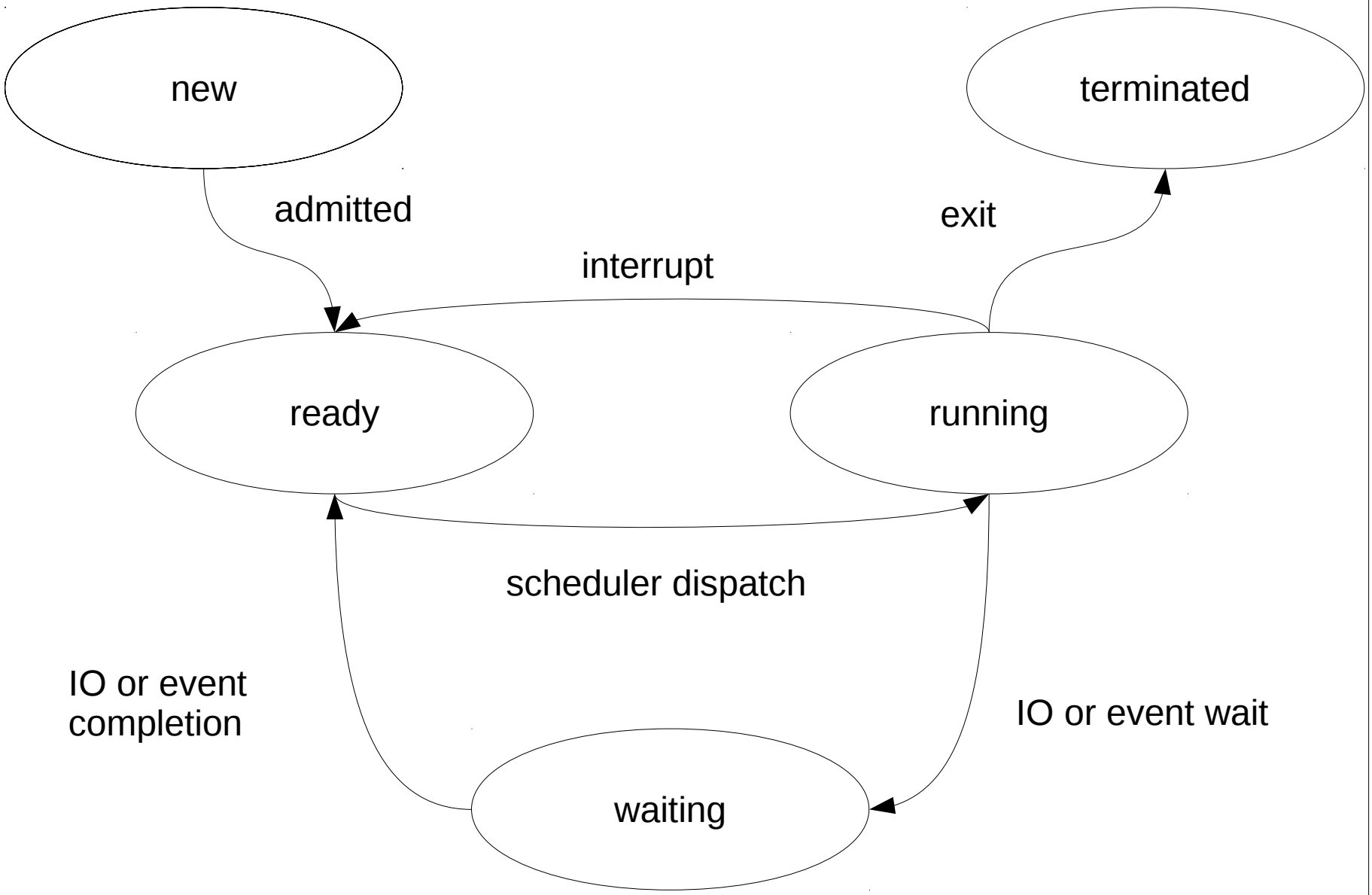
Threads

- A thread is a lightweight process that exists inside of a process.
- Threads within a process share memory space.
- Threads are multiplexed with in a process.
- Don't confuse them with *the thread of execution*.
- That is the smallest unit processing that can be *scheduled by the OS.*

Leger de main

- By switching quickly between processes and threads the OS gives the illusion of multiprocessing.
- Forking a process is an expensive operation.
- A thread costs less.
- Switching between threads requires less copying of memory.
- It still requires a context switch. (Remember this.)

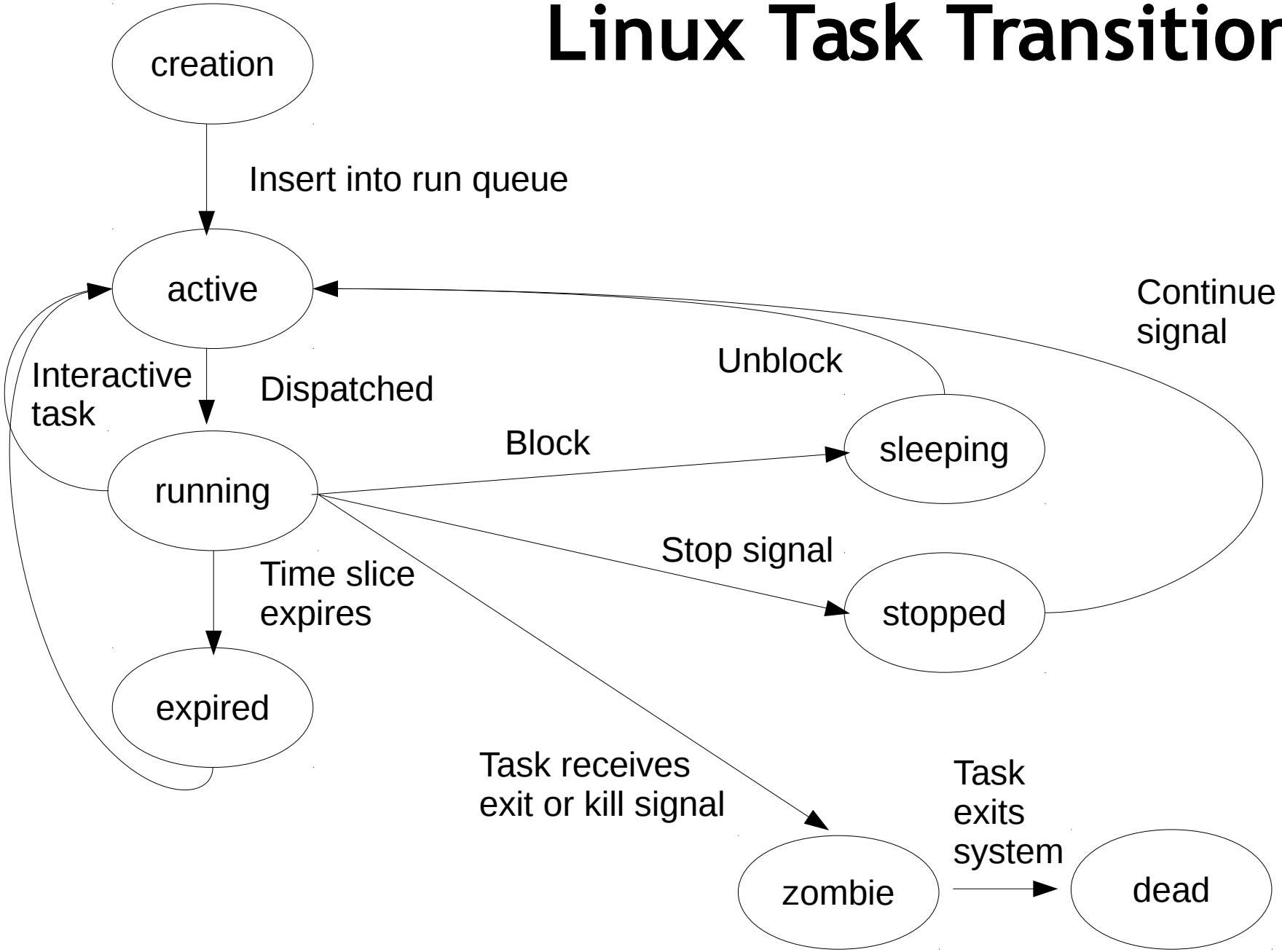
Process State



Linux Tasks

- Linux, unlike some other Posix systems, does not distinguish between processes and threads.
- Everything is a task.
- When you fork() you actually call clone()
- Copies only what is needed from the current task.
- Switching is extremely efficient.

Linux Task Transition



The Linux Scheduler

- Multi-level feedback queue
- Completely fair – what does this mean?
- It attempts to make use of all processors.
- It also means that if your process blocks it may see the core as available.
- This means a context switch and a loss of cache coherency.

The GIL

- Global Interpreter Lock
- A mutex that prevents multiple native threads from executing Python bytecodes simultaneously.
- Necessary mainly because CPython's memory management is not thread-safe.
- Must be held by the current thread before it can safely access Python objects

Python and Threads

- Python threads are full system threads.
- Shouldn't I use threads all the time?
- With the 2.7 GIL, running a threaded process on a multicore machine will slow you down.
- It executes a round robin every n ticks.
- The 3.3 GIL is better.
- It avoids GIL wars and is more equitable, but still does not implement priority scheduling.

The GIL

- The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads.
- It prevents signals from being delivered.
- The GIL is the single biggest point of controversy in Python.
- The best explanation is http://www.dabeaz.com/python/GIL.pdf

The GIL

- To emulate concurrency of execution, the interpreter regularly tries to switch threads
- The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.
- System call overhead (bookkeeping) is significant, especially on multicore hardware.

GIL Knowledge You Need

- For today's discussion you need to know the following:
- Threaded programs on multi-core machines can run significantly more slowly than on single core machines—if any still exist.
- The exception is that sometimes, when there is a lot of blocking IO, threaded programs can be efficient.
- Sockets fair better than persistent storage.

The Python 3 GIL

- Antoine Pitrou re-wrote the GIL to use a conditional object.
- This was changed in 3.2.2.
- The notion of a “check interval” to allow thread switches has been abandoned and replaced by an absolute duration expressed in seconds.
- 5ms but can be changed with sys.setswitchinterval(interval).

The GIL

- Why do I care about the GIL?
- It affects system threads.
- It affects performance.
- You need to understand how it works if you are going to write C extensions.

Bigfile Example

- Read a 60 gigabyte web log
- Find each record with a specific user name.
- Let's look at the results and the code.
- **bigfile_brute.py**
- **bigfile_regex.py**
- **bigfile_threads.py**
- **bigfile_mp.py**
- **bigfile_*.py for the rest**

Part 2

Python Generators

Generators

- A generator is a function that produces a sequence of results for iteration. **generators_1.py**

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

m = 10
x = countdown(m)
print(x)
for i in range(3):
    print("Countdown: {d}".format(d=x.next()))
```

Generators

- Function returns a generator object.
- The generator object executes the function when next is called.
- The function executes until it reaches yield statement.
- Yield produces a result and execution halts until the next invocation of next.
- Let's step through the code in debug.

itertools.chain()

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

Generator Pipelines

- Like shell pipes in UNIX.
- Create a function that receives a value and emits a value.
- You can hook these together and insert other generators as you alter your code.
- How does this affect Bigfile processing?
- Let's look at the timings.
- **bigfile_pipeline_1.py**

Yield as an Expression

- Python 2.5 (PEP-342) added yield as an expression.

```
def grep(pattern):
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

- What is the difference between a statement and an expression?
- So what does this do for me?

Part 3

Coroutines

Coroutines

- Using yield as an expression allows you to create coroutines.
- Coroutines execute when values are sent to them.
- (yield) returns the value.
- Generators only produce values.
- Coroutines consume values and return.
- Generators are about iteration.
- Coroutines are about consuming values.

Coroutines

- What is a coroutine in computer science?
- The concept of coroutines has been around since 1958 – M. E. Conway
- Coroutines are functions that save control state between calls.
- Why did threads overtake coroutines?
- Prime the coroutine with a decorator (Thanks to Dave Beazley).
- **coroutines_1.py**

Coroutines

- What would this do for the bigfile processing?
- **bigfile_pipeline_2.py**
- Let's look at the numbers.

Coroutines

- Python coroutines can be used to implement concurrency.
- An event loop can send data to a collection of coroutines that carry out diverse tasks.
- Python generators would be great if we had non-blocking IO
- Hold on, we are getting to that.

Part 4
Worthwhile Optimizations

Worthwhile Optimizations

- Everything is compounded at scale.
- Everything will eventually break at scale.
- **Code in optimal sub-dir.**
- Comprehensions over loops.
- Cdecimal (standard in 3.3) versus Decimal
- Memoize redundant calculations
- **fibonacci.py**
- Let's look at the memoized numbers.

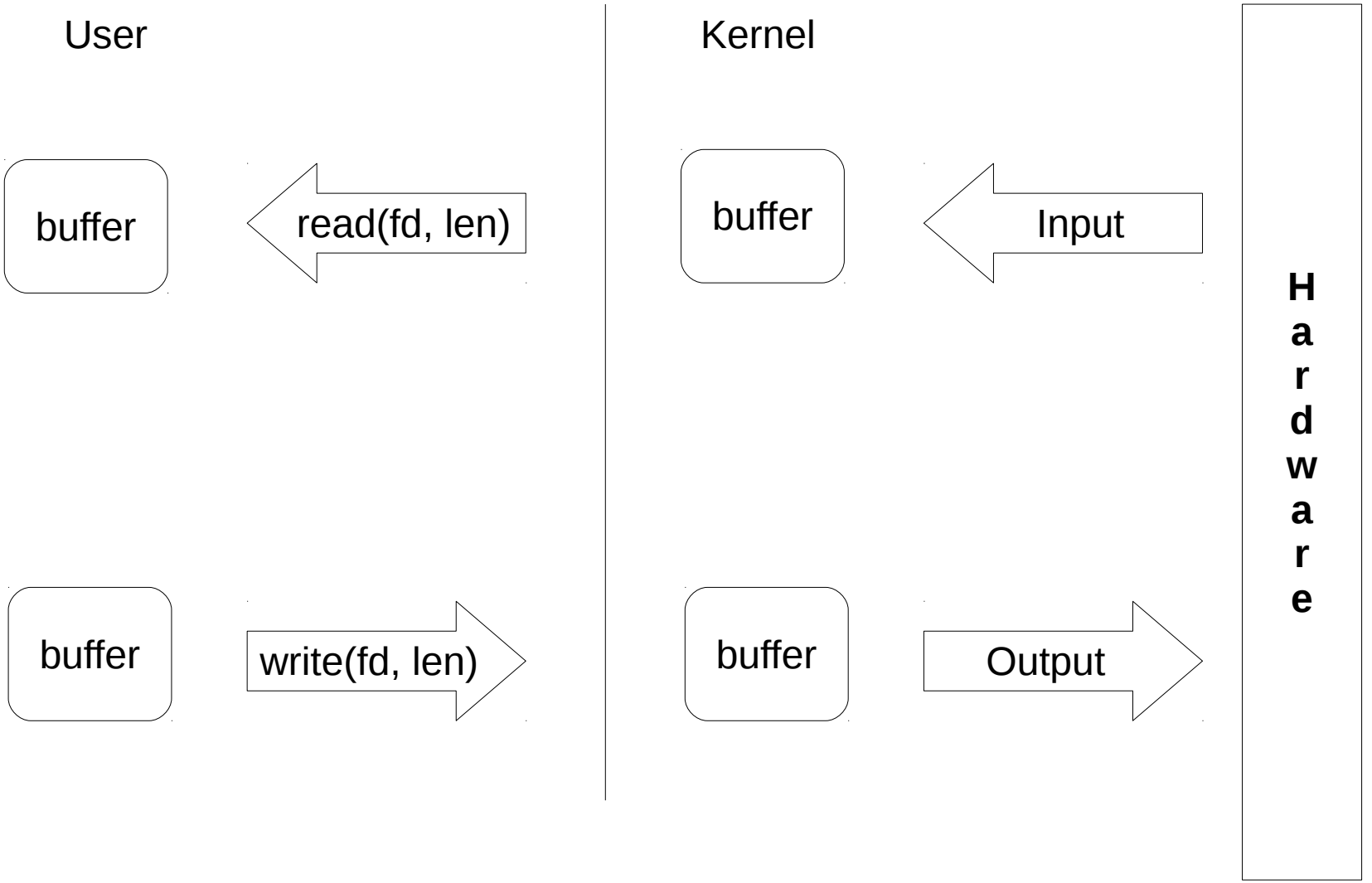
Memoizing versus Caching

- Memoizing - function calls avoid repeating the calculation of results for previously processed inputs.
- Caching – refers to a technique of storing data so that data can be served more quickly.
- Memoizing is a form of caching.

Part 5

IO

Buffered I/O

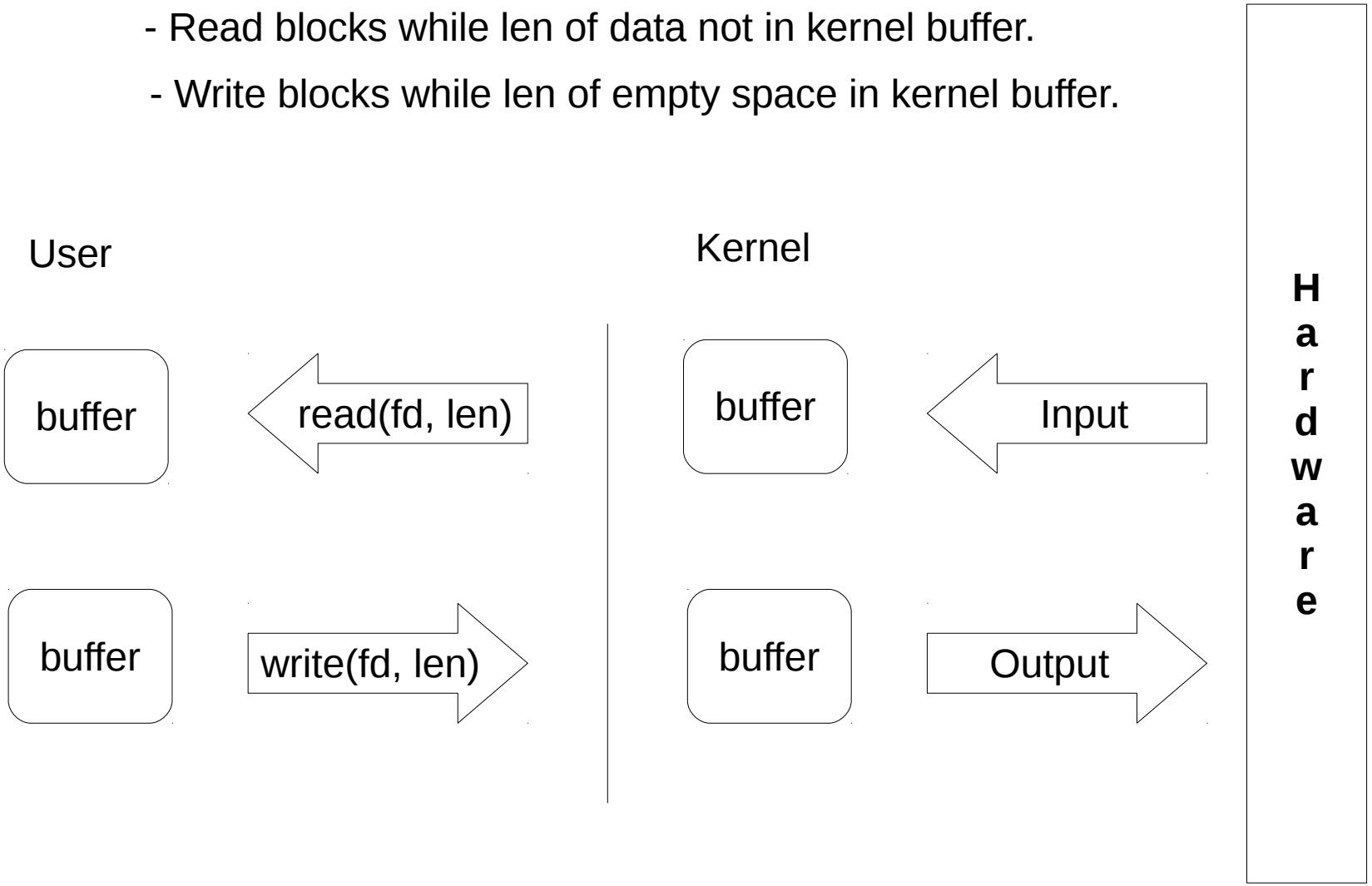


Blocking IO

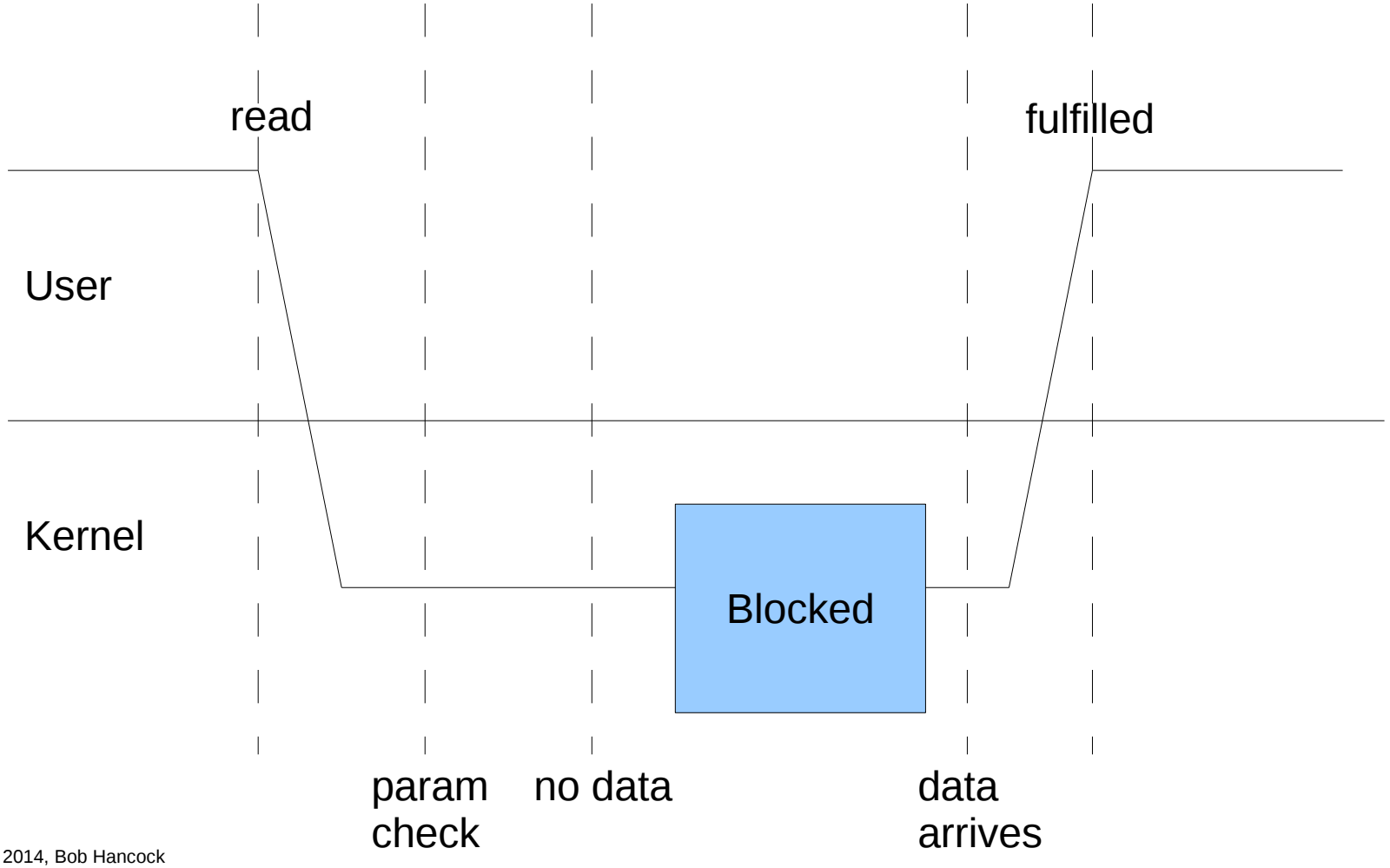
- An IO operation that may cause the requesting thread of execution to be blocked from further use of the processor.
- This means that the thread of execution and the IO run *sequentially*.

Blocking I/O

- Read blocks while len of data not in kernel buffer.
- Write blocks while len of empty space in kernel buffer.



Blocking IO Timeline



Blocking IO

blocking_io_1.py

```
import os
text = ""
while True:
    if text.lower() == "quit":
        os.write(1, "He's dead, Jim!\n")
        break
    elif text:
        os.write(1, "You said, '{n}'\n".format(n=text))
        text = ""
    os.write(1, "->")
    while True:
        readval = os.read(0, 4)
        if readval[-1] == "\n":
            text += readval[:-1]
            break
        text += readval
```

Non-blocking IO

- An IO operation that does not cause the thread of execution requesting the IO to be blocker from use of the processor.
- Thread of execution and the IO operation still run sequentially.
- Thread of execution will be notified when an IO operation is not initiated or partially initiated.
- You must tell the fd that this is non-blocking.

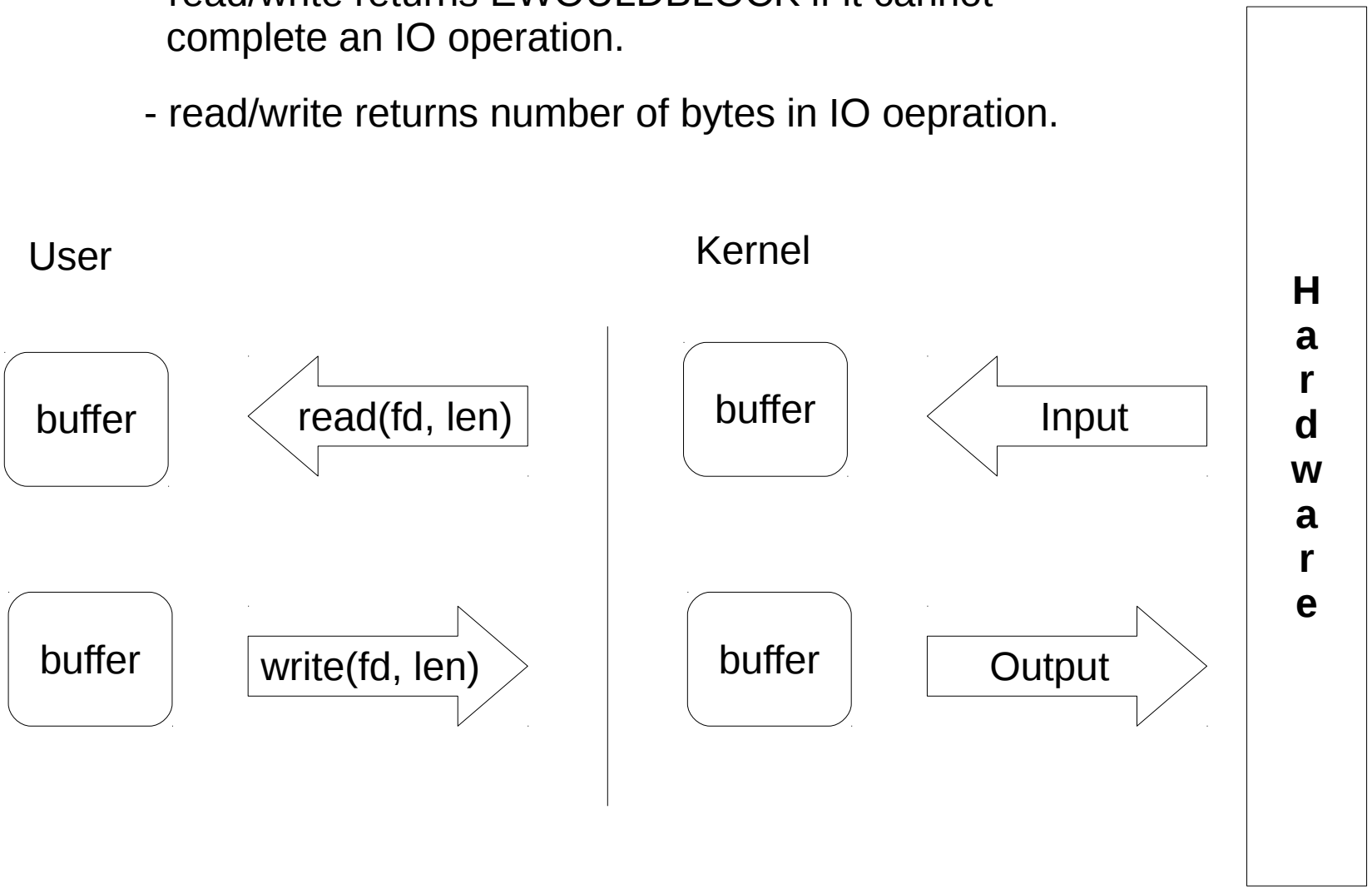
Blocking and Non-Blocking IO

- Blocking means that a thread of execution cannot do anything else until the IO is complete.
- Non-blocking IO makes the request and if it cannot be immediately fulfilled, it assumes that it will be sometime in the future.
- Operating systems schedule IO in an asynchronous manner.

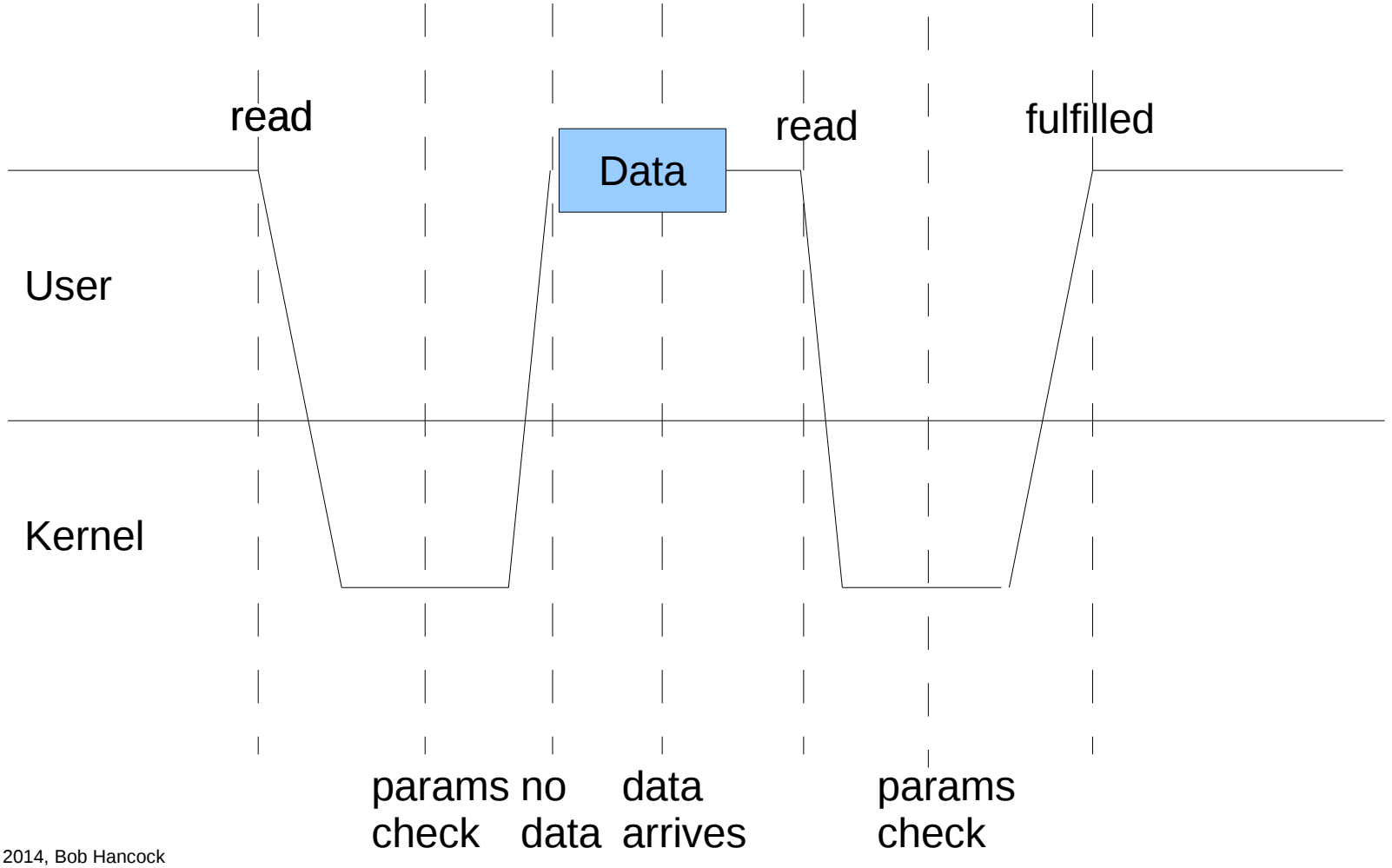
Non-Blocking I/O

- read/write returns EWOULDBLOCK if it cannot complete an IO operation.

- read/write returns number of bytes in IO oepration.



Non-blocking IO Timeline



Non-blocking IO

- Non-blocking IO waits for no man.
- What if there is no data for a time?
- **non_ blocking_io_1.py**

Non-blocking IO

- We create a procedure that we loop over that handles the exception.
- non_blocking_io_2.py
- This is getting really complicated and ugly!
- The OS offers you three system calls to deal with asynchronous IO.

select

- Returns a bitmask of all available file descriptors.
- Marks Fds that are eligible for IO.
- O(n) where n is the number of all available FDs.
- Uses up to 3 bits per FD.

poll

- Returns only registered FDs.
- Uses 64 bits per FD.
- Level triggered.
- Signals the presence of an unserviced FD.
- FDs will continue to be reported until you service them or the OS handles them.

epoll

- Edge triggered
- Signals a change in level on the interrupt line.
- If there is an FD to be serviced, you are only notified *once*--when it actually occurs.
- epoll is the fastest but requires more complex user logic and timing.

Event Loops C Libraries

- libevent - 2001
- Provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached.
- libev – 2007(?)
- Loosely modeled on libevent but without the limitations and bugs.
- Faster than libevent.

Part 6
greenlets

greenlets

- Comes from Stackless.
- Allows switching between coroutines anywhere within the call tree.
- The state of the OS thread is saved.
- C and assembler routines manipulate the stack.
- greenlets can only run the context of on Python thread (which is, remember a system thread).
- **greenlet_1.py**

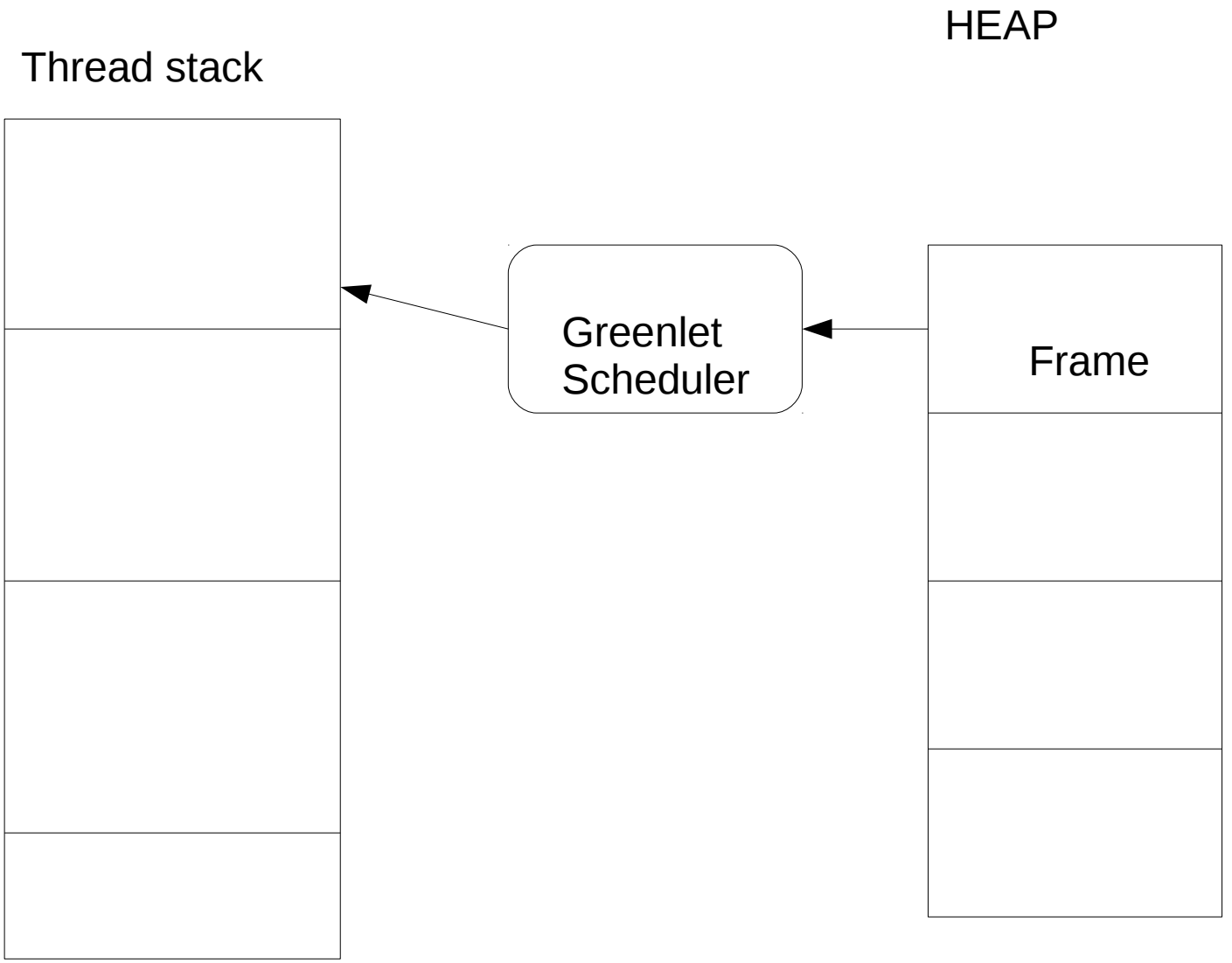
Python coroutines/greenlets

- You can use a subroutine within a greenlet to call other greenlets.
- greenlets are not “stack based” like Python coroutines.
- They allow arbitray invocations within the execution sequence.
- **greenlet_2.py**

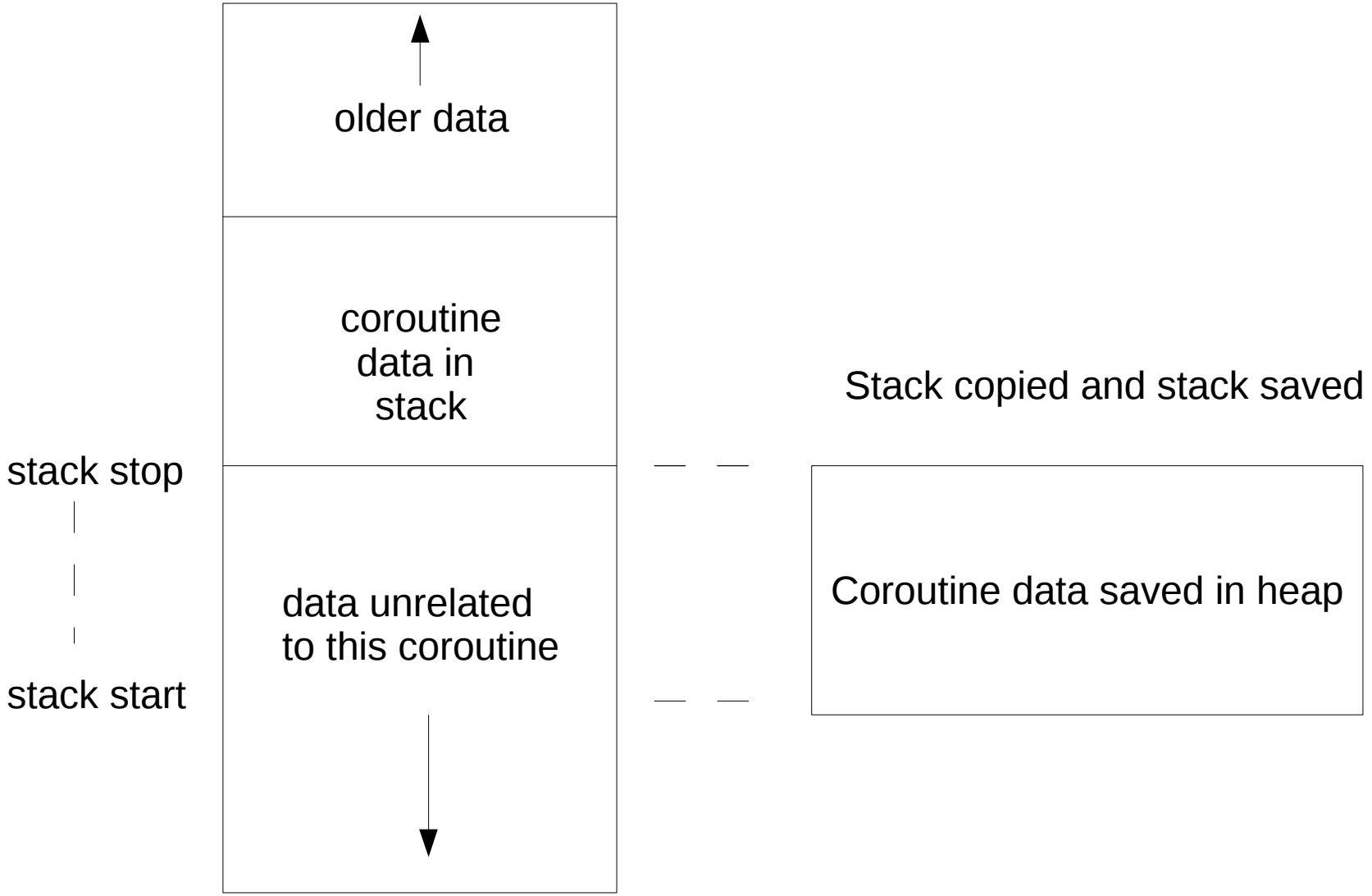
Coroutines and Threads

- Chunks of the thread execution stack are moved to the heap.
- As an event occurs for a frame on the heap, the frame is restored to the thread stack for execution.
- The scheduler controls the movement of frames.
- Non-blocking IO keeps the scheduler from stalling.
- Coroutines are tied to a thread.
- You cannot call coroutines across threads.
- The OS is unaware of coroutines.

Coroutines and Threads



Coroutines and Threads



Threading Models

- 1:1 Each program thread maps to a kernel thread.
  - Python and JVM (Java, Scala, etc.)
- M:1
  - Lightweight processes in one OS thread
  - Greenlet
- M:M
  - Beyond scope of this discussion

Scheduling

- By multiplexing coroutines in one thread, you avoid a context switch.
- The schedulers are the weak link.
- cpu bound process will block.
- Involuntary context switch will move it out of the system run queue.
- Performance relies on non-blocking IO.
- Before we put it all together..

Quiz

- What is a process?
- What is a thread?
- What is a context switch?
- What is blocking IO?
- What is non-blocking IO?
- What is a generator?
- What is a coroutine?

Part 7
Putting It Together

The Parts

- Non-blocking IO
  - Faster but you have to manage it.
- Python coroutines
  - Avoid the overhead of classes
  - Pipelines
- greenlets
  - Thread like performance without context switches

Packages

- twisted
- Stackless
- eventlet
- gevent
- concurrence
- There are probably more.

How Do They Work

- They all monkey patch the IO functions to make them non-blocking.
- They have an event loop that listens for IO eligible events.
- They dispatch via a cooperative scheduler
- What is pre-emptive scheduling?
- Asynchronous callbacks.

Event Loop

- There is a main event loop that services coroutines on a FIFO basis.
- If a CPU bound process blocks, you are toast.
- gevent, Go, node.js, Twisted, etc. all use this model.
- No migration of coroutines across tasks.

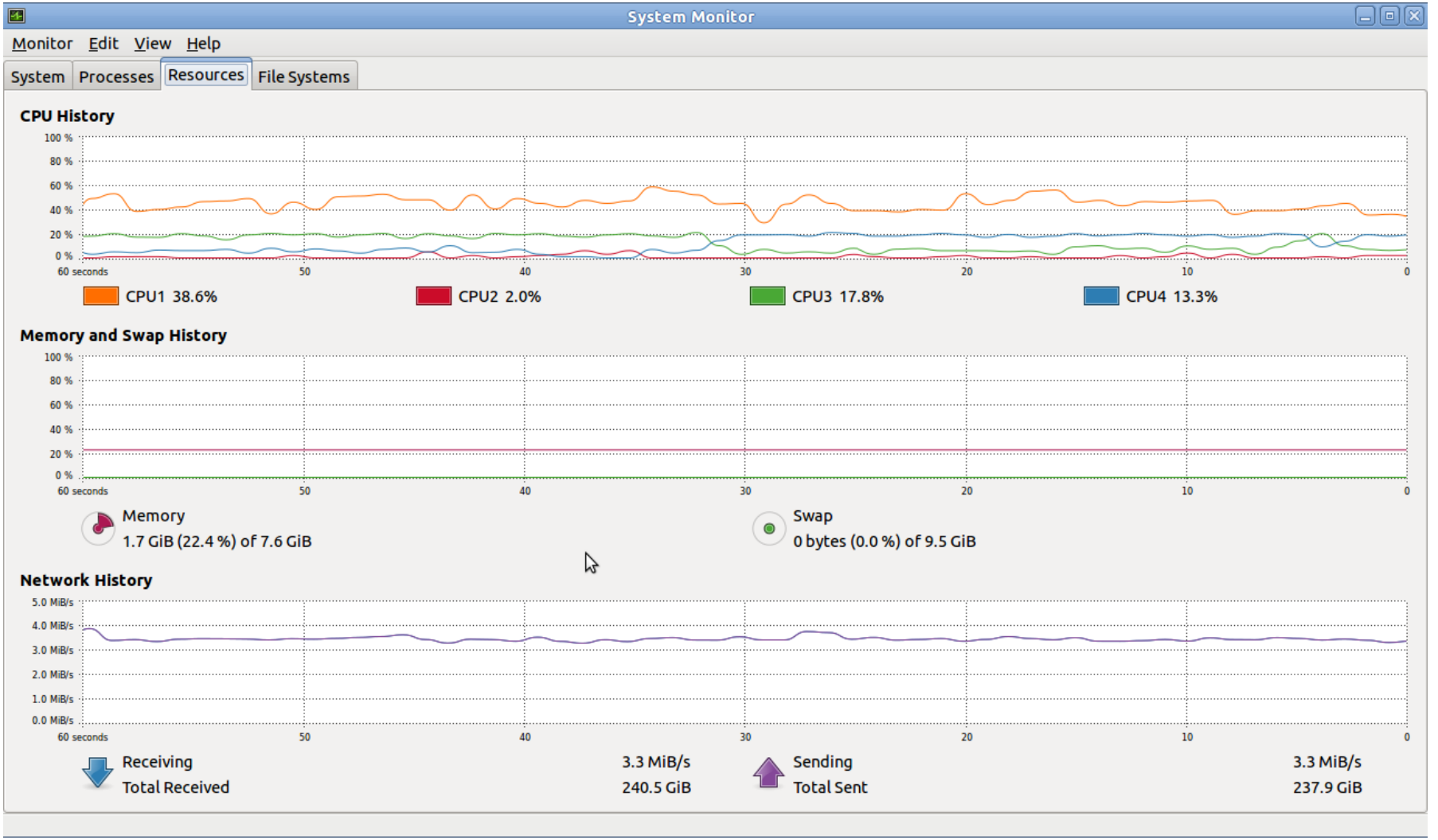
The Thundering Herd

- The thundering herd problem occurs when a large number of processes waiting for an event are awoken when that event occurs, but only one process is able to proceed at a time.
- After the processes wake up, they all demand the resource and a decision must be made as to which process can continue.

Not Every Server is HTTP

- You can write your own server for a specific service.
- An echo server that serves n concurrent requests.
- **echo-server-go.py**
- **echo-client-go**
- Let's look at the numbers.

gevent echo server



Gevent Advantage

- You don't need to write asynchronous code like for Tornado or Twisted.
- The memory and CPU usage are consistently small.
- Consistent behavior.
- Uses libev.

Gevent URL Crawler

- Retrieve the the data from 35 URLs n times
- **gevent_urlclient.py**
- Compare using futures for async IO in Python 3K.
- **futures_urlclient.py**

Gevent Synchronization

- Synchronized queues
- Use queues as channels.
- Events like Python threads – don't use.
  - Consumer/producer can miss events
- Course locking primitives
  - Avoid locks.
  - Revisit your program design.

libevent versus libev

- http://libev.schmorp.de/bench.html
- libevent was released on 11/14/2000 and has been a standard for years.
- libev was released on 11/12/2007 is faster and scales better.
- Check your third party libraries to see which they uses.

Part 8
Design

Design Points

- Can you decompose your data?
- Divide deterministic from non-dertiministic.
- Can the the deterministic processes run in parallel?
- Are any processes concurrent?
- Does it lend itself to map-reduce?
- Separate IO or CPU bound?
  - Have them communicate.

Design Points

- Do you need a separate coroutine for each process or can they run in a pool?
- How fast is fast enough?
- Is Python the right choice?
- Embrace eventual consistency.
- Check your database driver!
- Learn by surrogacy.

Parellelism

- Parallelism will likely help you scale, but it will not necessarily increase real time performance.
- Understand the bookkeeping involved.
- Make it as orthogonal as you can.

CPU Bound

- CPU bound processes won't give up kernel mode.
- Mathematical calculations
- Video rendering.
- Compression algorithms using floating point.
- Put these on a separate core, bus, machine.

Interprocess Communication

- Share memory by communicating, don't communicate by sharing memory.
- gevent queues can act as channels.
- A synchronized queue is the easiest and safest first choice to share data.

Separate IO and CPU Bound

- Separate IO and CPU bound sections.
- Consider placing them on separate cores or machines.
- Separate as RPC calls.
- Reduce at the end.
- Separate small server for each service.

More Is Not Necessarily Better

- Balance bookkeeping size of the problem.
- Consider starting more than one copy of the process.
- Measure – consider – measure again.

Part 8
Options

Cython

- Write in a dialect of Python and produce C code.
- See Wes McKinney's Pandas for an example of Cython.

Concurrent.futures

- Futures and promises.
- Futures are read only, and promises are writable (at least in Scala).
- A request will be fulfilled at some time in the future.
- Sequence is not guaranteed.

Go

- Created by Google for server side programming.
- Goroutines only take 4K to instantiate.
- The scheduler will multiplex goroutines between threads
- Strongly typed.
- C/Python like syntax – imperative.
- Team: Ken Thomson, Rob Pike, Russ Cox,

Golang

- **go_urlclient.go**
- echo-server-go
- Uses less memory and cpu than gevent.
- The language is approachable by Python programmers.
- It helps if you have some C experience.

Goroutines and CSP

- Communicating Sequential Process – Tony Hoare. (see bibliography)
- Intelligent use of channels can obviate the need for Mutexes.
- Google plans on this becoming their standard language for server side programming.
- An optimized gcc version is in the works.

Part 8
Conclusions

Conclusions

- The GIL makes Python easily extensible, but does limit performance.
- A monolithic kernel give the illusion of multiprocessing.
- Hiding scheduling from the OS by using coroutines in one process/thread can speed up IO bound processes.
- Coroutines are scheduled on a FIFO basis.
- CPU bound processes will still toast you.

Fix It!

- Why doesn't somebody fix it?
- Because it is a REALLY hard problem.
- How do you avoid starvation and priority inversion?
- How do you deal with the bookkeeping issues across multiple cores?
- Do you write an operating system on top of an operating system?
- How do you deal with distributed processing across multiple machines?

What Can You Do

- Come to the keynote tomorrow.