CONCURRENT PROGRAMMING IN PYTHON

An introduction to multi-threading, multi-processing and event loops

James R. Small

Michigan!/usr/group

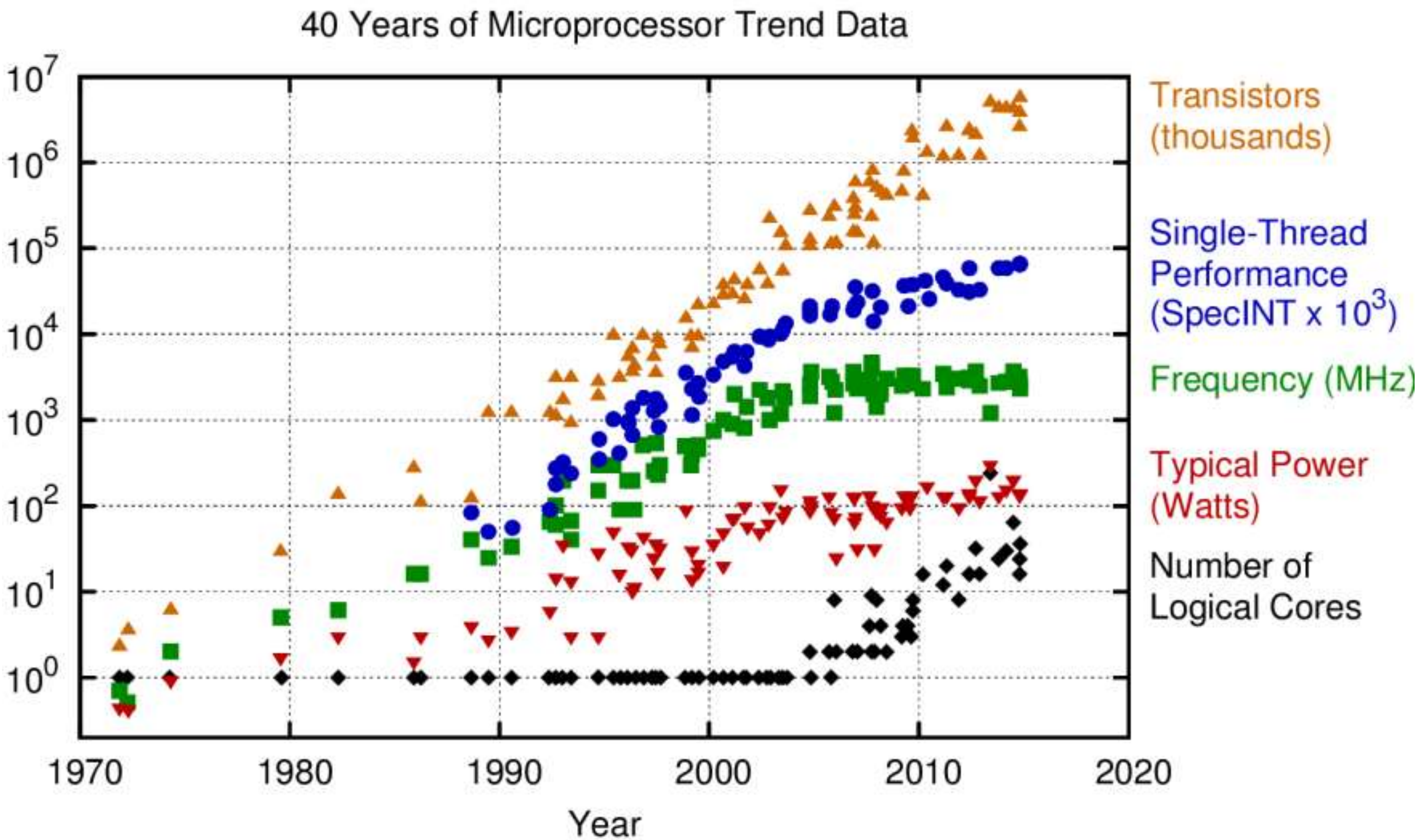
mug.org – A Free and Open Source Michigan Community

WHY CONCURRENCY?

- Performance, Responsiveness
- Why not just a faster processor/ computer?
- Moore's Law

See: Packet Pushers Show 315: Future Of Networking – Pradeep Sindhu

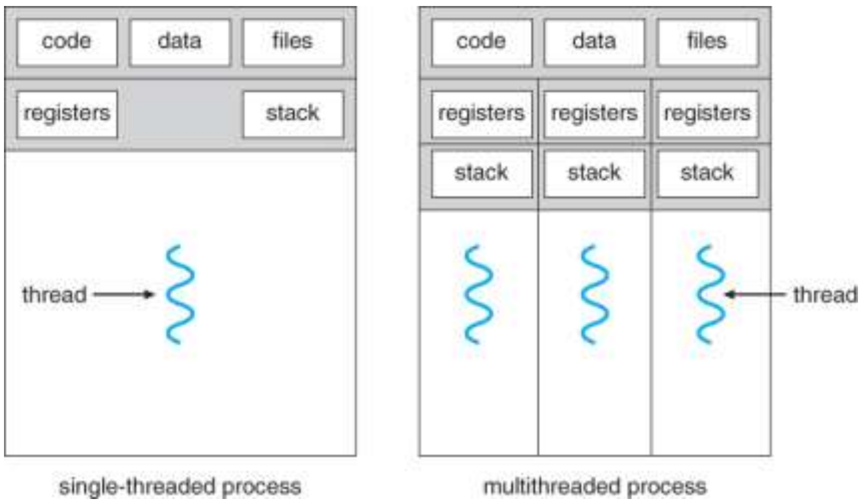
NECESSITY OF CONCURRENCY



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

BASICS – RUNNING A PROGRAM

- Process – essentially a container which hosts an executing binary or program



- Thread – what actually runs (the) code within a process, all processes have at least one thread; each thread is a separate “flow” of execution

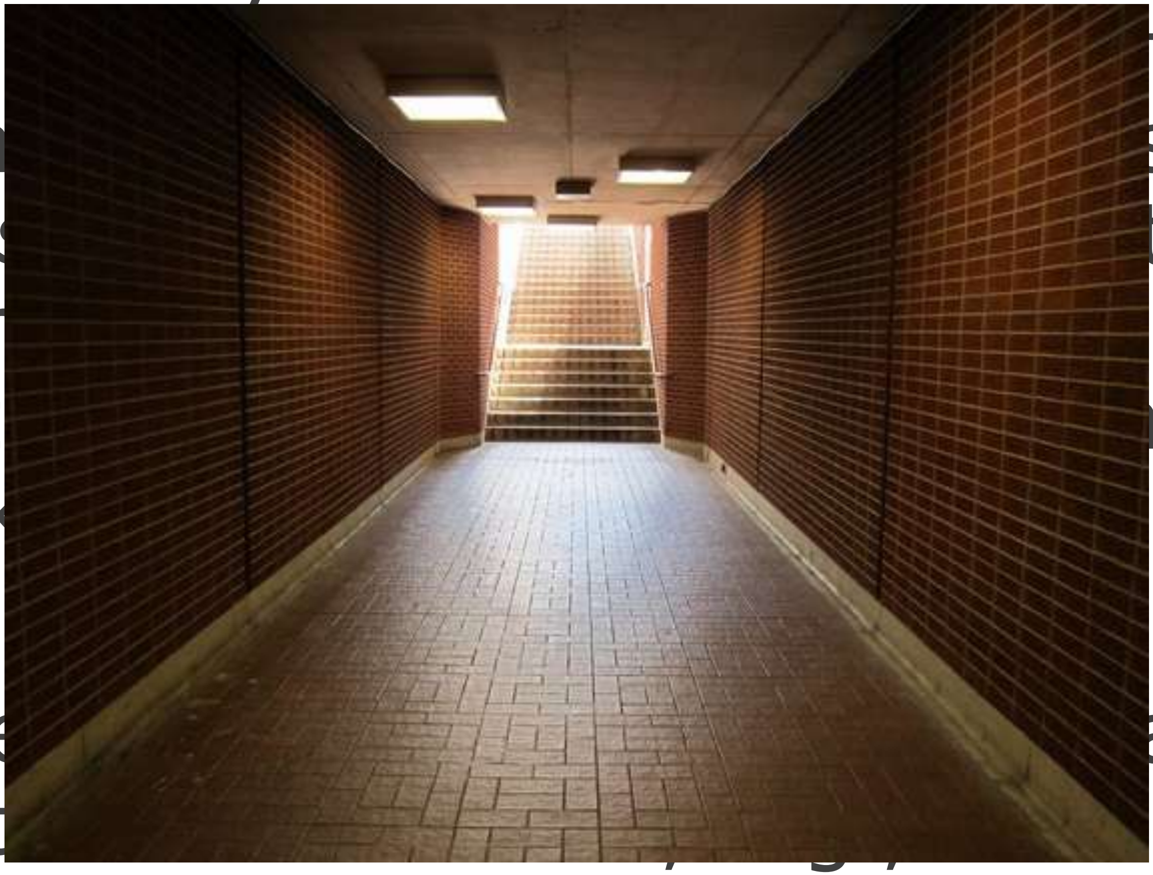
BASICS – SCALING PERFORMANCE, LIMITERS

- Should I always try to leverage multiple cores?

- Problem Type
  - » CPU-Bound – Program completion determined by speed of the central processor
  - » I/O-Bound – Program completion determined by time spent waiting for (external) input/output operations to finish

BASICS – SCALING PERFORMANCE, TERMS

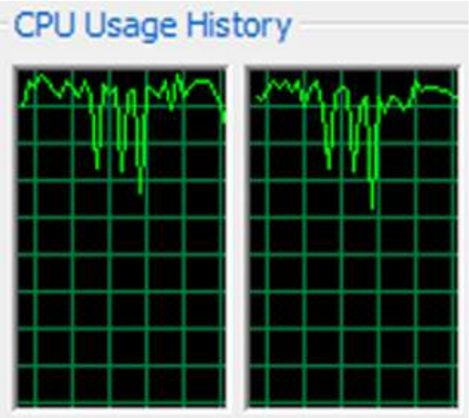
- Concurrency – when two or more tasks te in overla sn't neces th be runnin ., multit chine. (Stack



- Paralle ally run at multi-core machine. (StackOverflow)

EXAMPLE – CALCULATE FIBONACCI NUMBER

- CPU-Bound Problem



- Example Implementation:

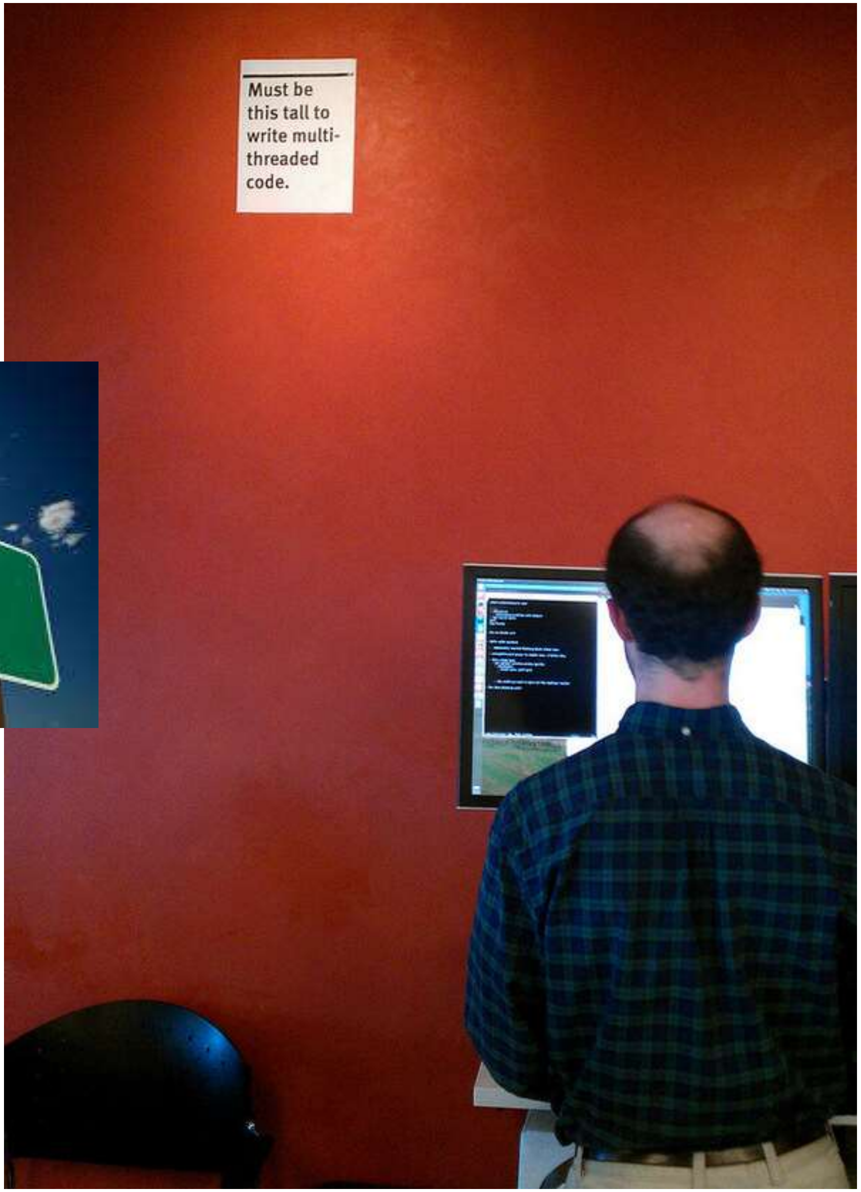
```python
# Simple, non-optimal fibonacci function
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

BASICS – DIFFERENCES

- Process – has its own address space (not shared with other processes)
  - Heavier weight, slower to instantiate
  - Independent state (safer)
  - Collaboration requires IPC

- Thread – shared address space with hosting process
  - Lighter weight, faster to instantiate
  - Shared state (must protect against corruption)
  - Native collaboration (shared state)

RESPECT FOR CONCURRENT PROGRAMMING



QUESTION – WHICH WILL BE FASTER?

- In Python, which technique will be faster for (concurrently) solving a list of Fibonacci numbers?

  » Multi-threading
  » Multi-processing

- Why?

BUILDING EXAMPLE SERVER

- Remaining examples will be I/O-Bound (network based)
- Will use simple echo server to demonstrate concurrency approaches

```python
# Simple, single-threaded echo server
from socket import *

def echo_server(address):
    # Create a socket, type IPv4, TCP
    sock = socket(AF_INET, SOCK_STREAM)
    # Set socket options
    # * Allow binding to address even if in use (also
    #   bypass TIME_WAIT delay)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    # Bind socket to passed address, port tuple
    sock.bind(address)
    # Listen for incoming connections
    # * Use a queue that can hold up to 5 pending
    #   connections
    sock.listen(5)
    while True:
        # Get socket and address, port tuple of
        # connecting client
        client, addr = sock.accept()
        # Dispatch to handler
        echo_handler(client, addr)
```

```python
def echo_handler(client, addr):
    print('Connection from {}'.format(addr))
    # Use context manager for socket "client":
    with client:
        # While not EOF (more data possible), loop
        while True:
            # Receive available data transmitted from
            # client
            data = client.recv(10000)
            # If no more data (EOF/EOT), break out of
            # loop
            if not data:
                break
            # Format data string
            data = 'Received:  '.encode('ascii') + data
            # Echo back to client
            client.sendall(data)
    print('Connection from {} closed'.format(addr))

# Start echo server on all available interfaces (with
# IPv4 addresses)
# * Use TCP port 25,000
echo_server(('', 25000))
```

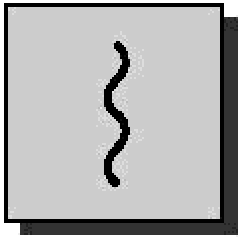
BUILDING EXAMPLE SERVER

- What’s wrong with this example echo server?

- More definitior
  - Blocking – a loesn’t immediately ou have to wait for it)
  - Non-blocking nediately returns

PROBLEM WITH EXAMPLE SERVER

- When highlighted section of code runs, the server is “blocked” until the client (echo_handler) finishes:



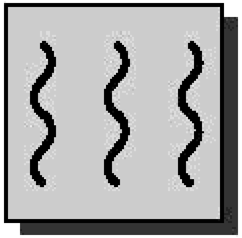
```python
# Simple, single-threaded echo server
from socket import *

def echo_server(address):
    # Create a socket, type IPv4, TCP
    sock = socket(AF_INET, SOCK_STREAM)
    # Set socket options
    # * Allow binding to address even if in use (also
    #   bypass TIME_WAIT delay)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    # Bind socket to passed address, port tuple
    sock.bind(address)
    # Listen for incoming connections
    # * Use a queue that can hold up to 5 pending
    #   connections
    sock.listen(5)
    while True:
        # Get socket and address, port tuple of
        # connecting client
        client, addr = sock.accept()
        # Dispatch to handler
        echo_handler(client, addr)
```

```python
def echo_handler(client, addr):
    print('Connection from {}'.format(addr))
    # Use context manager for socket "client":
    with client:
        # While not EOF (more data possible), loop
        while True:
            # Receive available data transmitted from
            # client
            data = client.recv(10000)
            # If no more data (EOF/EOT), break out of
            # loop
            if not data:
                break
            # Format data string
            data = 'Received:  '.encode('ascii') + data
            # Echo back to client
            client.sendall(data)
    print('Connection from {} closed'.format(addr))

# Start echo server on all available interfaces (with
# IPv4 addresses)
# * Use TCP port 25,000
echo_server(('', 25000))
```

OPTION 1 – USE MULTI-THREADING

- Use a new thread to service each incoming client. Frees up server so it's non-blocking:

```
# Simple, single-threaded echo server
from socket import *
from threading import Thread

def echo_server(address):
    # Create a socket, type IPv4, TCP
    sock = socket(AF_INET, SOCK_STREAM)
    # Set socket options
    # * Allow binding to address even if in use (also
    #   bypass TIME_WAIT delay)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    # Bind socket to passed address, port tuple
    sock.bind(address)
    # Listen for incoming connections
    # * Use a queue that can hold up to 5 pending
    #   connections
    sock.listen(5)
    while True:
        # Get socket and address, port tuple of
        # connecting client
        client, addr = sock.accept()
        # Create new thread for handler dispatch
        Thread(target=echo_handler,
               args=(client,addr)).start()
```

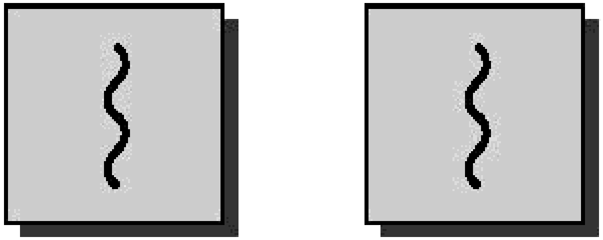


Good for I/O-Bound Problems

OPTION 2 – USE MULTI-PROCESSING

- Use a new process to service each incoming client. Frees up server so it's non-blocking:

```
# Simple, multi-process echo server
from socket import *
from multiprocessing import Process

def echo_server(address):
    # Create a socket, type IPv4, TCP
    sock = socket(AF_INET, SOCK_STREAM)
    # Set socket options
    # * Allow binding to address even if in use (also
    #   bypass TIME_WAIT delay)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    # Bind socket to passed address, port tuple
    sock.bind(address)
    # Listen for incoming connections
    # * Use a queue that can hold up to 5 pending
    #   connections
    sock.listen(5)
    while True:
        # Get socket and address, port tuple of
        # connecting client
        client, addr = sock.accept()
        # Create new thread for handler dispatch
        Process(target=echo_handler,
                args=(client,addr)).start()
```

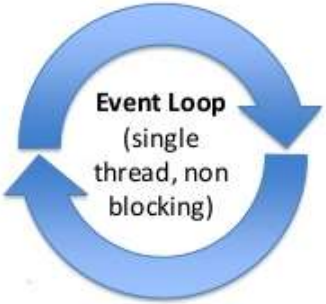


```
# Must start this way or won't work!
if __name__ == '__main__':
    echo_server(('', 25000))
```

Good for CPU-Bound Problems

OPTION 3 – USE AN EVENT LOOP

- In addition to processes and threads, there's another paradigm – an event loop.
- Why not stick with processes/threads?
- Scale – processes/threads good into thousands, what if tens of thousands (or more) of concurrent connections need to be serviced?



EVENT LOOPS – BASED ON SELECT (EXAMPLE)

```
from socket import *
import select
from Queue import Queue

# Socket lists
inputs = []
outputs = []

# Queues
msg_in = {}
msg_out = {}

timeout = 60

# Setup server socket and put into my_sockets list above
# <Code omitted>

def server_loop(srv_sock):
    # While there are sockets in the list, process them
    while inputs:
        readable, writable, exceptional = select.select(
                inputs, outputs, inputs, timeout)
        if not (readable or writable or exceptional):
            # select tiemout expired, next iteration of
            # loop
            continue
        # input
        for s in readable:
            # Main socket listening for inbound clients
            if s is listen_sock:
                clientsock, clientaddr = s.accept()
                clientaddr = clientaddr[0] + ':' + str(
                        clientaddr[1])
                print 'Accepted connection from ' + \
                        clientaddr
                # Make socket non-blocking
                clientsock.setblocking(False)
                # Add to input socket list
                inputs.append(clientsock)
                msg_out[clientsock] = Queue()
            else:
                data = s.recv(1024)
                if data:
                    # inbound client data
                    # Sloppy - might not get entire
                    # question on one recv call
                    if data.endswith('?'):
                        answer = reply(data)
                    # Incomplete question, queue...
                    msg_out[s].put(answer)
                    if s not in outputs:
                        outputs.append(s)
                else:
                    # empty client connection = close
                    # connection
                    print 'End of Transmission from ' + \
                        str(s.getpeername())
                    if s in outputs:
                        outputs.remove(s)
                    inputs.remove(s)
                    s.close
                    del messageoq[s]
        # output
        for s in writable:
            (...)
```

SELECT LIMITATIONS

- Select scales to monitoring dozens of sockets (file descriptors, handles, ...)

- For scalable implementation, it’s platform dependent:
  - Linux – epoll
  - OS X/MacOS/BSD – kqueue
  - Windows – iocp

- Painful – solution, Python’s asyncio abstracts all this away!

OPTION 3 – EVENT LOOP USING ASYNCIO

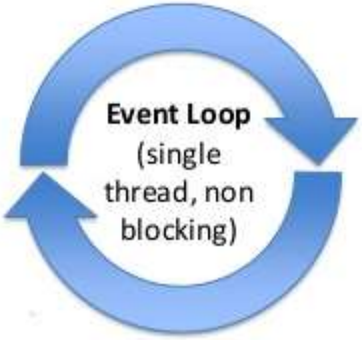
- Use a new process to service each incoming client. Frees up server so it’s non-blocking:

```python
# Simple, asyncio/event loop echo server
from socket import *
import asyncio

# Define asynchronous function
async def echo_server(address):
    # Create a socket, type IPv4, TCP
    sock = socket(AF_INET, SOCK_STREAM)
    # Set socket options
    # * Allow binding to address even if in use (also
    #   bypass TIME_WAIT delay)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    # Bind socket to passed address, port tuple
    sock.bind(address)
    # Listen for incoming connections
    # * Use a queue that can hold up to 5 pending
    #   connections
    sock.listen(5)
    # Must make socket non-blocking
    sock.setblocking(False)
    while True:
        # Use sock_accept from asyncio library
        # * Must "asynchronously wait" for it as it
        #   blocks (and async function)
        client, addr = await loop.sock_accept(sock)
        loop.create_task(echo_handler(client, addr))
```

```python
async def echo_handler(client, addr):
    print('Connection from {}'.format(addr))
    # Use context manager for socket "client":
    with client:
        # While not EOF (more data possible), loop
        while True:
            # Receive available data transmitted from
            # client
            data = await loop.sock_recv(client, 10000)
            # If no more data (EOF/EOT), break out of
            # loop
            if not data:
                break
            # Format data string
            data = 'Received:  '.encode('ascii') + data
            # Echo back to client
            await loop.sock_sendall(client, data)
    print('Connection from {} closed'.format(addr))

# Create "handle" to event loop
loop = asyncio.get_event_loop()
# Create running instance of fib_server
loop.create_task(fib_server(('', 25000)))
# Start event loop
loop.run_forever()
```

EXAMINING EVENT LOOPS AND ASYNC

- Event Loops work well when waiting for lots of I/O
  - » networking
  - » GUI



- Popular Uses:
  - » web browsers
  - » GUI/graphics toolkits
  - » highly scalable web servers (nginx)
  - » chat applications

A FEW MORE WORDS

- Multi-Threading Caveats – What’s Wrong?

```
# Broken Multi-threaded fibonacci server
#
from fib import fib
import time
from threading import Thread

def dispatcher(flist):
    dthreads = []
    for fnum in flist:
        t = Thread(target=fib, args=(fnum,))
        dthreads.append(t)
        t.start()

myfibs = [36, 36, 36, 36]
start = time.time()
dispatcher(myfibs)
end = time.time()
print('Ellapsed time:  {}'.format(end - start))
```

A FEW MORE WORDS

- Multi-Threading Caveats – Fixed:

```
# Multi-threaded fibonacci server
#
from fib import fib
import time
from threading import Thread

def dispatcher(flist):
    dthreads = []
    for fnum in flist:
        t = Thread(target=fib, args=(fnum,))
        dthreads.append(t)
        t.start()
    # Wait for all threads to finish
    for t in dthreads:
        t.join()

myfibs = [36, 36, 36, 36]
start = time.time()
dispatcher(myfibs)
end = time.time()
print('Ellapsed time:  {}'.format(end - start))
```

GREAT RESOURCES

- Raymond Hettinger's Thinking about Concurrency Talk
  - Raymond is a core Python developer
  - He wrote many of the standard libraries
  - Learn from the master
  - Notes from his talk
- Philip Roberts: What the heck is the event loop anyway? | JSConf EU 2014
- A Web Crawler With asyncio Coroutines by A. Jesse Jiryu Davis and Guido van Rossum

QUESTIONS

@sockduct

github.com/sockduct

ADDITIONAL RESOURCES USED

- Common asynchronous patterns in Python - PyOhion 2016
- Fear and Awaiting in Async - Beazley, PyOhio 2016 Keynote
- Shahriar Tajbakhsh - Parallelism Shootout: threads vs asyncio vs multiple processes
- Tulip: Async I/O for Python 3 by Guido van Rossum at Twitter
- Guido van Rossum on Tulip (January 2014)
- A. Jesse Jiryu Davis: What Is Async, How Does It Work, And When Should I Use It? - PyCon 2014
- Keynote David Beazley - Topics of Interest (Python Asyncio)
- David Beazley - Python Concurrency From the Ground Up: LIVE! - PyCon 2015
- Yury Selivanov - async/await in Python 3.5 and why it is awesome
- Understanding the Linux Kernel, 3rd Ed, Chapter 3 - Processes, Lightweight Processes and Threads
- Windows Internals, 6th Ed, Chapter 5 - Processes, Threads and Jobs
- Code Podcast, Episode 1 – Concurrency, Episode 2 – Concurrency – CSP & Actors, Episode 3 – Concurrency – Event Loop & Coroutines
- Software Engineering Radio Podcast, Episode 12 – Concurrent Part 1, Episode 19 – Concurrency Part 2, Episode 29 – Concurrent Part 3