

ELEC / COMP 177 – Fall 2015

Computer Networking

→ Parallel Network Programming

Some slides from Kurose and Ross, *Computer Networking*, 5th Edition

Upcoming Schedule

- **Project 2 – Python HTTP Server v2**
 - Starts today!
- Checkpoint 1 - **Due Oct 4th**
- Checkpoint 1 - **Due Oct 11th**
- Final Project - **Due Oct 18th**

Upcoming Schedule

■ Presentations on Thursday

1. **Craig Goble** - XMPP (Jabber)
2. **Taylor Yatogo** - Internet of Things (CoAP)
3. **Tyler Fernandez** - Secure Shell (SSH)
4. **Kenneth Thompson** - Building Automation and Control Networks (BACnet)
5. **Marcus Barnes** - Real Time Messaging Protocol (RTMP)
6. **Yunpeng Zhang** - Post Office Protocol (POP)
7. **Alexander Murray** - BitTorrent Protocol

Parallel Network Programming

Concurrency

- **Survey:**
 - Who has done parallel programming before?
 - What did you do?

Concurrency

- **Why do I need concurrency in a web server?**
 - **Many clients making requests in parallel**
 - What if several clients each attempt to download a large file?
 - Ugly to make everyone wait on the first user to finish
 - Eventually other clients would timeout and fail
 - A multi-CPU server should use all its resources (multiple cores) to satisfy multiple clients

Goals

MAXIMIZE

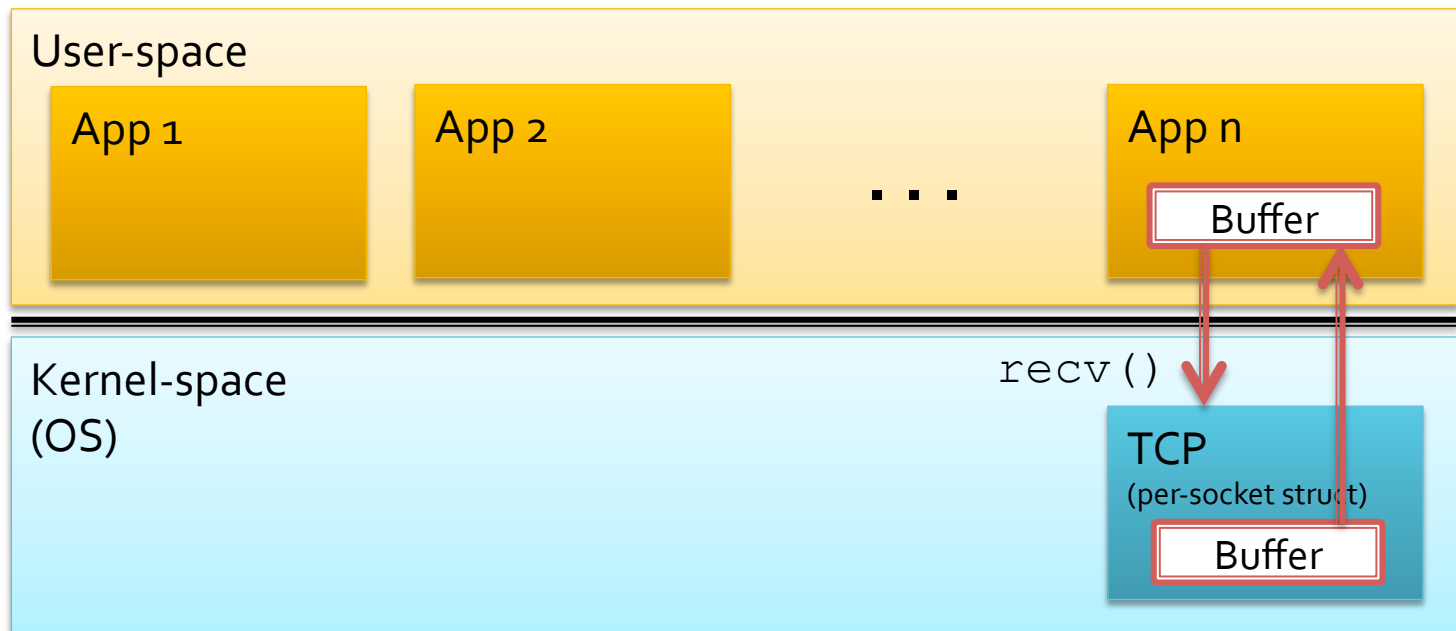
- Request throughput (#/sec)
- Raw data throughput (Mbps)
- Number of concurrent connections

MINIMIZE

- Response times (ms)
- Server CPU utilization
- Server memory usage

Socket `recv()`

- We'll use the `recv()` function for today's examples



`recv()` copies data from kernel space to user-space.
If data is available, the function returns immediately with data

Blocking -vs- Non-Blocking

`recv()` copies data from kernel space to user-space.
If data is available, the function returns immediately with data

BLOCKING

- ***Standard*** mode
- When your program calls `recv()`, if no data is available, the OS puts your program to **sleep**
- Your program is “blocked” on `recv()`

NON-BLOCKING

- ***Special*** mode for many socket calls, including `recv()`
- When your program calls `recv()`, if no data is available, `recv()` **immediately returns**

Synchronous -vs- Asynchronous

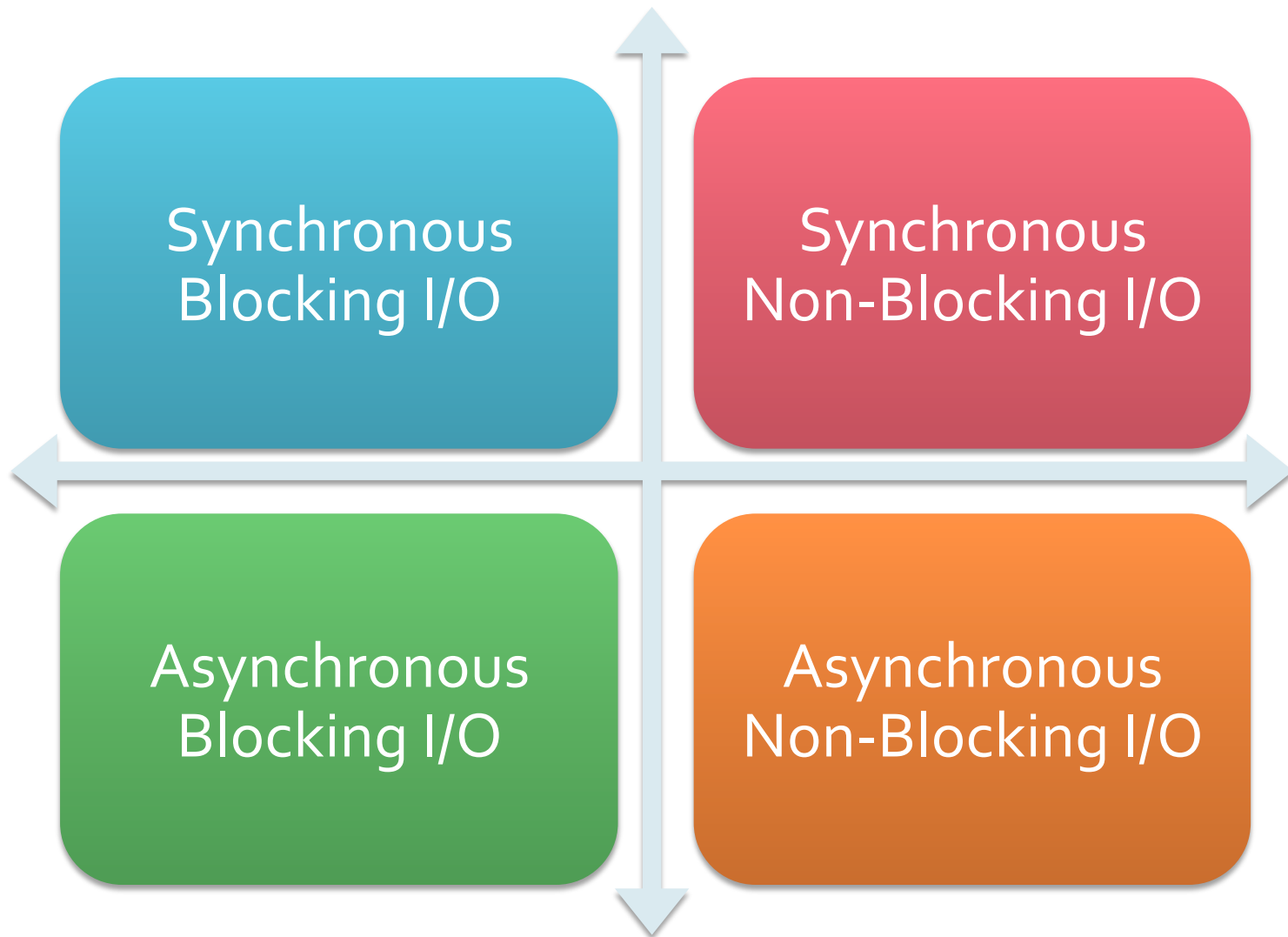
SYNCHRONOUS

- “With Synchronization”
- One operation at a time...
- Function calls to OS services do not return until action is complete

ASYNCHRONOUS

- “Without Synchronization”
- Function calls to OS services return immediately, while OS action can proceed independently of user program

Combine Methods



Synchronous Blocking I/O

- Program requests data from OS
- `recv()` only returns once data is available
- Works fine for managing one socket
 - **How about *two* sockets with different clients?**

Pseudo-code:

```
data = socket1.recv()  
# Data now available
```

Synchronous Non-Blocking I/O

- Program requests data from OS
- `recv()` will return immediately, but may not have any data
- Busy-wait loop wastes CPU time
- How would this work if we had *two* sockets to manage?

Pseudo-code:

```
socket1.blocking(off)
data = socket1.recv()
while(!data)
    data = socket1.recv()

# Data now available
```

Asynchronous Blocking I/O

- `recv()` still blocking
- Busy-wait loop replaced with **new** `select()` **function** that **tests multiple sockets at once**
- Give `select()` separate list of sockets
 - Want to `recv()`
 - Want to `send()`
 - Check for error

Pseudo-code:

```
list_recv = (socket1)
list = select(list_recv)
ready_sock = list[0]
data = ready_sock.recv()
# Data now available
```

- `select()` returns the subset of lists that are **ready** (for `send/recv/err`)
- Not the most efficient function...

Asynchronous Non-Blocking I/O

- `recv()` returns immediately
- In background, OS performs `recv()` work
- When ready, OS calls a “callback” function in your program

Pseudo-code:

```
data = socket.q_recv(done)
# Do something else
# in program

fun done()
    # When called, data
    # is available
```

Processes -vs- Threads



What's the difference?

Processes -vs- Threads

PROCESSES

- Use multi cores/CPU's
- **Separate memory** space
- Can communicate with other processes only by **IPC** (inter-program comm.)
- **"Safer"** to program (other processes can't hurt you)
- **"Heavy-weight"** - Slower to start a new process (lots of OS work)

THREADS

- Use multi cores/CPU's
- **Same memory** space
- Can communicate with other threads by **shared memory**
- **"Harder"** to program (other buggy threads can easily corrupt your memory + synchronization is hard!)
- **"Light-weight"** - Fast to start a new thread (minimal OS work)

Processes -vs- Threads

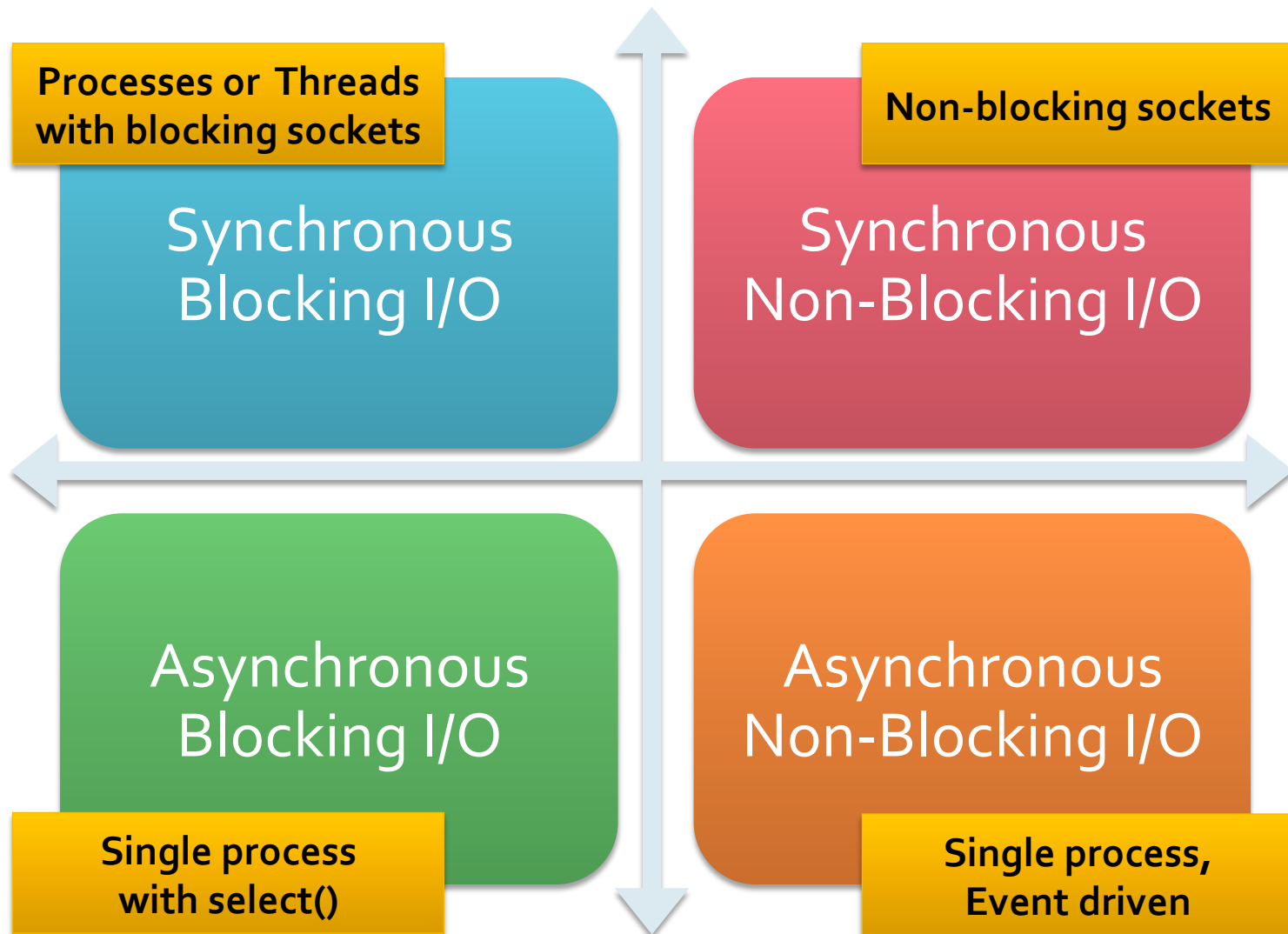
PROCESSES

- Slow start?
 - Typical servers start a “pool” of processes when launched
 - Requests are quickly assigned to an already-running process when received
- Shared data?
 - Need to use OS IPC mechanisms to communicate
 - Needed to assign requests to processes, store log data from processes to single file, ...

THREADS

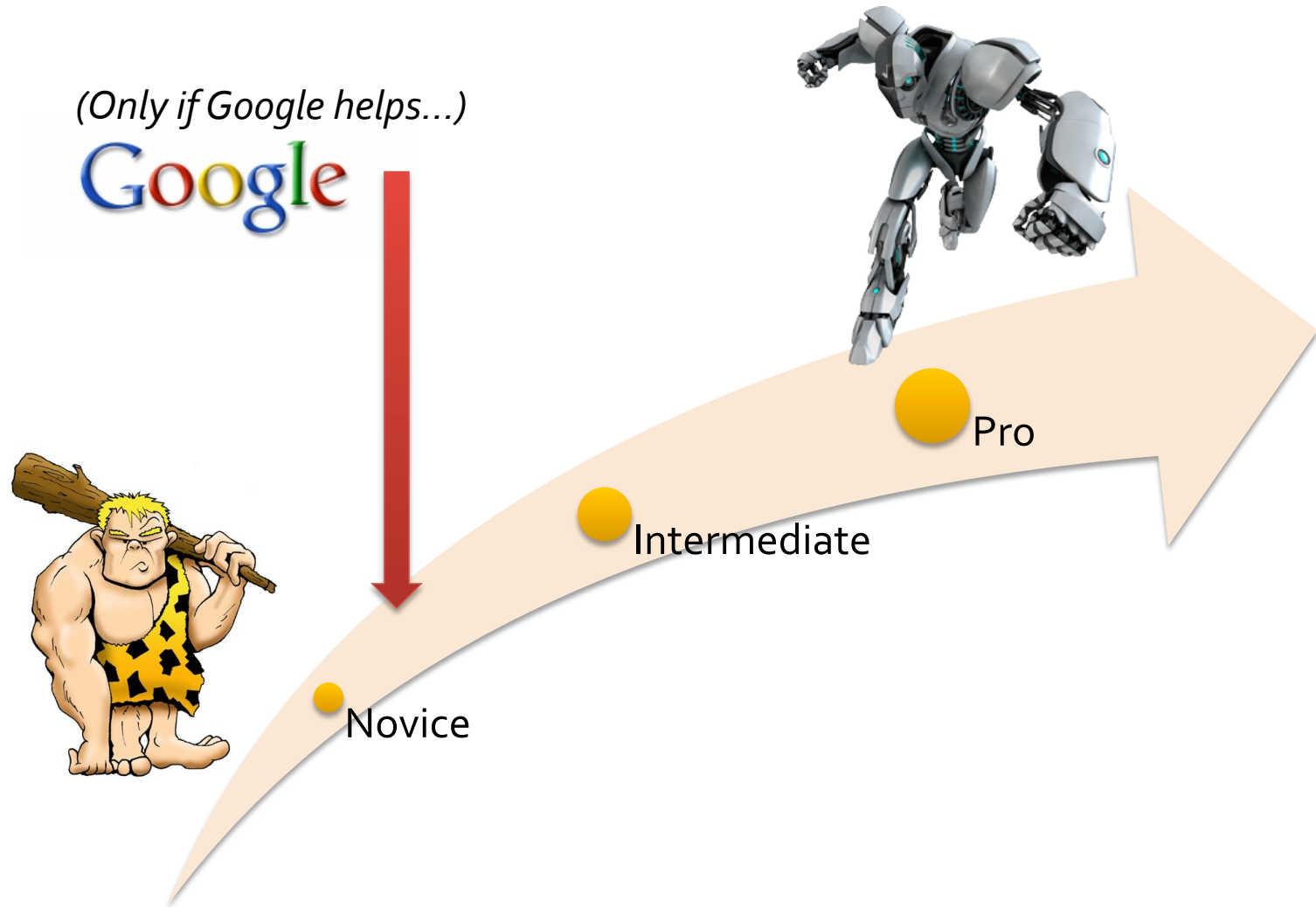
- Fast start?
 - OK to start threads “on demand”
- Shared data?
 - Need synchronization (locks, semaphores, etc...) to prevent corruption of shared data

How to Support Concurrency?



And now, a note
about Python...

My Skill Level in Python



So before assigning class projects, I wrote a Python web server using **threads**.

Once working, I measured its performance...

Results were “sub optimal”

Казахстан. Космодром Байконур

РОССИЯ 24

Not this bad, but it certainly did not scale well as the number of concurrent clients increased...

Пострадавших и разрушений при падении ракеты «Протон-М» нет

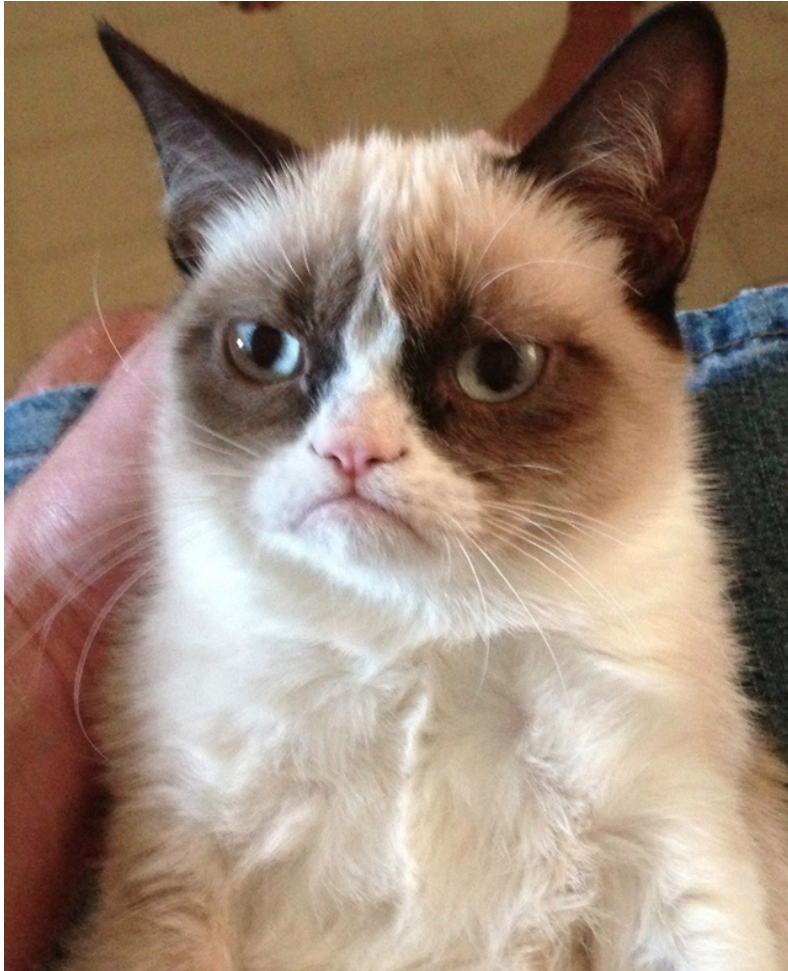
Threads in Python

- Python is an **interpreted language**
 - Several different interpreters exist...
 - **Most common interpreter** is written in C (“CPython”)
- CPython has a global lock
(GIL = Global Interpreter Lock)
 - Lock prevents two threads from running in the interpreter and manipulating memory at same time
 - Allows interpreter to run safely (correctly), perform garbage collection, etc...

Threads in Python

- Effect of GIL (lock) on concurrency
 - I can have multiple threads working on OS-related tasks (`send`, `recv`, ...) in parallel
 - But the GIL blocks multiple threads from running Python native code concurrently 😞
 - See: <http://www.dabeaz.com/python/UnderstandingGIL.pdf>
- So, while the Python *language* has nice threads, the CPython *implementation* limits the performance benefit

Threads in Python



- **Perfectly OK to use threads for class projects**
 - Educational
 - **Good practice** for other languages!
 - Server code will look elegant
- Just don't expect a massive performance boost from parallelism