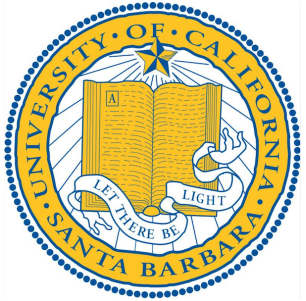
CS 290B

Scalable Internet Services

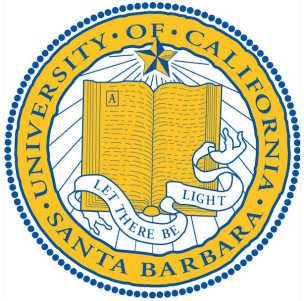
Andrew Mutz

October 14, 2014



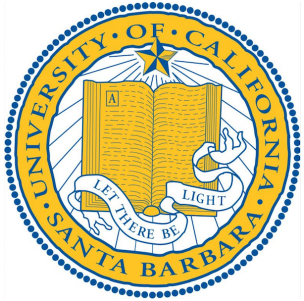
Today's Agenda

- Motivation
- HTTP Servers
- Application Servers
- For next time...

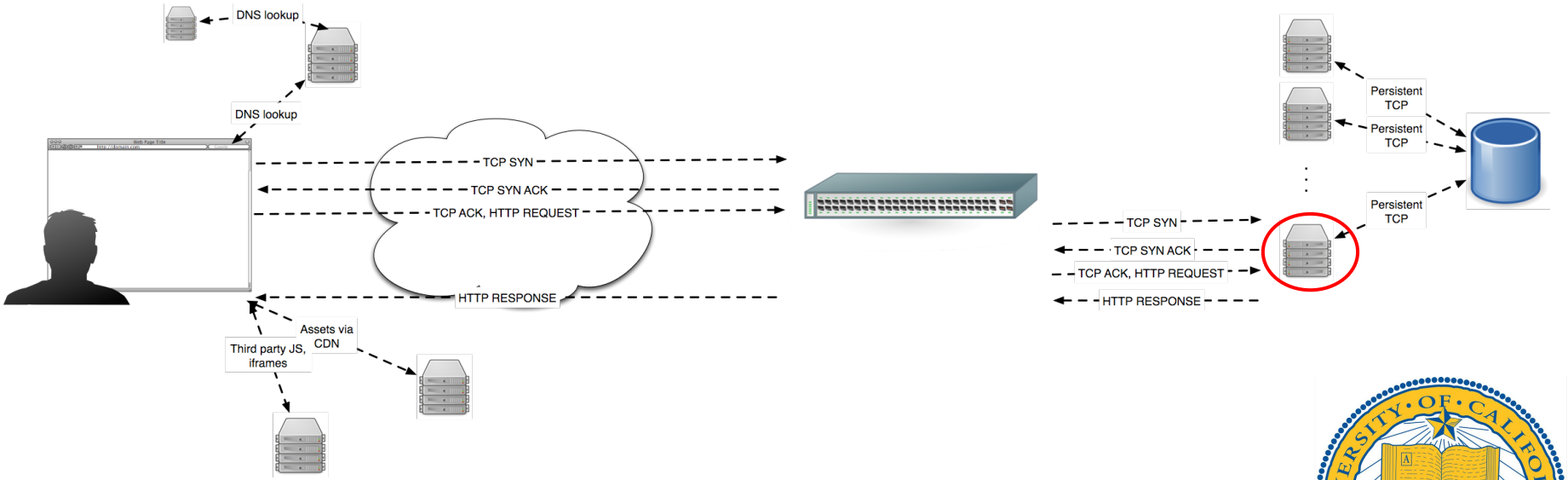


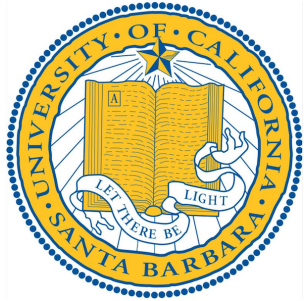
Quick Announcement

Tomorrow's lab (and all future labs) will be in Phelps 3525!



Motivation


DNS lookup
DNS lookup
TCP SYN
TCP SYN ACK
TCP ACK, HTTP REQUEST
HTTP RESPONSE
Assets via CDN
Third party JS, iframes
Persistent TCP
Persistent TCP
Persistent TCP
TCP SYN
TCP SYN ACK
TCP ACK, HTTP REQUEST
HTTP RESPONSE


UNIVERSITY OF CALIFORNIA
SANTA BARBARA
LET THERE BE LIGHT

Motivation

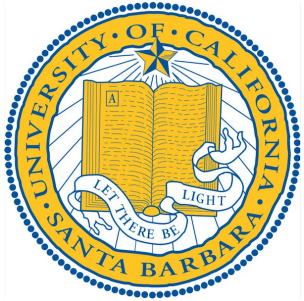
We've seen the HTTP protocol.

The world is full of browsers, apps & other clients that expect to be able to

- Open a TCP socket
- Send over a request (verb & resource)
- Have the request processed
- Receive data in a response
- Reuse the socket for multiple requests

The software systems that do this are generally divided into two parts

- HTTP Servers
- Application Servers



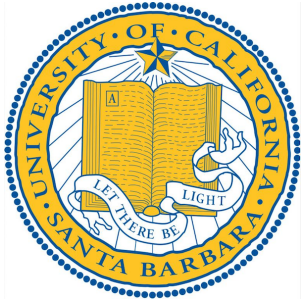
Motivation

Why not just have a single process that handles all this?

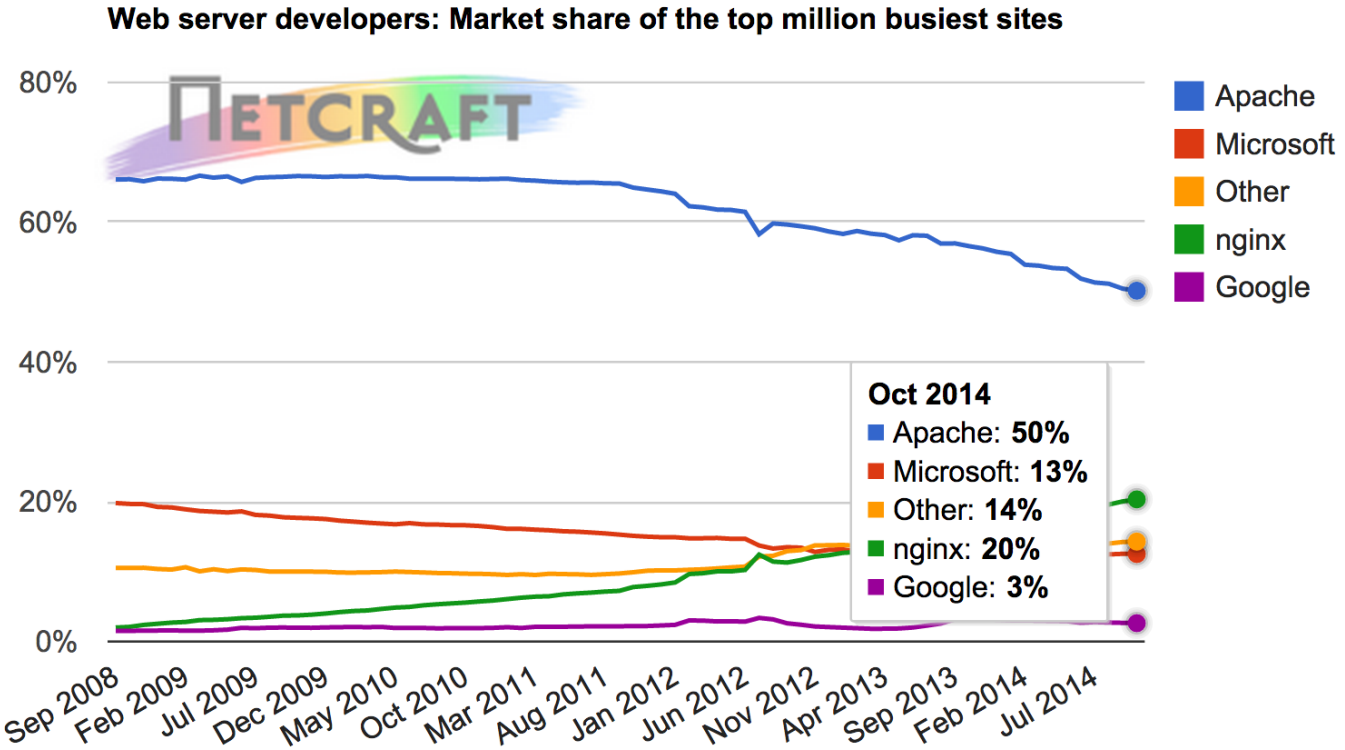
Why do we need two separate notions of an HTTP server and an App server?

The general answer is the two have separate concerns and separate design goals.

- HTTP Server:
  - High performance HTTP implementation
  - Stable, secure, relatively static
  - Highly configurable and language/framework agnostic
  - Concurrency concerns dealt with here (mostly)
- App Server:
  - Specific language, frequently lower-performance
  - Contains business logic and is very dynamic
  - More concerned with optimizing human resources
    - Commonly a large MVC architecture



HTTP Servers



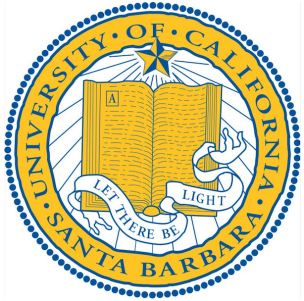
HTTP Servers

HTTP Server's responsibilities:

- Parse HTTP requests and and craft HTTP responses very fast
- Dispatch to the appropriate handler and return response
- Be stable and secure
- Provide clean abstraction for backing applications

Many possible ways to architect an HTTP server:

- Single Threaded
- Process per request
- Thread per request
- Process/thread worker pool
- Event-driven

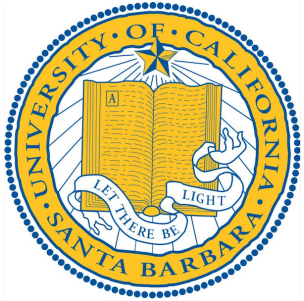


HTTP Servers - Single Threaded

Single threaded approach:

- Bind() to port 80 and listen()
- Loop forever and...
  - Accept() a socket connection
  - While we can still read from it
    - Read a request
    - Process that request
    - Write response
  - Close connection

If another request comes in before we get back around to accept() another, what happens?

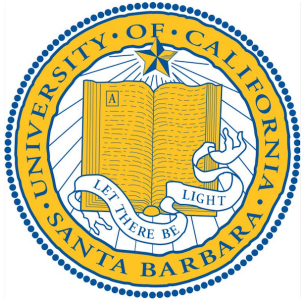


HTTP Servers - Single Threaded

Problem!

- If we don't quickly get back to accepting more connections, clients end up waiting or worse
- We are building web applications, not web sites:
  - These requests are usually much more than simply serving a file from disk
  - It is common to have a web request doing a significant amount of computation and business logic
  - It is common to have a web request talk to multiple external services: databases, caching stores, SOA services
  - These requests can be anything: lightweight or heavyweight, IO intensive or CPU intensive

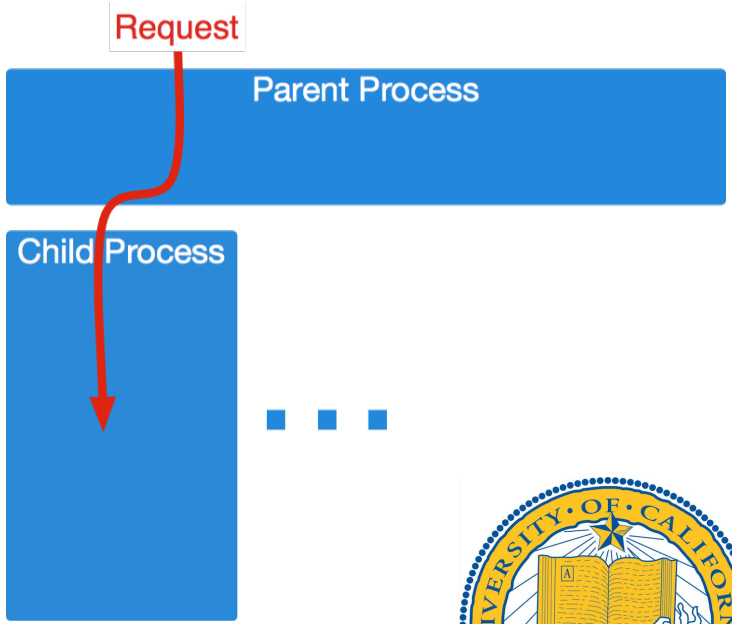
We can solve these problems if the thread of control that processes the request is separate from that listening and accepting new connections.



HTTP Servers - Process Per Request

Why not handle each requests as a subprocess?

- Bind() to port 80 and listen()
- Loop forever and...
  - Accept() a socket connection
  - if fork() == 0
    - While we can still read from it
      - Read a request
      - Process that request
      - Write response
    - Close connection, exit



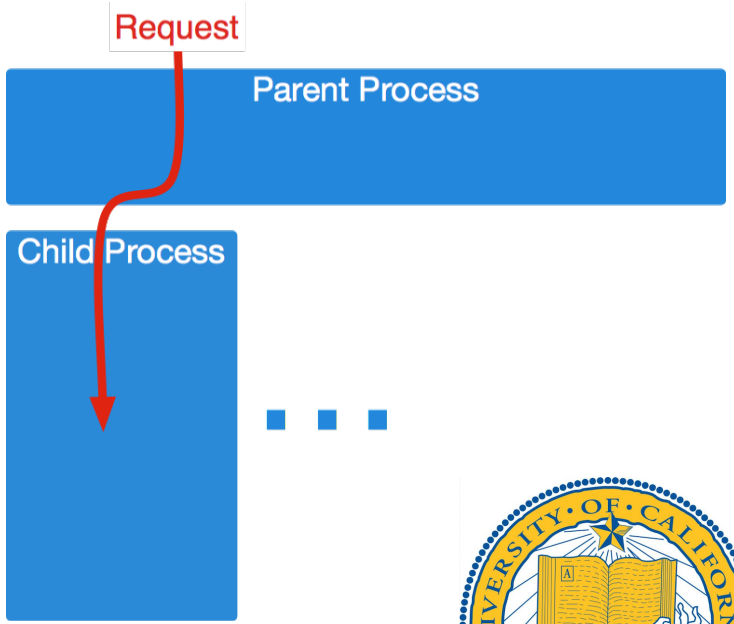
HTTP Servers - Process Per Request

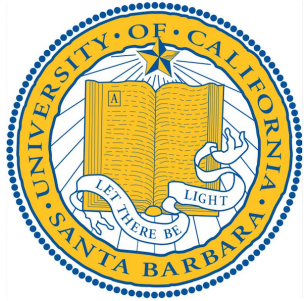
Strengths:

- Simple
- Great isolation between requests
- No problems with multiple threads

Weaknesses:

- Does each request duplicate process memory?
- What happens when load keeps rising?
- Is it efficient to be firing up a process on each request?
    - Each of these does setup work



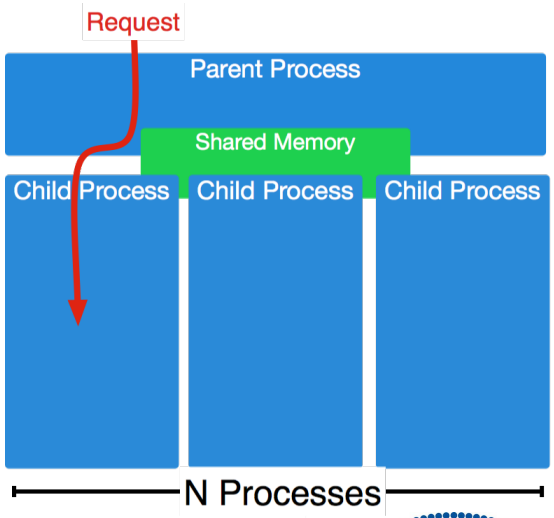


HTTP Servers - Process Pool

Instead of spawning a new process each time we get a request, we can create a pool of N processes at the beginning and dole out requests to them.

The children are responsible for accepting incoming connections, and use shared memory to coordinate.

The parent process watches the level of busy-ness of the children and adjusts the number of children as needed.



HTTP Servers - Process Pool

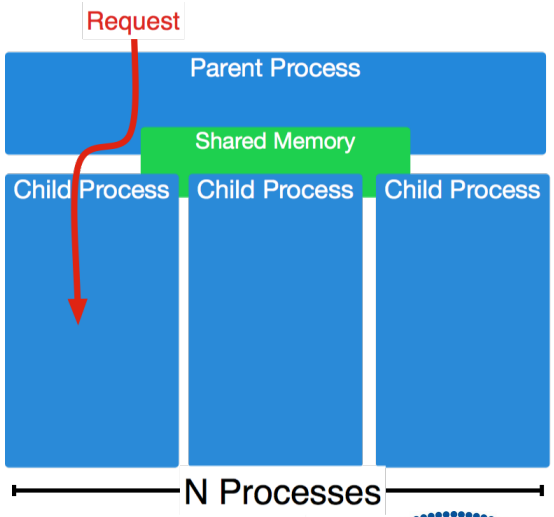
Strengths:

- Great isolation between requests. Children die after M requests to avoid memory leakage.
- Process startup/setup costs are avoided
- More predictable behavior under high load.
- Still no problems with multiple threads

Weaknesses:

- System more complex than before
- Many processes can mean a lot of memory consumption

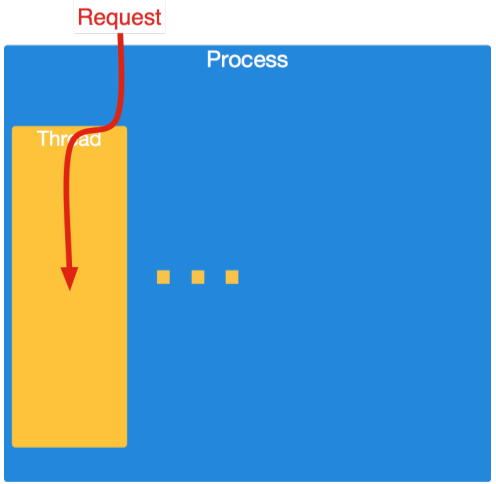
This basic structure is Apache 2.x MPM “Prefork”.

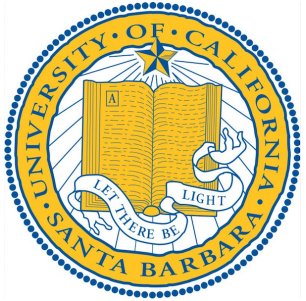


HTTP Servers - Thread per request

Why use multiple processes at all? Why not just have a single process, and each time we get a new connection we spawn another thread?

- Bind() to port 80 and listen()
- Loop forever and...
  - Accept() a socket connection
  - pthread_create a function that will...
    - While we can still read from it
      - Read a request
      - Process that request
      - Write response
    - Close connection, thread dies





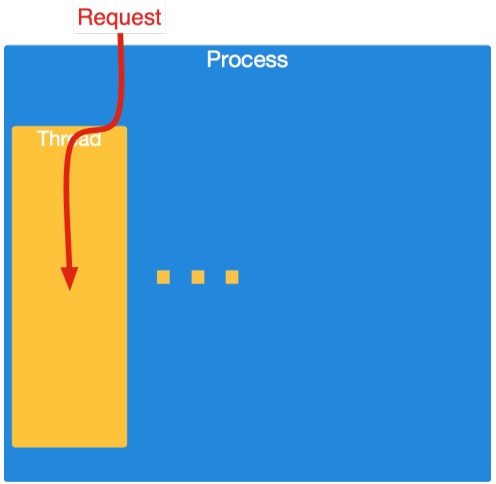
HTTP Servers - Thread per request

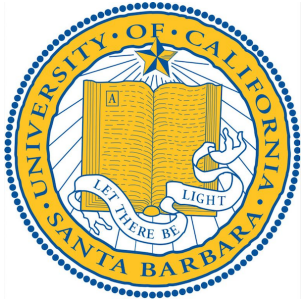
Strengths:

- Fairly simple
- Memory footprint is reduced versus processes

Weaknesses:

- The code handling each request must be thread safe
- Pushing thread-safety on to the application developer isn't ideal
- Setup (database connections, etc.) needs to happen each time





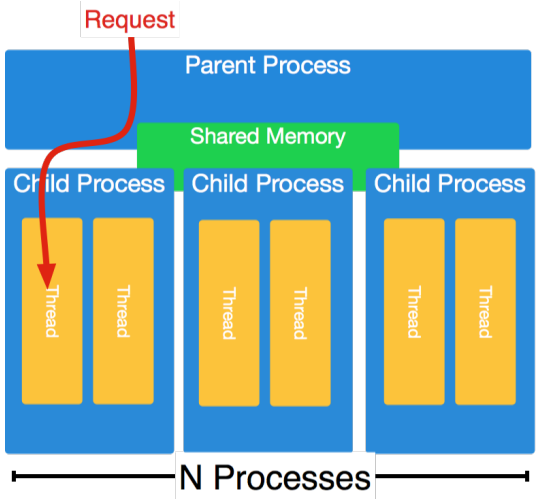
HTTP Servers - Process/Thread Pool

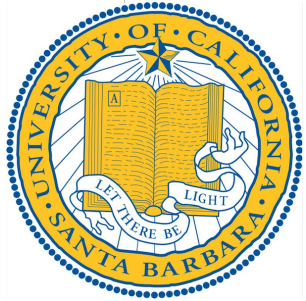
Can we see benefit from combining these techniques?

Master process spawns processes, each with many threads. Master maintains process pool.

Processes coordinate through shared memory to accept requests.

Fixed threads per request, scaling is done at the process level.





HTTP Servers - Process/Thread Pool

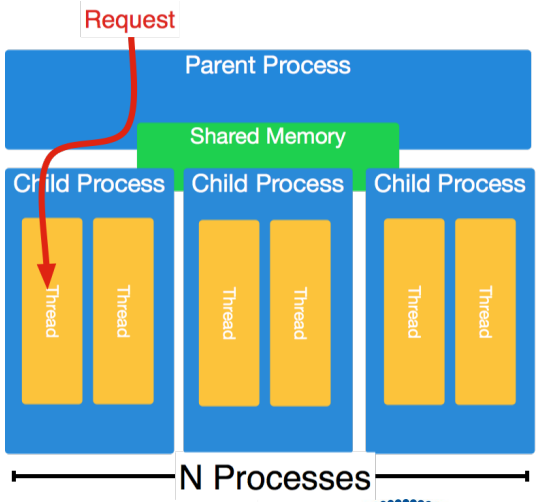
Strengths:

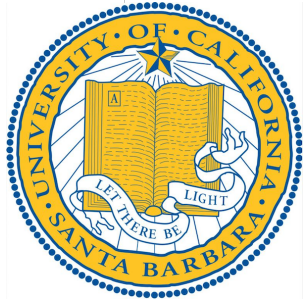
- Faults isolated between processes, but not threads
- Threads reduce our memory footprint and we still get a tuneable level of isolation
- Controlling the number of processes and threads allows predictable behavior under load

Weaknesses:

- Need thread-safe code
- Uses more memory than an all-thread based approach

This is Apache 2.x MPM “Worker”

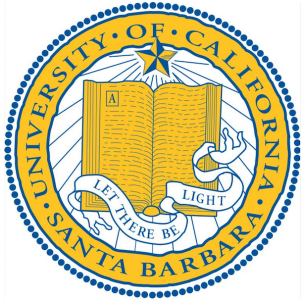




HTTP Servers

Next we will discuss event-driven architectures and nginx.

But first, a thought experiment: the C10K problem.



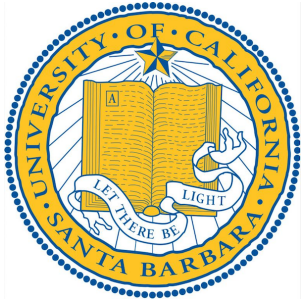
HTTP Servers

C10K Problem, originally posed in 2001

- Given a 1ghz machine with 2gb of RAM, and a gigabit ethernet card, can we support 10,000 simultaneous connections?
  - 10,000 clients means…
  - 100Khz CPU, 200Kbytes RAM, 100Kbits/second network for each
  - Shouldn't we be able to move 4kb from disk to network once a second?

This is difficult, but it seems like it shouldn't be.

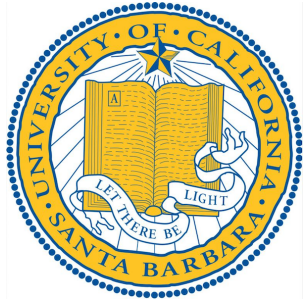
What are we spending time doing?



HTTP Servers

Lets say I've got 10K connections. Each is doing something like this:

```
Read from the network socket
Parse the request
Open the correct file on disk
Read the file into memory
Write the memory to network
```



HTTP Servers

Lets say I've got 10K connections. Each is doing something like this:

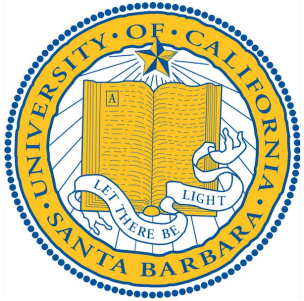
```
Read from the network socket (system call - WAIT)
Parse the request
Open the correct file on disk (system call - WAIT)
Read the file into memory (system call - WAIT)
Write the memory to network (system call - WAIT)
```

HTTP Servers

Each time I'm waiting on I/O, I'm not runnable, but I'm not cost-free.

- I need to be considered every time the scheduler does anything.
- Before I waited, my memory accesses pushed others' data out of caches

This massive concurrency slows down all processes.

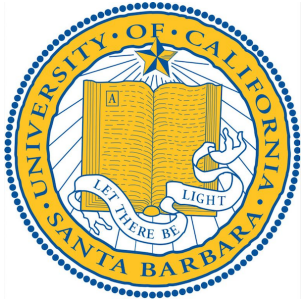


HTTP Servers

Since much of these problems have their root in these blocking system calls, can we accomplish all the same tasks without blocking?

Yes, with asyncronous io:

- select(): Here is a list of file descriptors. Block until ready for IO.
- epoll_*(): Lets keep a list of FDs in kernel space. Block until ready.

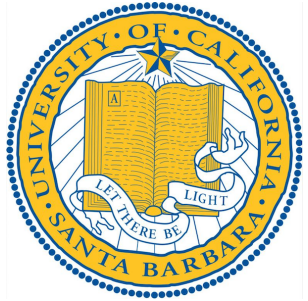


HTTP Servers - Event Driven

Let's say we have a list of sockets called `fd_list`

```
loop forever:
    select(fd_list, ...)  //block until one of this list is ready
    for each fd in fd_list
        if fd is ready for IO
            some_handler(fd)
        else do nothing.
```

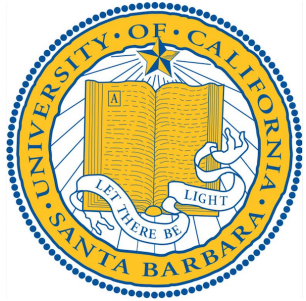
- `some_handler` can include socket acceptance.
- `some_handler` absolutely can't do blocking IO.
  - How do we handle this IO?
- What do we do if `some_handler` is doing a lot of computation?



HTTP Servers - Event Driven

These systems are called event driven systems.

- Only need a single thread (although can support more)
- Well known examples:
  - nginx
  - Tengine
  - LightTPD
  - netty (java)
  - node.js (javascript)
  - eventmachine (ruby)
  - twisted (python)



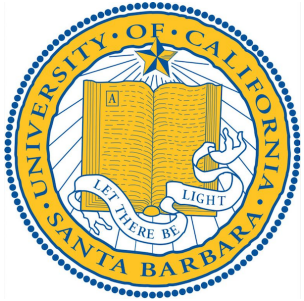
HTTP Servers - Event Driven

Strengths:

- High performance under high load
- Predictable performance under high load
- No need to be thread-proof

Weaknesses:

- Poor Isolation
  - If a bug causes an infinite loop, what happens?
- Fewer extensions, since code can't use blocking syscalls
- Very complex
  - See next slide...

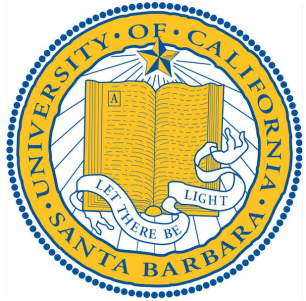


HTTP Servers - Event Driven

Code is dominated by callbacks:

```
EM.run {
  page = EM::HttpRequest.new('http://google.ca/').get
  page.errback { p "Google is down! terminate?" }
  page.callback {
    a = EM::HttpRequest.new('http://google.ca/search?q=em').get
    a.callback { # callback nesting, ad infinitum }
    a.errback  { # error-handling code }
  }
}
```

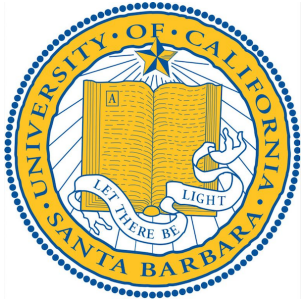
This can lead to code that is confusing and hard to maintain.



HTTP Servers

To recap, there are many possible ways to architect an HTTP server:

- Single Threaded
- Process per request
  - Greatest isolation, largest memory footprint
- Thread per request
  - Smaller memory footprint, less isolation
- Process/thread worker pool
  - Tuneable compromise between processes & threads
- Event-driven
  - Great performance under high load
  - Harder to extend and reduced isolation



Application Servers

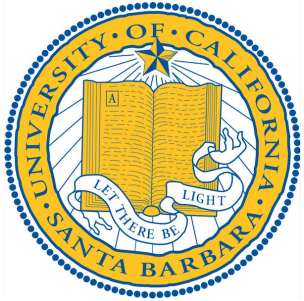
We are building web applications, so we will need complex server-side logic.

We can extend our HTTP servers to do this through modules, but there are benefits to breaking out application servers to a distinct process:

- Application logic will be dynamic, whereas HTTP is more static
- Application logic regularly uses high level (slow) languages vs. needs of high-performance
- Security concerns are easier: HTTP server can shield the app server from some things
- Startup/setup costs can be amortized if the app server is running continuously

Instead, we can have a separate Application server and forward each request to it for handling.

We will be looking primarily at Ruby application servers.

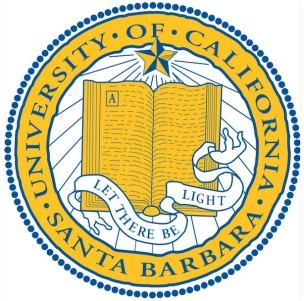


Application Servers

Our HTTP server needs to communicate each request to the App server, and the response needs to be sent back.

How is this done?

- CGI - Spawn a process, pass in HTTP headers as ENV variables
- FastCGI, SCGI - modifications to CGI to allow persistent processes.
- HTTP - Essentially a reverse-proxy configuration
    - Why does it make sense to have an HTTP server in front of a server that speaks HTTP?



Application Servers

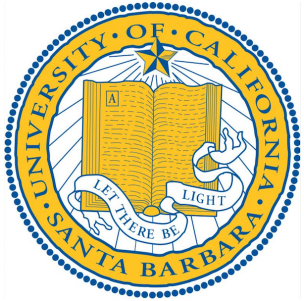
Many of the same questions regarding concurrency haven't gone away:

- Should we handle these requests via processes? Threads? Evented?

If we are using the standard C Ruby interpreter (Matz's Ruby Interpreter), then we have a Global Interpreter Lock to deal with.

- Only one thread of control can be executing in a given Ruby process at a given time
- JRuby has no GIL

The existence of the GIL simplifies things: using threads for concurrency won't get us very far.



Application Servers

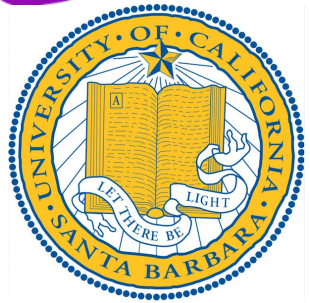
Mongrel

- Administrator sets up a pool of processes running the Mongrel app server
- Mongrel app server speaks HTTP
- Apache or nginx is set up to act as a reverse proxy and load balance between Mongrel processes using (for example) mod_proxy_balancer
- Monit watches pool of mongrels, restarts any that died.

Application Servers

Phusion Passenger

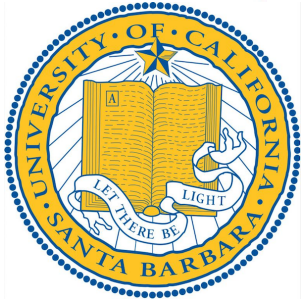
- A passenger module is added to Apache or nginx
- The code running inside the HTTP server knows what it is load balancing and actively controls the size of the pool.
- Two advantages:
  - Simple mechanism to increase/decrease the pool
  - Processes can be forked after ruby/rails is loaded.
    - Why is this good?



Application Servers

Unicorn:

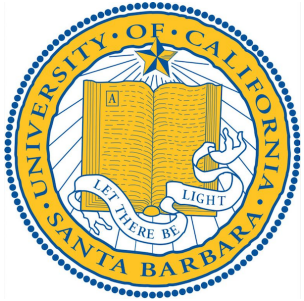
- Similar to passenger in that it manages a pool of processes to handle requests, and can take advantage of CoW.
- Similar to Mongrel, in that it needs* an HTTP server configured for load balancing in front of it
- Advantages over passenger:
  - Better monitoring of workers
  - Supports hot-restarts of code changes



Application Servers

Puma

- What if we can deploy on a Ruby VM without a GIL?
  - JRuby or Rubinius
- If we move away from the GIL, we can avoid process-based parallelism and choose threads instead
- Common setups involve a load balancer in front of multiple Ruby processes, each with multiple threads.
  - We can tune the isolation vs. memory footprint



For Next Time...

- **Tomorrow's lab (and all future labs) will be in Phelps 3525!**
- For Thursday read "Dynamic Load Balancing on Web-server Systems" by Cardellini, Colajanni, Yu
  - http://www.ics.uci.edu/~cs230/reading/DLB.pdf
- By Tomorrow
  - Complete through chapter 6 in AWDR
  - Attempt Bryce's challenges, come to lab with questions.