CS188
Scalable Internet Services

John Rothfels, 10/13/20

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
- Have the request processed
- Receive data in a response
- Reuse the socket for multiple requests

The software systems that do this are often divided into two parts:

- HTTP servers
- Application servers (“appserver”)

Motivation

⁉️ Why not just have a single process that handles everything?

⁉️ What are the benefits of separating an HTTP server and an application server?

Motivation

⁉️ Why not just have a single process that handles everything?

⁉️ What are the benefits of separating an HTTP server and an application server?

HTTP server

- High performance HTTP implementation
- Stable, secure, relatively static
- Highly configurable and language/framework agnostic
- Many concerns dealt with here

Motivation

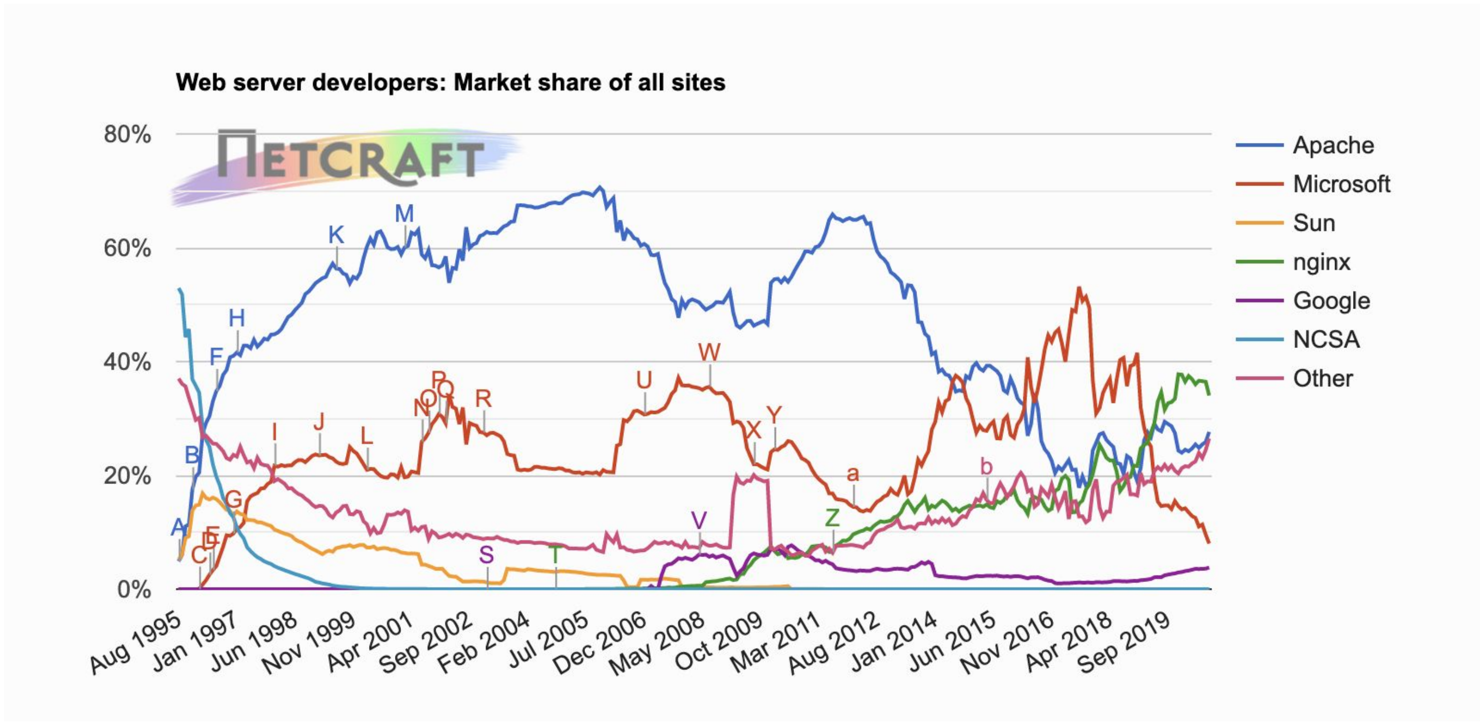
⁉️ Why not just have a single process that handles everything?

⁉️ What are the benefits of separating an HTTP server and an application server?

Application server

- Specific language, frequently lower performance
- Contains business logic and is very dynamic
- More concerned with optimizing human resources
- Commonly a large MVC architecture
    - We will talk a bit about MVC next lecture

HTTP servers



source

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, secure, and 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

Aside: process vs. thread

⁉️ What’s the difference between a **process** and a **thread**?

Aside: process vs. thread

Process:

- Executing instance of an application
- Can contain multiple threads
- Used for heavyweight tasks
- Processes do not share address space

Thread:

- Path of execution within a process
- Can do anything a process can do
- Used for small tasks
- Threads within process share address space ⚠️ 💣

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

😭 **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, APIs
- These requests can be anything: lightweight or heavyweight, IO intensive or CPU intensive

HTTP servers: single threaded

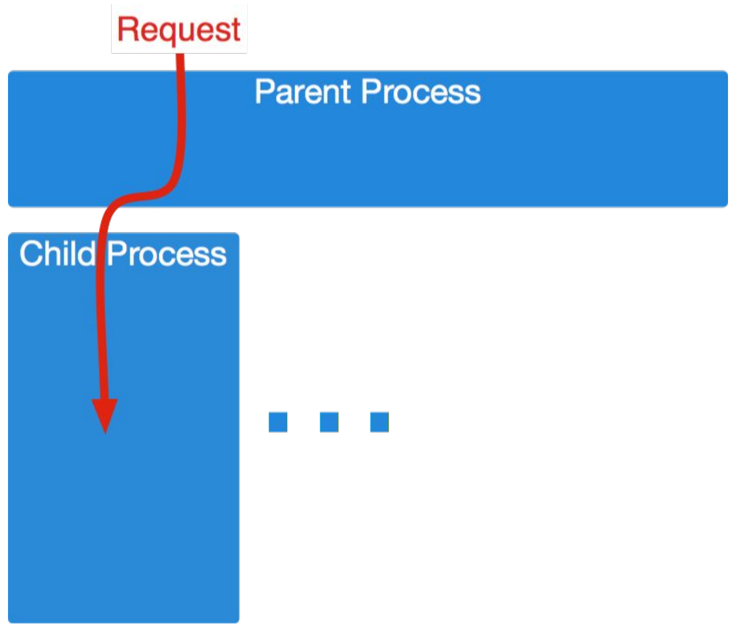
😭 **If we don’t quickly get back to accepting more connections, clients end up waiting or worse!** 😤

We can solve these problems if the thread of control that processes the request is separate from the one 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



Strengths? Weaknesses?

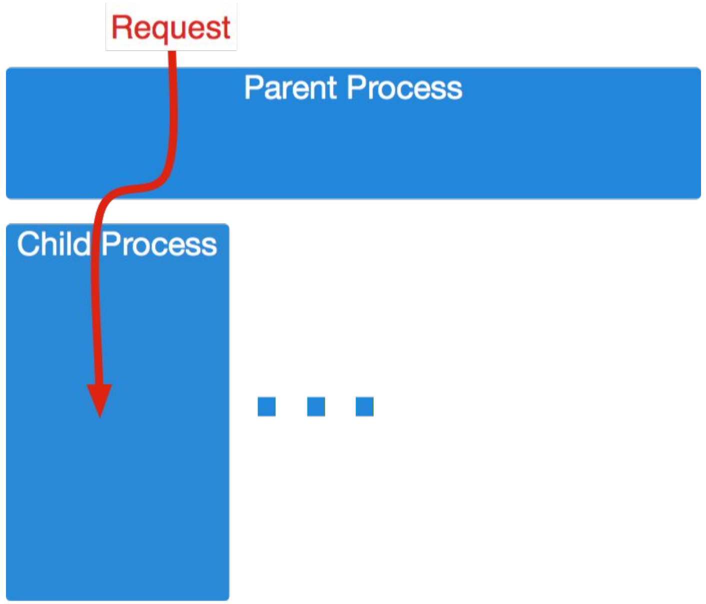
HTTP servers: process per request

Strengths:

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

Weaknesses:

- Does each request duplicate process memory?
- What happens if the load keeps rising?
- Is it efficient to spin up new processes? Each does setup work.

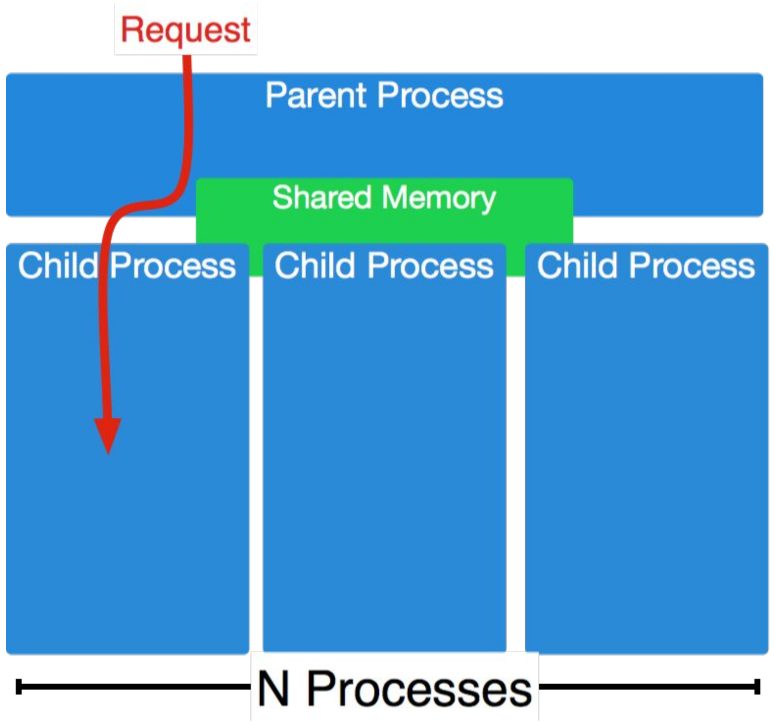


HTTP servers: process pool

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.

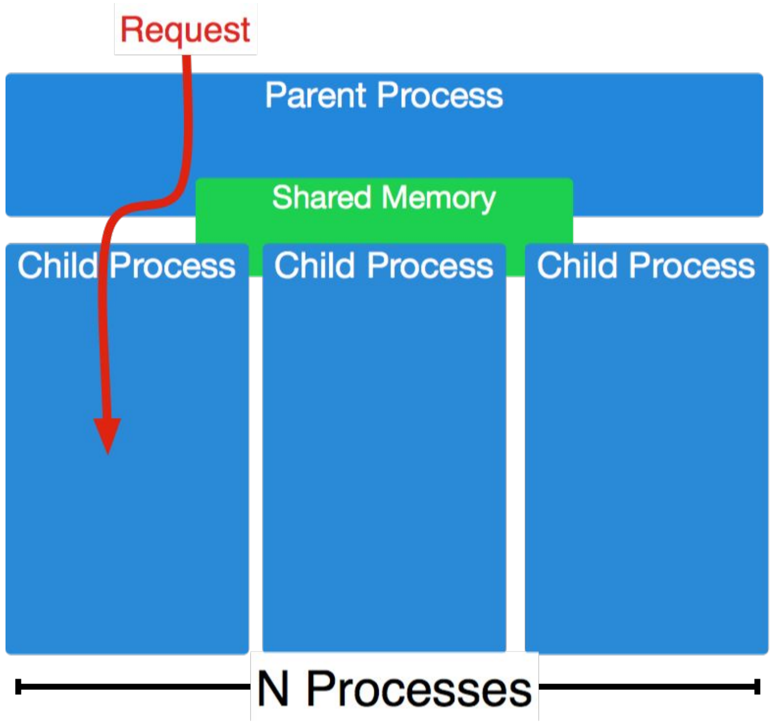
The parent process watches the level of busy-ness of the children and adjusts the number of children as needed.



HTTP servers: process pool

Strengths:

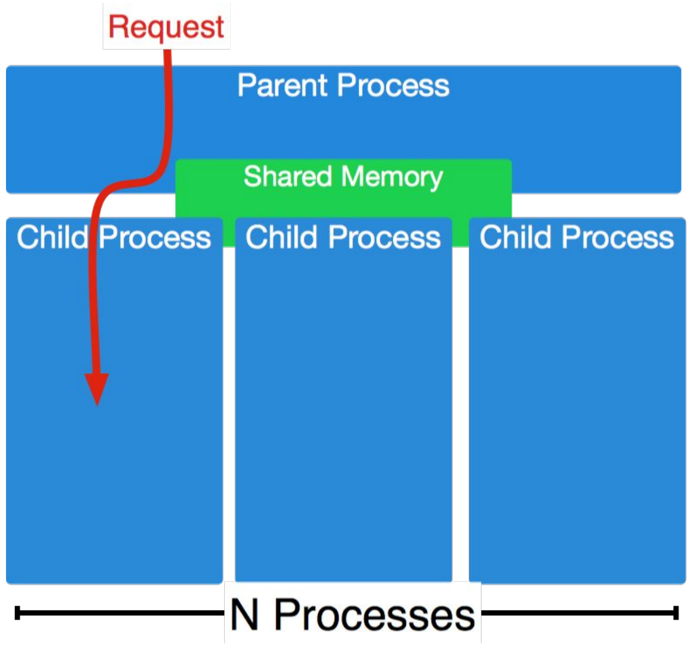
- Great isolation between requests. Children die after M requests to avoid memory leakage
- Process startup/setup costs are avoided
- More predictable behavior under high load
- Still no problems with multiple threads



HTTP servers: process pool

Weaknesses:

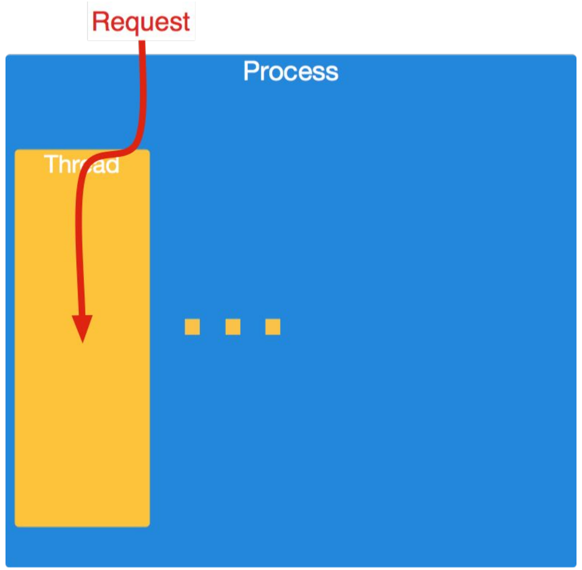
- System more complex than before
- Many processes can mean a lot of memory consumption



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

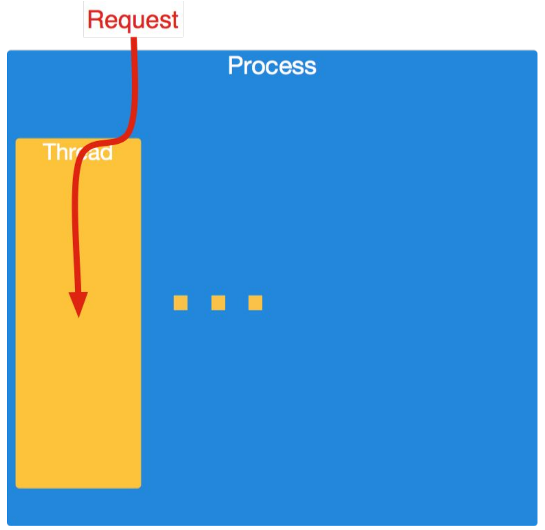


Strengths? Weaknesses?

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.) still needs to happen each time
    - It can be minimized with (language specific) software techniques

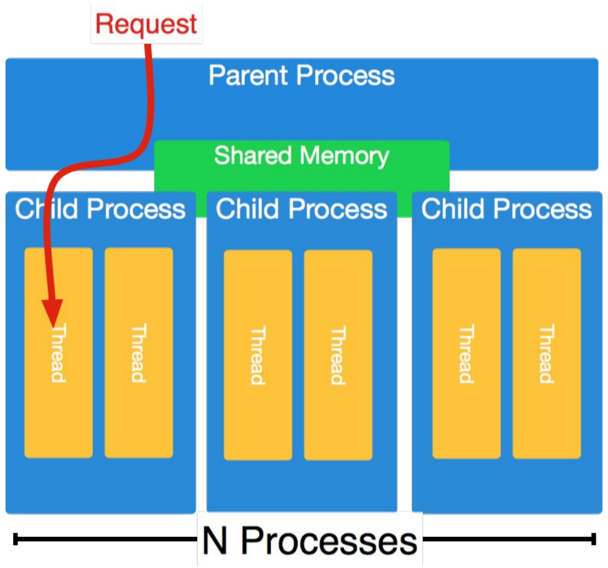
HTTP servers: process/thread pool

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.



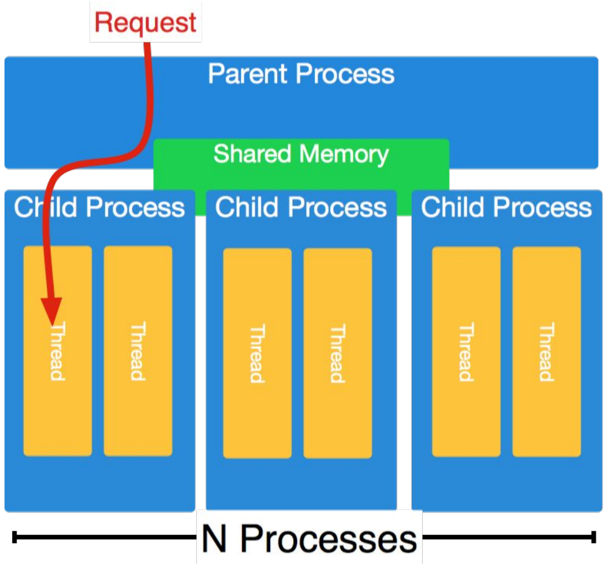
HTTP servers: process/thread pool

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



Aside: the perils of blocking

🤔 A thought experiment: **the C10K problem**

- 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?

Aside: the perils of blocking

Let’s 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
```

Aside: the perils of blocking

Let’s 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)
```

Aside: the perils of blocking

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! 🐢

Aside: the perils of blocking

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

Yes, with **asynchronous** 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.

https://jvns.ca/blog/2017/06/03/async-io-on-linux--select--poll--and-epoll/

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

What do we do if `some_handler` is doing a lot of computation?

HTTP servers: event driven

These systems are called event driven systems. They only need a single thread, but can support more. Examples:

- Nginx
- OpenResty
- 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 you include a library that does blocking syscall what happens?
- Fewer extensions, since code can't use blocking syscalls
- Programming hell...

HTTP servers: event driven

Code is dominated by callbacks 😢

```
app.get('/', (req, res) => {
    fs.readFile(filePath, (err, data) => {
        if (data) {
            ajax('https://twitter.com/api/ ...', (err, data) => {
                res.send(fetchData)
            })
        }
    })
})
```

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
  - Callback hell! 💡 Use JavaScript **`Promise`**s! (see later slides)

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.

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
    - This is the most common for large scale production deployments. It is what we'll be doing in this class for your class project.
    -

⁉️ 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?

In this class, we will we will be building **event driven application servers** that are also **HTTP servers**, using **Node.JS**.

We will take a quantitative look at some alternative approaches in lecture.

Synchronous programming

We’ve talked about **threads**. How do programs (& programming languages) handle the concept of parallel paths of execution?

In most imperative languages (e.g. Java), function invocations may **block** the current thread of execution. When the current thread of execution is blocked, a different thread (that isn’t blocked) can start running. We can **context shift** to this thread.

Synchronous programming

In most imperative languages (e.g. Java), function invocations may **block** the current thread of execution.

```
public void synchronousFunction() {
    var sqlResults = executeSql();
    // this line doesn’t execute until the previous is finished
    var apiResults = fetchFromApi();
    // this line doesn’t execute until the previous is finished
    var fileContents = readFile();
    // the function isn’t done until the file is read
}
```

Synchronous programming

In most imperative languages (e.g. Java), function invocations may **block** the current thread of execution.

```
public void synchronousFunction() {
    var sqlResults = executeSql(); // what happens here?
    // this line doesn’t execute until the previous is finished
    var apiResults = fetchFromApi(); // what happens here?
    // this line doesn’t execute until the previous is finished
    var fileContents = readFile(); // what happens here?
    // the function isn’t done until the file is read
}
```

Asynchronous programming

In most imperative languages (e.g. Java), **blocking** function calls may be run on a different thread. When you run the function on another thread, that thread gives the current thread a **future** value. The value isn't available until the other thread runs and "completes" the **future** value. A **future** is sometimes also called a **promise**.

```
public void asynchronousFunction() {
    var sqlFuture = executeSqlAsync();
    // executes immediately (before sqlFuture starts running)
    var apiFuture = fetchFromApiAsync();
    // executes immediately (before apiFuture starts running)
    var fileFuture = readFileAsync();
    // function ends before any SQL, fetch, or file read happens!
}
```

Asynchronous programming

In JavaScript, functions are **non-blocking** by default! Usually to receive an asynchronous value, you pass a callback:

```
function readFile() {
    fs.readFile(path, (err, data) => {
        // 2. file was read, this callback receives the contents
        console.log('finished reading file')
    })

    // 1. this line executes before #2; fs.readFile does not block!
    console.log('here')
}
```

Asynchronous programming

In JavaScript, **non-blocking** (asynchronous) functions execute their callbacks on an **event loop**. The event loop is a single thread that does the following:

- while(true)
  - if there is event (callback) queued
    - execute the event on the current thread
  - else wait for an event

When an asynchronous function is completed, it queues the provided callback on the event loop. The event loop processes events in the order they are received.

⚠️ Reminder: the event loop is a single thread!

Asynchronous programming

In JavaScript, **non-blocking** (asynchronous) functions can lead to **callback hell**.

Fortunately, future JavaScript provides a syntactic sugar called **`Promise`** which wraps the result of an asynchronous function.

```
function readFile() {
    const filePromise = readFilePromise(path)
    filePromise
        .then(contents => …)
        .catch(err => console.err(err))

    // this line executes before .then() or .catch() callback
    console.log('here')
}
```

Asynchronous programming

Fortunately, JavaScript provides a syntactic sugar called **`Promise`** which wraps the result of an asynchronous function. Unwrap a Promise with `await`.

```
async function readFile(): Promise<boolean> {
    const filePromise = readFilePromise(path)
    const fileContents = await filePromise // “blocks”
    // this line executes after the file is read!
    console.log('here')
    return true
    // async functions *always* return a Promise (implicitly)
}
```

ℹ️ To use `await`, you must be inside an `async` function.

Asynchronous programming

In JavaScript, using **`Promise`** with **`async`**/**`await`** lets us avoid callback hell! 🎉

It makes it **look* like your JavaScript code is blocking…but in reality there is nothing blocking the event-loop thread.

ℹ️ **`await`** effectively puts a callback on the loop that starts at the line following `await`. While waiting for the event, other events may be executing. This gives you the illusion in JavaScript of parallel threads of execution.

For next time

- Email me if you are interested in having a 5th teammate.
- If you did not meet with me to present your project last lab, you will have a chance at the beginning of the next lab. If you can’t make it to next lab, instructions on the course website for submitting your checkpoint by email.