# CS572 Modern Web Applications Programming
# Maharishi University of Management
## Department of Computer Science
## Assistant Professor Asaad Saad







PREV NEXT

OUTLINE

Search...

# Introduction

JavaScript was traditionally the language of the web browser, performing computations directly on a user's machine. This is referred to as "client-side" processing.

With the arrival of **Node.js**, JavaScript has become a compelling "server-side" language as well, which was traditionally the domain of languages like Java, Python, C# and PHP.





PREV NEXT

# Introduction to NodeJS

- Node.js is a server side platform built on Google's JavaScript Engine (V8 Engine).
- Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.
- Used by: Google, Amazon, Microsoft, eBay, General Electric, GoDaddy, PayPal, Uber, Wikipins, Yahoo!

Ryan Dahl





PREV NEXT

OUTLINE

Search...

# ECMAScript Specs

- ECMAScript is the standard JS is based on since there are many engines. ECMA is an organization. A standard that everyone agreed on.
- **ECMAScript 2015 specification**: http://www.ecma-international.org/ecma-262/6.0/
- Basically you can write your own JS engine based on those specifications.
- **JS Engine**: A program that convert JS code into something that computer processor understands, it should follow the ECMAScript standard and how the language should work and what features it should have.

# Other JS Engines VMs

- **Google**: V8
- **Mozilla**: Spider Monkey
- **Microsoft**: Chakra Core
- **Apple**: JavaScript Core





PREV NEXT

# NodeJS in brief

- Server-side JavaScript
- Built on Google's V8 (Supports other VM like Chakra)
- Evented, non-blocking I/O
- CommonJS module system
- Allows script programs do I/O in JavaScript
- Focused on Performance

```
100 concurrent clients
1 megabyte response

node     822 req/sec
nginx    708
thin     85  (Ruby)
mongrel  4   (Ruby)

(bigger is better)
```

# Process vs Thread

- When you execute a computer program, an instance of this program will live in the memory, we call it: **Process**.
- Every process may have multiple threads. A **thread** is some sort of instructions to be executed by the CPU (todo list).
- **OS scheduler**: scheduling is the logic the OS uses to decide which thread to process at any given time. Urgent thread should be processed quickly.
- To improve scheduling threads we can add more CPU cores, however one core can process multiple threads by **Hyper-threading** feature (examining the work in threads and pause one when an expensive IO operation occur)

# I/O

A communication between CPU and any other process external to the CPU (memory, disk, network)

How to handle I/O:

- Synchronous code (slow)
- Fork new process (doesn't scale)
- Threads (complicated and we need to lock shared resources)
- Single thread (event loop)

For the client, **I/O takes the form of AJAX requests**. AJAX is by default asynchronous. If AJAX were synchronous then it would lock up the front-end.

# I/O needs to be done differently

- Many web applications have code like this:

```
var result = db.query("select * from T");
// use result
```

- What is the software doing while it queries the database? In many cases, just waiting for the response. Either **blocks the entire process** or implies **multiple execution stacks**.
- But a line of code like this allows the program to return to the event loop immediately.

```
db.query("select..", function (result) {
// use result
});
```

- This is how I/O should be done.





PREV NEXT

# I/O latency

| | | |
|---|---|---|
| **1 CPU cycle** | 0.3 ns | 1 s |
| **Level 1 cache access** | 0.9 ns | 3 s |
| **Level 2 cache access** | 2.8 ns | 9 s |
| **Level 3 cache access** | 12.9 ns | 43 s |
| **Main memory access** | 120 ns | 6 min |
| **Solid-state disk I/O** | 50-150 µs | 2-6 days |
| **Rotational disk I/O** | 1-10 ms | 1-12 months |
| **Internet: SF to NYC** | 40 ms | 4 years |
| **Internet: SF to UK** | 81 ms | 8 years |
| **Internet: SF to Australia** | 183 ms | 19 years |
| **OS virtualization reboot** | 4 s | 423 years |
| **SCSI command time-out** | 30 s | 3000 years |
| **Hardware virtualization reboot** | 40 s | 4000 years |
| **Physical system reboot** | 5 m | 32 millenia |

# Synchronous and Asynchronous

- *Asynchronous* means more than one process running simultaneously.
- *Synchronous* means one process is executing at a time.
- **JavaScript/ V8 is Synchronous**
- **Node does things Asynchronously.**





PREV NEXT

# Why JavaScript?

JavaScript is designed specifically to be used with an event loop:

- Anonymous functions, closures.
- Only one callback at a time, no need to lock variables.
- I/O through event callbacks.

The culture of JavaScript is already geared towards **event-driven programming.**

Client side **JavaScript has no traditional I/O**. Node.js was created in the first place because JavaScript had no existing I/O libraries so they could start clean with non-blocking I/O.

# Node.js features – Writing Servers

**Asynchronous and Event Driven** All APIs of Node.js library are asynchronous that is, non-blocking, unless stated otherwise.

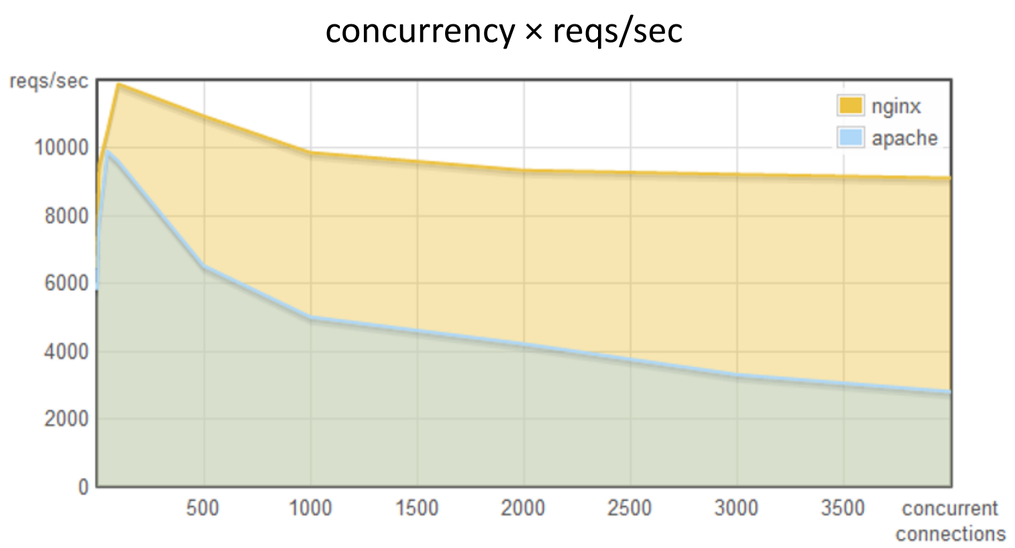
**Very Fast** Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

**Single Threaded but Highly Scalable** - Node.js uses a single threaded model with event looping. Traditional servers create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

**No Buffering** - Node.js applications never buffer any data. The applications simply output data in chunks (streams).

# A look at Apache and NGINX

Context switching when using threads is not free, it costs CPU time.

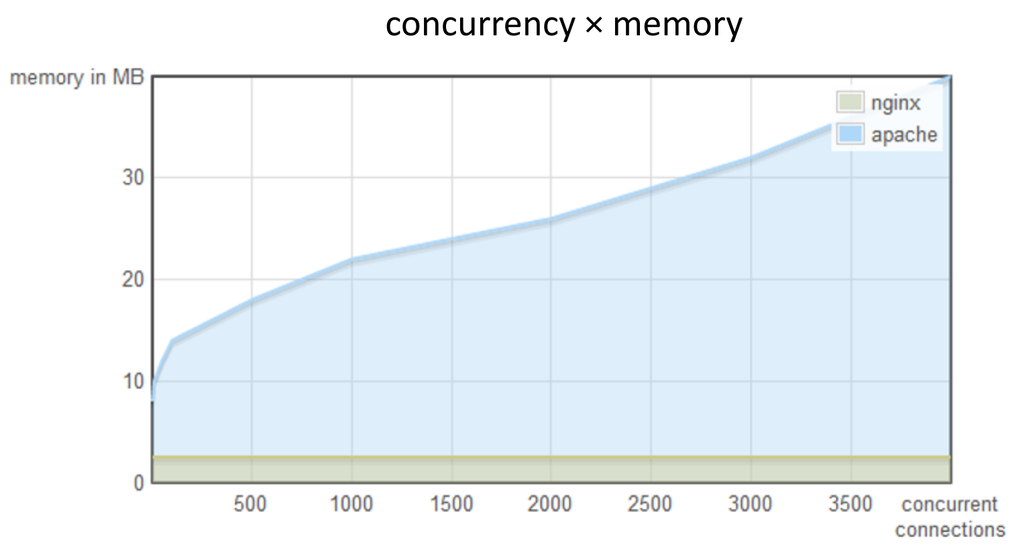



(Number of clients on your server)

http://blog.webfaction.com/

# Apache vs NGINX

Execution stacks take up memory



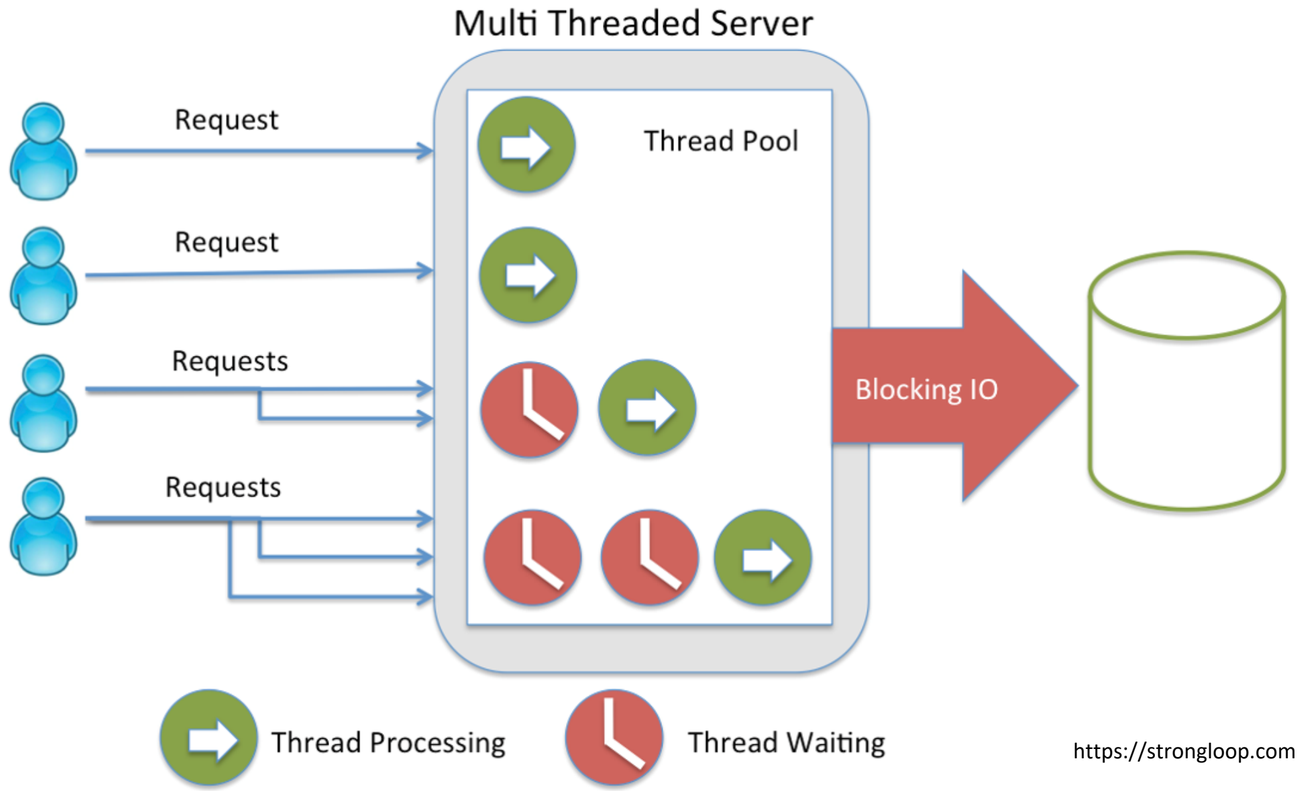

(Number of clients on your server)

http://blog.webfaction.com/

# High concurrency matters

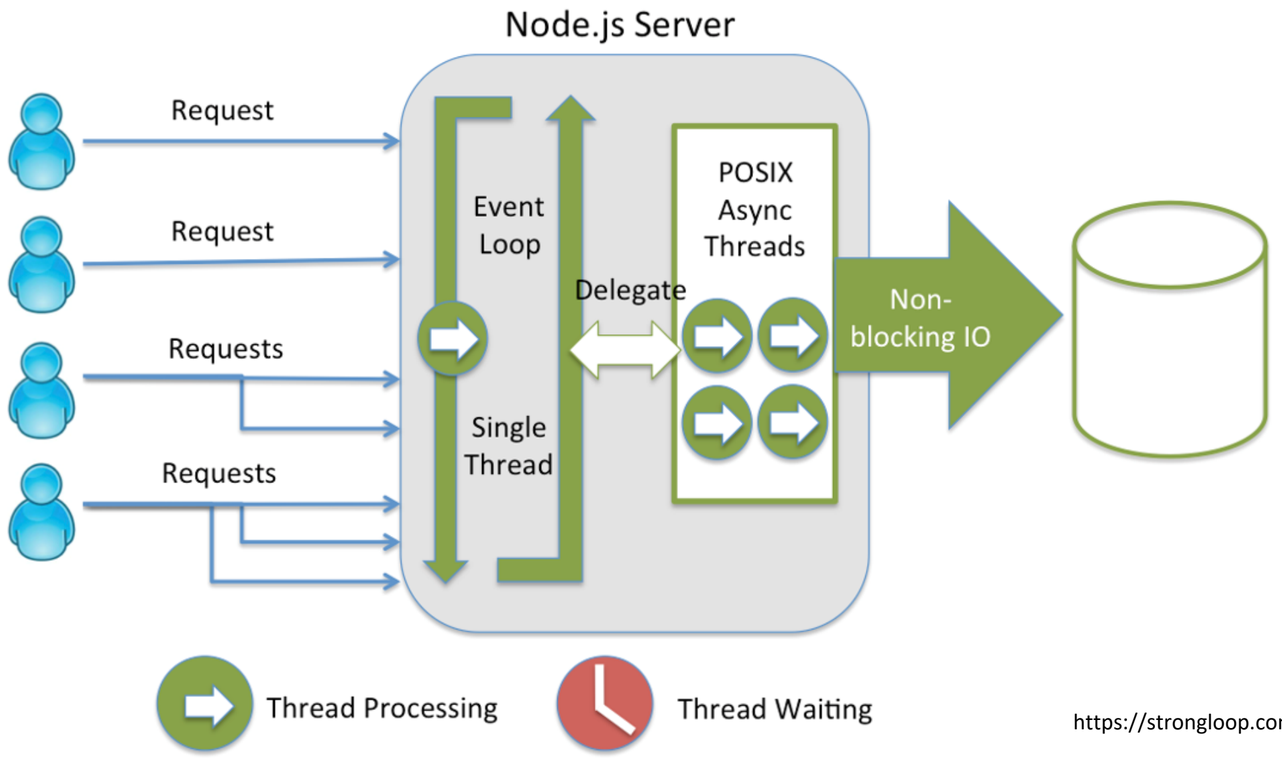
At high levels of **concurrency** (thousands of connections) your server needs to be smart and go to asynchronous non-blocking IO, but the issue is that if any part of your server code blocks you're going to need a thread. And at these levels of concurrency, you **can't go creating threads** for every connection. So the **whole code path needs to be non-blocking** and async, including the IO layer. This is where Node excels.

# Threading Java




https://strongloop.com

# Threading Node

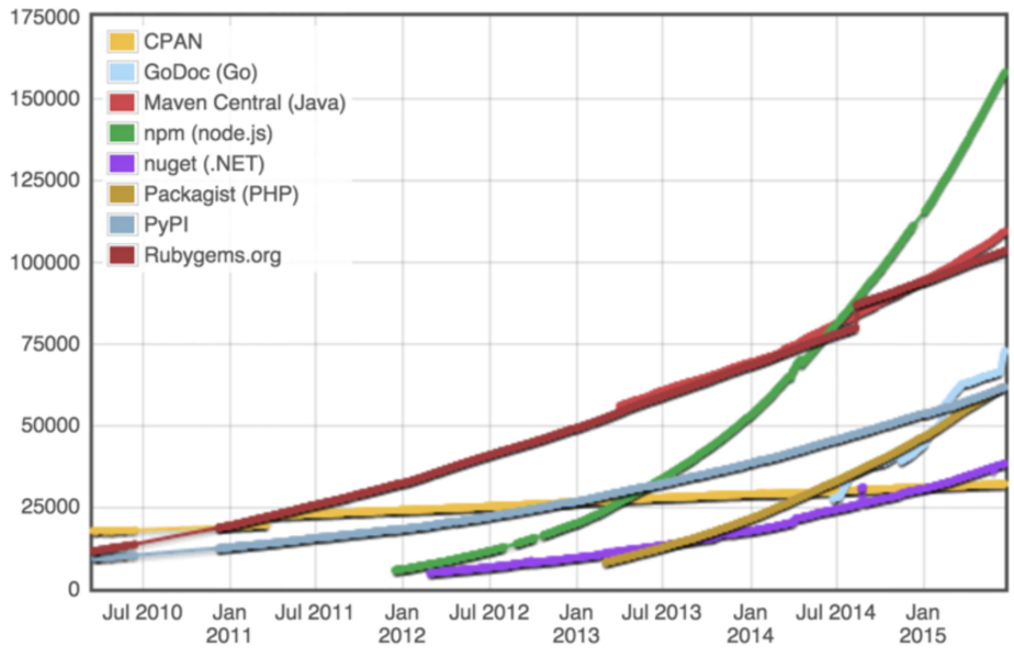



https://strongloop.com

# Node for Enterprise Applications

There is no other server has the non-blocking ecosystem like Node.js today. Over half millions modules all written in the Async style by 7 million developers, ready to use.

Projects that need big concurrency will choose Node because it’s the best way to get their project done. Most companies nowadays are transforming their backend API to be written in Node.




npm is the package manager for JavaScript and the world’s largest software registry.

# JavaScript on the server

- **What does JavaScript need to manage a Server?**
  - Organize our code into **reusable pieces** (MVC)
  - Ways to deal with **Files**
  - Ways to deal with **Databases**
  - Ability to **communicate over the Internet** – Accept requests and send responses in standard format
  - Deal with **work** that takes a **long time**.

# Download Node.JS

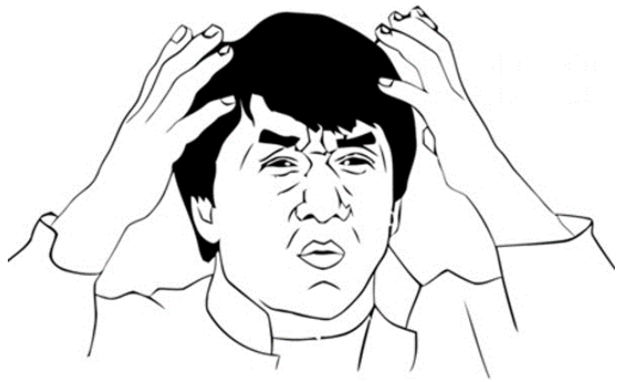
Go to nodejs.org and download node. After installing Node we will be able to use it using the command line interface.

- If Node is installed properly, Try this command: **node –v**
- Hit **Ctrl+C** twice to quit Node.

**Note:** Node sends the request input to V8 engine and writes back the response for us

# Node Versions

- NodeJS 0.12.7 *(started by Node people)*
- V8 was updated very fast by Google. For every change in V8 will require change in Node.
- io.JS 1.0 - 2.0 - 3.0 *(Took the initiative and started their own versioning for Node and they called it io)* It was updated fast.
- Eventually, they have merged and came to an agreement after v4.x
- Even numbers are always LTS, good for production, Odd numbers have latest features but they won't become LTS.



# Try these commands:

Check number of processors that Node can use

```
node -p "os.cpus()"
```

Check the CPU architecture

```
node -p "process.arch"
```

Check V8 version

```
node -p "process.versions.v8"
```

Check V8 heap

```
node -p "v8.getHeapStatistics()"
```

Check the environment variables

```
node -p "process.env"
```

# Node REPL (Read, Eval, Print, Loop)

Run JS scripts

```
node script.js
```

Autocomplete your commands

```
> (tab) (tab)
> global.(tab)
> var a = [];  a.(tab)
```

Shows all the modules exported by Node by default:
Primitives, Wrapper types, Node Core Modules and Global Objec

Underscore: Access to last evaluated value

```
> Math.random(); _
```

The Dot (.) commands

```
.help, .break, .load, .save, .editor, .exit
```

# First Program

```
setTimeout(function () { console.log("world"); }, 2000);
console.log("hello");
```
hello_world.js

```
% node hello_world.js
Hello
// 2 seconds later...
World
```

**node.js** file name is reserved in Node

Node exits automatically when there is nothing else to do (end of process). Let's change it to never exit, but to keep it in loop!

Node API is not all asynchronous. Some parts of it are synchronous like, for instance, some file operations. Don't worry, they are very well marked: they always end with "Sync". They should only be used when initializing.

# Node Startup

- **Bootstrap**: Node loads its own JavaScript (cold-start in serverless)
- **Main scope**: Node loads and executes the user code
- **Event Loop**: starts after Main scope exits, only if async tasks is scheduled, then it runs all async scheduled callbacks.

# Example HTTP Server

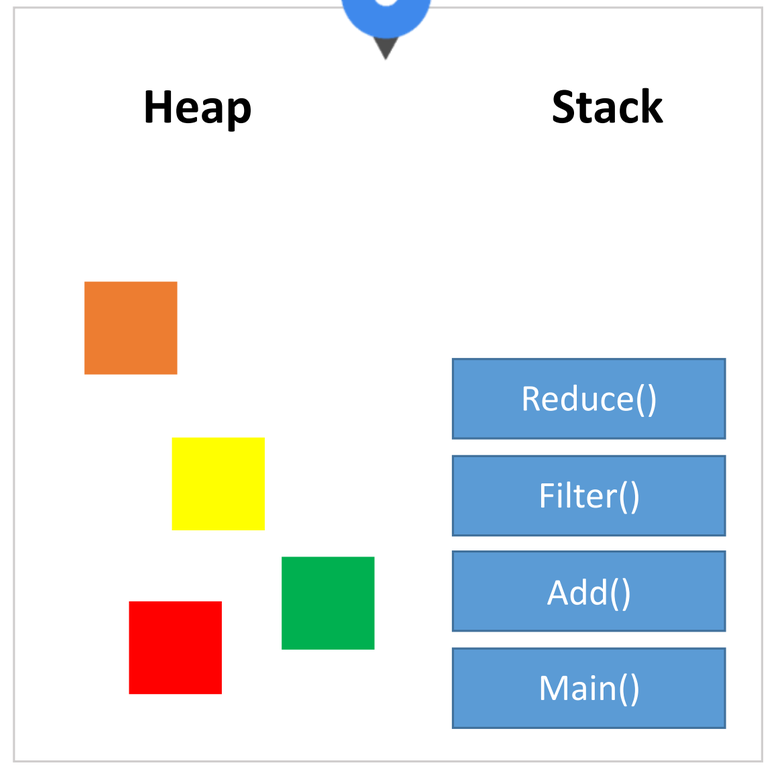
```
const http = require('http');

http.createServer(function(req, res){
        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end('Hello World\n');
}).listen(1337, ()=> console.log('127.0.0.1'));
```

Working in Node requires a deep understanding the command line interface!
Let's dive deeper in Node core.

# Google V8 JavaScript Engine

- **Google V8 engine** is an open source and you may use or change. It's used in Google Chrome. It implements ECMAScript and runs on many processors.
- You may find its code at https://github.com/v8/v8
  - V8 is Google's open source JavaScript engine.
  - V8 implements ECMAScript as specified in ECMA-262.
  - V8 is written in C++ and is used in Google Chrome.
  - **V8 can run standalone, or can be embedded into any C++ application**.
  - It's extremely fast, Google developers worked on it for years.
  - Features groups: shipping, staged (--harmony), in progress

# V8 Core




Memory Allocation

Execution Context

One thread = One call stack = One thing at a time

# Node is written in C++

Yes we will only write our code in JavaScript but it's important to understand that Node itself was written in C++. Node is just a program (server) that’s written in C++.

But why Node was written in C++ in the first place? Because V8 (The JavaScript Engine made by Google) that converts JS to machine code is written in C++.





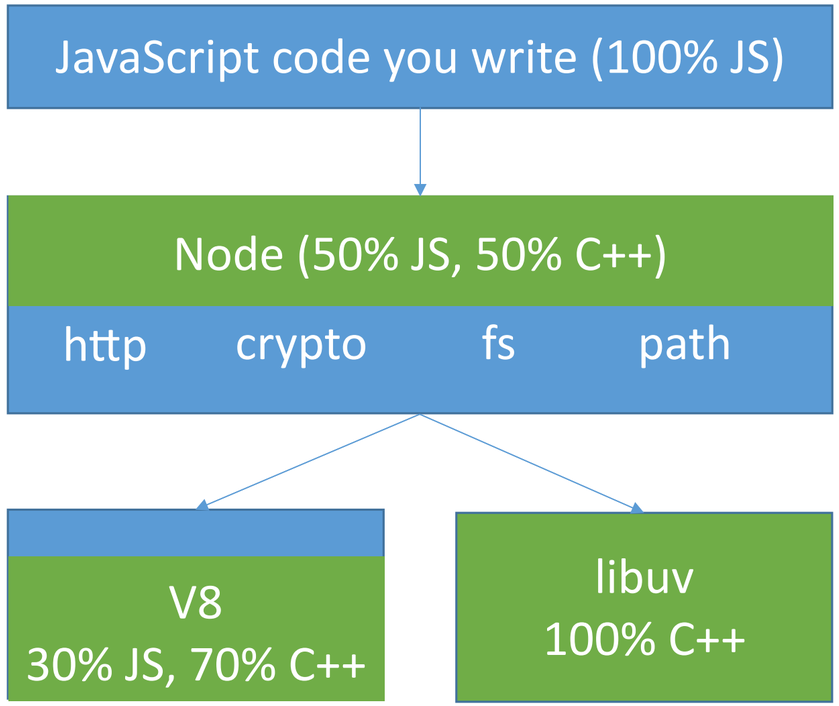
PREV NEXT

# What's inside Node?

**C++ Core:** Node JS source code: https://github.com/nodejs

- Dependencies "***deps***" folder:
  - Google V8 Engine
  - npm
- All C++ core features and utilities are built in C++ and made available to JavaScript via the hooks in the V8 engine.
- **The JavaScript Core:** Inside "***lib***" folder, it's actually wrappers around the C++ features. So we usually use JavaScript libraries and these wrap C++ libraries for us.
- **Other Node dependencies**: http-parser, c-ares, OpenSSL, zlib

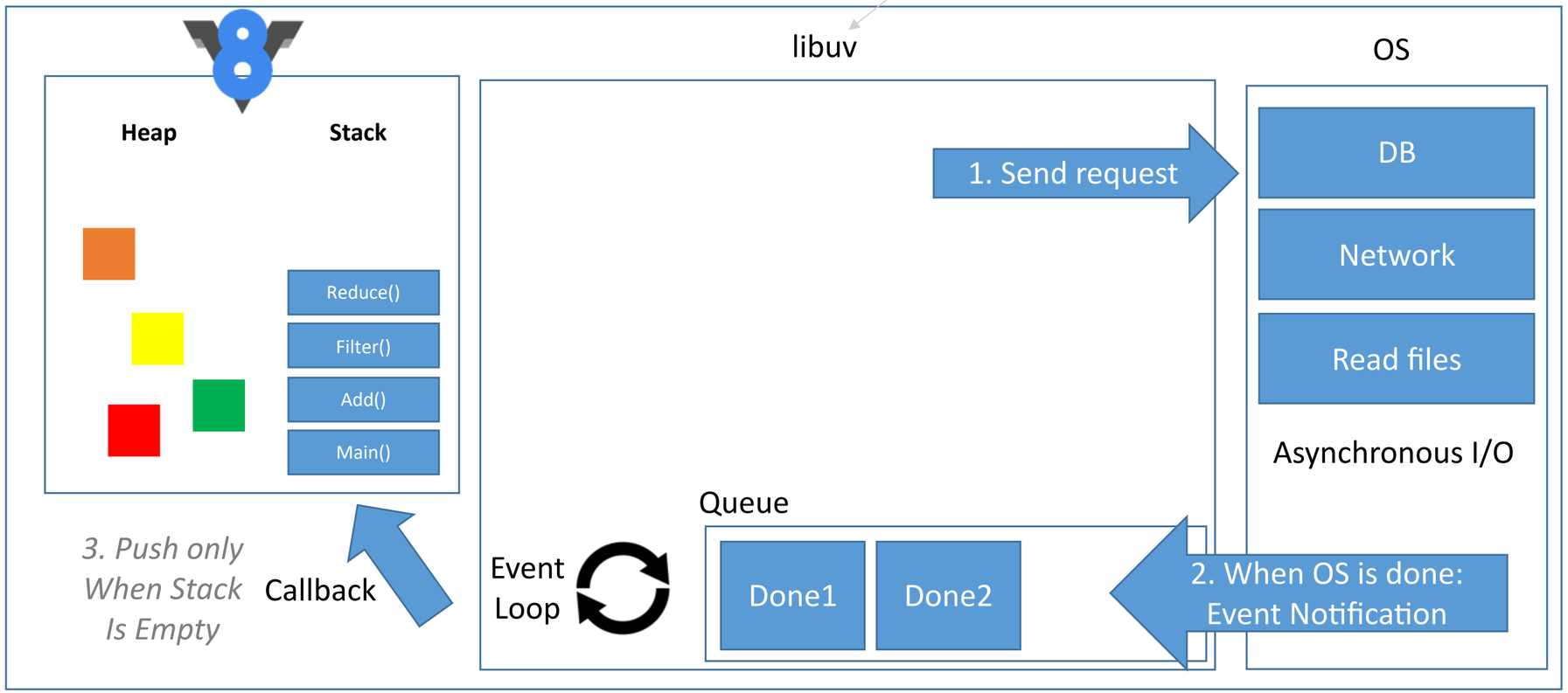
# Node.js Structure



# JS on the Server

An abstract non-blocking IO operations (Async) using thread pool

Part of NodeJs




48

# Blocking vs Non-Blocking?

```
var open = false;

setTimeout(function() {
        open = true;
}, 1000)

while(!open) { // wait }

console.log('opened!');
```

# Blocking vs Non-Blocking?

```
const add = (a,b)=>{
        for(let i=0; i<9e7; i++){}
        console.log(a+b);
}

console.log('start');
const A = add(1,2);
const B = add(2,3);
const C = add(3,4);
console.log('end');
```

# Blocking vs Non-Blocking?

```
const add = (a,b)=>{
        setTimeout(()=>{
                for(let i=0; i<9e7; i++){};
                console.log(a+b)}
        , 5000);
}

console.log('start');
const A = add(1,2);
const B = add(2,3);
const C = add(3,4);
console.log('end');
```

What happen if we set the timer to 0?

# The Server Global Environment

- Buffer
- console
- URL, URL SearchParams
- global
- process
- __dirname, __filename, exports, module, require()
- setInterval() and clearInterval()
- setTimeout() and clearTimeout()
- setImmediate() and clearImmediate()

*Note: Global objects can be used in any script without import.*

# Global Scope in Node

- Browser JavaScript by default puts everything into its **`window`** global scope.
- Node.js was designed to behave differently with **everything being local by default (variables and functions)**. In case we need to set something globally, there is a **`global`** object that can be accessed by all modules. (However, declaring a variable without **`var`** will create the variable on the **`global`** object)
- The document object that represents DOM of the webpage is nonexistent in Node.js.

# Node Single-Thread Model

We know that the event loop has a queue of callbacks that are processed by Node on every **tick** of the event loop. So, even if you are running Node on a multi-core machine, you will not get any parallelism in terms of actual processing - all events will be processed only one at a time. For every I/O task, you can simply define a callback that will get added to the event queue. The callback will fire when the I/O operation is done, and in the mean time, the application can continue to process other I/O bound requests.





PREV NEXT

# What's the event loop?

An Entity that handles external events and convert them to callback invocations.

A loop that picks events from the event queue and pushes their callbacks into the call stack.

Node.js runs using a **single thread**, at least from a Node.js developer's point of view. Under the hood Node uses many threads through **libuv**.

When JavaScript (V8) is running the Event Loop is Not.

# The Event Loop Simplified

- **`node myFile.js`**
- Read the file content and run the JS code + track all operations (timers, listeners and operations)
- **`while (shouldContinue()){`**

One Tick

```
        1. cb for timers (setTimeOut, setInterval)
        2. cb for OS tasks and operation tasks
        3. pause.. Continue when new cb from (OS task is done,
           Operation task is done, timer is complete)
        4. cb for timer (setImmediate)
        5. Handle cb functions of 'close' event.
}
```

- Exit to terminal

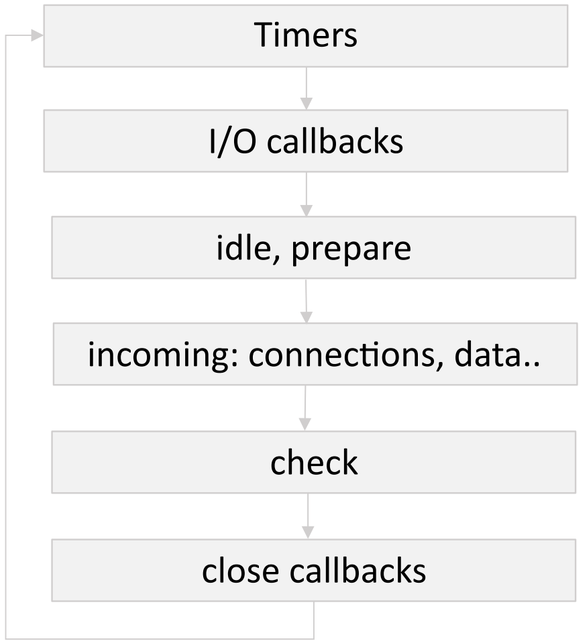
```
function shouldContinue(){
   any pending timer?
   any pending OS task (server listener)?
   any pending operation (fs)?
}
```



PREV NEXT

# Event Loop order of operations

Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.




https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

# Node Asynchronous Code

- setImmediate()
- setTimeout()
- setInterval()
- Promise
- Event Emitter
- async/await
- process.nextTick()
- queueMicrotask()

Making something asynchronous does not mean it's non-blocking



PREV NEXT

# `setTimeout` vs `setImmediate`

While **`setTimeout`** schedules a callback to run after a specific time, the functions are registered in the **timers phase** of the event loop. On the other hand, **`setImmediate`** will schedule a callback to run at **check phase** of the event loop after IO events' callbacks.





PREV NEXT

# `process.nextTick(callback)`

`process.nextTick()` is not part of the event loop, it adds the callback into the nextTick queue. Node processes **all the callbacks** in the nextTick queue after the current callback operation completes and before the event loop continues.

Which means it runs **before** any additional I/O events or timers fire in subsequent ticks of the event loop.

**Note**: the next-tick-queue is completely drained on each pass of the event loop before additional I/O is processed. As a result, recursively setting `nextTick` callbacks will block any I/O from happening, just like a while(true) loop.





PREV NEXT

# process.nextTick(callback)

```
function foo() {
        console.error('foo');
}
process.nextTick(foo);
console.log('bar');
```

Notice that bar will be printed in the console before foo, as we have delayed the invocation of foo() till the next tick of the event loop. We can get the same result by using setTimeout() this way:

```
setTimeout(foo, 0);
console.log('bar');
```

However, process.nextTick() is not just a simple alias to setTimeout(fn, 0).

What's the difference and why it's more efficient?

# Keeping callbacks truly asynchronous

When you are writing a function that takes a callback, you should always ensure that this callback is fired asynchronously. Let's look at an example which violates this convention:

```
function asyncFake(data, callback) {
   callback('result');
}
asyncFake('foo', (r)=> console.log(r));
console.log('Done'); // will run after finishing asyncFake!
```

We can correct asyncFake() to be always asynchronous this way:

```
function asyncReal(data, callback) {
   process.nextTick(callback, data);
}
asyncReal('foo', (r)=> console.log(r));
console.log('Done'); // will run first then calling the anonymous function!
```

You can pass arguments to the callback as extra parameters:
`process.nextTick(callback, param1, param2...)`

# queueMicrotask(callback)-Node 11+

The microtask queue is managed by **V8** and may be used in a similar manner to the **process.nextTick()** queue, which is managed by Node.js. The **process.nextTick()** queue is always processed before the microtask queue within each turn of the Node.js event loop.

# setTimeout vs setImmediate vs process.nextTick

```
(() => new Promise((resolve) => resolve('promise')))()
        .then((p) => console.log(p))
setTimeout(() => console.log('timeout'), 0);
setImmediate(() => console.log('immediate'));
queueMicrotask(() => console.log('microtask'));
process.nextTick(() => console.log('nexttick'));
```

What's the output of this code and why?