Event Loop

SD540 Server-Side Programming

Maharishi University of Management

Masters of Software Development

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

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.

In 2009, Ryan Dahl wrote down the first version of Node.js, and JavaScript has become a compelling "server-side" language as well, which was traditionally the domain of languages like Java, Python, C# and PHP.

What is Node?

Node 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 applications are written in JavaScript and run within the Node runtime on all platforms (OS X, Microsoft Windows, and Linux).

Node is used by: Google, NASA, Amazon, Microsoft, eBay, General Electric, GoDaddy, PayPal, Uber, Walmart, Netflix, CitiBank, Morgan Stanley, MasterCard, Medium, The New York Times, The Washington Post, Airbnb, Spotify, Slack, Evernote, Salesforce, IBM, Trello.

NodeJS in brief

- Server-side JavaScript, built on Google’s V8
- Event-Driven Non-blocking I/O
- CommonJS / ES Modules
- Focused on Performance

ECMAScript Specs

ECMAScript is the standard specification that many JS engines implement.

JS Engine: A program that convert JS code into something that the CPU (computer processor) understands:

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

Process vs Thread

When you execute a computer program, an instance of this program lives in the memory, we call it: **process**.

Every process may have multiple threads. A **thread** is some sort of instruction to be executed by the CPU.

OS scheduler: scheduling is the logic the OS uses to decide which thread to process at any given time. Urgent threads 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 (concurrently examining the work in threads and pausing one when an expensive IO operation occurs)

Parallel, Concurrent, Synchronous and Asynchronous

Parallel is when tasks are executed at the same time.

Concurrency is when the execution of multiple tasks is interleaved, instead of each task being executed sequentially one after another.

Asynchronous delegate and process away from the current thread, and receive the response in the form of a callback function later, which gives the impression of concurrent or parallel tasking but in reality, it is normally used for a process that needs to do work away from the current application and we don't want to wait and block our application awaiting the response.

Synchronous means one task is executed at a time, one after another.

I/O

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

Many web applications have code like this:

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

The allocated thread is waiting for the response and **blocking the entire thread.**

But a line of code like this allows the thread to return to the event loop immediately (**async + non-blocking**).

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

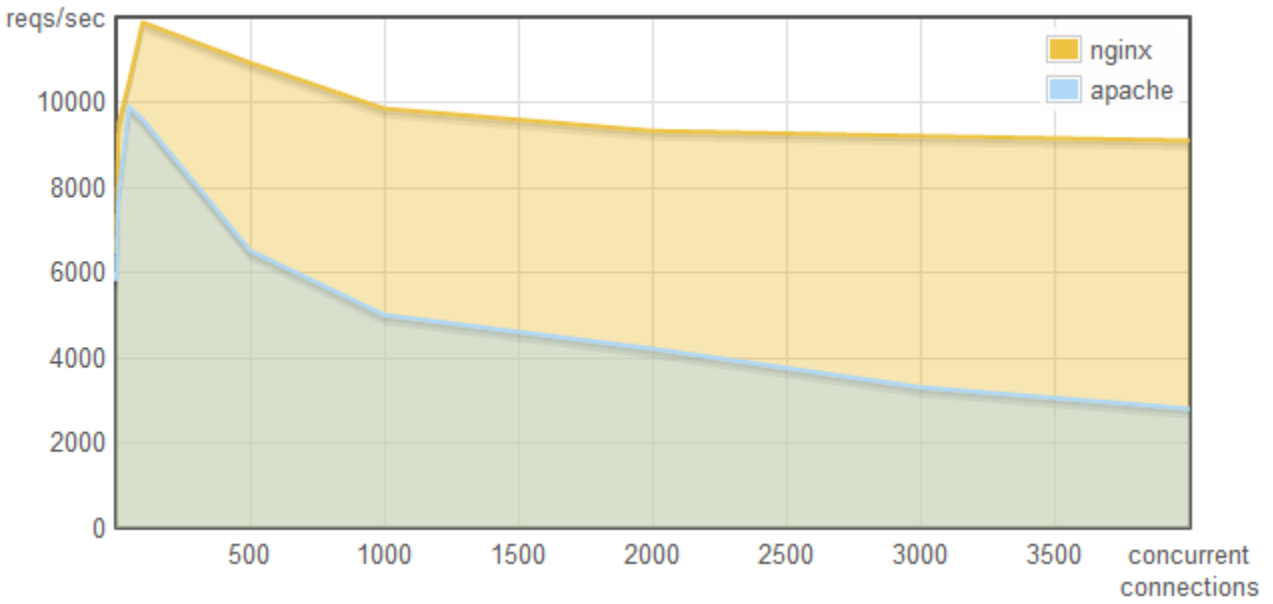
I/O latency

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

A look at Apache and NGINX

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

concurrency × reqs/sec



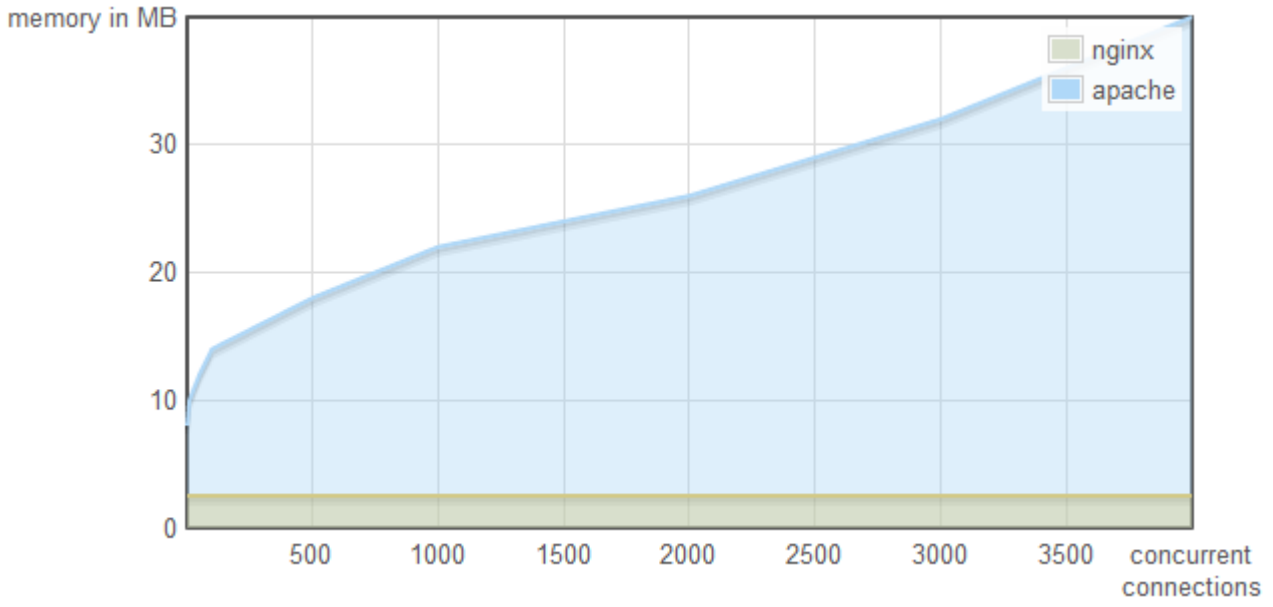
(Number of clients on your server)

http://blog.webfaction.com/

Apache vs NGINX

Execution stacks take up memory

concurrency × memory



(Number of clients on your server)

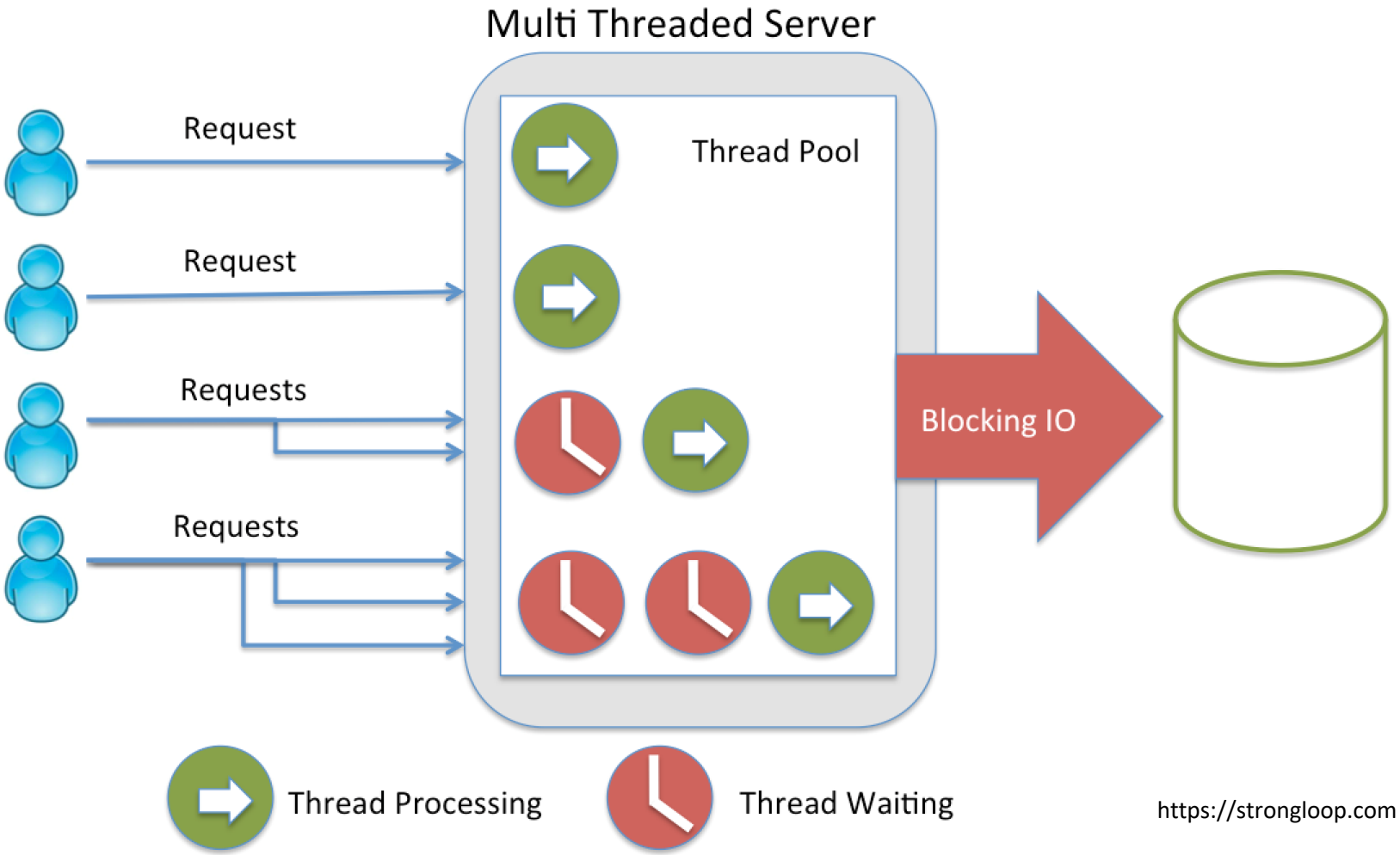
http://blog.webfaction.com/

Node Paradigm

At high levels of **concurrency** (thousands of connections) your server needs to be smart and go to **asynchronous and non-blocking IO**.

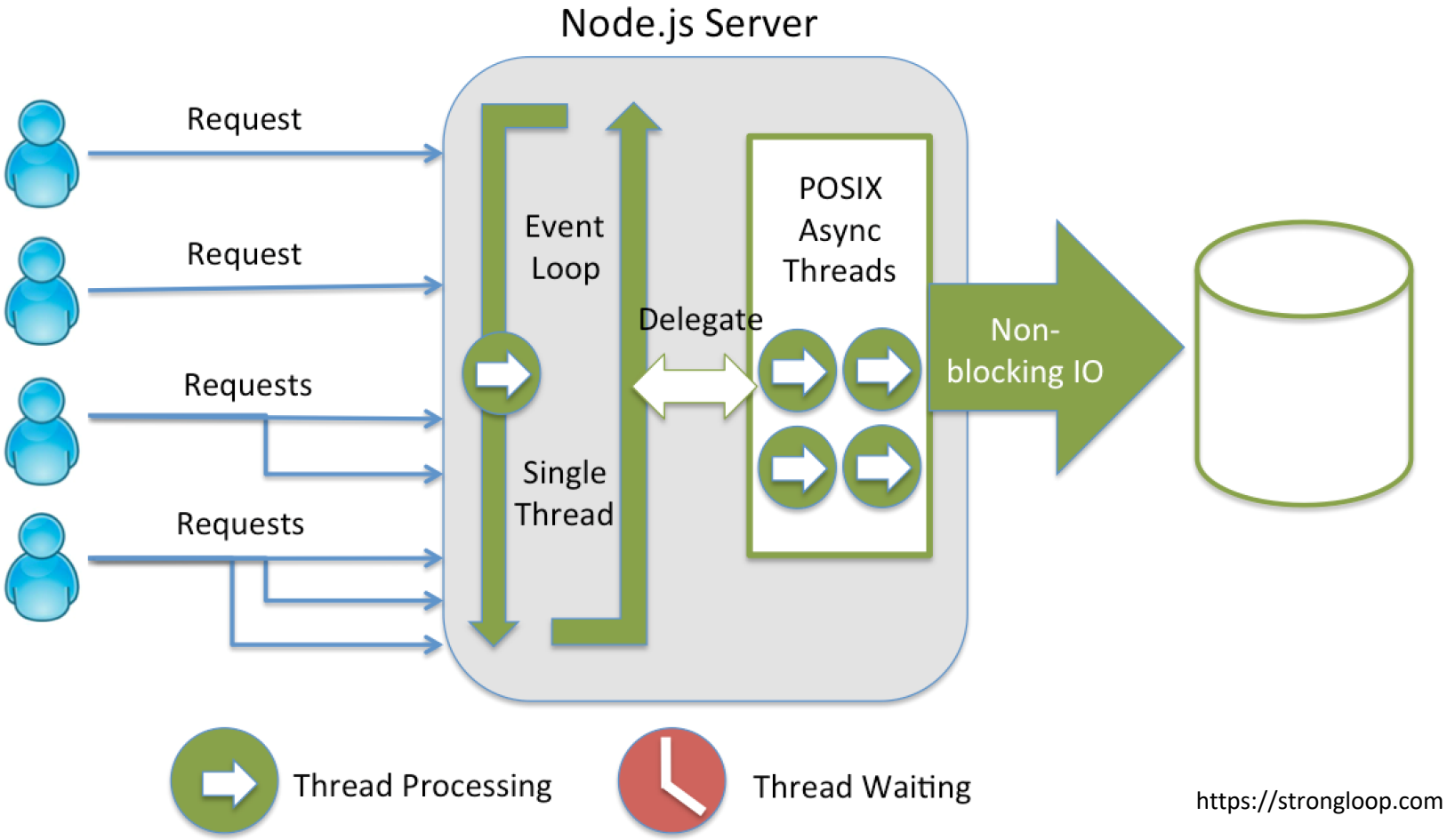
At these levels of concurrency, you **can't go creating threads** for every connection. So the **whole code path needs to be non-blocking** and **async**, including the IO layer.

Threading Java



https://strongloop.com

Threading Node



https://strongloop.com

Download Node

Go to nodejs.org and download the **stable LTS version** of node.

Note: to run TypeScript files, you will need a few dependencies, in a folder, run the following commands:

- `npm init --y`
- `npm i nodemon ts-node @types/node –D`
- `create a script: {"start": "nodemon file.ts"}`
- `npm run scriptname`

Node Startup

init-phase: Node loads its own JavaScript (cold-start in serverless)

main-scope-phase: Node loads and executes the user code

event-loop-phase: starts after the main scope phase exits (all sync code is finished), but only if there are async tasks scheduled, so it assigns all async scheduled callbacks one-by-one to run synchronously in the JS engine, when possible.

What's inside Node?

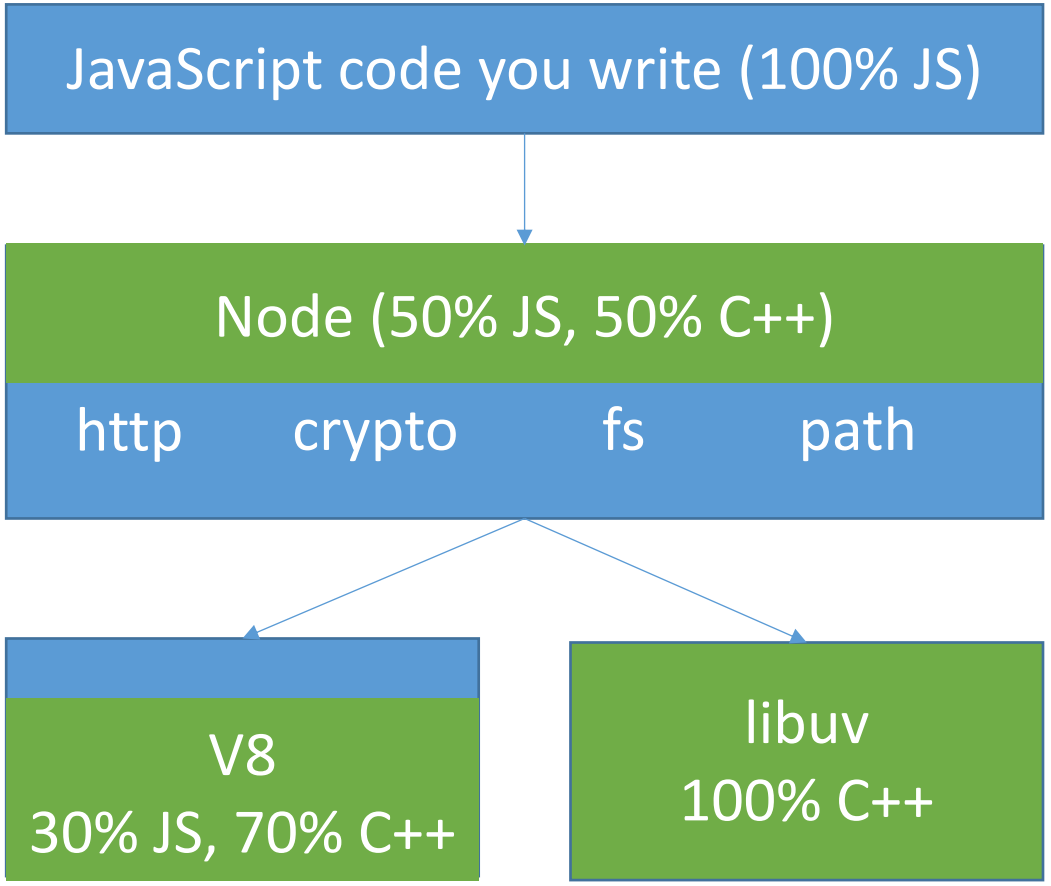
Node source code: https://github.com/nodejs

Node C++ core: all C++ core features and utilities.

The JavaScript core: hooks/wrappers around the C++ features.

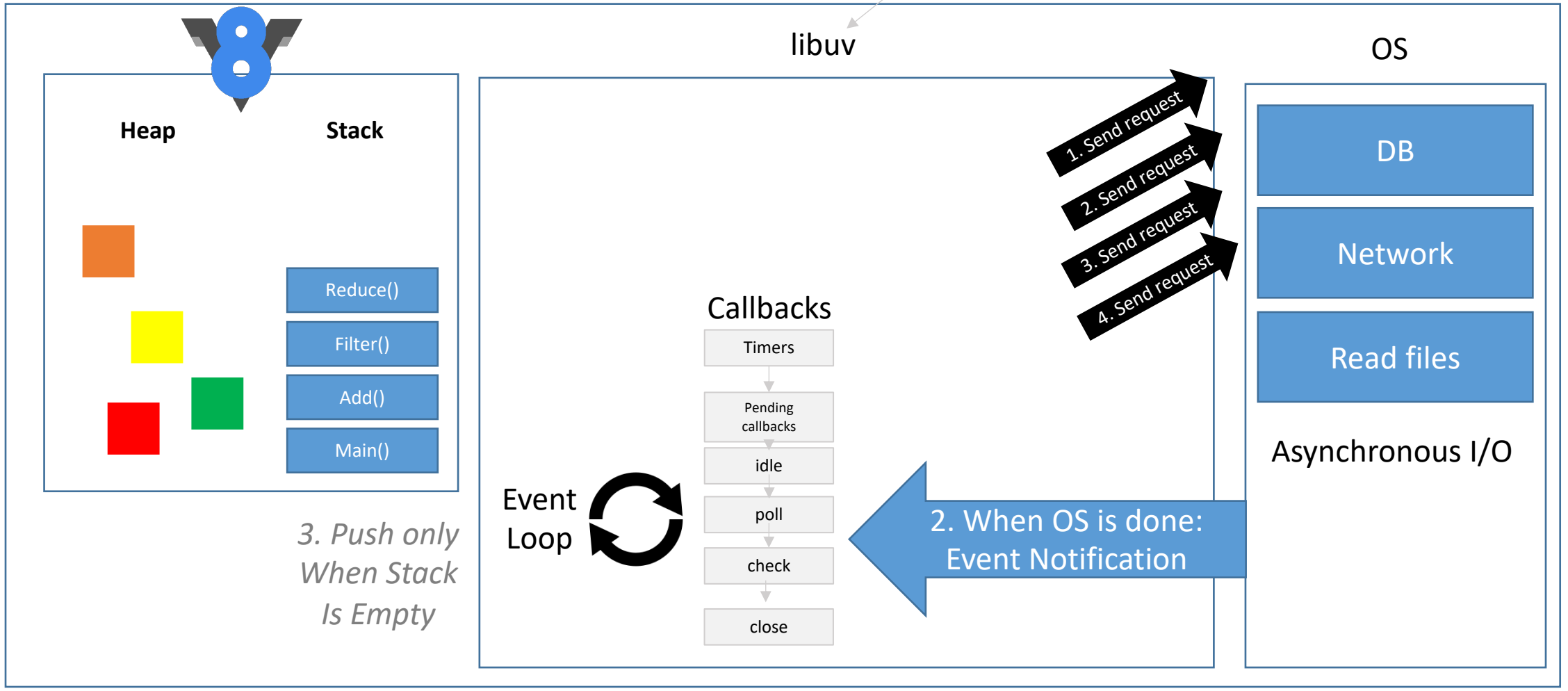
Third-party dependencies: v8, npm, uv, openssl, zlib

Node.js Structure



Node

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



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?

Global Scope in Node

Browser JavaScript by default puts everything into its **`window`** global scope, this object is nonexistent in Node.

Node 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.

Node Global Environment

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

Global objects can be used in any script without import.

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 meantime, the application can continue to process other I/O bound requests.

Node.js and Threads

Node uses the Event-Driven Architecture, it has an Event Loop for orchestration and a Worker Pool for expensive tasks. To put it in a simple way: there are two types of threads one **Event Loop**, and a **pool of Workers** (threadpool).

Node IO thread pool is a multi-threaded execution model where **threads are pre-allocated and kept on hold until a thread is needed, saving the overhead of thread allocation**. It's about the fastest way to run multi-threaded code. A thread executes its task and, once done, returns to the pool and back on hold.

The default number of the pre-allocated pool is 4 worker threads.

What code runs on the Worker Pool?

Only Node Core Modules take advantage of the Worker Pool. Some of the Node Core modules that make use of the Worker Pool:

I/O-intensive

- DNS
- File System (fs)

CPU-intensive

- Crypto
- Zlib

What's the event loop?

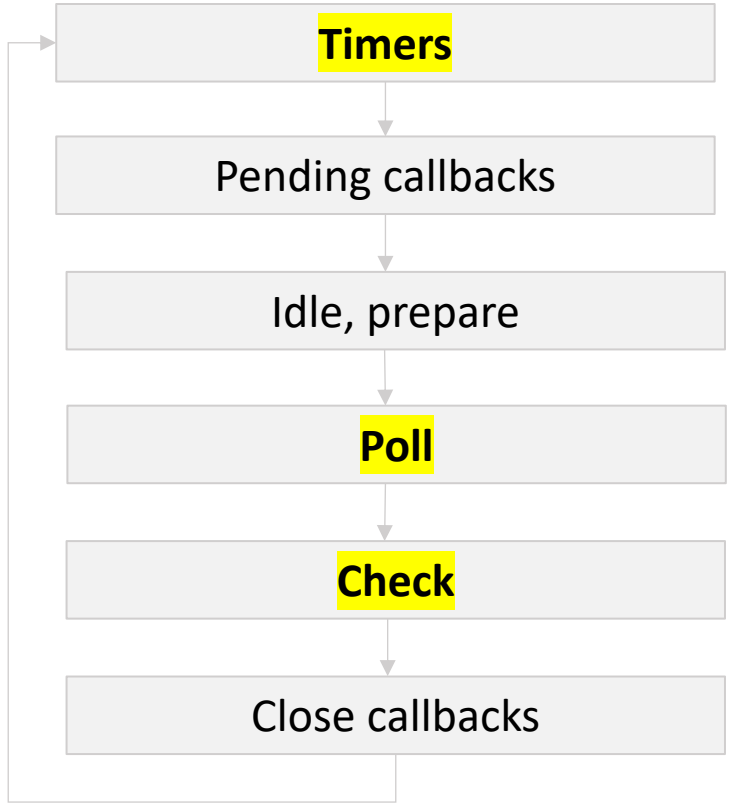
A loop that picks events from the event queue and pushes their callbacks into the call stack.

Node.js runs uses a **single-thread event loop**, and **multiple I/O threads** through **libuv**.

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

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

API	Queue	Event Loop
`setTimeout(f, time)`	timers	yes
`setInterval(f, time)`	timers	yes
`setImmediate(f)`	check	yes
`Node API`	poll	yes
`Promise .then(f)`	microtask	no
`async/await`	microtask	no
`queueMicrotask(f)`	microtask	no
`process.nextTick(f)`	next tick	no

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

`queueMicrotask(callback)`

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.

`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.

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?

Note

The order in which the two timers are executed is non-deterministic, as it is bound by the performance of the process:

```
// timeout_vs_immediate.js
setTimeout(() => { console.log('timeout') }, 0);
setImmediate(() => { console.log('immediate') });

$ node timeout_vs_immediate.js
timeout
immediate

$ node timeout_vs_immediate.js
immediate
timeout
```

However, if you move the two calls within an I/O cycle, the immediate callback is always executed first.

Processing CPU intensive tasks

Every node process is single-threaded by design (has one event loop). Converting a blocking code to run asynchronously doesn't mean it won't block the event loop when it runs in V8.

We can process any custom-user CPU-intensive task in many ways:

- **Clone/Fork Child Process**
- **Create Additional Custom Task Worker Threads**

Use child-process and worker_threads only for custom development and other heavy-duty work (not to be used to handle a task from Node Core Modules).