NODE JS

An open source, cross platform JavaScript runtime environment

Javascript is synchronous, blocking and single threaded language

What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment

Open source - source code is publicly available for sharing and modification

Cross platform - available on Mac, Windows and Linux

JavaScript runtime environment - provides all the necessary components in order to use and run a JavaScript program *outside the browser*

ECMAScript Summary

ECMA-262 is the language specification

ECMAScript is the language that implements ECMA-262

JavaScript is basically ECMAScript at its core but builds on top of that

Chrome's V8 Engine Summary

A JavaScript engine is a program that executes JavaScript code

In 2008, Google created its own JavaScript engine called V8

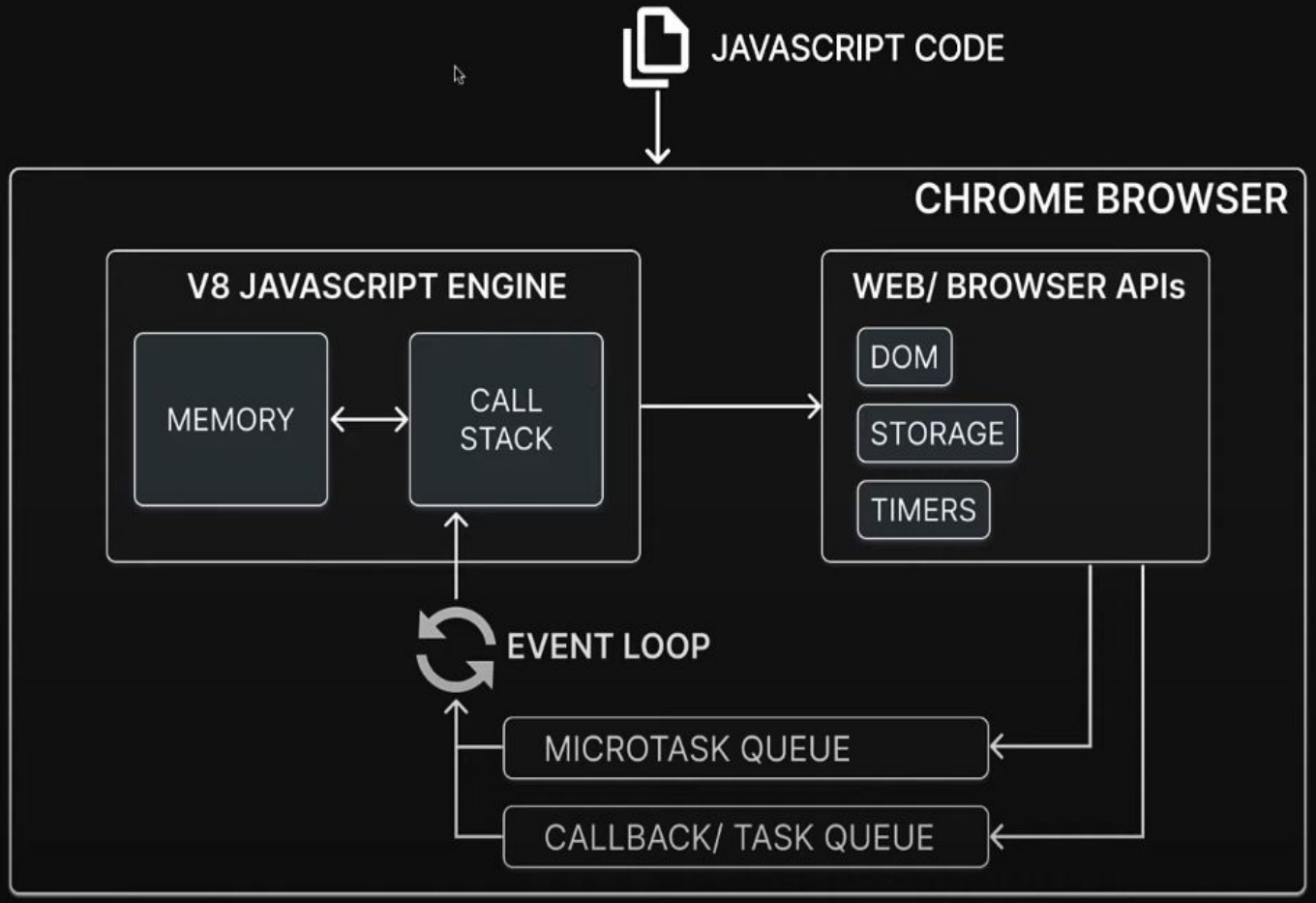
V8 is written in C++ and can be used independently or can be embedded into other C++ programs

That allows you to write your own C++ programs which can do everything that V8 can do and more

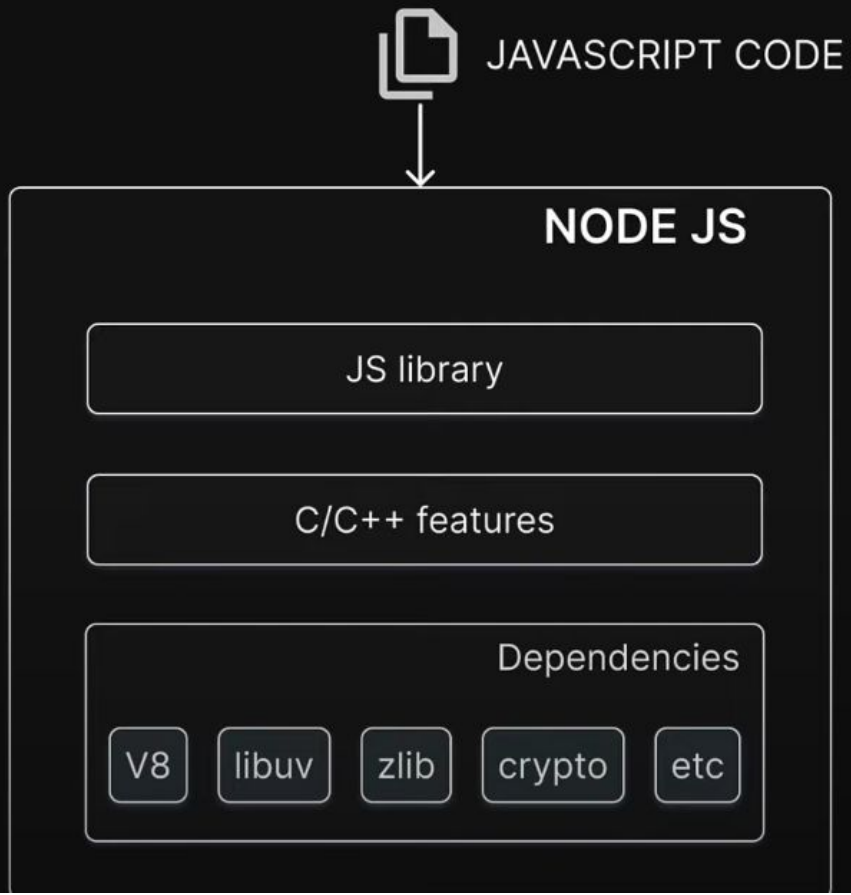
Your C++ program can run ECMAScript and additional features that you choose to incorporate

For example, features that are available in C++ but not available with JavaScript

Chrome Browser JavaScript Runtime



Node.js JavaScript Runtime



Node.js Summary

Node.js is an open-source, cross-platform JavaScript runtime environment

It is not a language, it is not a framework

Capable of executing JavaScript code outside a browser

It can execute not only the standard ECMAScript language but also new features that are made available through C++ bindings using the V8 engine

It consists of C++ files which form the core features and JavaScript files which expose common utilities and some of the C++ features for easier consumption

Browser vs Node.js

In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. You don't have the document, window and all the other objects that are provided by the browser.

In the browser, we don't have all the nice APIs that Node.js provides through its modules. For example the filesystem access functionality.

With Node.js, you control the environment.

With a browser, you are at the mercy of what the users choose

Modules

A module is an encapsulated and reusable chunk of code that has its own context

In Node.js, each file is treated as a separate module

Types of Modules

1. Local modules - Modules that we create in our application
2. Built-in modules - Modules that Node.js ships with out of the box
3. Third party modules - Modules written by other developers that we can use in our application

Immediately Invoked Function Expression (IIFE) in Node.js

Before a module's code is executed, Node.js will wrap it with a function wrapper that provides module scope

This saves us from having to worry about conflicting variables or functions

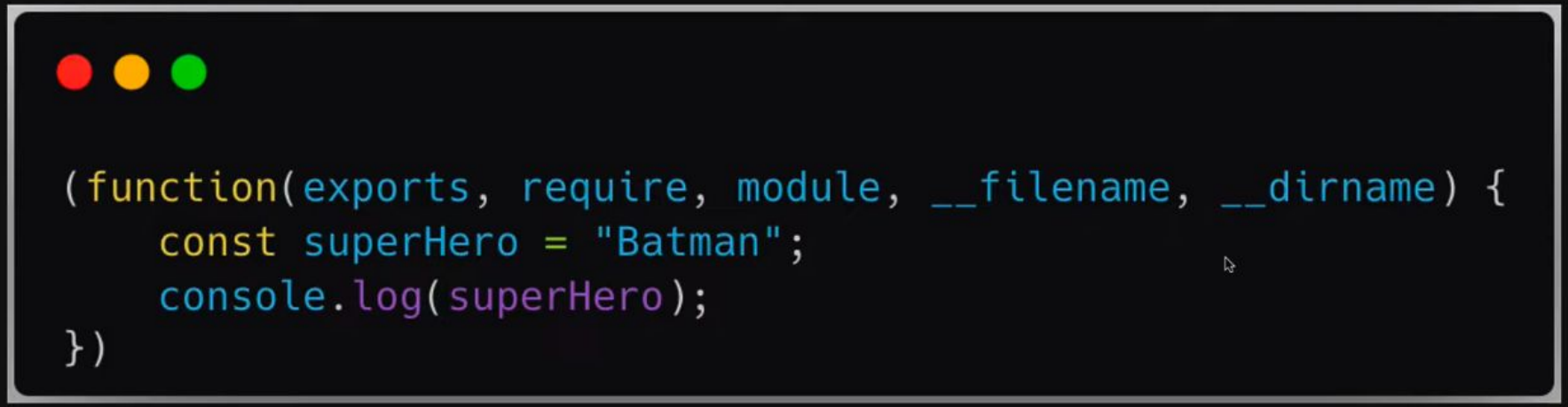
There is proper encapsulation and reusability is unaffected

Module Scope Summary

Each loaded module in Node.js is wrapped with an IIFE that provides private scoping of code

IIFE allows you to repeat variable or function names without any conflicts

Module Wrapper contd.



```
(function(exports, require, module, __filename, __dirname) {
    const superHero = "Batman";
    console.log(superHero);
})
```

CommonJS

Each file is treated as a module

Variables, functions, classes, etc. are not accessible to other files by default

Explicitly tell the module system which parts of your code should be exported via *module.exports* or *exports*

To import code into a file, use the require() function

ES Modules Summary

ES Modules is the ECMAScript standard for modules

It was introduced with ES2015

Node.js 14 and above support ES Modules

Instead of *module.exports*, we use the *export* keyword

The export can be default or named

We import the exported variables or functions using the *import* keyword

If it is a default export, we can assign any name while importing

If it is a named export, the import name must be the same

Section Summary

What is a module and what is the need for modules?

Types of modules in Node.js

Local modules

CommonJS module format

Module wrapper (IIFE)

Module caching

Patterns for importing and exporting modules in CommonJS and ESM format

Importing JSON data and watch mode

1. Higher order functions are those functions which accepts functions as arguments or returns a function.
2. Callback function is a function as an argument to another function.
3. Synchronous callback: A callback which is executed immediately.

```
let numbers = [1, 2, 4, 7, 3, 5, 6]
numbers.sort((a, b) => a - b)
numbers.filter(n => n % 2 === 0)
numbers.map(n => n\2)
```

4. Asynchronous callback: A callback that is often used to continue or resume code execution after an asynchronous operation is completed.

Events Module

The events module allows us to work with events in Node.js

An event is an action or an occurrence that has happened in our application that we can respond to

Using the events module, we can dispatch our own custom events and respond to those custom events in a non-blocking manner

Streams

A stream is a sequence of data that is being moved from one point to another over time

Ex: a stream of data over the internet being moved from one computer to another

Ex: a stream of data being transferred from one file to another within the same computer

Process streams of data in chunks as they arrive instead of waiting for the entire data to be available before processing

Ex: watching a video on YouTube

The data arrives in chunks and you watch in chunks while the rest of the data arrives over time

Ex: transferring file contents from fileA to fileB

The contents arrive in chunks and you transfer in chunks while the remaining contents arrive over time

Prevents unnecessary data downloads and memory usage

Node.js Tutorial - 28 - Streams

index.js file.txt file2.txt

index.js > ...

```
1  const fs = require("node:fs");

2

3  const readableStream = fs.createReadStream("./file.txt", {
4    encoding: "utf-8",
5    highWaterMark: 2,
6  });

7

8  const writeableStream = fs.createWriteStream("./file2.txt");

9

10 readableStream.on("data", (chunk) => {
11   console.log(chunk);
12   writeableStream.write(chunk);
13 });
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL zsh

```
Hello Codevolution
vishwas@Vishwas-Personal-MBP NodeJS % node index
He
ll
o
Co
de
vo
lu
ti
on
```

Piping

Suppose you want to write a server that serves some files. In this case, you will need to write code to handle the reading operation on the files and code to write the data to a network socket. But with the piping facility provided by NodeJS, you can easily connect the file stream to the network socket stream to send the data. There is a pipe() function in NodeJS to handle this operation which is provided with a stream object and accepts a stream object.

HTTP and Node

We can create a web server using Node.js

Node.js has access to operating system functionality like networking

Node has an event loop to run tasks asynchronously and is perfect for creating web servers that can simultaneously handle large volumes of requests

The node server we create should still respect the HTTP format

The HTTP module allows creation of web servers that can transfer data over HTTP

Section Summary

Built-in modules

Path module

Callback pattern

Events module

EventEmitter class

Character sets and encoding

Streams and buffers

Asynchronous JavaScript

fs module

Streams

Pipes

HTTP module

libuv

What?

libuv is a cross platform open source library written in C language

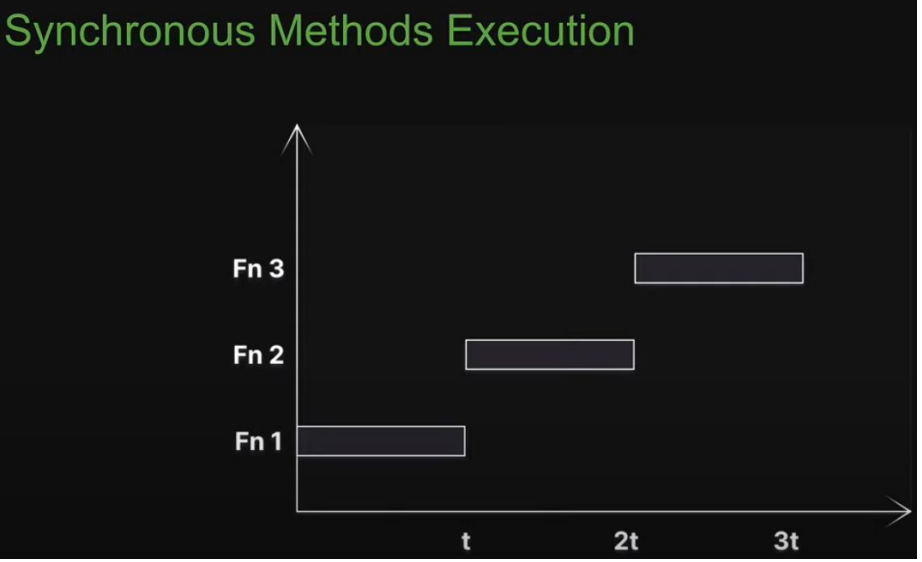
Why?

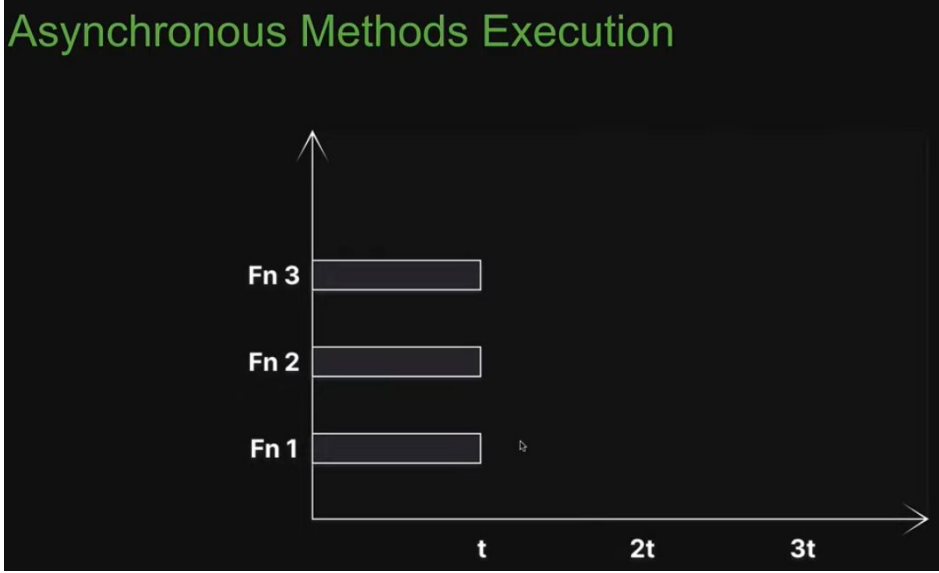
handles asynchronous non-blocking operations in Node.js

How?

Thread pool

Event loop





Every method in node.js that has the "sync" suffix always runs on the main thread and is blocking

1. A few async methods like fs.readFile and crypto.pbkdf2 run on a separate thread in libuv's thread pool. They do run synchronously in their own thread but as far as the main thread is concerned, it appears as if the method is running asynchronously
2. Libuv’s thread pool has 4 threads

index.js > ...

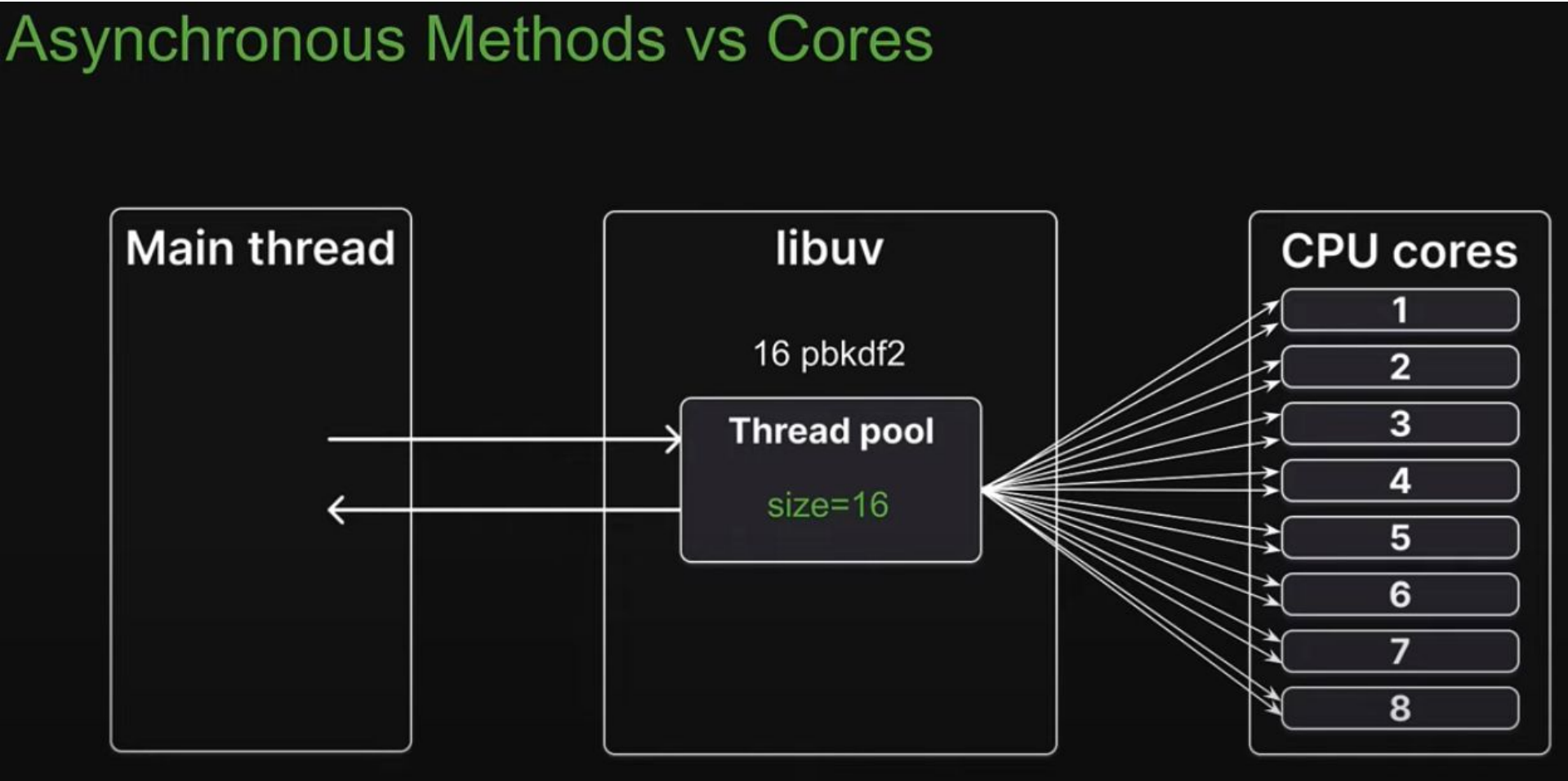
```
1   const crypto = require("node:crypto");
2
3   process.env.UV_THREADPOOL_SIZE = 6;
4   const MAX_CALLS = 6;
5
6   const start = Date.now();
7   for (let i = 0; i < MAX_CALLS; i++) {
8     crypto.pbkdf2("password", "salt", 100000, 512, "sha512", () => {
9       console.log(`Hash: ${i + 1}`, Date.now() - start);
10    });
11  }
12
13  // const start = Date.now();
```

By increasing the thread pool size, we are able to improve the total time taken to run multiple calls of an asynchronous method like pbkdf2

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
vishwas@Vishwas-Personal-MBP NodeJS % node index
Hash: 5 308
Hash: 4 312
Hash: 1 323
Hash: 3 325
Hash: 6 363
Hash: 2 383
vishwas@Vishwas-Personal-MBP NodeJS %
```

Increasing the thread pool size can help with performance but that is limited by the number of available CPU cores



1. Although both crypto.pbkdf2 and https.request are asynchronous, https.request method does not seem to use the thread pool
2. https.request does not seem to be affected by the number of CPU cores either
3. https.request is a network input/output operation and not a CPU bound operation
4. It does not use the thread pool
5. Libuv instead delegates the work to the operating system kernel and whenever possible, it will poll the kernel and see if the request has completed

Libuv and Async Methods Summary

In Node.js, async methods are handled by libuv

They are handled in two different ways

1. Native async mechanism
2. Thread pool

Whenever possible, Libuv will use native async mechanisms in the OS so as to avoid blocking the main thread

Since this is part of the kernel, there is different mechanism for each OS. We have epoll for Linux, Kqueue for MacOS and IO Completion Port on Windows

Relying on native async mechanisms makes Node scalable as the only limitation is the operating system kernel

Example of this type is a network I/O operation

If there is no native async support and the task is file I/O or CPU intensive, libuv uses the thread pool to avoid blocking the main thread

Although the thread pool preserves asynchronicity with respect to Node's main thread, it can still become a bottleneck if all threads are busy

Few Questions

Whenever an async task completes in libuv, at what point does Node decide to run the associated callback function on the call stack?

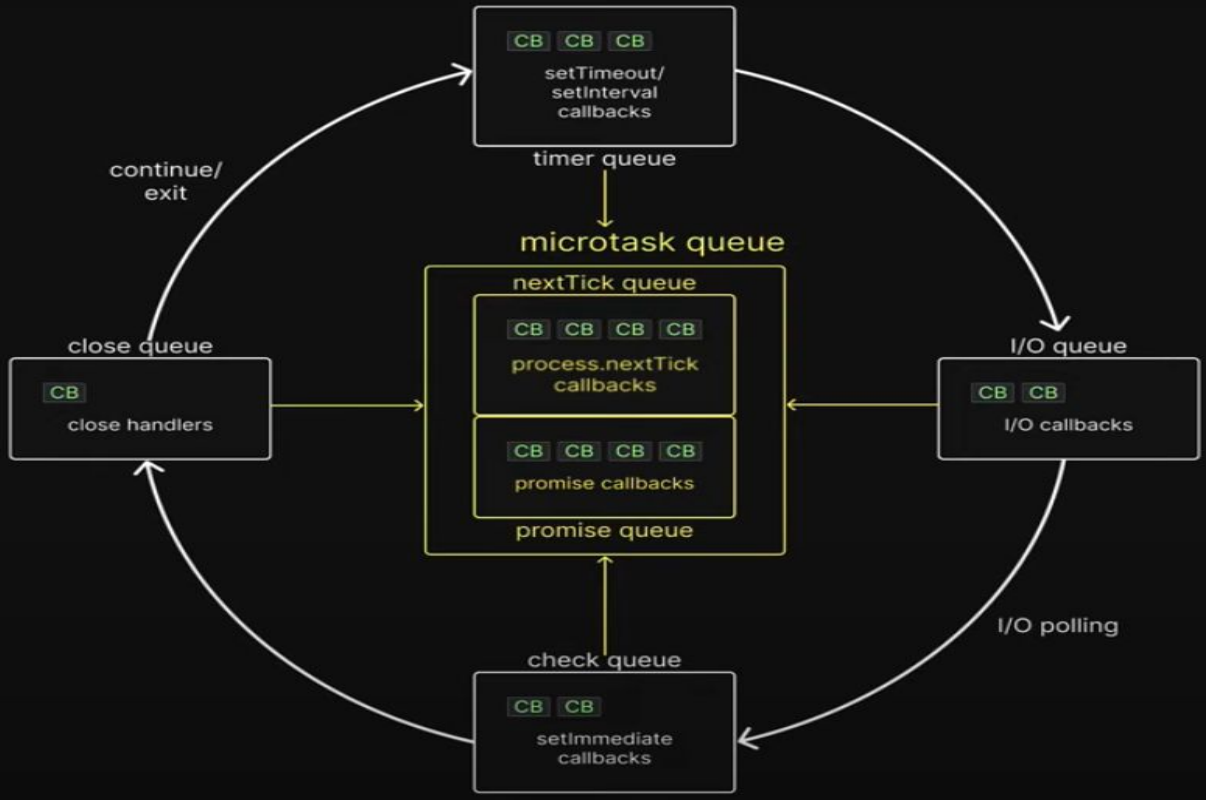
What about async methods like setTimeout and setInterval which also delay the execution of a callback function?

If two async tasks such as setTimeout and readFile complete at the same time, how does Node decide which callback function to run first on the call stack?

Event Loop

It is a C program and is part of libuv

A design pattern that orchestrates or co-ordinates the execution of synchronous and asynchronous code in Node.js



I/O events are polled and callback functions are added to the I/O queue only after the I/O is complete

Event Loop - Execution Order

1. Any callbacks in the micro task queues are executed. First, tasks in the nextTick queue and only then tasks in the promise queue
2. All callbacks within the timer queue are executed
3. Callbacks in the micro task queues if present are executed. Again, first tasks in the nextTick queue and then tasks in the promise queue
4. All callbacks within the I/O queue are executed
5. Callbacks in the micro task queues if present are executed. nextTick queue followed by Promise queue.
6. All callbacks in the check queue are executed
7. Callbacks in the micro task queues if present are executed. Again, first tasks in the nextTick queue and then tasks in the promise queue
8. All callbacks in the close queue are executed
9. For one final time in the same loop, the micro task queues are executed. nextTick queue followed by promise queue.

Few Questions

Whenever an async task completes in libuv, at what point does Node decide to run the associated callback function on the call stack?
Callback functions are executed only when the call stack is empty. The normal flow of execution fill not be interrupted to run a callback function

What about async methods like setTimeout and setInterval which also delay the execution of a callback function?
setTimeout and setInterval callbacks are given first priority

If two async tasks such as setTimeout and readFile complete at the same time, how does Node decide which callback function to run first on the call stack?
Timer callbacks are executed before I/O callbacks even if both are ready at the exact same time

```
process.nextTick(() => console.log("this is process.nextTick 1"));
process.nextTick(() => {
  console.log("this is process.nextTick 2");
  process.nextTick(() =>
    console.log("this is the inner next tick inside next tick")
  );
});
process.nextTick(() => console.log("this is process.nextTick 3"));

Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
Promise.resolve().then(() => {
  console.log("this is Promise.resolve 2");
  process.nextTick(() =>
    console.log("this is the inner next tick inside Promise then block")
  );
});
Promise.resolve().then(() => console.log("this is Promise.resolve 3"));
```

Console

```
nextTick 1
nextTick 2
nextTick 3
nextTick inside nextTick
promise 1
promise 2
promise 3
nextTick inside promise
```

process.nextTick()

Use of process.nextTick is discouraged as it can cause the rest of the event loop to starve

If you endlessly call process.nextTick. the control will never make it past the microtask queue

Two main reasons to use process.nextTick

1. To allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues
2. To allow a callback to run after the call stack has unwound but before the event loop continues

Callbacks in microtask queues are executed in between the execution of callbacks in the timer queue

Event Loop Summary

The event loop is a C program that orchestrates or co-ordinates the execution of synchronous and asynchronous code in Node.js

It co-ordinates the execution of callbacks in six different queues

They are nextTick, Promise, timer, I/O, check and close queues

We use *process.nextTick()* method to queue into the nextTick queue

We *resolve or reject a Promise* to queue into the Promise queue

We use *setTimeout* or *setInterval* to queue into the timer queue

Execute an *async method* to queue into the I/O queue

Use *setImmediate* function to queue into the check queue and finally

Attach *close event listeners* to queue into the close queue

The order of execution follows the same order listed here

nextTick and Promise queues are executed in between each queue and also in between each callback execution in the timer and check queues

Versioning Rules

When you fix a bug and the code stays backwards-compatible you increment the patch version.
For example 1.1.1 to 1.1.2

When you add new functionality but the code still stays backwards-compatible, you increment the minor version
You also reset the patch version to zero
For example 1.1.1 to 1.2.0

When you make changes and the code is no more backwards compatible, you increment the major version
You have to reset the minor and patch version to zero
For example 1.1.1 to 2.0.0.

Semantic versioning always starts with 0.1.0

0.Y.Z (a major version of zero) is used for initial development

When the code is production-ready, you increment to version 1.0.0

Even the simplest of changes has to be done with an increase in the version number

Wrapping Up

What is Node.js?

User defined modules

Built-in modules

Internals of Node.js

npm

CLI tool

Misc topics