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

PREV NEXT

OUTLINE

Search...

Packages and Package Manager

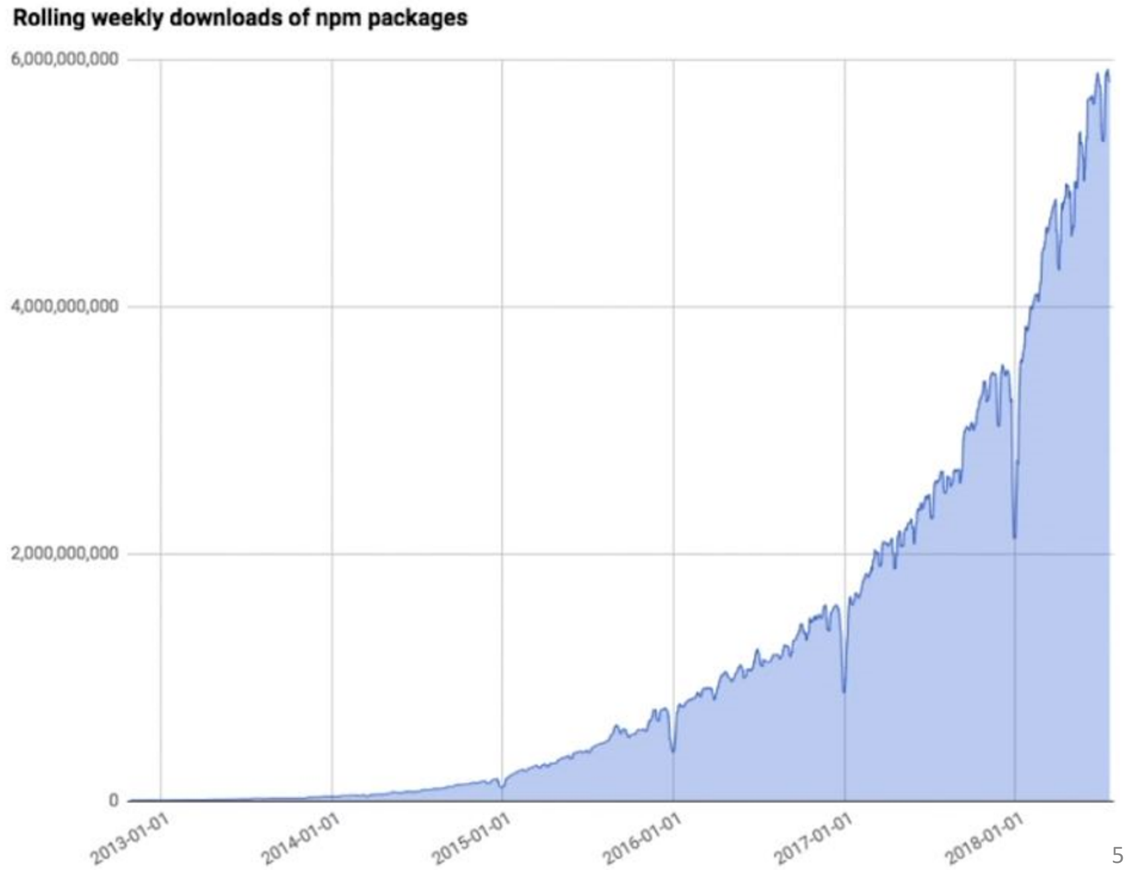
Package is a collection of code (module) you can use it in your code. It's managed by package manager.

Package management system: software that automates installing and updating packages. Deals with what version you have or need, and manages dependencies. Update them when needed.

Dependencies: code (module) that another set of code (module) depends on to work. If you use that code in your app, it is a dependency and your app depends on it. And that code you depend on might depend on another package. this is why we need a strong package manager to manage all dependencies and version control.

10 Million active users are using npm with 6 Billions download per week

97% of code in Modern Web Applications comes from npm



https://www.npmjs.com

https://npm.community

Laurie Voss

PREV NEXT

OUTLINE

Search...

npm: Node Package Manager

The npm registry: source of all codes we are installing: www.npmjs.com
npm is not part of node but a dependency complementing Node.
Another package manager is Yarn from Facebook www.yarnpkg.com

When we install a package:

- Notice dependencies changes in **package.json**
- notice folder: **node_modules**
- This structure separates code from dependencies, at any later time we can copy our code and run: **npm install** (will read all dependencies and install them for us) helpful when we deploy.
- We have three kind of packages: dependency, development and global.
- Add **.gitignore** to your project listing **node_modules** folder

DEMO

```
npm -v            // will print npm version
npm init          // will create package.json
npm i <package> // download & install & audit code from last commit of git repo
                     npm will update package.json automatically
                     other options are: --save-dev (-D) --save-optional (-O)
npm i <package> -g  // download & install a package globally
npm i <package> --dry-run
npm ls -g --depth=0 // show all global packages in your system
npm update        // check versions in package.json and update
npm i npm -g      // update to the latest version of npm
npm outdated -g // show all outdated global packages
npm prune         // delete packages that are not defined in package.json
npm audit         // detailed report of all vulnerabilities in your app
npm audit fix     // try to fix all vulnerabilities, --force for adventurous
npx <package>     // for commands you use once a while, it will download and install
                     and run the package and delete itself
```

MORE...

```
npm config list l // display the default npm settings
npm config set init-author-name "Asaad Saad"
npm config delete init-author-name
npm config set save true // automatically --save (-S)

npm search lint // search online for package with lint in the name
npm home <package> // open browser to package homepage
npm repo <package> // open browser to package repository
```

Semantic Versioning

We will use semantic versioning to Giving a version of code meaning:

MAJOR.MINOR.PATCH

- **PATCH**: Some bugs were fixed. Your code will work fine
- **MINOR**: some new features were added. Your code will work fine.
- **MAJOR**: Big changes. Your code will break (maybe)

www.semver.org

package.json Manifest

```
{
        "name": "nodejs-test-application",
        "version": "1.0.0",
        "description": "NodeJS Test App",
        "main": "app.js",
        "scripts": {
                "test": "echo \"Error: no test specified\" && exit 1"
        },
        "author": "Asaad Saad",
        "license": "ISC",
        "dependencies": {
                "moment": "^2.10.6" // MAJOR.MINOR.PATCH
        }
}
// "scripts" defines commands we can run using (npm run commandName)
// ^ (caret) only update minor and patches
// ~ (tilde) only update patches
```

Dev and Global Packages

- **Development Dependencies**: Needed only while I'm developing the app. It's not needed for running the app.

```
npm install jest --save-dev
// notice devDependencies entry now in package.json
```

- **Global Dependencies**: Available to all applications

```
npm install –g nodemon
nodemon app.js
```

package-lock.json

- The purpose of the **package-lock** is to avoid the situation where installing modules from the same **package.json** results in two different installs. (npm 6 locks by default)
- **package-lock** is a large list of each dependency listed in your **package.json**, the specific version that should be installed, the location of the module (URI), a hash that verifies the integrity of the module, and a list of dependencies, so the install it creates will be the same, every single time.
- **package.json** overrules the **package-lock** if **package.json** has been updated.
- You should commit your **package-lock** to source control

Using a Package

```
npm install moment // moment is a package that parse, validate, manipulate and
display dates
```

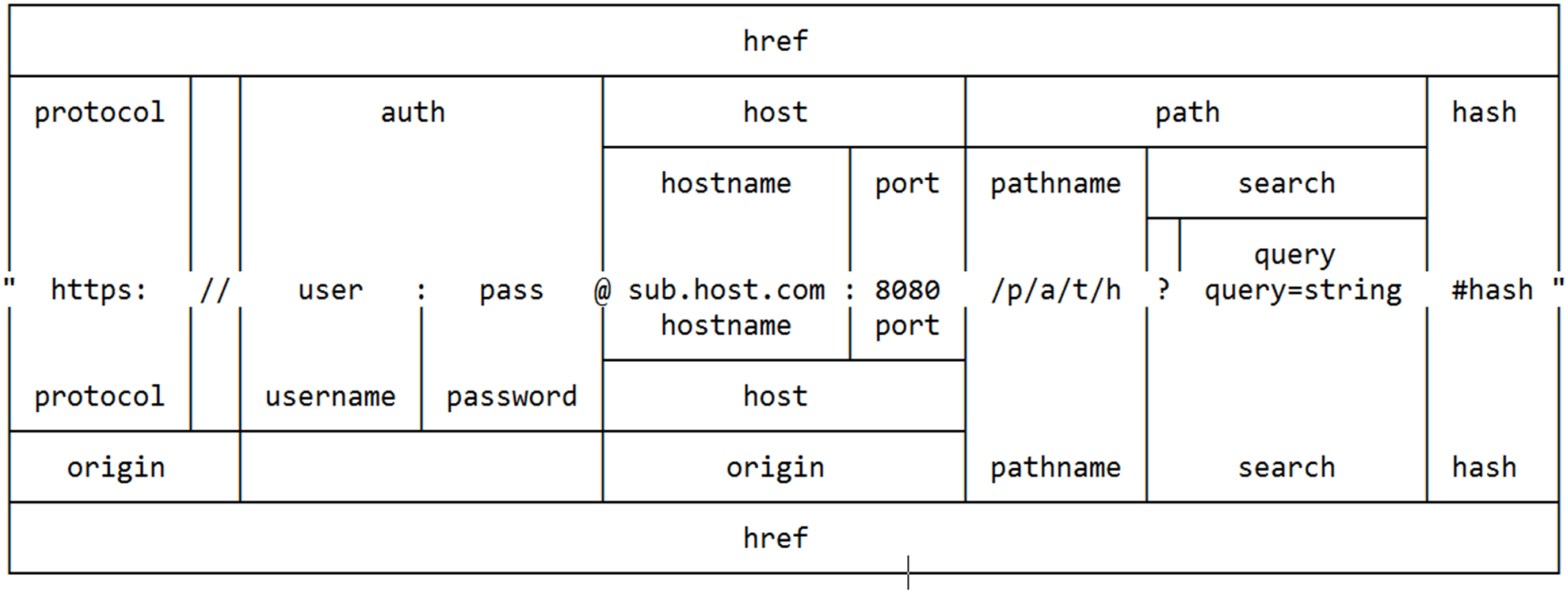
```
var moment = require('moment');
console.log(moment().format("ddd, hA")); // Mon, 10AM
```

How Node resolves a core module path without (./): If it is not a Core Node module then: it looks at the paths list found in module

Reading Get and Post Data

- Handling basic GET & POST requests is relatively simple with Node.js.
- Handling file uploads (multi-part POST requests) is not simple with Node.js, and therefore we will learn more about it by using middleware in Express Framework.
- We use the `url` module to parse and read information from the URL.
- The `url` module uses the WHATWG URL Standard (https://url.spec.whatwg.org/)

Elements in URL Object



https://nodejs.org/api/url.html

Using `url` core Module

You may read the request url from `req.url`

```
const url = require('url');
const myURL = url.parse('https://user:pass@sub.host.com:8080/p/a/t/h?name=asaad#hash');

myURL { protocol: 'https:',
        slashes: true,
        auth: 'user:pass',
        host: 'sub.host.com:8080',
        port: '8080',
        hostname: 'sub.host.com',
        hash: '#hash',
        search: '?name=asaad',
        query: 'name=asaad',
        pathname: '/p/a/t/h',
        path: '/p/a/t/h?query=string',
        href: 'https://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash'
}
```

`url.parse(str)` will return URL object with properties (protocol, hostname, port, pathname, hash)

Parsing the Query String

```
const url = require('url');
const myURL = url.parse('https://user:pass@sub.host.com:8080/p/a/t/h?name=asaad#hash', true);
myURL { protocol: 'https:',
        slashes: true,
        auth: 'user:pass',
        host: 'sub.host.com:8080',
        port: '8080',
        hostname: 'sub.host.com',
        hash: '#hash',
        search: '?name=asaad',
        query: { name: 'asaad' },
        pathname: '/p/a/t/h',
        path: '/p/a/t/h?query=string',
        href: 'https://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash'
}
```

17

PREV NEXT

OUTLINE

Search...

Format a URL

```
const urlObject = {
        protocol: 'http',
        host: 'www.mum.edu',
        search: '?q=CS572',
        pathname: '/search', };

console.log( url.format(urlObject) );  // http://www.mum.edu/search?q=CS572
```

Using `querystring` core module

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

querystring.stringify({
                        name: 'Asaad Saad',
                        course: 'CS572 Modern Web Applications'
                        })
// 'name=Asaad%20Saad&course=CS572%20Modern%20Web%20Applications'

querystring.parse('name=Asaad%20Saad&course=CS572%20Modern%20Web%20Applications');
// { name: 'Asaad Saad', course: 'CS572 Modern Web Applications' }
```


PREV NEXT

Reading Post Data

- Handling POST data is done in a **non-blocking way**, by using asynchronous callbacks. Because POST requests can potentially be very large - multiple megabytes in size. Handling the whole bulk of data in one go would result in a blocking operation.
- To make the whole process non-blocking, Node.js serves our code the POST data in small chunks (**stream**), callbacks that are called upon certain events. These events are **`data`** (a new chunk of POST data arrives) and **`end`** (all chunks have been received).
- We need to tell Node.js which functions to call back to when these events occur. This is done by adding listeners to the `request` object

Reading Post Data

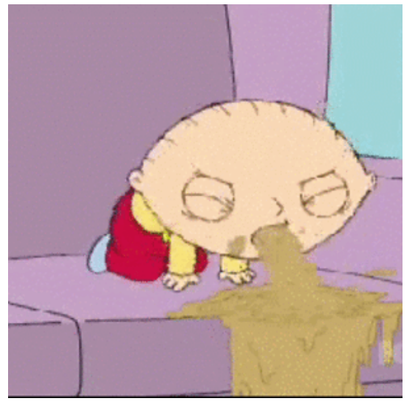
```
const http = require("http");
const querystring = require("querystring");

function onRequest(request, response) {
        let postData = "";
        request.setEncoding("utf8");
        request.on("data", function(postDataChunk) {
                postData += postDataChunk;
        });
        request.on("end", function() {
                const myTextAreaValue = querystring.parse(postData).text;
        });
        response.end();
}
http.createServer(onRequest).listen(8888);
console.log("Server has started on port 8888.");
```

```
<form action="/upload" method="post">
   <textarea name="text"></textarea>
   <input type="submit" />
</form>
```

21

Routing

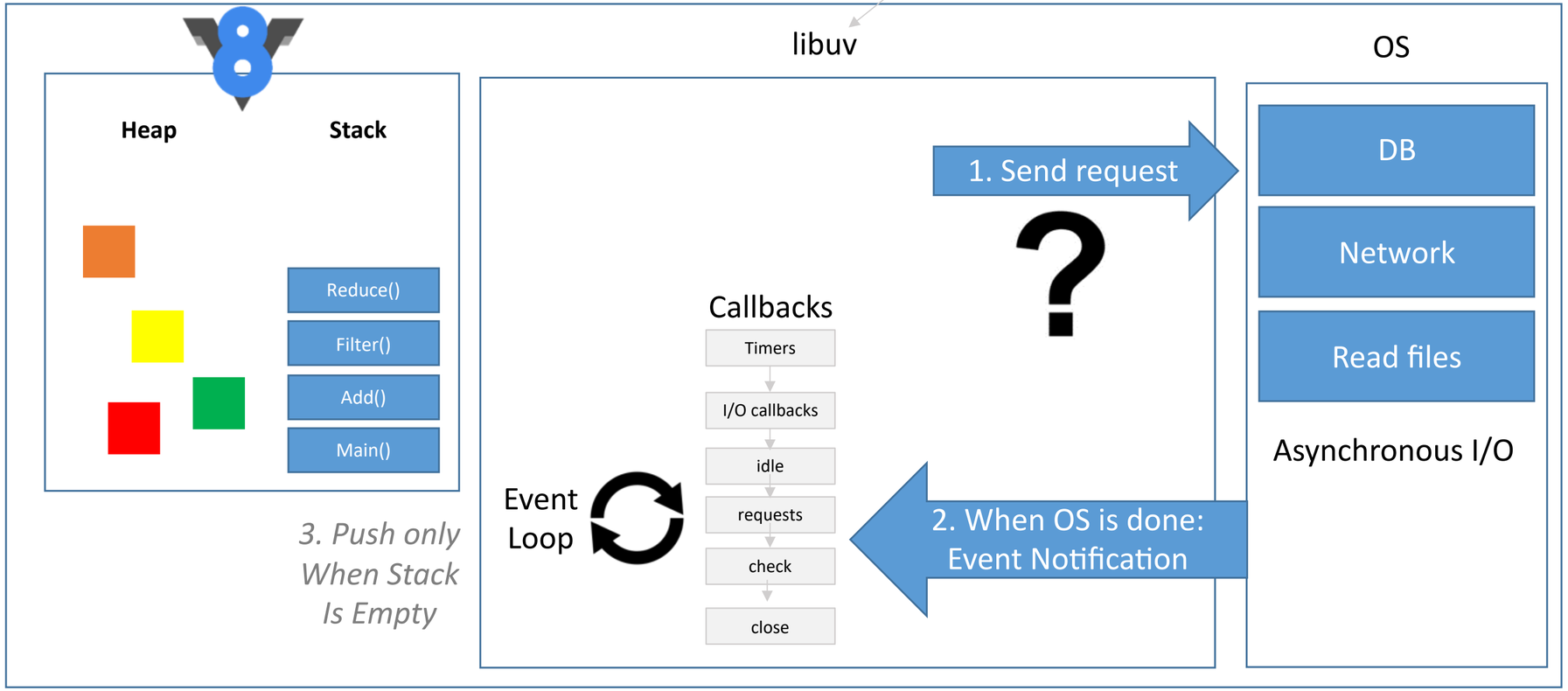


```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res) {
        if (req.url === '/') {
                fs.createReadStream(__dirname + '/index.htm').pipe(res);
        } else if (req.url === '/api') {
                res.writeHead(200, { 'Content-Type': 'application/json' });
                var obj = { firstname: 'Asaad', lastname: 'Saad' };
                res.end(JSON.stringify(obj));
        } else {
                res.writeHead(404);
                res.end();
        }
}).listen(1337, '127.0.0.1');
```

JS on the Server

Part of NodeJs



23

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** (main loop, main thread), 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 pre-allocated pool is 4 worker threads.

What code runs on the Worker Pool?

These are the Node module APIs that make use of this Worker Pool:

I/O-intensive

- DNS
- File System

CPU-intensive

- Crypto
- Zlib

PREV NEXT

Code Example Using Thread Pool

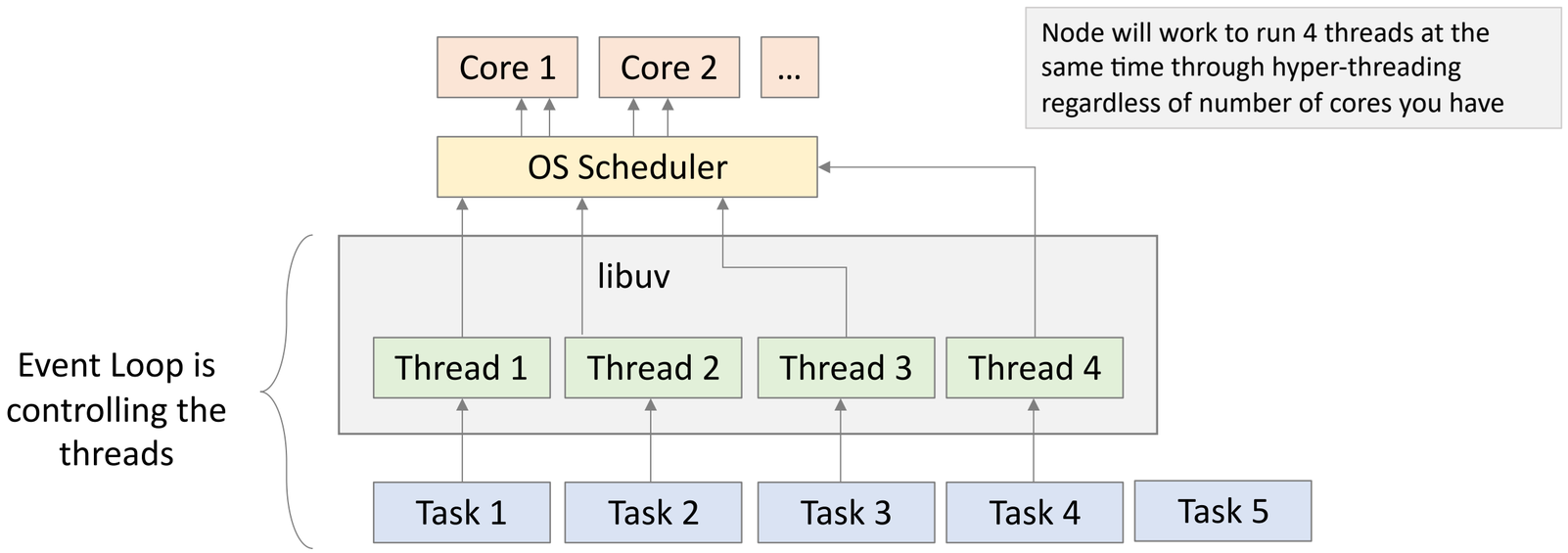
```
//SET UV_THREADPOOL_SIZE=2 && node threads.js
//SET UV_THREADPOOL_SIZE=5 && node threads.js

const crypto = require('crypto');

const start_time = Date.now();

crypto.pbkdf2("A", "B", 100000, 512, 'sha512', () => {
        console.log(`1. ${Date.now() - start_time}`);
})
crypto.pbkdf2("A", "B", 100000, 512, 'sha512', () => {
        console.log(`2. ${Date.now() - start_time}`);
})
crypto.pbkdf2("A", "B", 100000, 512, 'sha512', () => {
        console.log(`3. ${Date.now() - start_time}`);
})
```

Threading



Node will work to run 4 threads at the same time through hyper-threading regardless of number of cores you have

All **fs** module functions use the threadpool, the tasks callback functions are usually scheduled as phase 2 in the event loop. If libuv sees that we are attempting to do an HTTP request, since neither libuv nor Node has any code to handle it, libuv delegates the request to the OS.

Code Example Using OS Async Helpers

```
const https = require('https')
const start_time = Date.now()
function requestMe() {
        https.request('https://www.google.com', res => {
                res.on('data', () => { })
                res.on('end', () => { console.log(Date.now() - start_time) })
        }).end();
}
requestMe()
requestMe()
requestMe()
requestMe()
requestMe()
requestMe()
requestMe()
requestMe()
```

Scalability

Every node.js process is single threaded by design. Therefore to get multiple threads, you have to have multiple processes.

Making blocking code or code that isn't ready to run now asynchronous, doesn't mean it won't block the event loop in the future when it runs in V8. We can achieve scalability for our Node application in three ways:

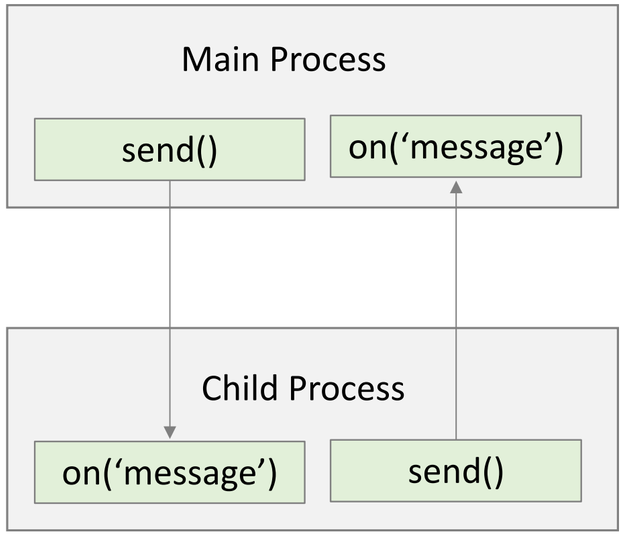
- **Cloning** (using multi processors on single machine)
- **Decomposing** (micro services)
- **Sharding** (using multi machines)

To clone a new child process from the master process you may use:

- **`spawn()`** lunch a command in **new process**
- **`fork()`** same as spawn with ability to **exchange msgs** with the parent process

Child Process

fs and **crypto** and other Node core modules, by default, take advantage of Node multi threads system, there is no need to fork new processes for it. Use child-process only for custom development and other heavy duty work.



Long Operation

```
const http = require('http');
const server = http.createServer();

const longOperation = () => {
        let sum = 0;
        for (let i=0; i<1e9; i++) { sum += i; };
        return sum;
};

server.on('request', (req, res) => {
        const sum = longOperation();
        return res.end(`Sum is ${sum}`);
});

server.listen(3000);
```

Using Child Process

```
const http = require('http');
const { fork } = require('child_process');
const server = http.createServer();

server.on('request', (req, res) => {
    const childProcess = fork('longOperation.js');
    childProcess.send('start');
    childProcess.on('message', sum => {
       res.end(`Sum is ${sum}`);
    });
});

server.listen(3000);
```

```
const longOperation = () => {
   let sum = 0;
   for (let i=0; i<1e9; i++) {
       sum += i;
   };
   return sum;
};

process.on('message', (msg) => {
   const sum = longOperation();
   process.send(sum);
});
```

longOperation.js

Cluster core Module & Load Balancer

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
        const cpus = os.cpus().length;
        console.log(`Forking for ${cpus} CPUs`);
        for (let i = 0; i<cpus; i++) {
                cluster.fork();
        }
} else {
        require('./server');
}
```

When you use load balancing you may need to change your app to handle the shared state cases.

Instance of app will be running on different CPU cores. Every time cluster.fork() triggers it will run again the cluster.js but this time in worker mode

Load balancer keeps a copy of things you create in memory in the master process but you better write your app to use a single store for your app state.

```
const http = require('http');
const pid = process.pid;
http.createServer((req, res) => {
   res.end(`Handled by process ${pid}`);
}).listen(8080, () => {
   console.log(`Started process ${pid}`);
});
```

server.js

Worker Threads

Experimental

The **`worker_threads`** core module enables the use of threads that execute JavaScript in parallel.

Worker threads are useful for performing CPU-intensive JavaScript operations. They will not help much with I/O-intensive work. Node.js's built-in asynchronous I/O operations are more efficient than Workers can be.
Workers, unlike child processes or when using the cluster module, can also share memory efficiently.

Example

Experimental

```
const { Worker } = require('worker_threads');

const worker = new Worker('./worker.js', { workerData: { name: 'Asaad Saad' } });
worker.on('message', (msg) => { console.log(msg); });
```

app.js

```
const { workerData, parentPort } = require('worker_threads');

const data = { ...workerData, course: 'CS572' }
parentPort.postMessage({ data });
```

worker.js