# Advanced Node Scalability

## CS571 – Mobile Application Development

**Maharishi University of Management**

**Department of Computer Science**

**Associate Professor Asaad Saad**

## Maharishi International University - Fairfield, Iowa

## NodeJS

- 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

## 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)

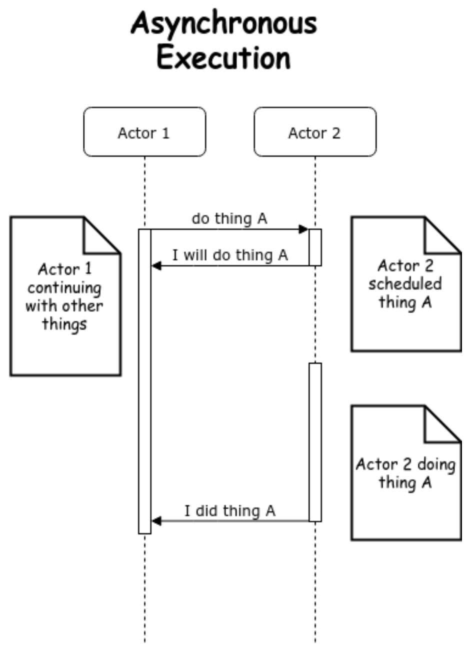
For the client, **I/O takes the form of AJAX requests**. AJAX is by default asynchronous. If AJAX were synchronous then it would lock up the front-end.

# I/O 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 |

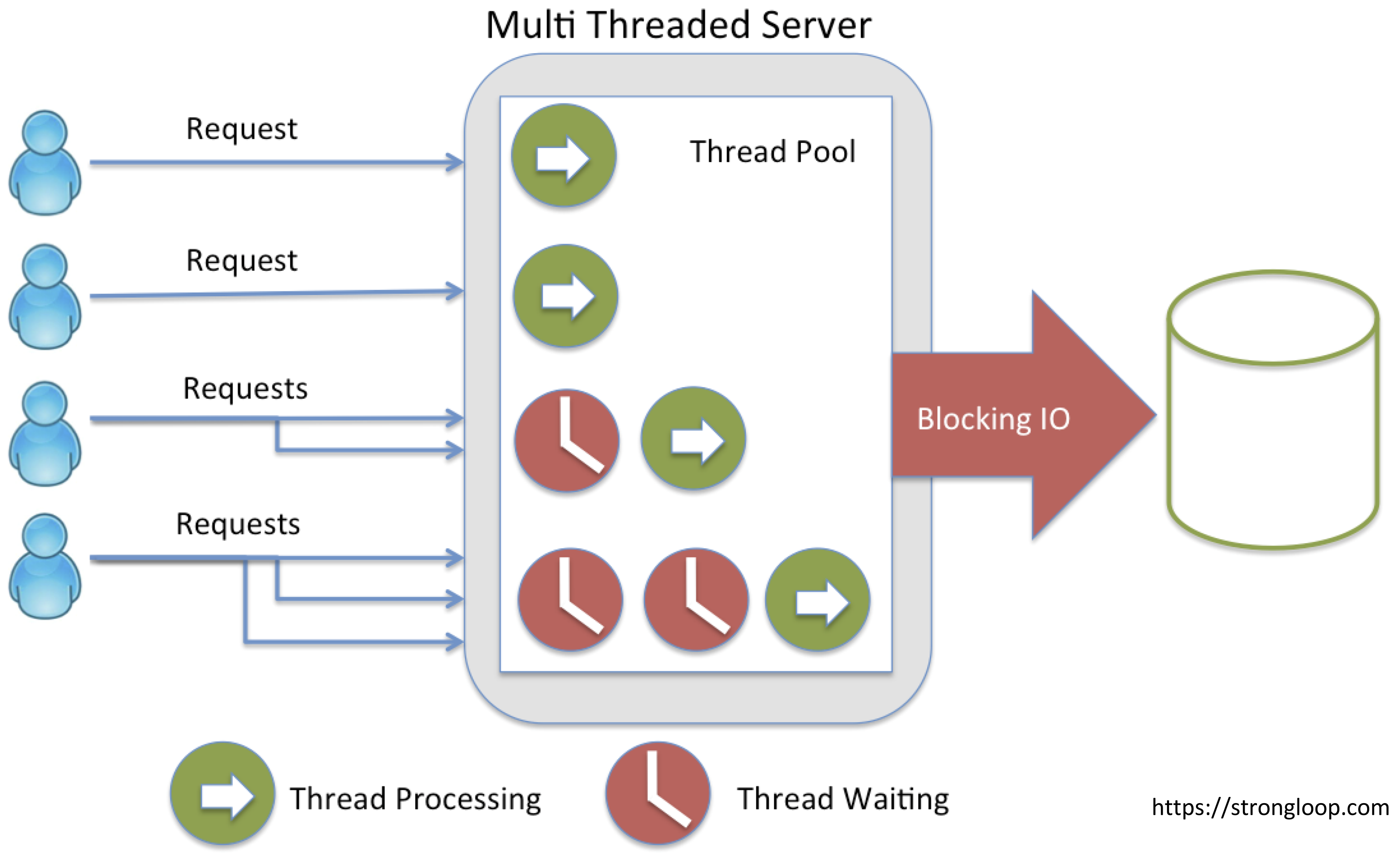
# Parallel, Concurrent, Synchronous and Asynchronous

- **Parallelism** is when tasks are executed in parallel.
- **Concurrency** is when the execution of multiple tasks is interleaved, instead of each task being executed sequentially one after another.
- **Asynchronous** 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 executing at a time one after another.



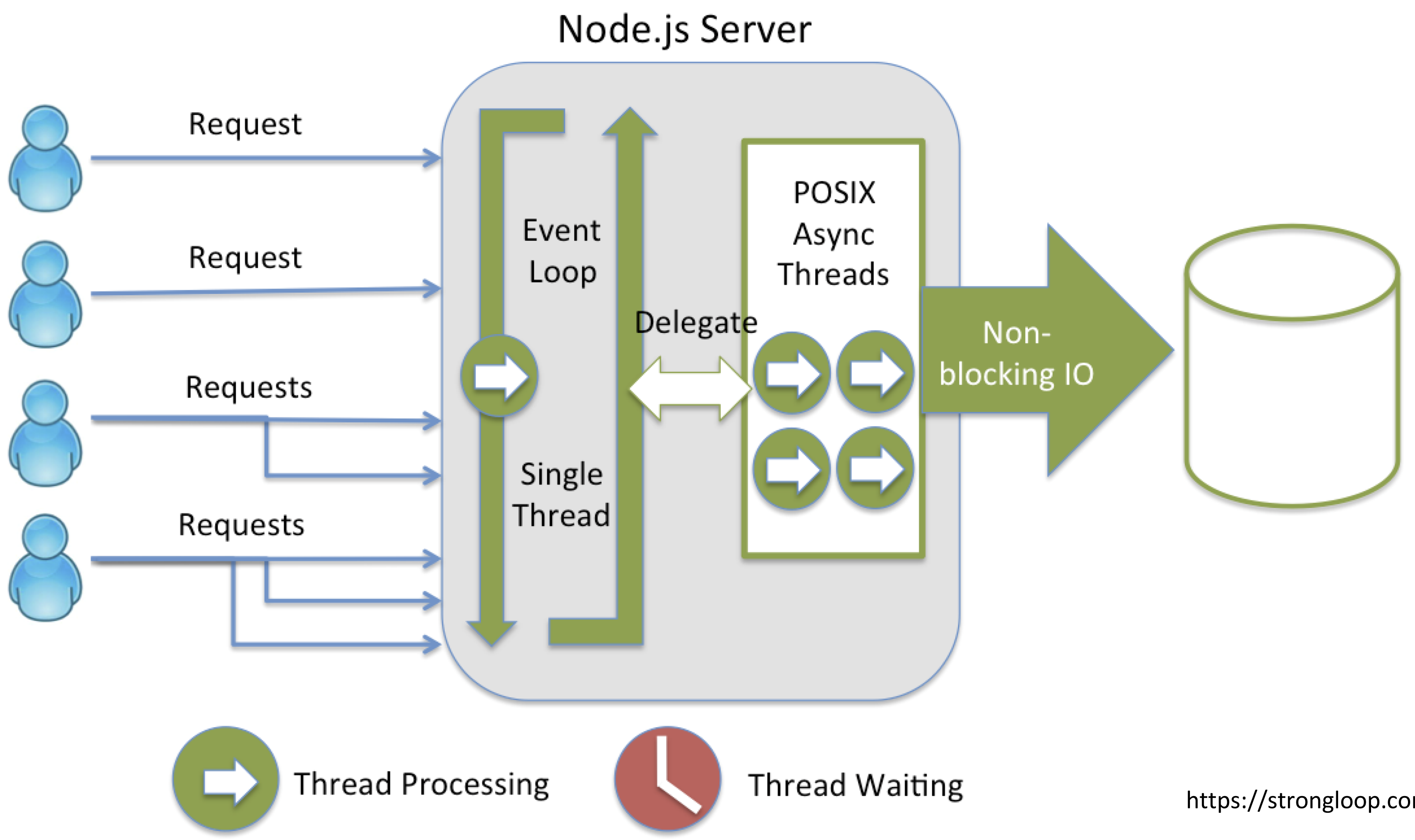
**JavaScript/ V8 is Synchronous**
**Node does things Asynchronously**.

# Threading Java



https://strongloop.com

# Threading Node



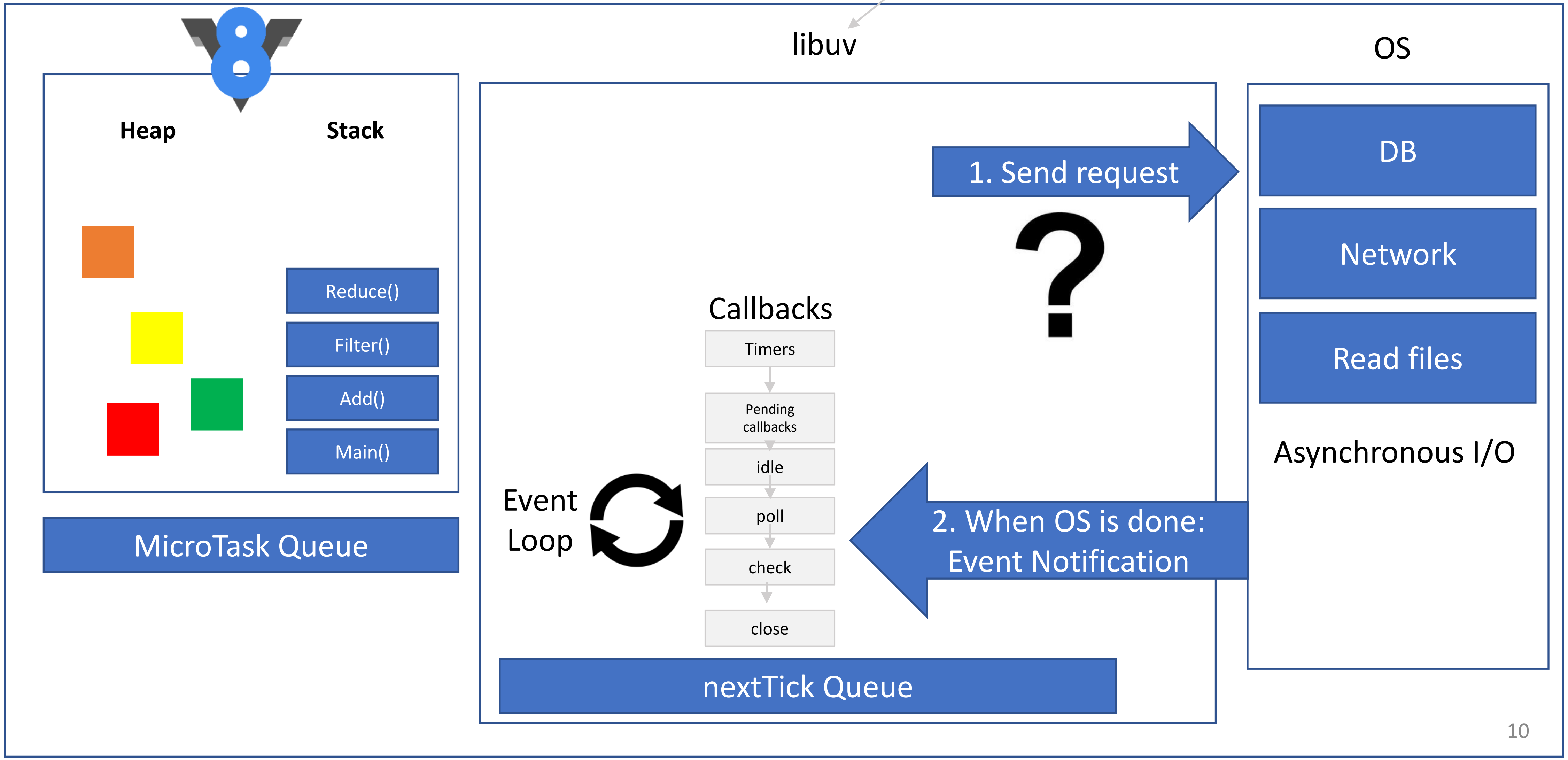
https://strongloop.com

## 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.

# Node JS

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?

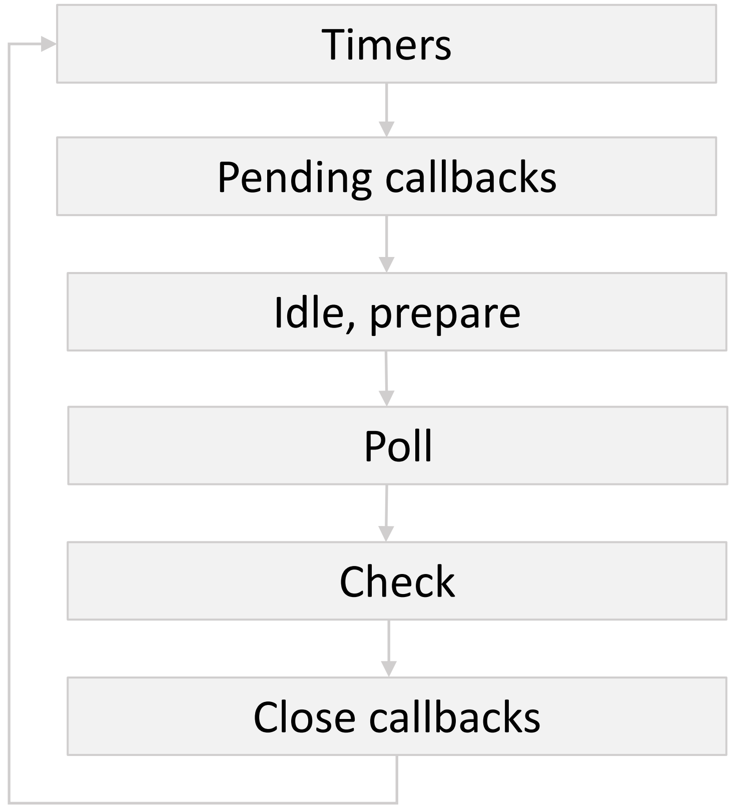
```
const add = (a,b)=>{
       setTimeout(()=>{
              for(let i=0; i<9e7; i++){};
              console.log(a+b)}
       , 5000);
}

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?

## 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

## setTimeout vs setImmediate vs process.nextTick

```
(() => new Promise((resolve) => resolve('promise')))()
       .then((p) => console.log(p))
setTimeout(() => console.log('timeout'), 50000);
setImmediate(() => console.log('immediate'));
queueMicrotask(() => console.log('microtask'));
process.nextTick(() => console.log('nexttick'));
```

What's the output of this code and why?

## Understanding the Event Loop

```javascript
const fs = require('fs');

fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8', function(err, data) {
        setTimeout(() => { console.log('timeout'); }, 0);
        setImmediate(() => { console.log('immediate'); });
        process.nextTick(()=> console.log('nexttick'));
});
```

What is the output of the code?

# 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

## Code Example Using Thread Pool

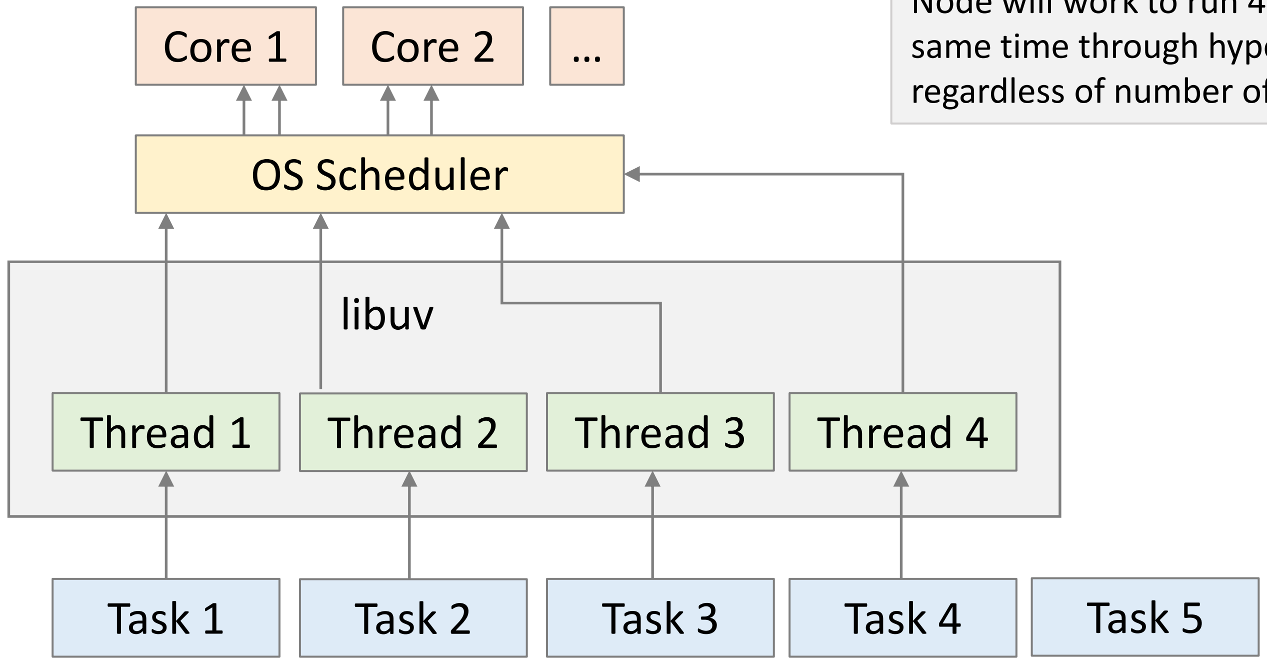
```
//SET UV_THREADPOOL_SIZE=2 && node threads.js (Windows)
//export UV_THREADPOOL_SIZE=2 && node threads.js (MacOS)

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.

## Scalability

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

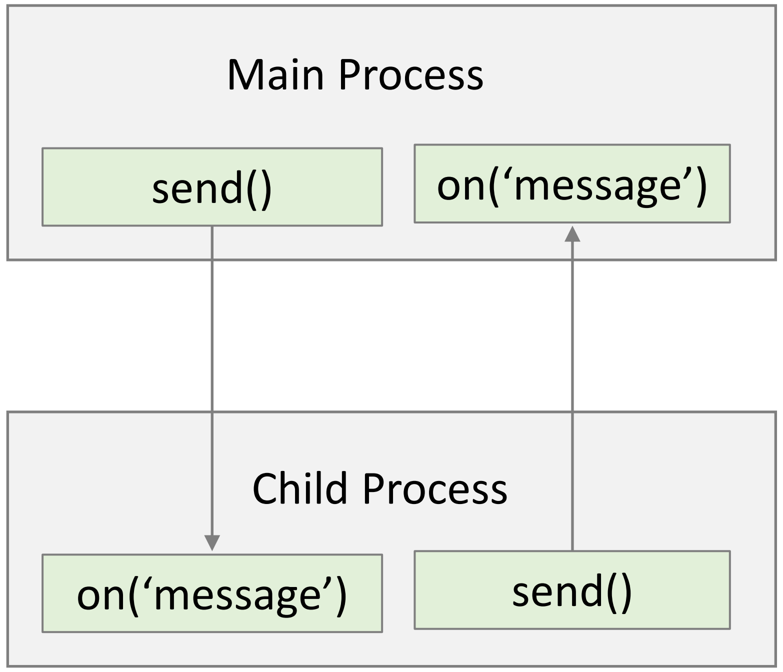
- **Cloning** Child Process
- Using Worker Threads
- Running Node Application as a cluster using all CPU cores
- **Decomposing** (microservices)
- **Sharding** (using multi machines)

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.



## Blocking 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

## Worker Threads

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 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

```
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
```

# 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

```
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

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.

## Reactive benefits

We can add logging, buffer, filter, merge and many more operations on events, because of the large number of operations in RxJS library.

We can handle errors and memory deallocation easily.

Reactive programming makes reasoning about asynchronous events, controlling their timing, and combining them simpler. It is much simpler than using callbacks, and even simpler than using Promises. Since the server logic naturally deals with multiple identical request events, a stream where the events can flow and be processed is a better mental model compared to single-execution Promise abstraction.

## Reactive Server with RxJs

```
const { Subject } = require('rxjs’);
const subject = new Subject();

function sendHello(reqres) {
       reqres.res.end('Hello \n');
}

Subject.subscribe(sendHello)
subject.subscribe(FilterIP)
subject.subscribe(LogToDB)

const http = require('http');
http.createServer((req, res) => {
       subject.next({ req: req, res: res });
}).listen(1337, ()=> console.log('127.0.0.1'));
```