6. Node.js Event Loop & Thread Pool

Wednesday, June 7, 2023 6:18 PM

node.js:
-V8 (JS & C++)
-libuv (C++)
File Access, Network I/O, Event Loop,
Thread Pool (Tread#1, #2, #3, #4)
Access to C++ capabilities through JS

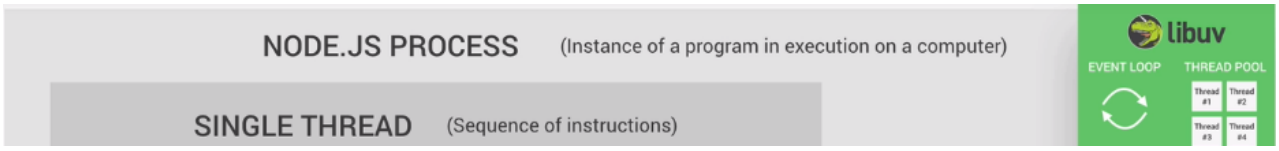
-http-parser
-c-ares (DNS requests)
-OpenSSL (crypto)
-zlib (compression)

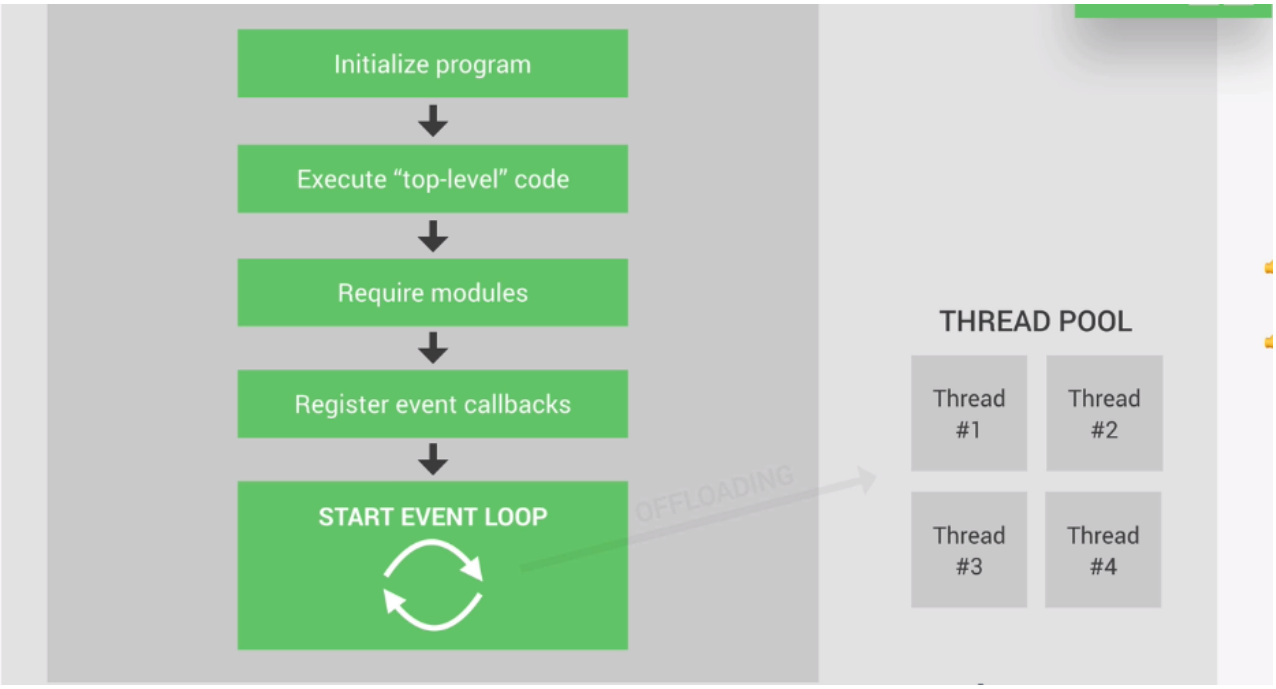
====**Processes, Threads, and Thread Pool**
Threads & Thread Pool:

Node.js process running in a server
-Single Threaded architecture(Sequence of instructions)

0. Initialize program
1. Execute 'top-level' code
2. Require modules
3. Register Event Callbacks
4. Start Event Loop (most of the work)

Thread Pool (by **libuv**):
-4 additional threads (up to 128 threads ~ 32 CPUs)





Event Loop can auto-distribute tasks to Thread Pool

Heavy tasks:

-File System APIs

-Crypto (Password Encryption)

-Compression

-DNS lookups (map domain name to public IPs)

====**Node.js Event Loop**

Heart of Node.js

Event Loop:

1. All Callback functions (non top-level code) run in Event Loop

2. Node.js is built around Callbacks functions
(some operations that finish & return something in the future)

3. Event-driven architecture:

-Events are emitted

-Event loops pick them up

-Callbacks are called

Event Loop receives events each time
something important happens:
-Orchestration:
1. Receives Events
2. Call their callback funcs
3. Off-load resource intensive tasks to Thread Pool

==========

Start - Official Documentation for Offloading

2 options for a destination Worker Thread Pool to
which offload work.

1. Use the built-in Node.js Worker Pool by
developing a **C++ addon**.
Older versions of Node C++ addon,
built using
https://github.com/nodejs/nan

Newer version **N-API**
https://nodejs.org/api/n-api.html

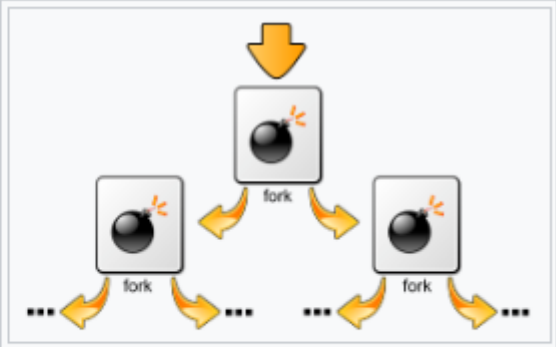
Child Process or Cluster
https://nodejs.org/api/child_process.html

We should NOT simply create a Child Process for every client, as Node.js can receive client requests more quickly than it can create & manage children

From Wikipedia, the free encyclopedia

"Rabbit virus" redirects here. For the disease used in an attempt to exterminate rabbits in Australia, see Myxomatosis.

In computing, a **fork bomb** (also called **rabbit virus**) is a denial-of-service (DoS) attack wherein a process continually replicates itself to deplete available system resources, slowing down or crashing the system due to resource starvation.



The concept behind a fork bomb — the processes continually replicate themselves, potentially causing a denial of service

History [edit]

Around 1978, an early variant of a fork bomb called wabbit was reported to run on a System/360. It may have descended from a similar attack called **RABBITS** reported from 1969 on a Burroughs 5500 at the University of Washington.[1]

Some suggestions for Offloading

We may wish to distinguish between CPU-intensive & I/O-intensive tasks, as they have markedly different characteristics.

A **CPU-intensive** task **only makes progress when** its **Worker** is **scheduled**, and the Worker must be scheduled onto 1 of our machine's logicial cores.

If we have 4 Logical Cores & 5 Workers, 1 of these Workers cannot make progress. We will be paying overhead (memory & scheduling costs) for this Worker & getting no return for it.

I/O-intensive tasks involve querying an external provider such as **fetching API servers**, **DNS**, **file system**

**** Dealing with Variations in task-lengths ****
Uneven resource distributions across

the same Essence of I/O-driven tasks
e.g.
File System fs.readFile() fs.writeFile()

Variation example: Long-running file system reads

Suppose your server must read files in order to handle some client requests. After consulting the Node.js File system APIs, you opted to use `fs.readFile()` for simplicity. However, `fs.readFile()` is (currently) not partitioned: it submits a single `fs.read()` Task spanning the entire file. If you read shorter files for some users and longer files for others, `fs.readFile()` may introduce significant variation in Task lengths, to the detriment of Worker Pool throughput.

For a worst-case scenario, suppose an attacker can convince your server to read an *arbitrary* file (this is a directory traversal vulnerability). If your server is running Linux, the attacker can name an extremely slow file: `/dev/random`. For all practical purposes, `/dev/random` is infinitely slow, and every Worker asked to read from `/dev/random` will never finish that Task. An attacker then submits `k` requests, one for each Worker, and no other client requests that use the Worker Pool will make progress.

Cryptography crypto.randomBytes()

Variation example: Long-running crypto operations

Suppose your server generates cryptographically secure random bytes using `crypto.randomBytes()`. `crypto.randomBytes()` is not partitioned: it creates a single `randomBytes()` Task to generate as many bytes as you requested. If you create fewer bytes for some users and more bytes for others, `crypto.randomBytes()` is another source of variation in Task lengths.

Task partitioning
Tasks with variable time costs can harm the throughput of the Worker Pool.
Minimizing Task times by partitioning each

Task into comparable-cost sub-Tasks.

Manually partitioning via **fs.read()**
Automatically partitioned via **ReadStream**

For CPU-bound tasks, **asyncAvg** might be
inappropriate for Event Loop, but it is well suited to Worker Pool.

Shorter Tasks expand into a small number of
sub-Tasks, and longer Tasks expand into
a larger number of sub-Tasks.

Number of sub-Tasks completed is **NOT**
a userful metric for the throughput of the Worker Pool.
Concern with the number of Tasks completed.

Avoiding Task partitioning
Purpose of **Task partitioning** is to **minimize**
the **variation** in **Task times**.

e.g.
Summing an array = Shorter Tasks
Sorting an array = Longer Tasks

2 Ways to reduce variation in Task times
1. We can create 1 Worker Pool for each class of Task.
2. Routing shorter Tasks & longer Tasks to separate Worker Pool

However, **Partitioning Tasks** incurs **Overhead**
[The costs of creating a Worker Pool Task representation
Manipulating the Worker Pool queue]

Measuring Overhead in Node.js

1. Computational cost

How much CPU time the API consumes

2. Memory usage

How much memory the API operation uses

3. Response time

How long does it take to get a response from the API

4. Bandwidth

How much data is being sent & received

a. Logging Execution Time

You can manually log the start &
end time of your API calls to measure how long they take

javascript 複製

```javascript
const startTime = Date.now();
apiCall().then(() => {
    const endTime = Date.now();
    console.log(`API call took ${endTime - startTime} ms`);
});
```

b. Profiling CPU Usage

Node.js provides built-in modules like
perf_hooks that can help in profiling:

javascript 複製

```javascript
const { performance, PerformanceObserver } = require('perf_hooks');

let obs = new PerformanceObserver((items) => {
  console.log(items.getEntries()[0].duration + " ms");
  performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });

performance.mark('A');
```

```
apiCall().then(() => {
  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
});
```

c. Monitoring Memory Usage
Monitor memory usage using the process global object:

```javascript
console.log(`Memory Usage before API call: ${process.memoryUsage().heapUsed / 1024 / 1024} MB`);
apiCall().then(() => {
  console.log(`Memory Usage after API call: ${process.memoryUsage().heapUsed / 1024 / 1024} MB`);
});
```

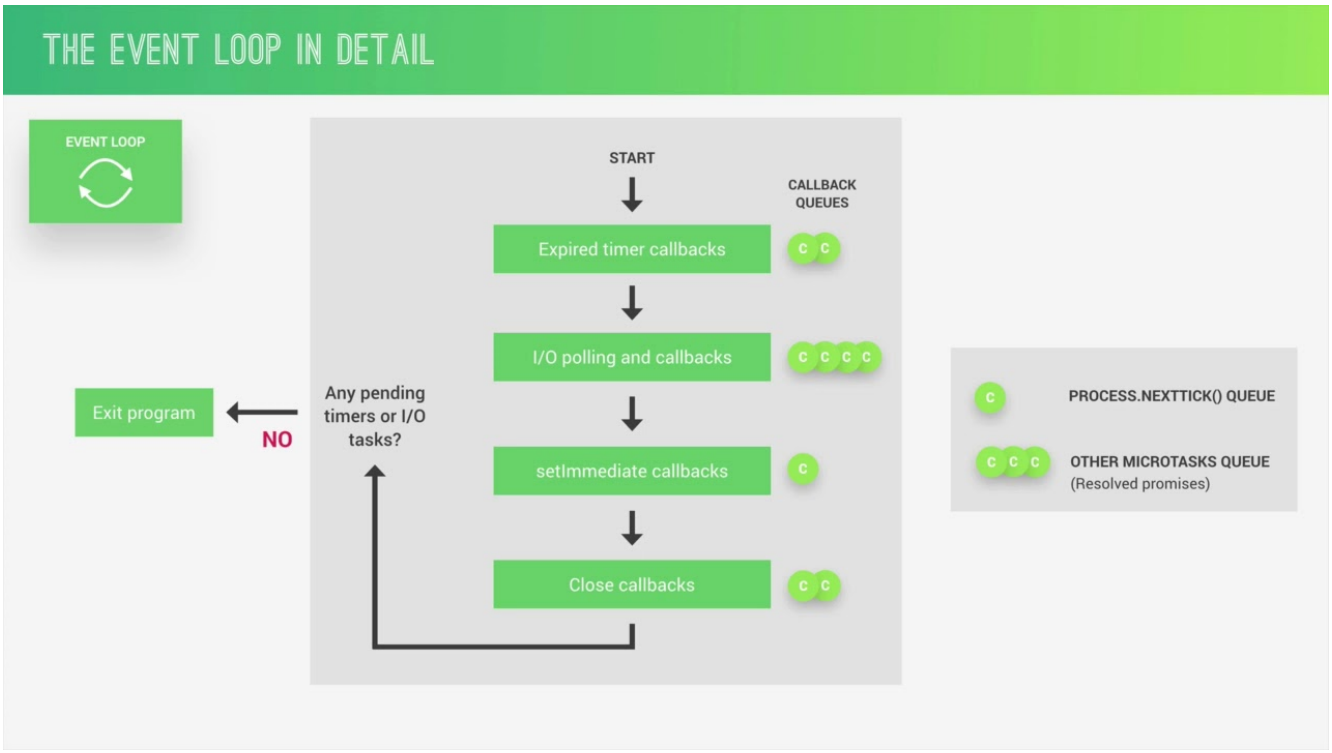
d. Using Profilers & Node.js Tools
clinic.js, 0x, or Node Inspector

3. Analyze Asynchronous Behavior
Understanding how asynchronous API calls
impact the Event Loop is crucial in Node.js:

```javascript
const { async_hooks } = require('async_hooks');

const hook = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    console.log(`${type} async operation started`);
  },
});
hook.enable();
```

End of Official Documentations Offloading
==========
Node runtime starts
Callback Queues:
Callbacks coming from events that Event Loop receives,
sometimes, there's only 1 event queue, each phase has its own

sometimes, there's only 1 event queue, each phase has its own callback queues.



4 Most important Event Loop phases:

0. Start

1. Call 'Expired timer callbacks'
Example:

```
setTimeout(() => {
        console.log('Timer expired!');
        }, ms)
```

If there're Callback funcs from timers that just expired, these are the 1st ones to be proceeded by the Event Loop.

**If timer expires later, during the time, when 1 of the other phases are being processed, then the Callback of that timer will only be

then the callback of that timer will only be
called when Event Loop comes back to this 1st phase.

*Callbacks in each queue are processed 1 by 1,
until there're no one left in the queue,
only then Event Loop will enter next phase.

2. I/O polling & callbacks (looking for new I/O events to callback queue)
Example:
**Networking & File Access, 99% of
our code gets executed here

```
fs.readFile('file.txt', (err, data) => {
    if (err) console.log(err);
    console.log('File read!\n', 'data: \n', data);
    return data;
    });
```

3. setImmediate callbacks
A special kind of timer if we want to
process callbacks immediately after the I/O polling & callbacks phase.
This is only used for some really advanced use-cases.

4. Close callbacks
Not that important
All close events are processed
When a web server / web socket shutdown

There're also 2 other Queues:
i. process.nextTick() queue:

==========

Start of Official Documentation - The Node.js Event Loop
https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick

process.nextTick() will be processed after the current operation is completed, regardless of the current phase of the Event Loop.

process.nextTick() will be resolved before the event loop continues.
This behaviour can create some bad situations because **it allows us to "starve" our I/O by making recursive process.nextTick() calls**, which prevents the event loop from reaching the **poll** phase.

Q/A Why would that be allowed?
All Node.js APIs are designed to be always asynchronous even it does NOT have to be asynchronous.

```javascript
function apiCall(arg, callback) {
  if (typeof arg !== 'string')
    return process.nextTick(
      callback,
      new TypeError('argument should be string')
    );
}
```

JavaScript Copy to clipboard

This passes an error back to the user but
only after we have allowed the rest of the user's code to execute.
By using **process.nextTick()** ,
we guarantee that **apiCall()** always runs
its callback after the rest of the code &
before the event loop is allowed to proceed.

before the event loop is allowed to proceed.

To achieve this, the JS call stack is allowed
to unwind then immediately execute
the provided callback which allows a person
to make recursive calls to **process.nextTick()**
without reaching a
RangeError: Maximum call stack size exceeded from v8.
**

Example of problematic situations

```
let bar;

// this has an asynchronous signature, but calls callba
function someAsyncApiCall(callback) {
  callback();
}

// the callback is called before `someAsyncApiCall` com
someAsyncApiCall(() => {
  // since someAsyncApiCall hasn't completed, bar hasn'
  console.log('bar', bar); // undefined
});

bar = 1;
```

JavaScript Copy to clipboard

The code above is NOT compiled,
bar is undefined during runtime => Errors

Solution:

```
let bar;

function someAsyncApiCall(callback) {
```

```
  process.nextTick(callback);
}

someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});

bar = 1;
```

JavaScript Copy to clipboard

Inside function someAsyncApiCall(),
by placing the callback in a **process.nextTick()**,
the script is still able to run to completion,
allowing all variables, functions etc. to be
initialized prior to the callback being called.
It also has the advantage of not allowing the event loop to continue.

```
const server = net.createServer(() => {}).listen(8080);

server.on('listening', () => {});
```

JavaScript Copy to clipboard

When only a port is passed, the port is bound immediately. So, the `'listening'` callback could be called immediately. The problem is that the `.on('listening')` callback will not have been set by that time.

`process.nextTick()` VS `setImmediate()`

We have two calls that are similar as far as users are concerned, but

their names are confusing.

- `process.nextTick()` fires immediately on the same phase
- `setImmediate()` fires on the following iteration or 'tick' of the event loop

There's a Generic Artifact error in naming process.nextTick() & setImmediate() , however the NPM team cannot
fix this naming issue as there're a large number of
npm packages built & published to npm everyday,
risks of potential breakages are extremely high.

process.nextTick() fires more immediately than setImmediate() ,

> We recommend developers use `setImmediate()` in all cases because it's easier to reason about.

Two main reasons:
1. Allow users to handle errors,
cleanup any then unneeded resources,
or perhaps try the request again before the event loop continues.

2. At times it's necessary to allow
a callback to run after the call stack
has unwound but before the event loop continues.

One example is to match the user's expectations. Simple example:

```
1 const server = net.createServer();
2 server.on('connection', conn => {});
3
4 server.listen(8080);
```

```
server.on('listening', () => {});
```

JavaScript Copy to clipboard

Say that `listen()` is run at the beginning of the event loop, but the listening callback is placed in a `setImmediate()`. Unless a hostname is passed, binding to the port will happen immediately. For the event loop to proceed, it must hit the **poll** phase, which means there is a non-zero chance that a connection could have been received allowing the connection event to be fired before the listening event.

server.on() & server.listen() are both top level code
server.listen(8080); should be at the bottom

Why This Order?

This sequence ensures that:

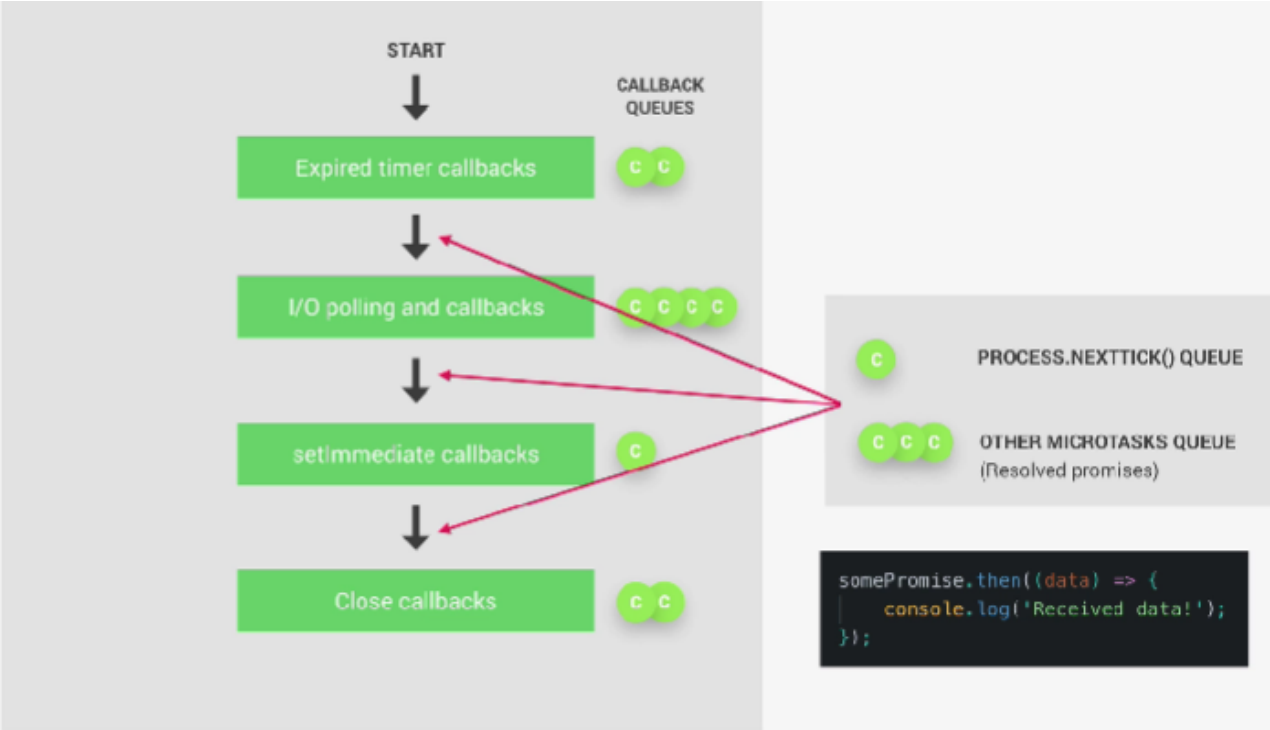
- The server is ready and listening before any connections are actually attempted.
- All event handlers are set up correctly and are in place to handle events as soon as they start occurring.

By placing `server.listen(8080)` before the `server.on('listening', () => {})`, you ensure that the event listener for 'listening' is registered right after the server starts the process of listening but before the event is actually emitted. This is important because if you reverse the order:

javascript 複製

```javascript
const server = net.createServer();
server.on('connection', conn => {});

server.on('listening', () => {});
server.listen(8080);
```

In this reversed order, `server.on('listening', () => {})` is still registered before the server emits the 'listening' event (which is triggered by `server.listen(8080)`). Thus, even in this reversed order, the event handling for 'listening' will work correctly. However, the original order is typically preferred for clarity and logical grouping of related operations (i.e., setting up the server and then starting it).



**** Referring to Event Loop life-cycle ****

Code	Behaviour in Event Loop	Essence
server.listen	runs at the **beginning**	top-level code
server.on	runs during **Poll phase**	top-level code

Another example is extending an `EventEmitter` and emitting an event from within the constructor:

```
const EventEmitter = require('node:events');

class MyEmitter extends EventEmitter {
  constructor() {
    super();
    this.emit('event');
  }
}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
```

```
13 });
```

JavaScript Copy to clipboard

You can't emit an event from the constructor immediately because the script will not have processed to the point where the user assigns a callback to that event. So, within the constructor itself, you can use `process.nextTick()` to set a callback to emit the event after the constructor has finished, which provides the expected results:

```
1  const EventEmitter = require('node:events');
2
3  class MyEmitter extends EventEmitter {
4    constructor() {
5      super();
6
7      // use nextTick to emit the event once a handler is
8      process.nextTick(() => {
9        this.emit('event');
10     });
11   }
12 }
13
14 const myEmitter = new MyEmitter();
15 myEmitter.on('event', () => {
16   console.log('an event occurred!');
17 });
```

JavaScript Copy to clipboard

**** Important concepts ****

We cannot emit an event during inheritance phase from the constructor immediately because MyEmitter class has NOT been instantiated yet

because MyEmitter class has NOT been instantiated yet.
Thus, we can use **process.nextTick()** to set
a callback to emit the event after
the constructor has finished, which provides the expected results.

https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick
End of Official Documentation - The Node.js Event Loop

==========

ii. Other Microtasks queue: Resolved Promises

If there are any Callbacks in 1 of
these special queues to be processed,
they'll be executed right after the current phase finishes,
instead of waiting for the entire Event Loop to finish.

Execute Callbacks right after current Event Loop phase

Other Microtasks queue

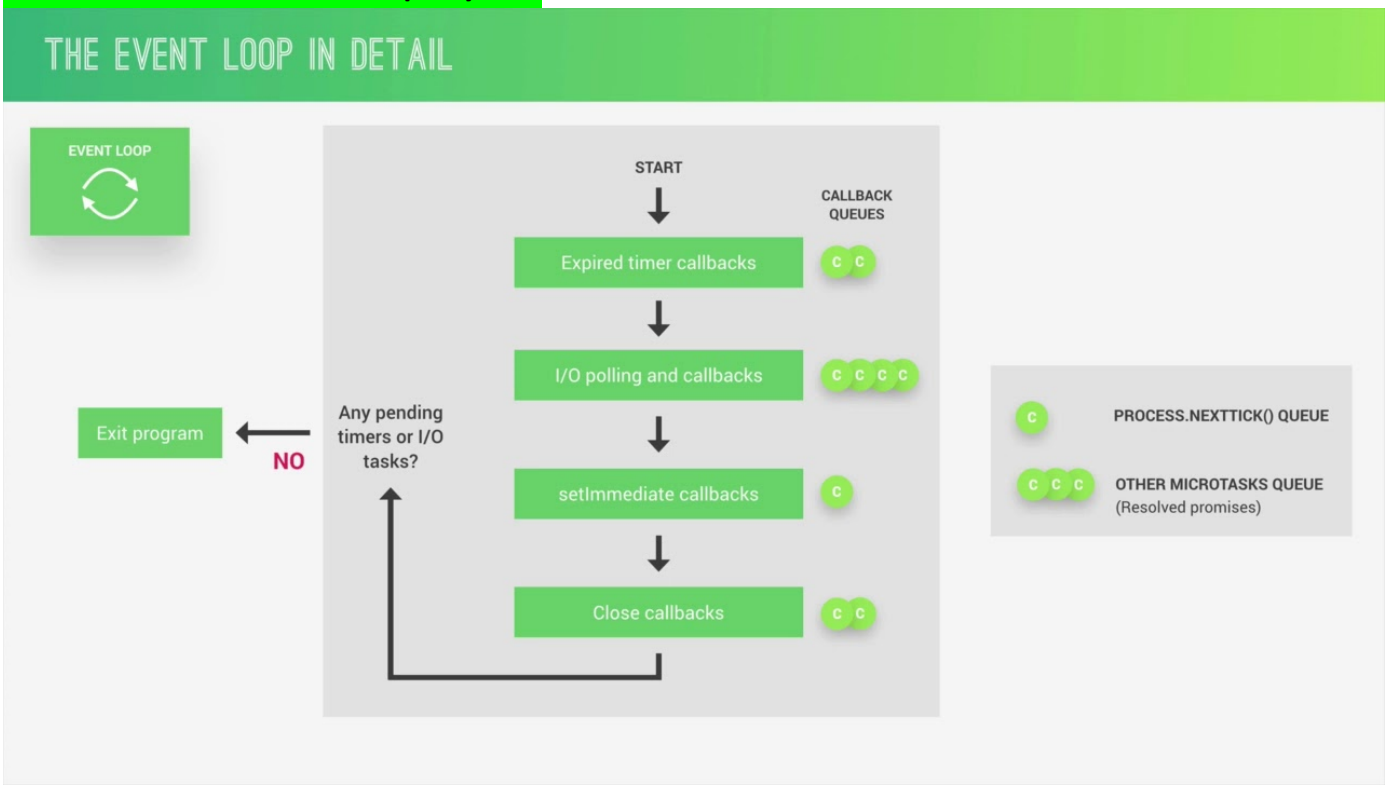
```
somePromise.then((data) => {
    console.log('Received data!');
    });
```

i. Process.nextTick() queue:
Process.nextTick() queue is only used,
when we **really need to execute a certain**
Callback right after the current Event Loop phase.

Similar to setImmediate, yet **setImmediate only runs right after I/O callback phase finishes**)

[really advanced use-case]

1 Tick = 1 Event Loop cycle



Event Loop => Any pending timers or I/O tasks ?
(Proceed NextTick) : (Exit Program)

i.e. Node-farm project
-When we're listening for incoming HTTP requests,
we're basically running an I/O tasks, thus node.js
keeps running & listening to HTTP requests coming in,
rather than Exiting the app

-When fs.readFile or fs.writeFile, there's also an I/O task,
node won't Exit

It's really important to grasp the concepts of
Node.js Event Loop to debug.

It's the **Event Loop makes**

It's the **Event Loop makes**
Asynchronous programming possible in Node.js,
making it different from other platforms.

Event Loop does the orchestration to
off-load heavy tasks such as File Access,
Networking to the Thread Pool & doing the simpler works itself.

Why we need the Node.js Event Loop?
Cuz, in Node.js, everything works in 1 single thread,
thus you can have thousands of users
using the same thread at the same time.

However, there's a danger of blocking the single thread,
making our entire app slow or
even stopping all users from accessing the entire app.

Apache + PHP:
A new thread is created for each new user.
Way more resource-intensive,
there's no danger of blocking Event Loop.

****How NOT to block Event Loop in Node.js:**
1. Do NOT use SYNC functions in fs,
crypto and zlib modules in Callback functions

2. Do NOT perform complex calculations in Event Loop
(e.g. Loops inside Loops)

3. Be careful with JSON in very large objects
It takes long-time to
JSON.parse() || JSON.stringify()
These are **O(n)** in the length of the input, for large n they can take

surprisingly long

Example: JSON blocking. We create an object `obj` of size 2^21 and `JSON.stringify` it, run `indexOf` on the string, and then JSON.parse it. The `JSON.stringify`'d string is 50MB. It takes 0.7 seconds to stringify the object, 0.03 seconds to indexOf on the 50MB string, and 1.3 seconds to parse the string.

```javascript
let obj = { a: 1 };
let niter = 20;

let before, str, pos, res, took;

for (let i = 0; i < niter; i++) {
  obj = { obj1: obj, obj2: obj }; // Doubles in size ea
}

before = process.hrtime();
str = JSON.stringify(obj);
took = process.hrtime(before);
console.log('JSON.stringify took ' + took);

before = process.hrtime();
pos = str.indexOf('nomatch');
took = process.hrtime(before);
console.log('Pure indexof took ' + took);

before = process.hrtime();
res = JSON.parse(str);
took = process.hrtime(before);
console.log('JSON.parse took ' + took);
```

JavaScript Copy to clipboard

npm modules that offer asynchronous JSON APIs:

- JSONStream

- JSONStream
https://www.npmjs.com/package/JSONStream

- Big-Friendly JSON
https://www.npmjs.com/package/bfj
This uses partitioning-on-the-Event-Loop paradigm outlined below

Complex calculations without blocking the Event Loop
Partitioning

Partitioning

You could *partition* your calculations so that each runs on the Event Loop but regularly yields (gives turns to) other pending events. In JavaScript it's easy to save the state of an ongoing task in a closure, as shown in example 2 below.

For a simple example, suppose you want to compute the average of the numbers `1` to `n`.

Example 1: Un-partitioned average, costs `O(n)`

```
1 for (let i = 0; i < n; i++) sum += i;
2 let avg = sum / n;
3 console.log('avg: ' + avg);
```

JavaScript Copy to clipboard

Example 2: Partitioned average, each of the `n` asynchronous steps costs `O(1)`.

```
1 function asyncAvg(n, avgCB) {
```

```javascript
  // Save ongoing sum in JS closure.
  let sum = 0;
  function help(i, cb) {
    sum += i;
    if (i == n) {
      cb(sum);
      return;
    }

    // "Asynchronous recursion".
    // Schedule next operation asynchronously.
    setImmediate(help.bind(null, i + 1, cb));
  }

  // Start the helper, with CB to call avgCB.
  help(1, function (sum) {
    let avg = sum / n;
    avgCB(avg);
  });
}

asyncAvg(n, function (avg) {
  console.log('avg of 1-n: ' + avg);
});
```

JavaScript | Copy to clipboard

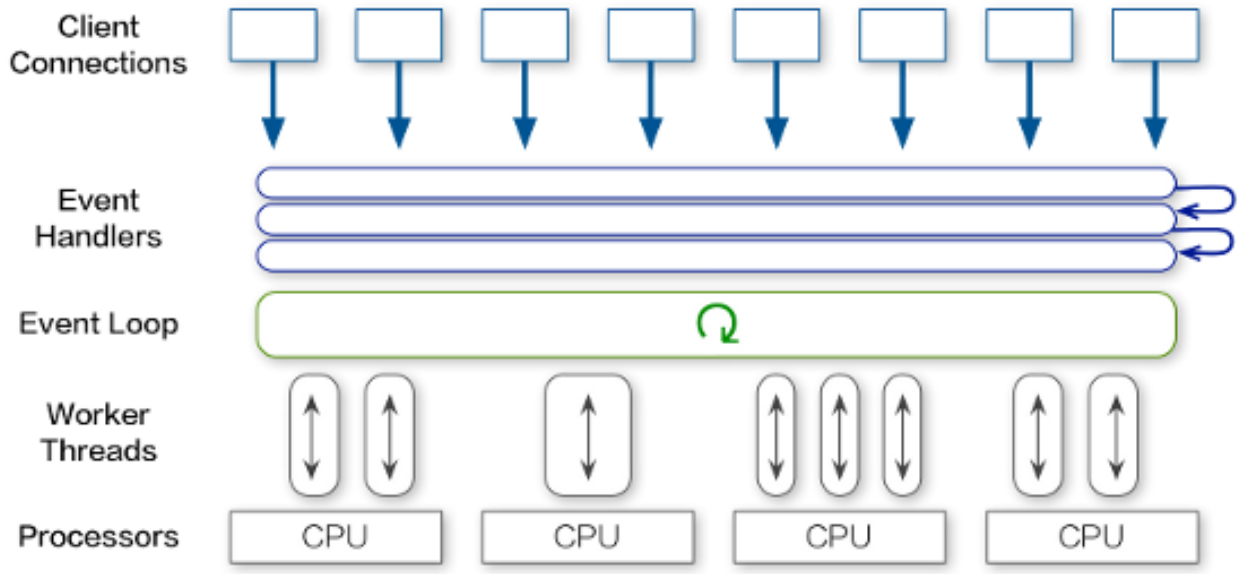
Offloading

Moving expensive tasks from
Event Loop onto the Worker Thread Pool
https://www.npmjs.com/package/webworker-threads

Event Loop Orchestration:
Juggle Events, Listeners and Callbacks quickly & without any hiccups nor interruptions that would ruin its performance

Illustrated Writeup

There is an illustrated writeup for the original use case of this module:



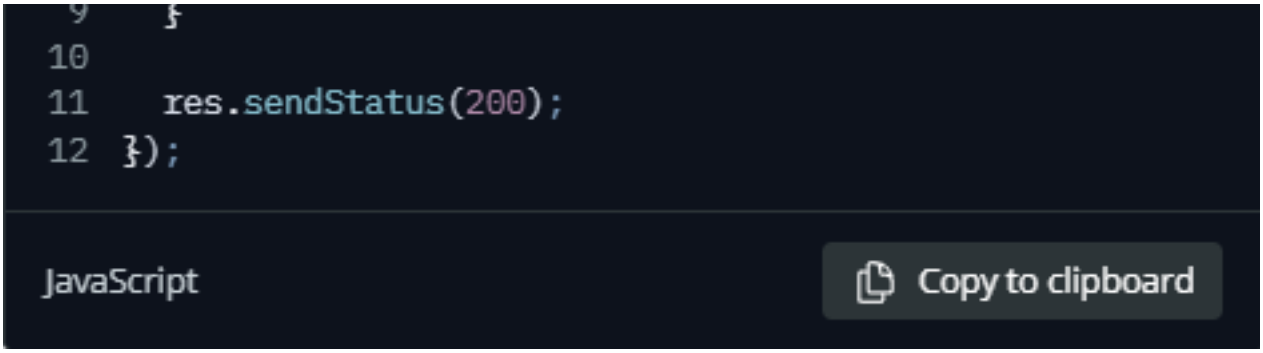
4. Do NOT use too complex Regular Expressions
e.g. Nested Quantifiers
Vulnerable Regex = Taking O(n^2) <u>exponential time</u>
i. Avoid nested quantifiers like (a+)*
V8's regexp engine can only handle some of these quickly

ii. Avoid **OR**'s with overlapping clauses, like (a|a)*

iii. **indexOf** for string match = cheaper resources & never > O(n)

example of vulnerable regexp

```
app.get('/redos-me', (req, res) => {
  let filePath = req.query.filePath;

  // REDOS
  if (filePath.match(/(\/.+)+$/)) {
    console.log('valid path');
  } else {
    console.log('invalid path');
```



This has doubly-nested quantifier:
If a client queries with filePath ///.../\n
[100 /'s followed by a newline character
that the regexp's "." won't match),
then the Event Loop will take effectively forever,
blocking the Event Loop.

This client's REDOS attack causes
all other clients not to get a turn until the regex match finishes.

Tools to check for REDOS (vulnerable) regex
https://github.com/davisjam/safe-regex

https://github.com/superhuman/rxxr2

Expensive / heavy lifting / resource draining APIs:
- Encryption
- Compression
- File system
- Child process

These APIs are only intended for
scripting convenience, but are not intended for
use in the server context.
If you execute them on the Event Loop,
they will take far longer to complete than

they will take far longer to complete than
a typical JavaScript instruction, blocking the Event Loop.

In a Node.js server, we should NOT use these Synchronous APIs:
- Encryption:
    - **crypto.randomBytes** (synchronous version)
    - **crypto.randomFillSync**
    - **crypto.pbkdf2Sync**

- Compression:
    - **zlib.inflateSync**
    - **zlib.deflateSync**

- File system:
    - Do not use the synchronous file system APIs e.g. NFS
    - **fs.readFileSync**
    - **fs.writeFileSync**

- Child process:
    - **child_process.spawnSync**
    - **child_process.execSync**
    - **child_process.execFileSync**

<u>Potential Solutions</u>:
-Manually off-loading Heavy Tasks to Thread Pool
-Using Child processors

====Node.js Event Loop in Practice

```
// import file system module
const fs = require('fs');
// 0. Start app
```

```
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop
setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));
fs.readFile('./test-file.txt', () => {
    console.log('I/O finished');
})
console.log('Hello from top-level code');
```

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
```

```
// ========================
// import file system module
const fs = require('fs');
// 0. Start app
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop
setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));
fs.readFile('./test-file.txt', () => {
    console.log('I/O finished');
    setTimeout(() => console.log('Timer 2 finished'), 0);
     // Event Loop finds Timer 3 as pending after I/O finished
    setTimeout(() => console.log('Timer 3 finished'), 3000);

     // When there's no I/O callbacks in the queue
     // Event Loop checks if there's any setImmediate()
     // And execute setImmediate() right away after I/O finished
     // Even before expired timers
    setImmediate(() => console.log('Immediate 2 finished'));
})
console.log('Hello from top-level code');
```

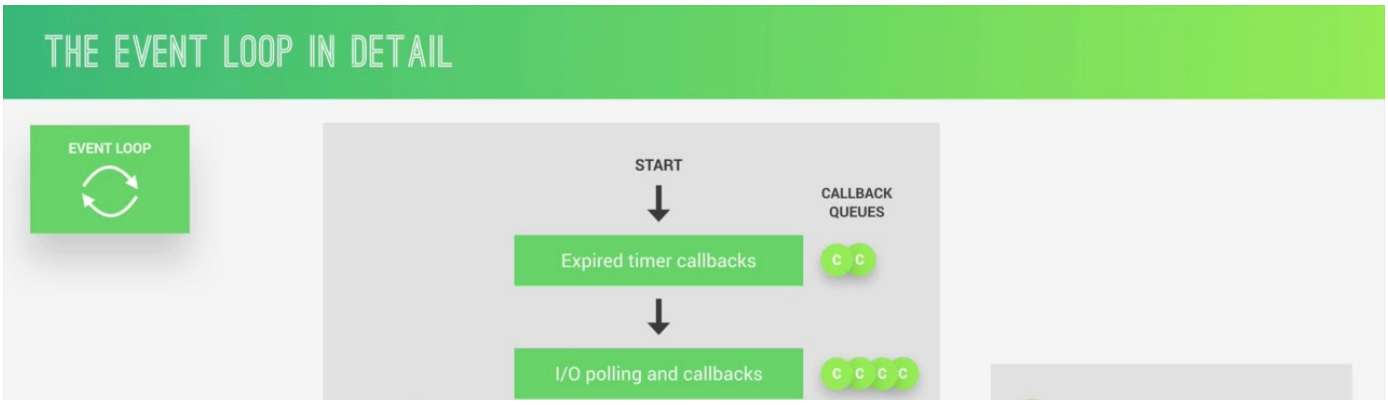
```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
```

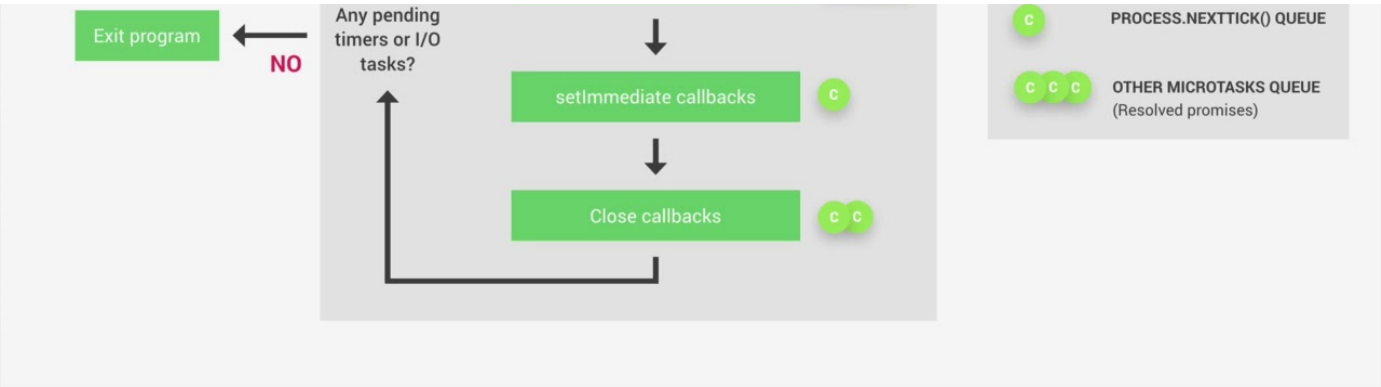
```
Immediate 2 finished
Timer 2 finished
```

Program kept running until Timer3 finished

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Immediate 2 finished
Timer 2 finished
Timer 3 finished
```

```
// import file system module
const fs = require('fs');
// 0. Start app
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop
setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));
fs.readFile('./test-file.txt', () => {
    console.log('I/O finished');
    setTimeout(() => console.log('Timer 2 finished'), 0);
    setTimeout(() => console.log('Timer 3 finished'), 3000);

    // When there's no I/O callbacks in the queue
    // Event Loop checks if there's any setImmediate()
    // And execute setImmediate() right away after I/O finished
    // Even before expired timers
    setImmediate(() => console.log('Immediate 2 finished'));
```






```
    // process.nextTick() can run after each phase
    // In this case, process.nextTick() is in I/O polling and
    callbacks
    // Thus, process.nextTick() is run right after I/O finishes
    &&
    // before setImmediate callbacks

    // process.nextTick() is actuall process.nextPhase() that
    // happens before the next Event Loop phase, instead of
    entire Tick
    process.nextTick(() => console.log('Process.nextTick()'));
})
console.log('Hello from top-level code');
```

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
```

Program kept running until Timer3 finished

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
```

By the way, these are really advanced use-cases

----Thread Pool => Complex operations to be off-loaded to Thread Pool =>
Time how long it takes to run &
how to change Thread Pool size =>
Crypto to encrypt a password

```
// import file system module
const fs = require('fs');
const crypto = require('crypto');
const start = Date.now(); // Current Date in ms
// 0. Start app
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop
setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));
fs.readFile('./test-file.txt', () => {
    console.log('I/O finished');
    setTimeout(() => console.log('Timer 2 finished'), 0);
    setTimeout(() => console.log('Timer 3 finished'), 3000);
    // When there's no I/O callbacks in the queue
    // Event Loop checks if there's any setImmediate()
    // And execute setImmediate() right away after I/O finished
    // Even before expired timers
    setImmediate(() => console.log('Immediate 2 finished'));
    process.nextTick(() => console.log('Process.nextTick()'));
    // To encrypt a password
    // crypto.pbkdf2('password', 'salt', key length, algorithm,
callback)
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
})
console.log('Hello from top-level code');
```

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
```

```
Process.nextTick()
Immediate 2 finished
Timer 2 finished
```

// program kept running

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
```

// program kept running

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3266 Password encrypted
```

// almost 3.2 secs to encrypt this password

// Making 4 instances of Password Encryptions

```
// import file system module
const fs = require('fs');
const crypto = require('crypto');
const start = Date.now(); // Current Date in ms
// 0. Start app
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop
setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));
fs.readFile('./test-file.txt', () => {
    console.log('I/O finished');
```

```
    console.log('I/O finished');
    setTimeout(() => console.log('Timer 2 finished'), 0);
    setTimeout(() => console.log('Timer 3 finished'), 3000);
    // When there's no I/O callbacks in the queue
    // Event Loop checks if there's any setImmediate()
    // And execute setImmediate() right away after I/O finished
    // Even before expired timers
    setImmediate(() => console.log('Immediate 2 finished'));
    process.nextTick(() => console.log('Process.nextTick()'));
    // Making 4 instances of password encryption
    // To encrypt a password
    // crypto.pbkdf2('password', 'salt', key length, algorithm,
callback)
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
})
console.log('Hello from top-level code');
```

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
```

```
Immediate 2 finished
Timer 2 finished
```

// Program kept running

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
```

// Program kept running

```
$ node event-loop.js
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
4666 Password encrypted
4689 Password encrypted
4779 Password encrypted
4904 Password encrypted
```

// All 4 instances of Password Encryptions almost completed at same time
// By default, Thread Pool has 4 threads
// There're 4 full threads working at the same time
// Thus, these 4 instances of Password Encryptions almost completed
// at the same time

// We can change Thread Pool size
// In Windows, we'll have to create a package.json to specify Thread Pool

npm init -y

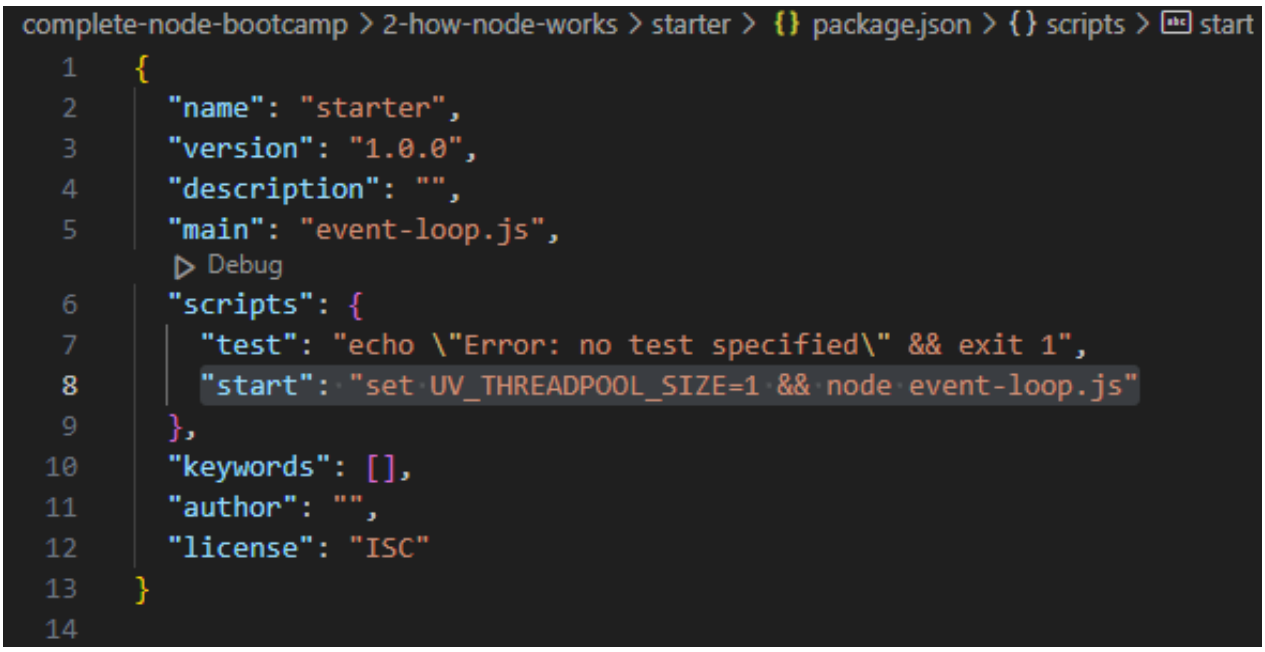
npm init -y

package.json:

"script": {

"start": "set UV_THREADPOOL_SIZE=1 && node event-loop.js"

}



event-loop.js:

```
// import file system module
const fs = require('fs');
const crypto = require('crypto');
const start = Date.now(); // Current Date in ms

// environmental variable
// UV => libuv
// We will only have 1 thread in Thread Pool
process.env.UV_THREADPOOL_SIZE = 1;

// 0. Start app
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {
    console.log('I/O finished');
```

```
    setTimeout(() => console.log('Timer 2 finished'), 0);
    setTimeout(() => console.log('Timer 3 finished'), 3000);

    // When there's no I/O callbacks in the queue
    // Event Loop checks if there's any setImmediate()
    // And execute setImmediate() right away after I/O finished
    // Even before expired timers
    setImmediate(() => console.log('Immediate 2 finished'));
    process.nextTick(() => console.log('Process.nextTick()'));

    // Making 4 instances of password encryption
    // To encrypt a password
    // crypto.pbkdf2('password', 'salt', key length, algorithm,
callback)
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', ()
=> {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
})
console.log('Hello from top-level code');
```

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
```

```
Immediate 2 finished
```

// program kept running

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
```

// program kept running

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3141 Password encrypted
```

// program kept running

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3141 Password encrypted
6333 Password encrypted
```

// program kept running

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3141 Password encrypted
6333 Password encrypted
```

```
9371 Password encrypted
```

// program kept running

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3141 Password encrypted
6333 Password encrypted
9371 Password encrypted
12414 Password encrypted
```

// Each instance of Password Encryption completes 1 after another

---

// If we set UV_THREADPOOL_SIZE=2

complete-node-bootcamp > 2-how-node-works > starter > {} package.json > {} scripts > start

```
{
  "name": "starter",
  "version": "1.0.0",
  "description": "",
  "main": "event-loop.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "set UV_THREADPOOL_SIZE=2 & node event-loop.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
```

```
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3880 Password encrypted
3916 Password encrypted
```

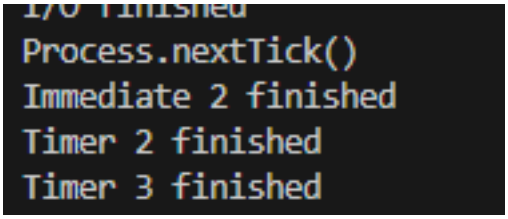
```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
Process.nextTick()
Immediate 2 finished
Timer 2 finished
Timer 3 finished
3880 Password encrypted
3916 Password encrypted
7597 Password encrypted
7628 Password encrypted
```

// Let's test set UV_THREADPOOL_SIZE=4
// Execute 4 instances of Synchronous Password Encryptions in a callback

```
// import file system module
const fs = require('fs');
const crypto = require('crypto');
const start = Date.now(); // Current Date in ms

// environmental variable
// UV => libuv
// We will only have 1 thread in Thread Pool
process.env.UV_THREADPOOL_SIZE = 1;

// 0. Start app
// 1. Execute 'top-level' code
// 2. Require module
// 3. Register Event Callbacks
// 4. Start Event Loop
setTimeout(() => console.log('Timer 1 finished'), 0);
setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {
```

```
    console.log('I/O finished');
    setTimeout(() => console.log('Timer 2 finished'), 0);
    setTimeout(() => console.log('Timer 3 finished'), 3000);
    // When there's no I/O callbacks in the queue
    // Event Loop checks if there's any setImmediate()
    // And execute setImmediate() right away after I/O finished
    // Even before expired timers
    setImmediate(() => console.log('Immediate 2 finished'));

    process.nextTick(() => console.log('Process.nextTick()'));

    // Making 4 instances of SYNC password encryption
    // To encrypt a password
    // crypto.pbkdf2('password', 'salt', key length, algorithm,
callback)
    crypto.pbkdf2Sync('password', 'salt', 100000, 1024,
'sha512', () => {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2Sync('password', 'salt', 100000, 1024,
'sha512', () => {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2Sync('password', 'salt', 100000, 1024,
'sha512', () => {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
    crypto.pbkdf2Sync('password', 'salt', 100000, 1024,
'sha512', () => {
        // To time how long it takes to encrypt a password
        // Date.now() - start
        console.log(Date.now() - start, 'Password encrypted');
    });
})
console.log('Hello from top-level code');
```

```
Hello from top-level code
Timer 1 finished
Immediate 1 finished
I/O finished
```



// Entire Event Loop execution was BLOCKED!

====**Events & Event-driven Architecture**

Event emitter =emits events> Event Listener
=>calls> Attached callback func
i.e.
-requests hitting server
-timer expiring
-file finishing to read

```
const server = http.createServer();
server.on('request', (req, res) => {
        console.log('Request received');
        res.end('Request received');
});

// server.on('', () => {} ) is an Event Listener
// Server acts as an emitter =>
// auto-emit a 'request' event each time request hits the server
New Request on Server 127.0.0.1:8000
```

Behind the scenes,
Server = Instance of EventEmitter class

Observer pattern = Reacting, rather than calling funcs

====**Events in Practice**

We'll need EventEmitter for Events in Node.js