Node.js event-loop

Synchronous code

```
1  const filesystem=require('fs');
   // readFileSync takes as arguments the filepath and the character encoding
2  var textread= filesystem.readFileSync('./txt/filetoread.txt','utf8');
3  console.log(textread);
```

-> Synchronous code -> blocking code

- This means that each statement is processed one after the other
- So each line waits for the result of the previous one
- Thus each line blocks the execution of the rest of the code

Asynchronous code

But not everything needs to be attended to immediately.

- Ie. when we send a network request, the process executing our code shall wait for data-> so much time wasted

- Asynchronous code allow us to transfer heavy work in the background, in order for the rest of the code to continue being executed

- Asynchronous code - > non-blocking code

Asynchronous code

With asynchronous code, we can **offload long-running tasks to a background thread** to avoid blocking

When a task is complete-> aforementioned tasks' **data is put back on the main single thread**

- When a JavaScript engine executes a script -> it creates **the execution contexts**

- The execution context has two phases:
  - **creation** phase
  - **execution** phase.

Call stack

- call stack -> is a "Last in, first out" or LIFO stack
  - used by Javascript engine to manage execution contexts: *Global Execution Context* & *Function Execution Contexts*

- When we **execute** a **script**
  - JavaScript engine creates ***a Global Execution Context*** and pushes it on top of the call stack.

- When a **function** is **called**
  - JavaScript engine creates *a* ***Function Execution Context*** for the function, pushes it on top of the Call Stack, and starts executing the function.

Call stack

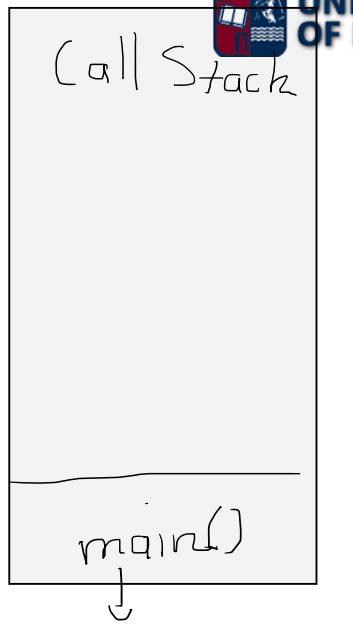
- If a function calls another function
  - JavaScript engine creates a ***new Function Execution Context*** for the function that is being called and pushes it on top of the call stack.

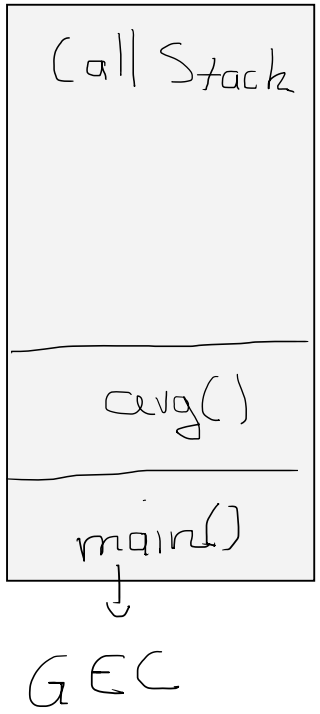
- When the current function completes
  - JavaScript engine **pops it off the call stack** and resumes the execution where it left off in the last code listing.

- When the call stack is empty
  - script stops

```
function sum(a, b) {
    return a + b;
}
function avg(a, b) {
    return sum(a, b) / 2;
}
var x = avg(5,5);
```

When script runs -> JS engine places the **global execution context** (denoted by main() or global() function) in the call stack.

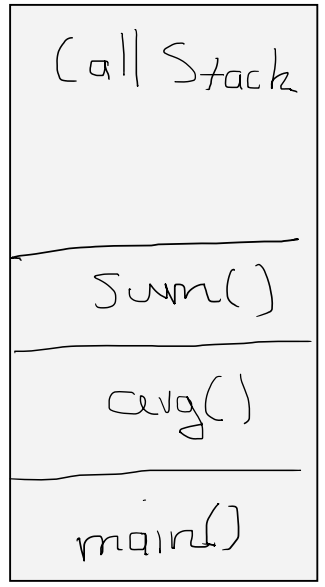




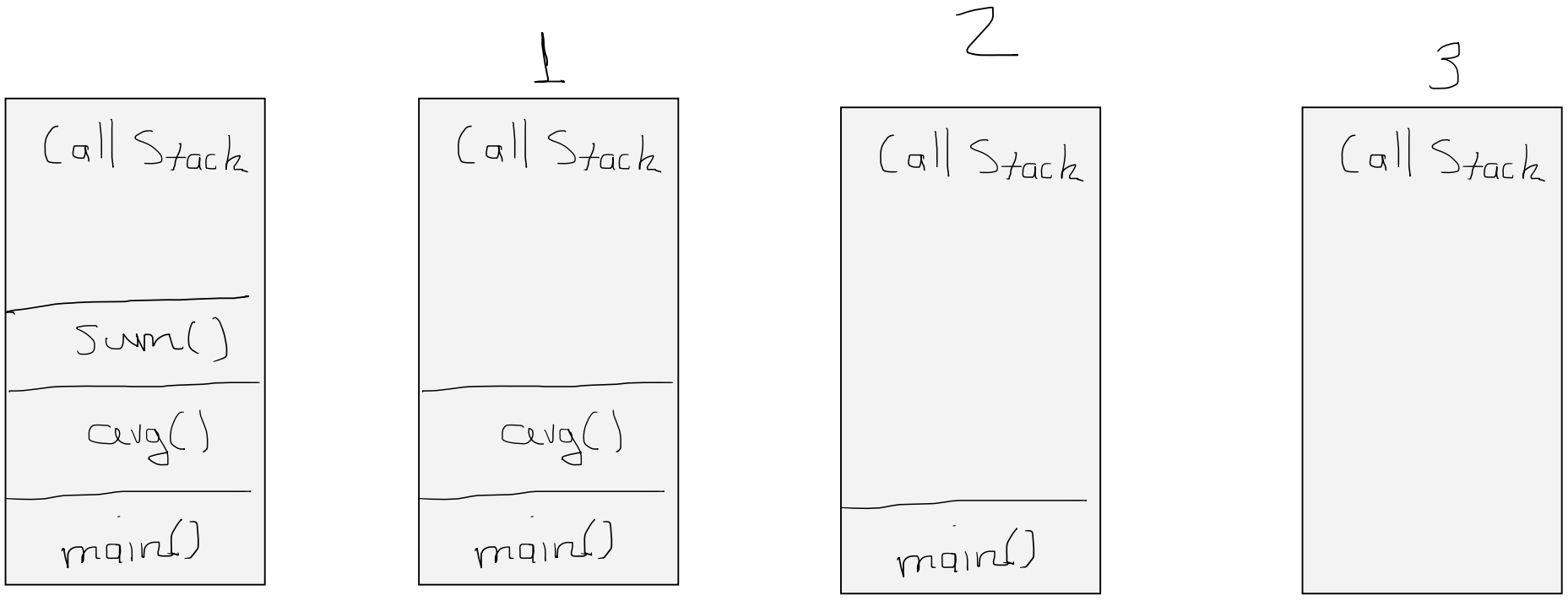
JS engine executes the call to the avg() function -> creates a **function execution context** it and pushes it on top of the call stack:

JS engine starts executing avg() since it is at the top of the call stack.

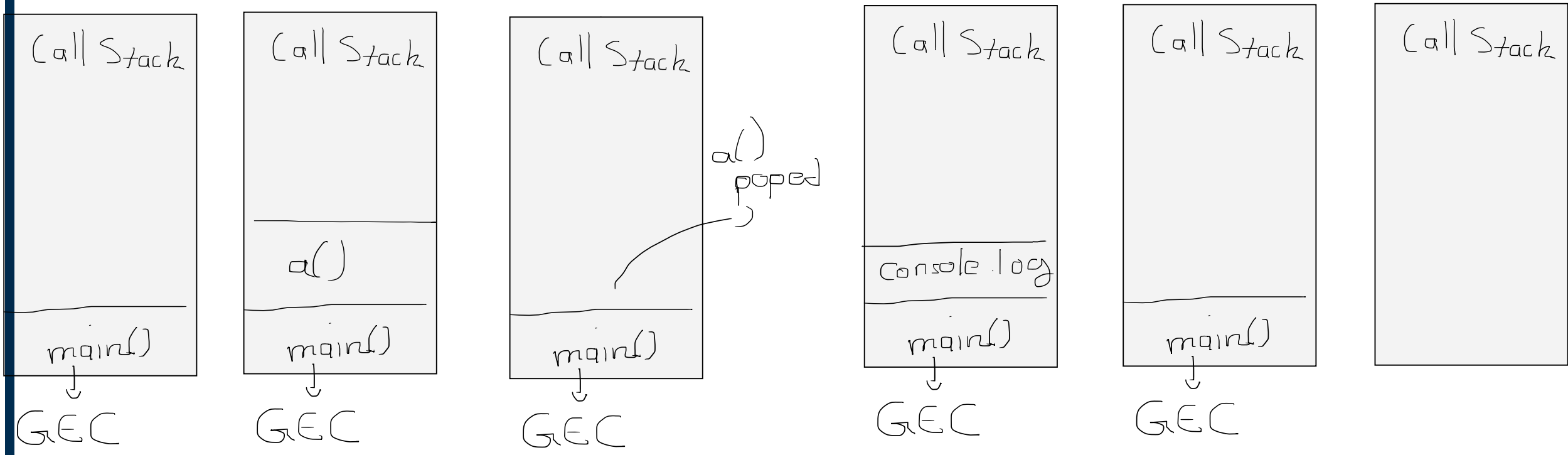
Avg() calls sum() function-> **JS engine** creates another **function execution contex**t for sum() function -> places it on the top of the call stack



1. JS engine executes sum() function and **pops** it off the call stack
2. JS engine executes avg() function and **pops** it off the call stack
3. call stack is empty so the script stops executing



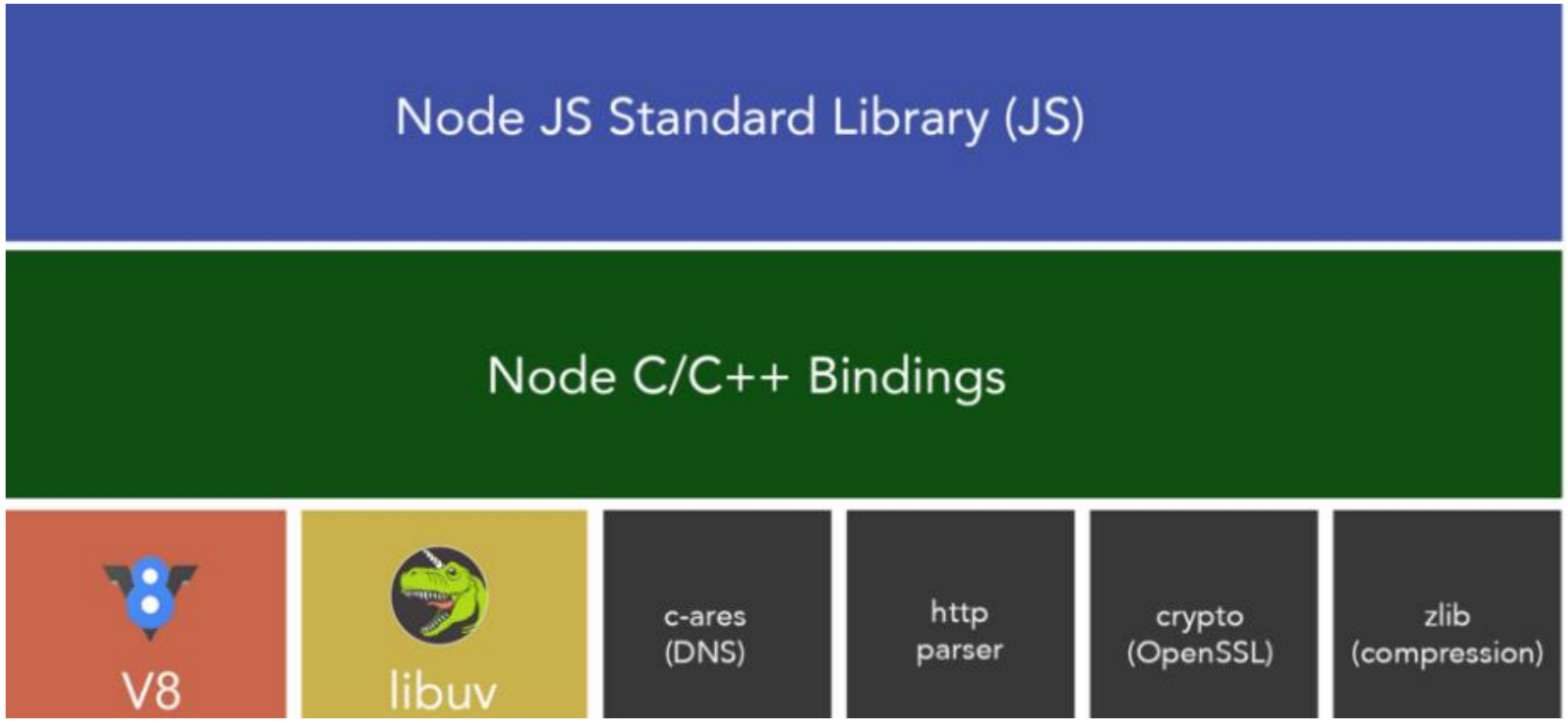
```
function a() {
    console.log("a");
}
a();
console.log('end');
```



How it works behind the scenes

- Node.js is a JavaScript runtime environment -> based on Google's V8 Engine
- Node.js allows us to run JavaScript outside of the browser

- Node.js Architecture is made of:
  - Chrome V8 engine (which is written in C++)
  - Libuv -> a multi-platform support library that focuses on asynchronous I/O based events on event loops and thread loops
  - More on libuv http://docs.libuv.org/en/v1.x/



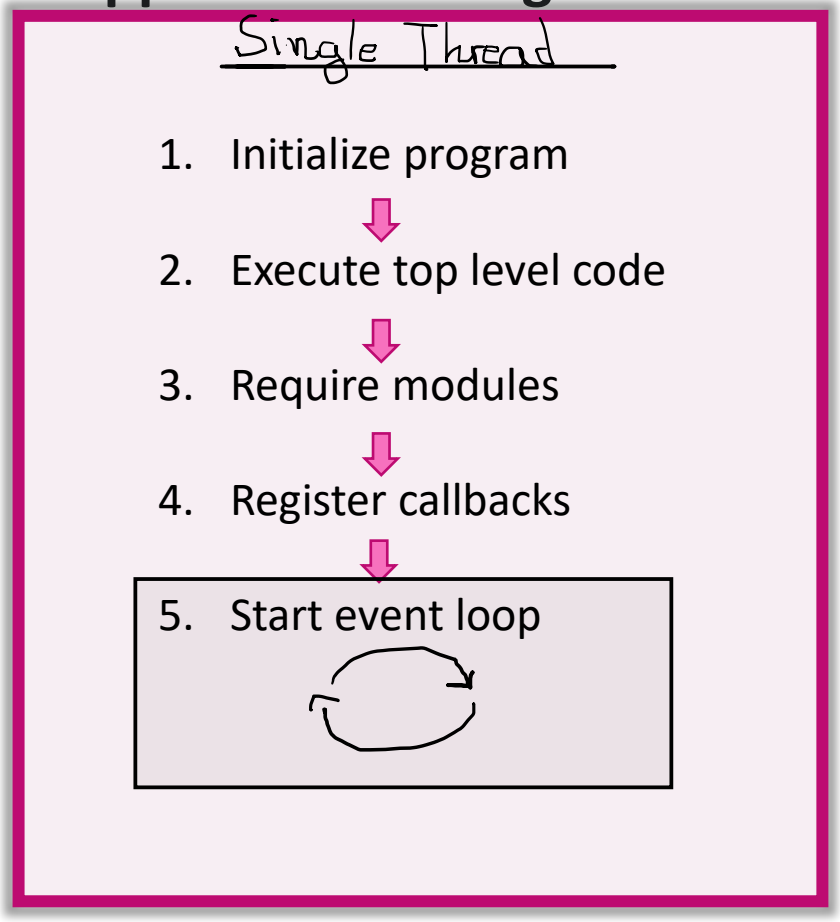
https://blog.insiderattack.net/handling-io-nodejs-event-loop-part-4-418062f917d1

Libuv

- is an open source library => focuses on asynchronous IO (gives Nodejs access to the underlying computer operating system, file system, networking, and more)
- implements event loop & thread pool.
- https://libuv.org/

Nodejs process: instance of program in execution

App runs on a single thread



Node runs in a single thread -> we must not bock this thread

In a single thread -> when we run our node app

1. program is initialized
2. top level code is executed-> code outside callbacks
3. modules are required
4. Callbacks functions are registered
5. event loop starts running

Event loop

event loop

- enables Node.js to perform non-blocking, asynchronous I/O operations
- making it one of the most important environmental features.

- Objects in Node.js can fire events, ie example receiving an HTTP request on our server or a file finishing to read will emit events & event loop **will then pick up these events** & call the **callback functions** that are **associated** with each event.

Event loop

event loop has ***multiple phases***

- ***each phase*** has a **FIFO queue of callbacks** to execute
- callbacks in each queue are processed one by one until there are no ones left in the queue
- then it moves on to next phase

- Event loop
  - is an endless loop: waits for tasks, executes them and then sleeps until it receives another one and so on
    - (ie when listening for incoming HTTP requests , we were basically running an I/O task, so the event loop keeps running & keep listening for new HTTP requests coming in instead of exiting the app)
  - executes tasks from the task queue only when the call stack is empty
  - allows us to use callbacks and promises.
  - executes the tasks starting from the oldest first

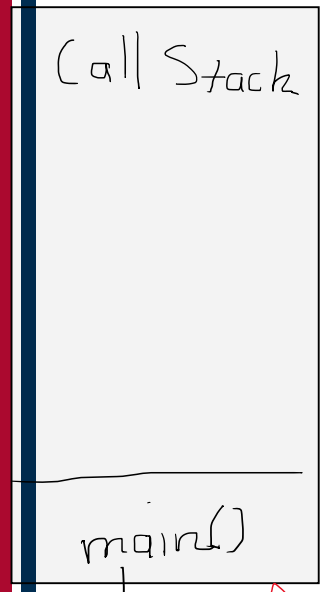
```
console.log('Hi I start');
setTimeout(function cb (){
    console.log("Callback");
},6000);

console.log('end');
```

cb is registered, code moves to next line it does not wait..

Console

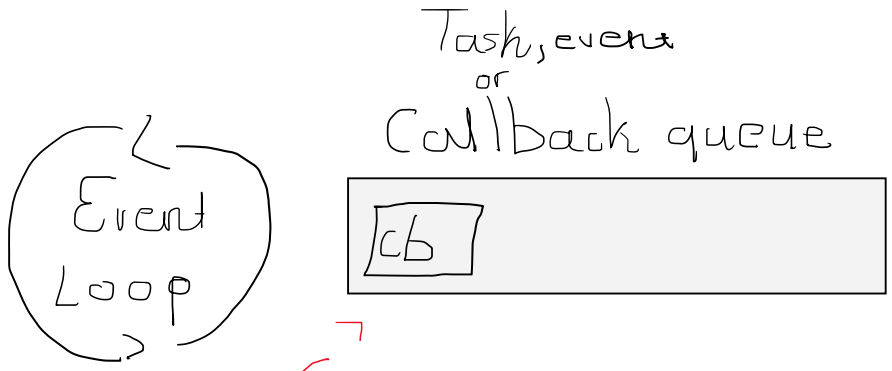
Hi I start
End
Callback



Firstly console.logs are executed

After 6 ms pass, though, our callback needs to be executed ->thus callback needs to get inside our call stack in order to execute it

When timer expires cb is put in callback queue



Check callback queue, if it finds something, it pushes it in call stack

Event loop: monitors all the time the call stack and the callback queue..

If **call stack is empty** and there is **something** in the callback queue, it puts it in the call stack in order to execute it!

We need a queue because we may have more than one tasks to be executed in the task queue

- We said that event loop has multiple phases -> each phase has a callback/task/event queue
- let's now take a look at the four most important phases

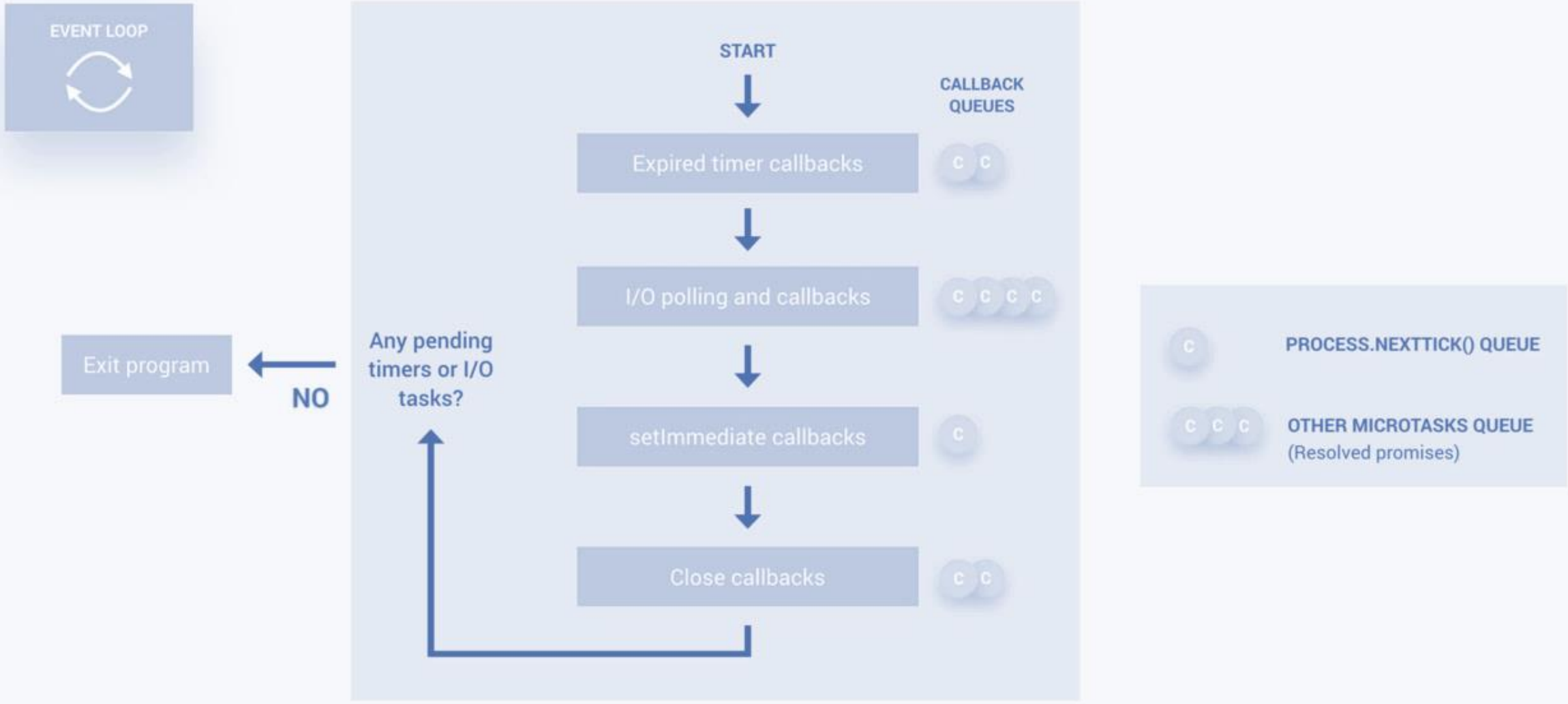
- **1st** phase takes care of callbacks of expired timers (ie setTimeout() function we saw above (If a timer expires later when another phase is being processed-> cb of that timer will be called when event loop comes back to 1rst phase)

- **2nd** phase Input /Output polling and execution of I/O callbacks

- **3rd** phase: setImmediate used in order to process callbacks immediately

- **4th** phase: close callbacks: close events are processed

THE EVENT LOOP IN DETAIL



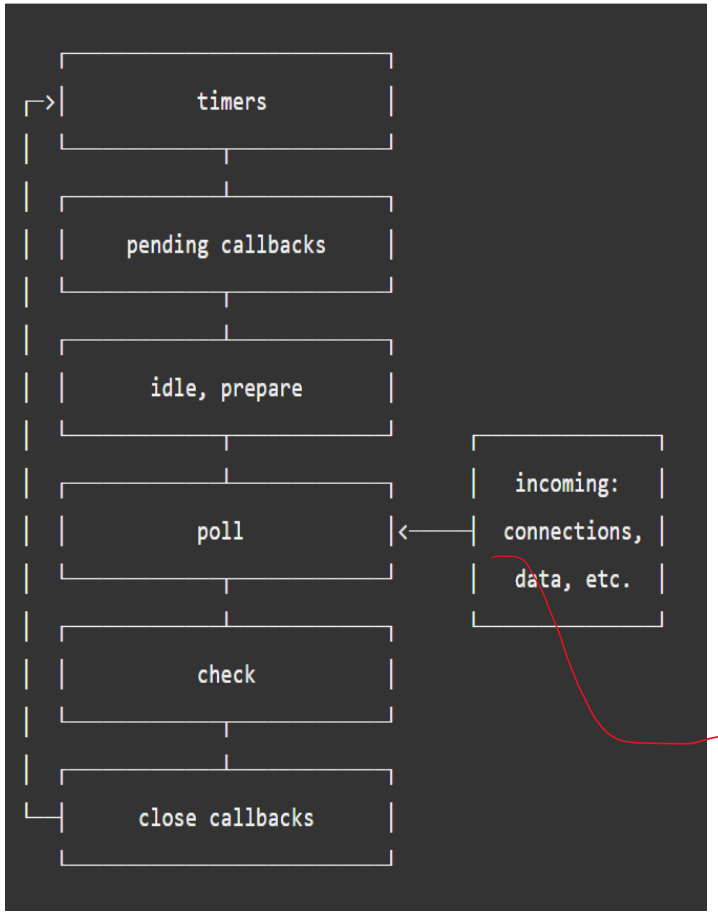
https://www.youtube.com/watch?v=6YgsqXlUoTM

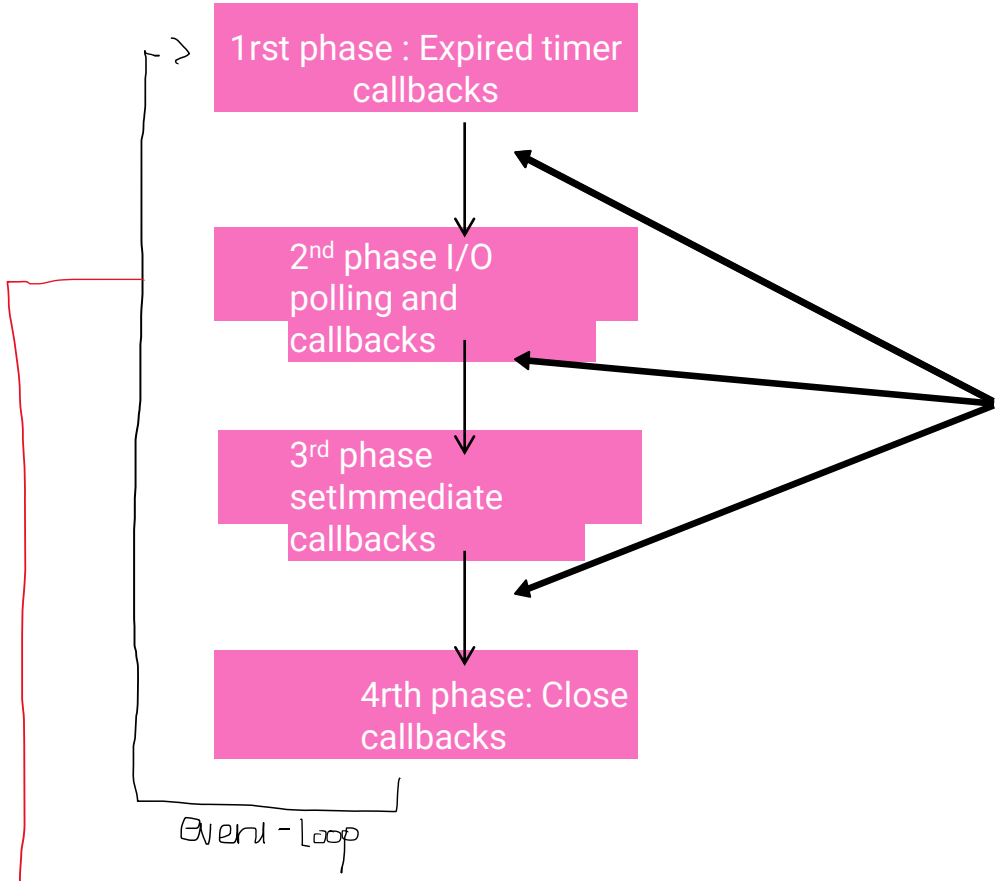
2 other queues also exist:

- nextTick() queue (nextTick() is a function we use when we wish to execute a certain callback right after the current event loop phase)
- microtasks queue (mainly for resolved promises)

- if there are any callbacks in one of these two queues to be processed: they will be executed right after current phase of the event loop

Complete diagram with overview of the event loop's order of operations in https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/





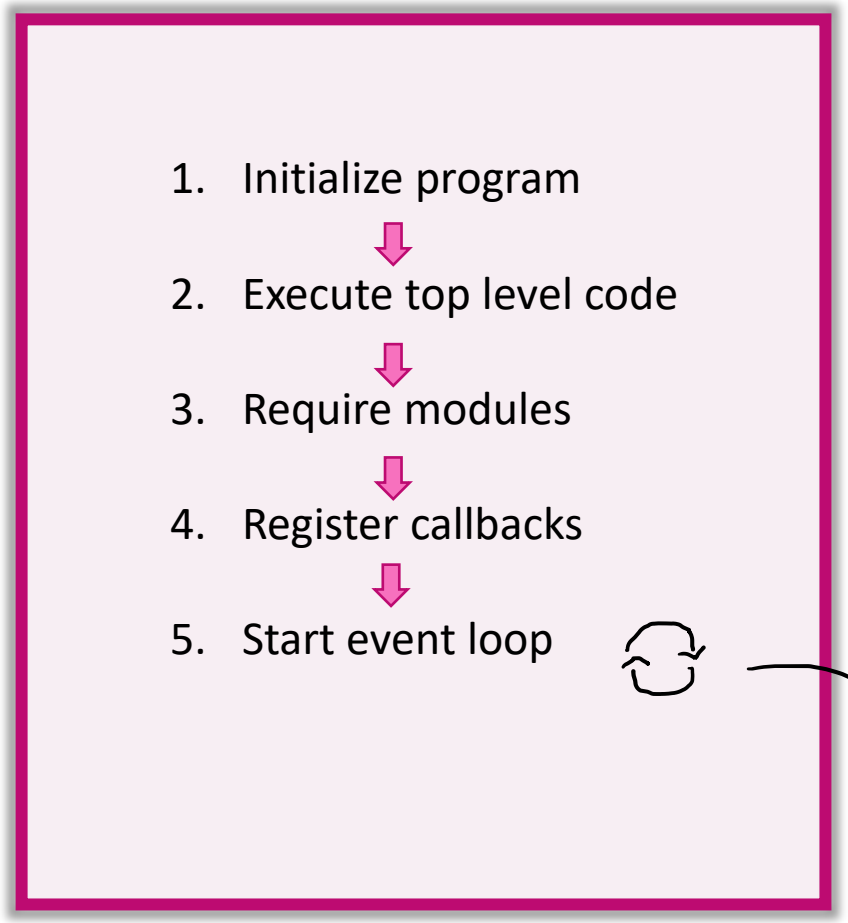
PROCESS.NEXTTICK() QUEUE

MICROTASKS QUEUE
(Resolved promises)

Most important phase to understand as this phase waits for and executes :

- asynchronous IO related callbacks. (eg. callbacks from fs.read(), fetch() etc.)
- incoming connections or requests

Nodejs program in execution
App runs on a single thread

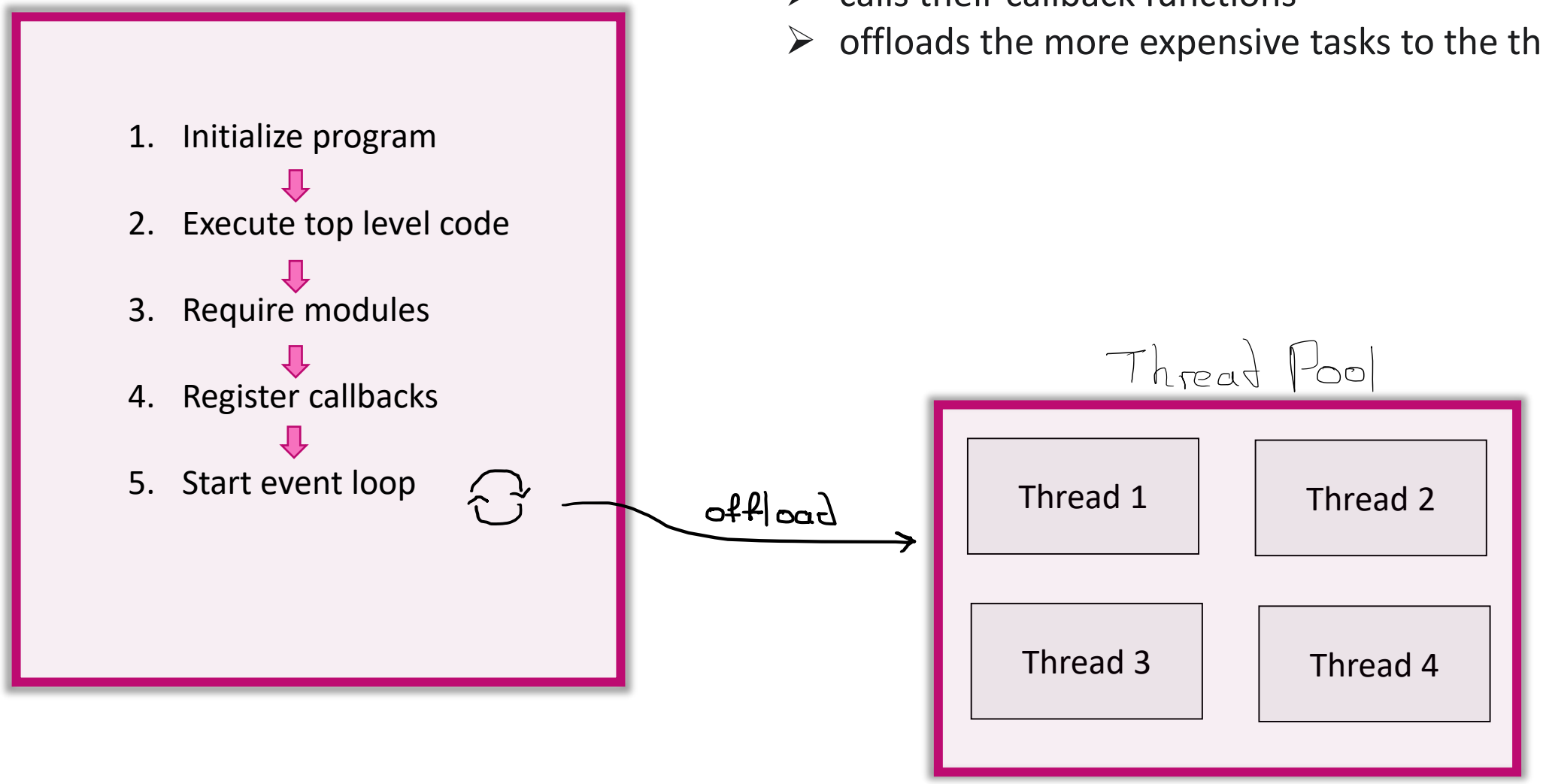


Some tasks are so heavy -> in such case the thread pool comes in,

thread pool gives additional threads (by default 4)-> completely separate from the main single thread.

event loop automatically offloads heavy tasks to the thread pool (ie expensive tasks : operations with files, cryprography related tasks)

Nodejs program in execution
App runs on a single thread



event loop does the orchestration

- it receives events,
- calls their callback functions
- offloads the more expensive tasks to the thread pool.

UNIVERSITY OF PIRAEUS

To be continued…

Sources and interesting articles:

https://blog.insiderattack.net/javascript-event-loop-vs-node-js-event-loop-aea2b1b85f5c

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

https://nodejs.org/en/about/

https://www.geeksforgeeks.org/how-node-js-works-behind-the-scene/

http://docs.libuv.org/en/v1.x/

https://dharmanikheem.medium.com/what-are-threads-in-nodejs-9ae3203f2dff

https://medium.com/payu-engineering/the-node-js-event-loop-ecde345dbc57

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