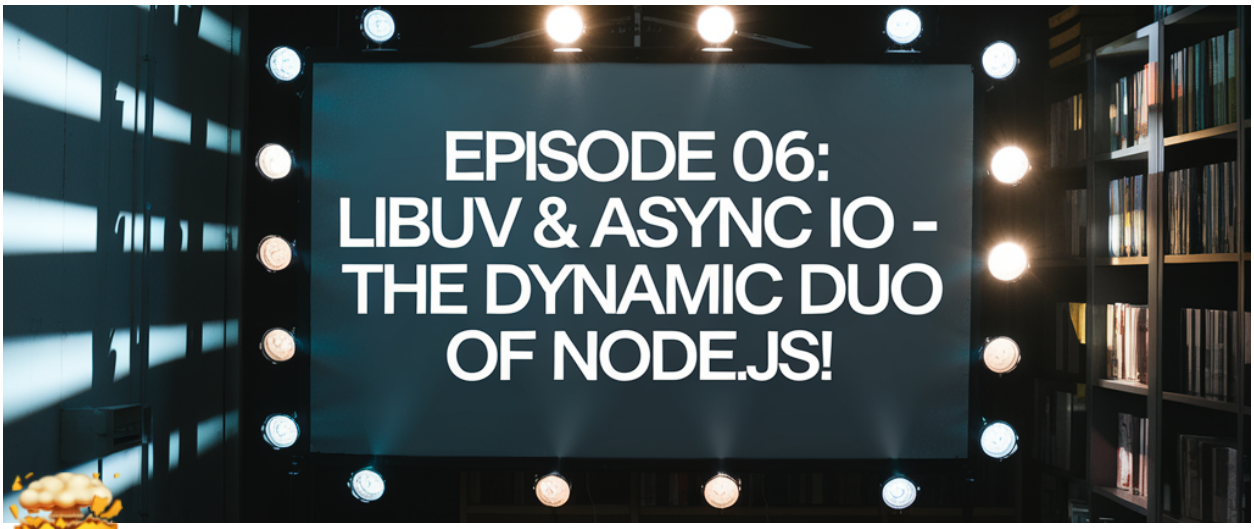


Episode-06 | libuv & async IO

I hope you remember that in the first episode, we read through the Node.js Wikipedia page. One of the key points we came across was the statement that **"Node.js has an event-driven architecture capable of asynchronous I/O."**

So, this episode is focused on explaining that concept.

WIKIPEDIA
The Free Encyclopedia

Create account Log in

Node.js

50 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

Node.js is a cross-platform, open-source JavaScript runtime environment that can run on Windows, Linux, Unix, macOS, and more. Node.js runs on the V8 JavaScript engine, and executes JavaScript code outside a web browser.

Node.js lets developers use JavaScript to write command line tools and for server-side scripting. The ability to run JavaScript code on the server is often used to generate dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm,[6] unifying web-application development around a single programming language, as opposed to using different languages for the server- versus client-side programming.

Node.js has an event-driven architecture capable of asynchronous I/O. These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for real-time Web applications (e.g., real-time communication programs and browser games).[7]

Node.js

Original author(s)	Ryan Dahl
Developer(s)	OpenJS Foundation
Initial release	May 27, 2009; 15 years ago[1]
Stable release	22.6.0[2] / August 6, 2024; 15 days ago
Repository	github.com/nodejs/node
Written in	JavaScript, C++, Python, C
Operating system	z/OS, Linux, macOS, Microsoft Windows, SmartOS, FreeBSD, OpenBSD, IBM AIX[3]
Type	Runtime environment

Lets start with the fundamentals of JavaScript first

Q: What is Thread?

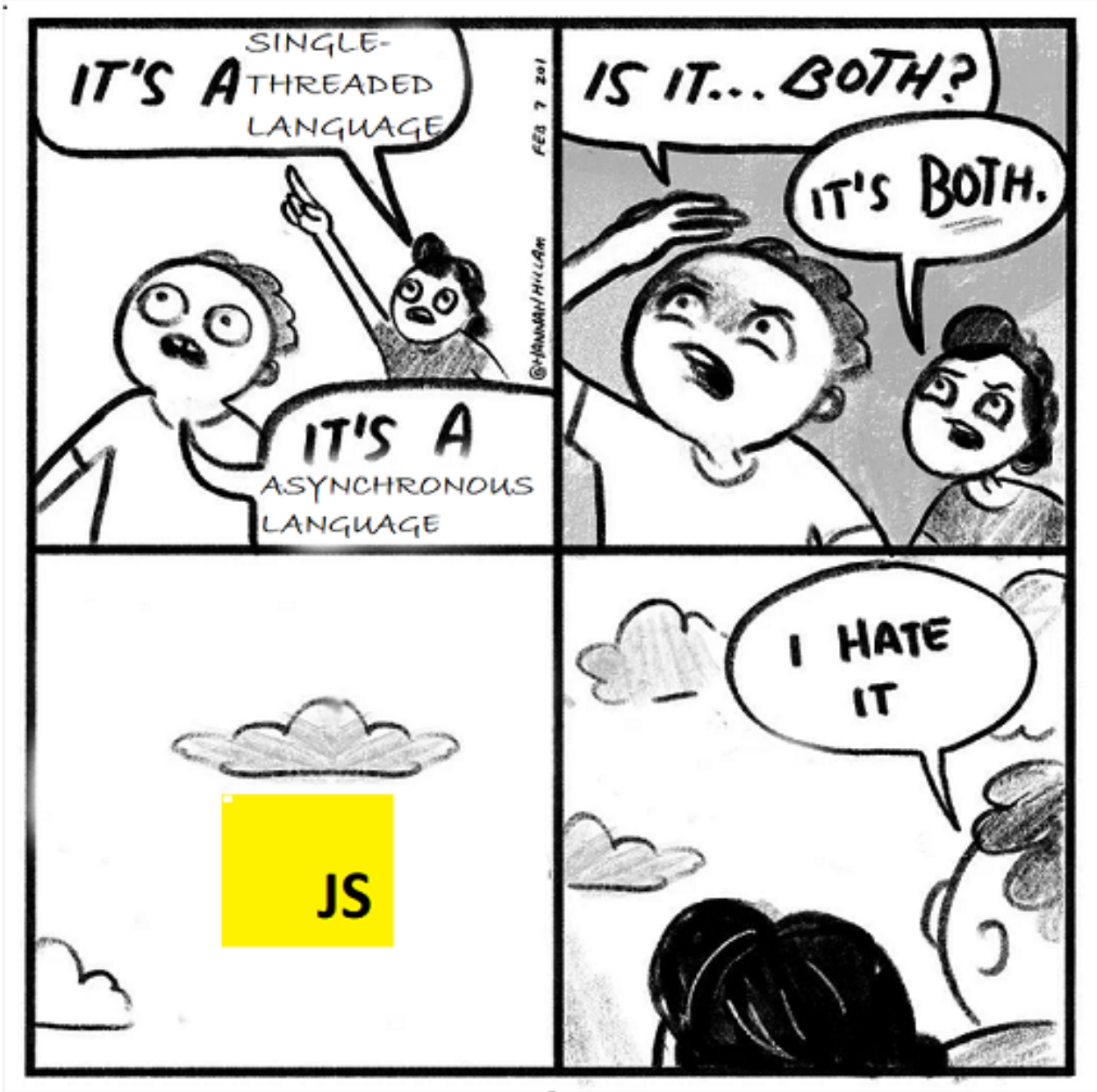
A thread is the smallest unit of execution within a process in an operating system. It represents a single sequence of instructions that can be managed independently by a scheduler. Multiple threads can exist within a single process, sharing the same memory space but executing independently. This allows for parallel execution of tasks within a program, improving efficiency and responsiveness.
Threads can be either:

1) Single-threaded

2) Multi-threaded

Q: What type of threading does JavaScript use?

- JavaScript is a synchronous, single-threaded language, meaning there is only one thread in which the JavaScript engine (such as the V8 engine) runs. In JavaScript, code is executed line by line within this single thread.
- In other languages like C++ or Java, code can be executed across multiple threads. For example, a portion of the code might be executed in one thread, while another part runs simultaneously in a different thread. However, JavaScript handles this process more straightforwardly—executing code one line after the other in sequence.
- So, if you're executing line 2 in JavaScript, it will only run after line 1 has finished executing. This is the essence of synchronous execution: each task is performed one after the other, without overlap.



Synchronous vs Asynchronous JavaScript

Q: What is a Synchronous System?

In a synchronous system, tasks are completed one after another.

Think of this as if you have just one hand to accomplish 10 tasks. So, you have to complete one task at a time.

Here, an order can only be fulfilled ,once the previous order is fulfilled.

Q: What is an Asynchronous System?

In this system, tasks are completed independently.

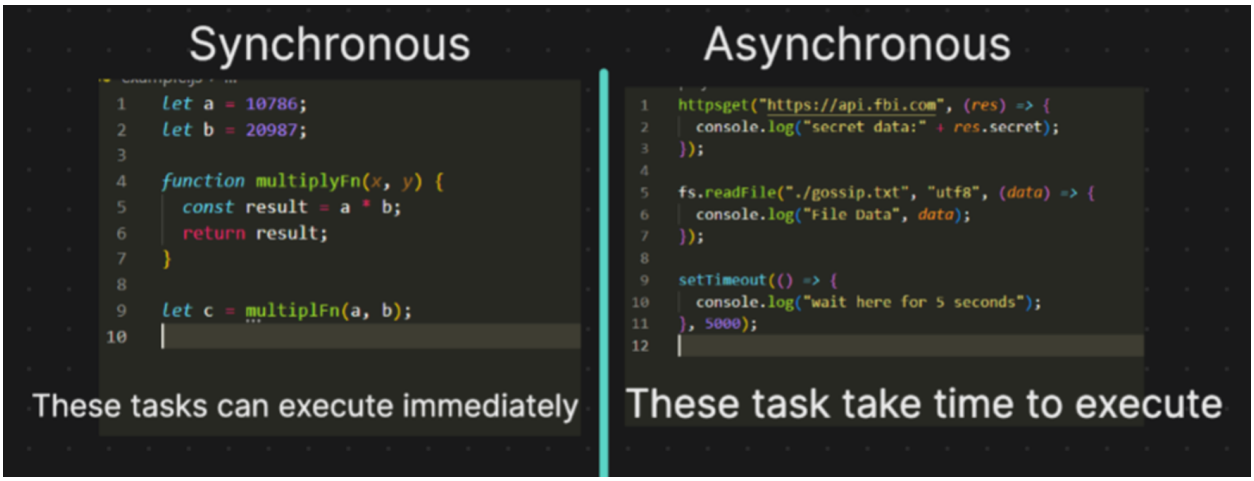
Here, imagine that for 10 tasks, you have 10 hands. So, each hand can do each task independently and at the same time.

All orders with Coke will get it at 0 min, the second will get it in 10 min, and all the ice cream orders will be fulfilled in 5 min.

No one has to wait for any other order.

So, JavaScript itself is synchronous, but with the power of Node.js, it can handle asynchronous operations, allowing JavaScript to perform multiple tasks concurrently.

Examples Incoming!



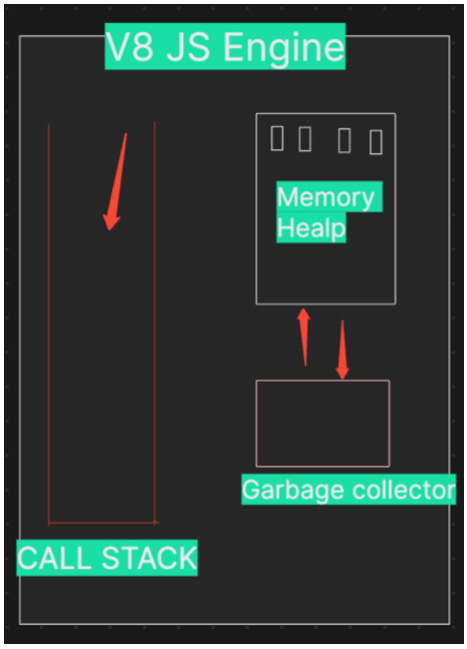
Q: what are the portions inside the JS engine and How synchronous code is executed By JS Engine ?

A:

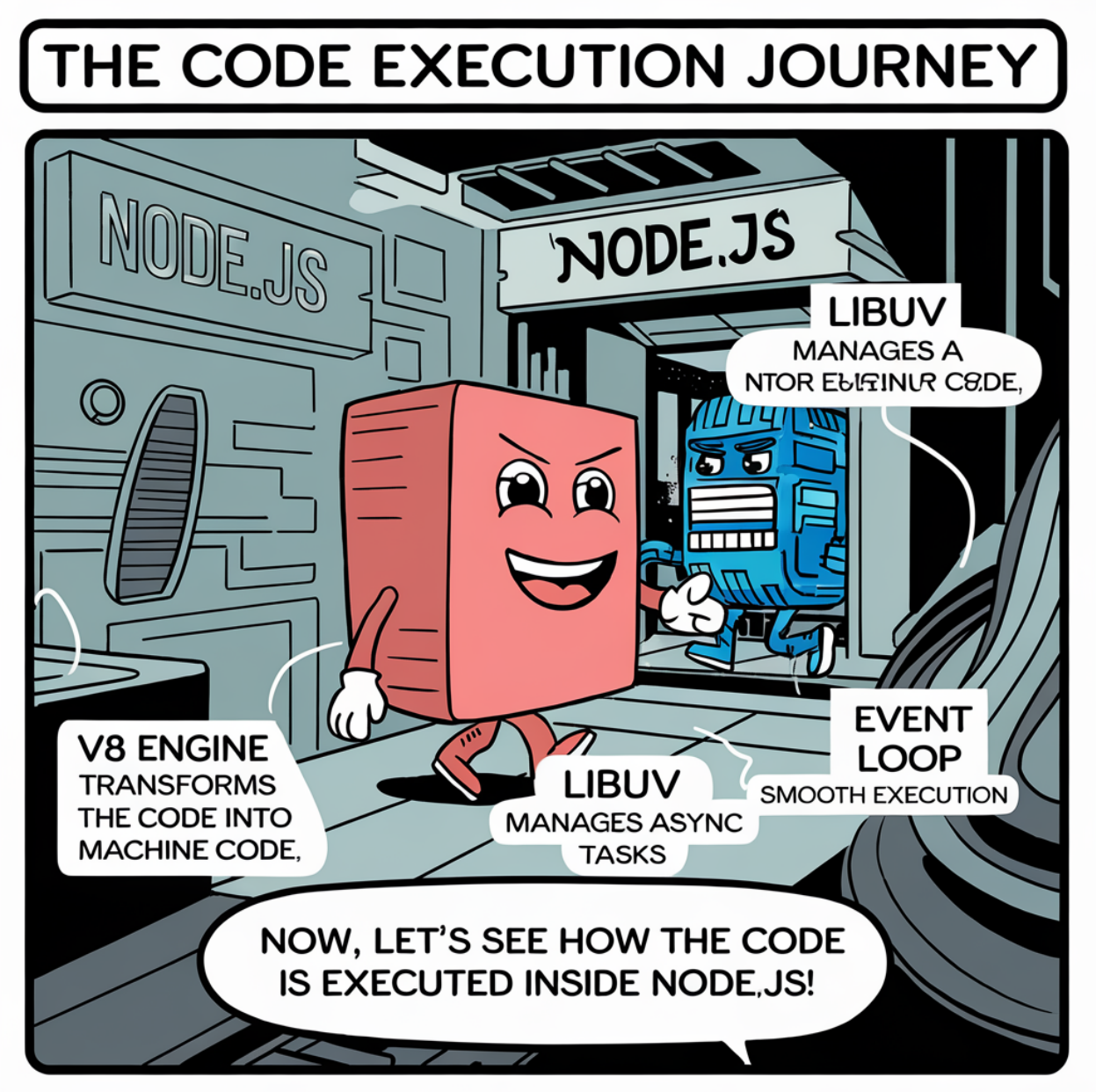
- The JavaScript engine operates with a single **call stack**, and all the code you write is executed within this **call stack**. The engine runs on a **single thread**, meaning it can only perform one operation at a time.

- In addition to the call stack, the JavaScript engine also includes a **memory heap**. This memory heap stores all the variables, numbers, and functions that your code uses.

- One key feature of the JavaScript V8 engine is its **garbage collector.** The garbage collector automatically identifies and removes variables that are no longer in use, freeing up memory. Unlike languages like C++, where developers need to manually allocate and deallocate memory, JavaScript handles this process automatically. This means you don't have to worry about memory management—it's all taken care of by the engine.

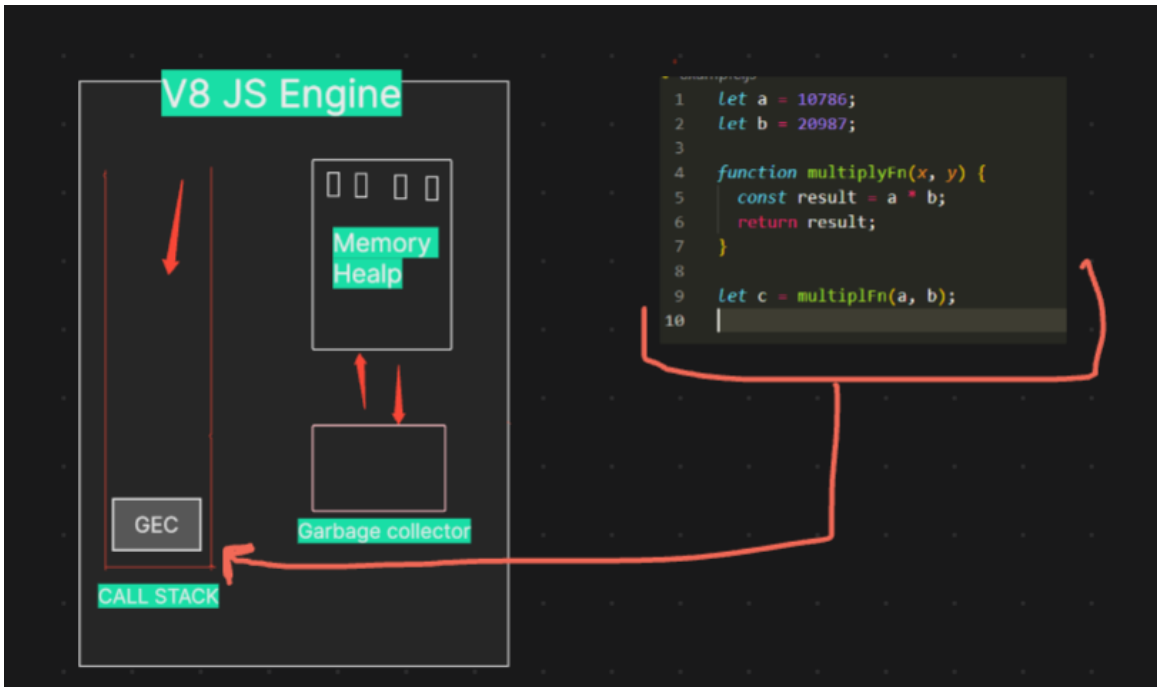


Now, lets see how code is executed inside



Step 1: Global Execution Context Creation

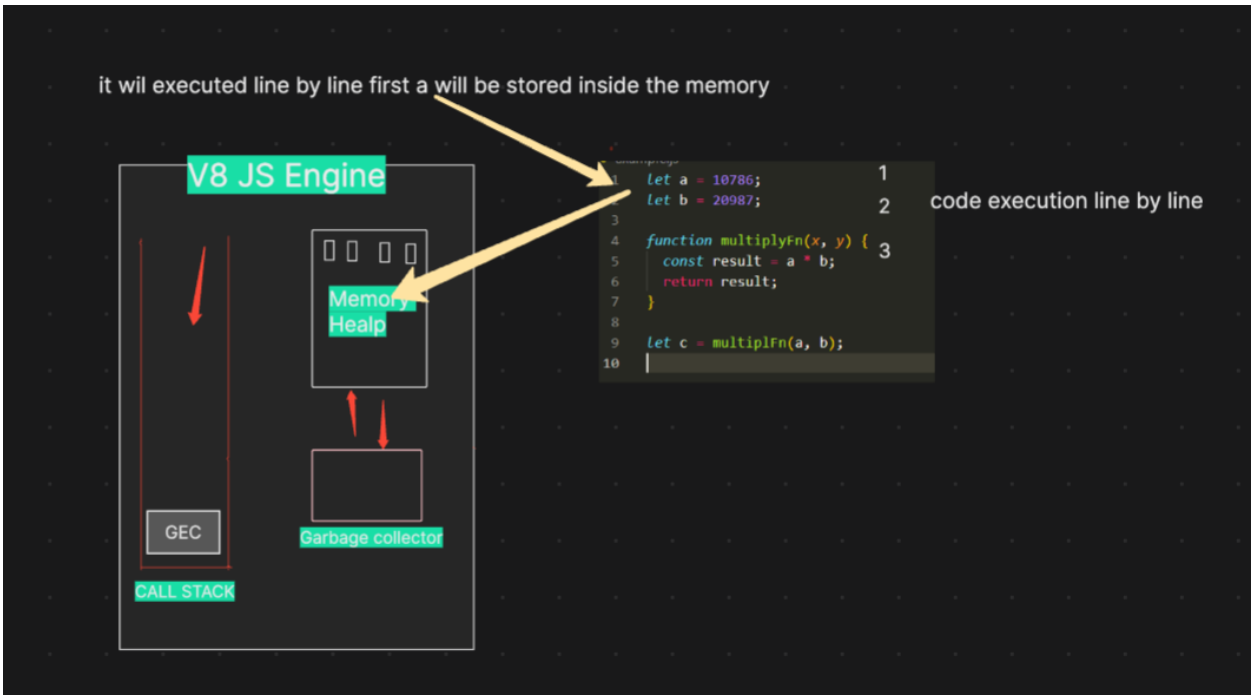
As soon as the JavaScript engine begins executing the code, it creates a **Global Execution Context**. This is the main environment where the top-level code is executed. The global execution context is unique and is always the first to be pushed onto the **call stack**.



Step 2: Memory Creation Phase

Before any code is executed, the JavaScript engine enters the **memory creation phase**. During this phase:

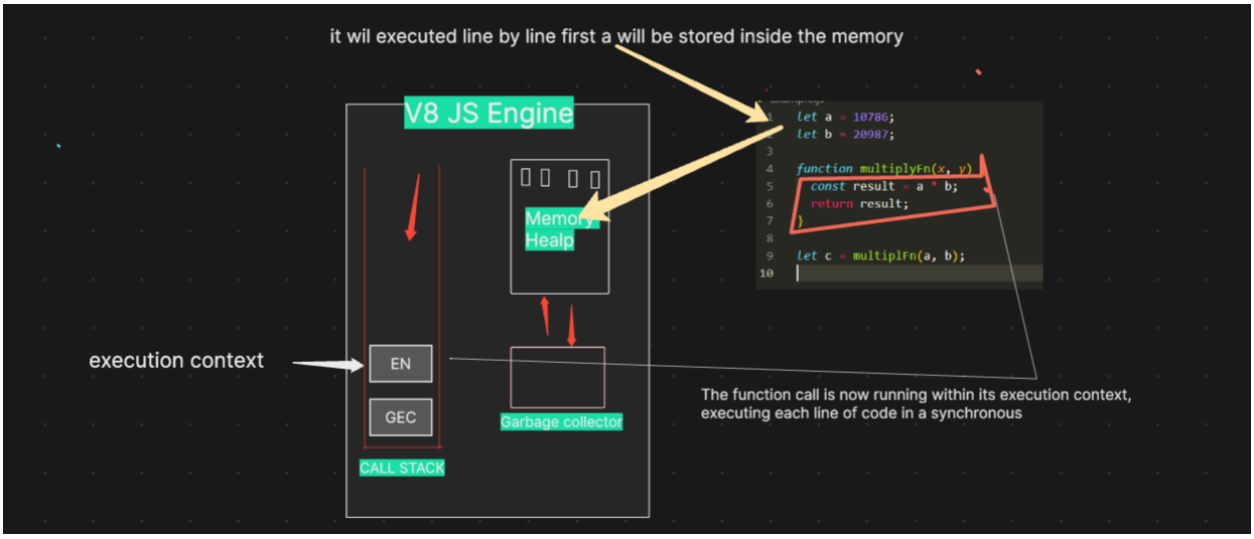
- Variables `a` and `b` are declared in memory and initialized to `undefined`.
- The function `multiplyFn` is also stored in memory, with its value set to the entire function definition.



Step 3: Code Execution Phase

Once the memory creation phase is complete, the engine moves to the **code execution phase**:

- **Execution of** `let a = 10786;` **and** `let b = 20987;`: The variables `a` and `b` are now assigned their respective values.
- **Execution of** `let c = multiplyFn(a, b);`: The function `multiplyFn` is invoked, creating a new execution context specifically for this function.



Step 4: Function Execution Context Creation

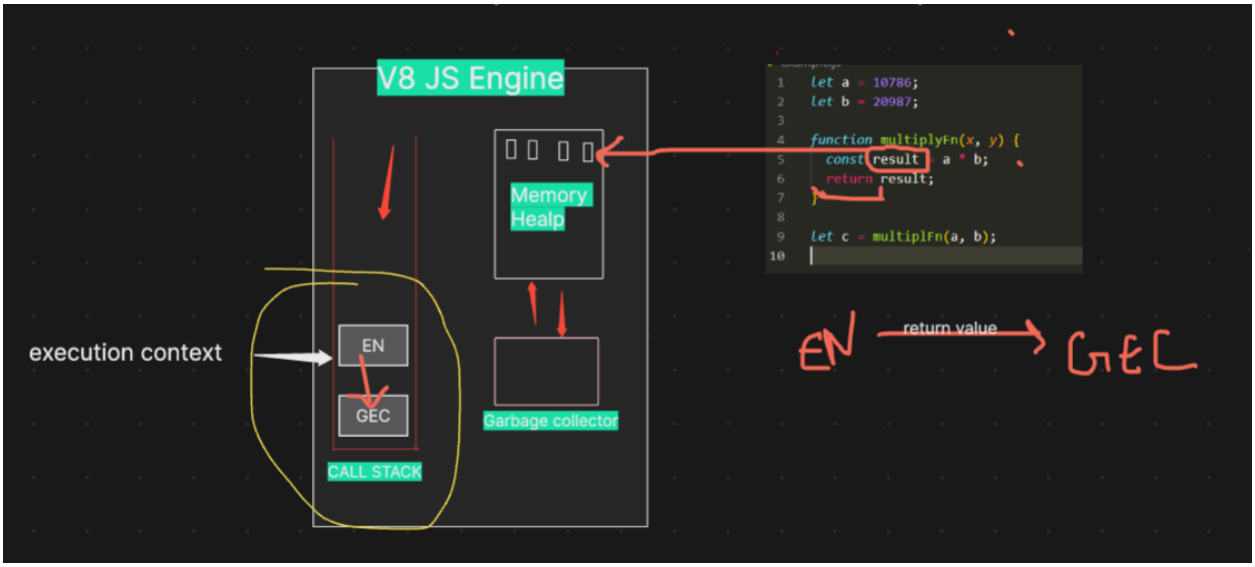
When `multiplyFn(a, b)` is called, the JavaScript engine:

- Creates a new **execution context** for `multiplyFn` and pushes it onto the top of the call stack.
- In this new context, the parameters `x` and `y` are assigned the values of `a` and `b`.

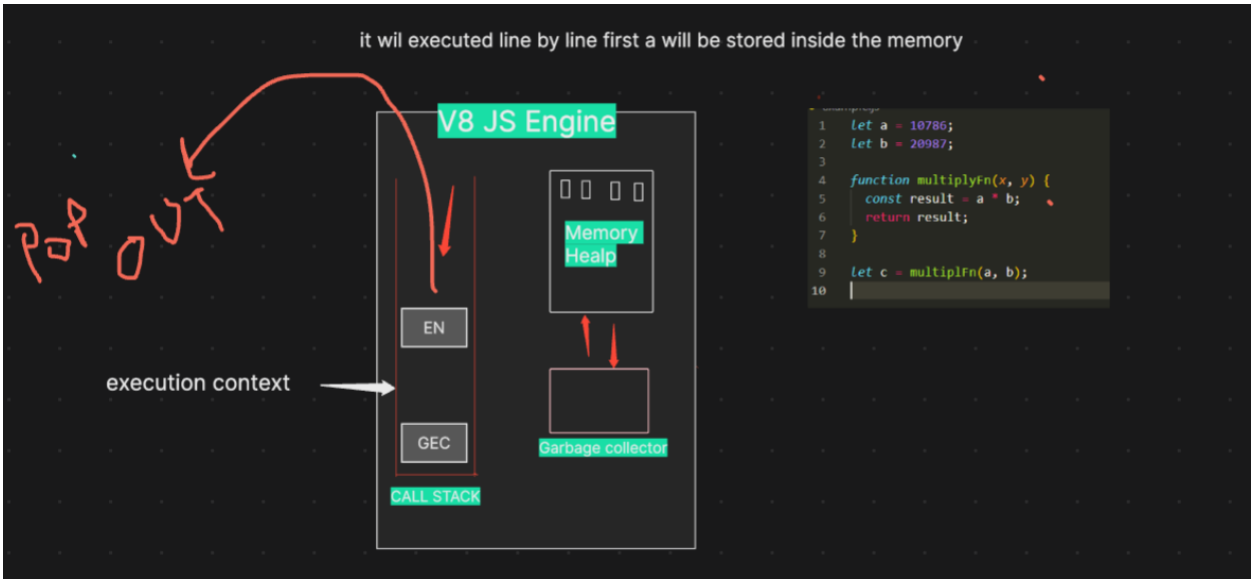
Step 5: Memory Creation and Code Execution Inside `multiplyFn`

Inside `multiplyFn`, the memory creation phase initializes `result` in memory with `undefined`.

- **Execution of** `const result = a * b;`: The multiplication is performed, and `result` is assigned the value `226215682`.

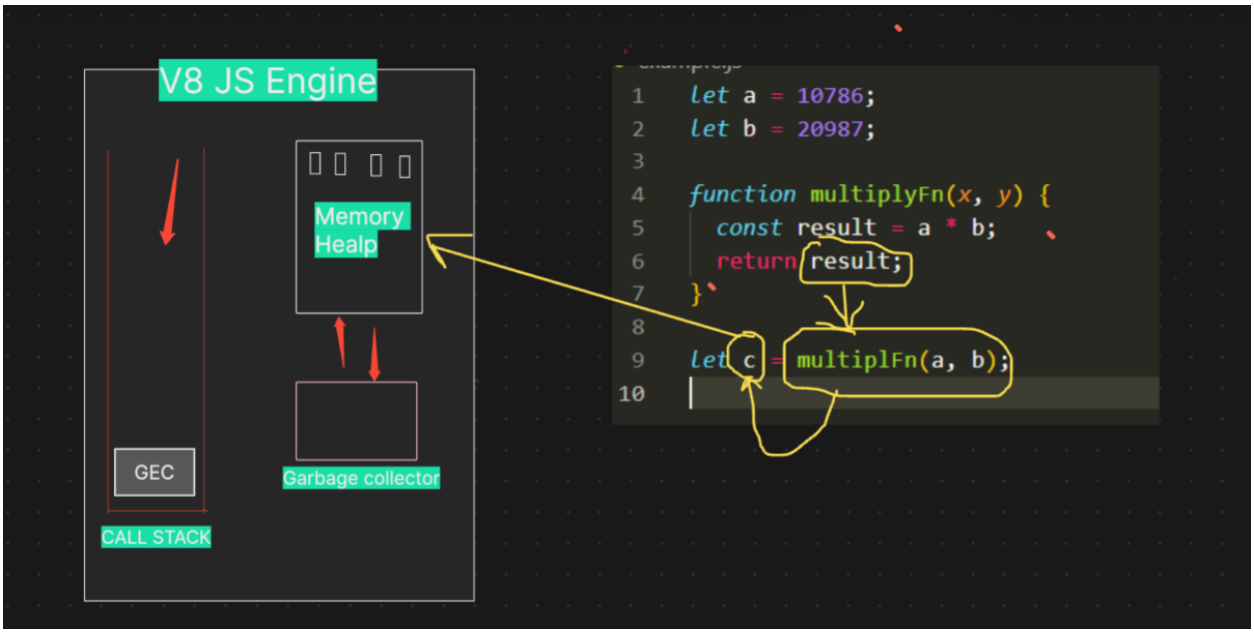


- **Execution of** `return result;`: The function returns `226215682`, and the `multiplyFn` execution context is popped off the call stack.



Step 6: Resuming Global Execution Context

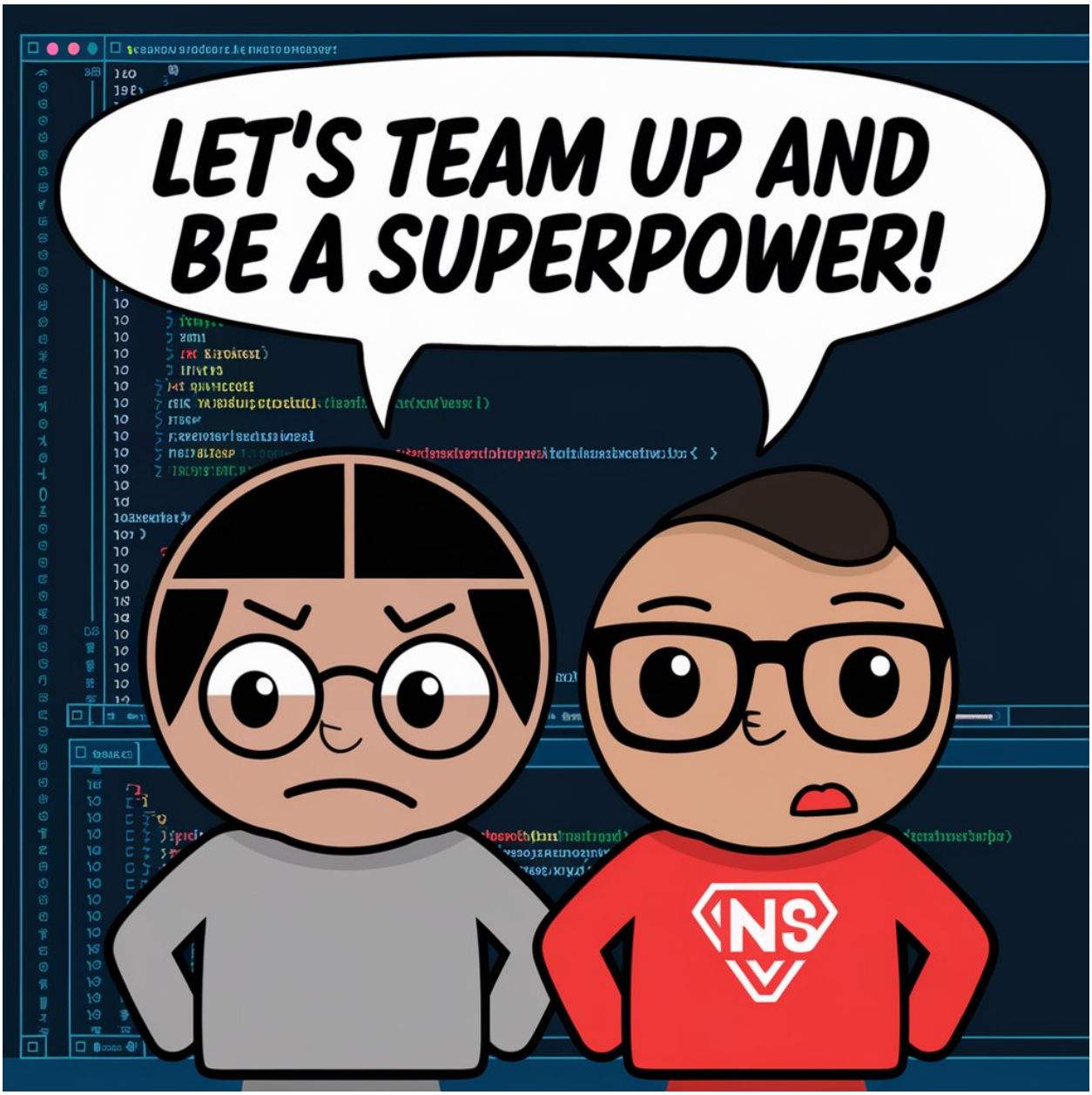
Back in the global execution context, the returned value from `multiplyFn` (`226215682`) is assigned to the variable `c`.



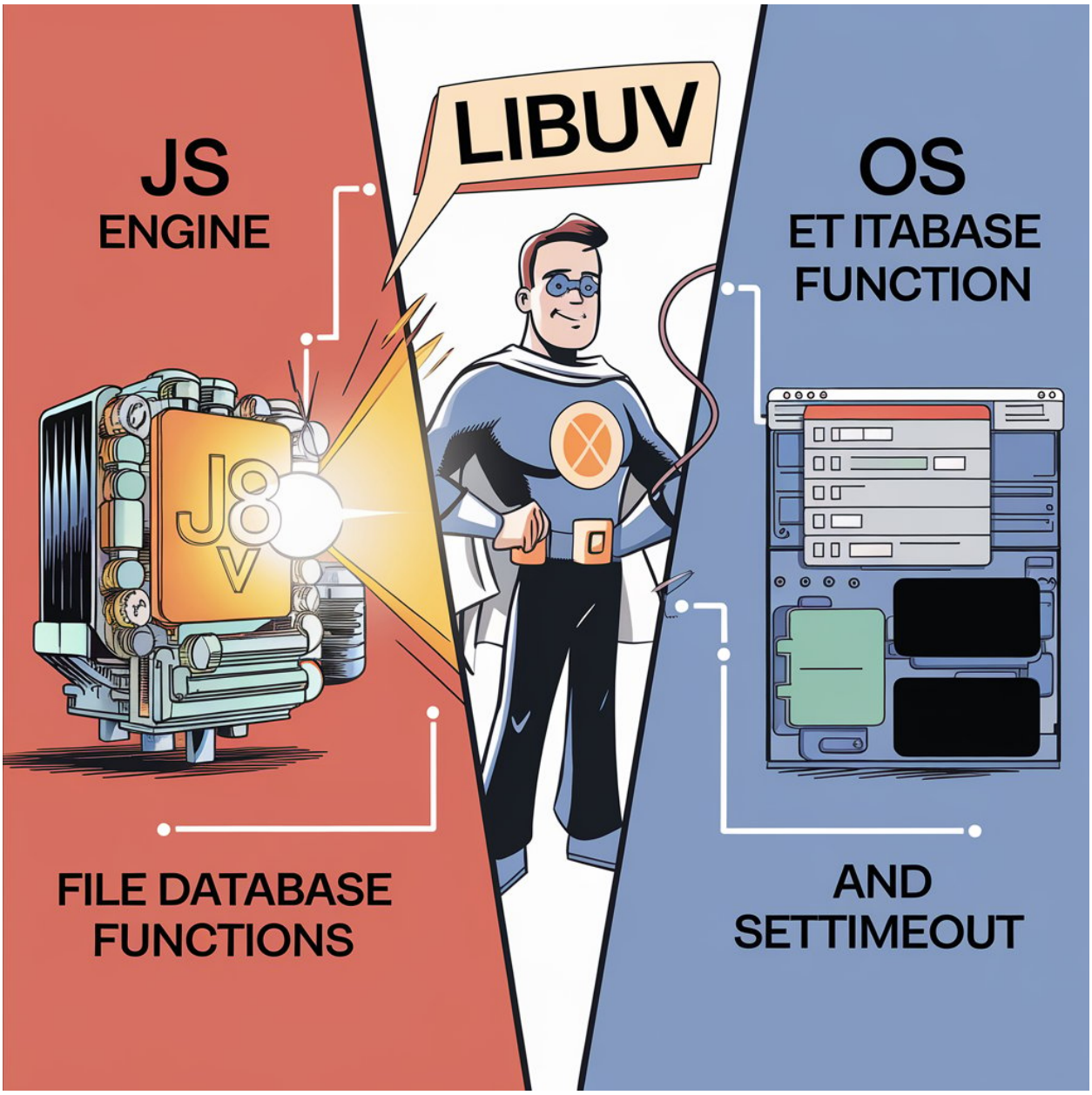
Step 7: Once the entire code is executed, the global execution context is also popped out, and the call stack becomes empty.

Q: How asynchronous code executed?

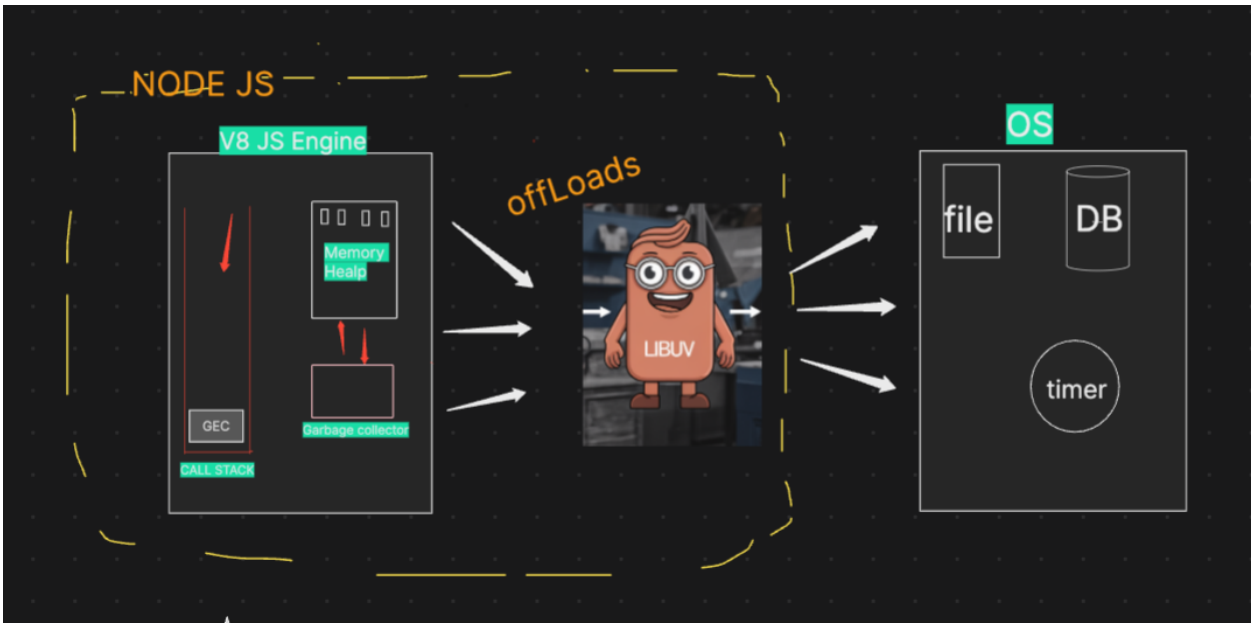
The JavaScript engine cannot do this alone; it needs superpowers. This is where Node.js comes into the picture, giving it the ability to interact with operating system functionalities.



The JS engine gains its superpowers from Node.js. Node.js grants these powers through a library named Libuv—**our superhero.**

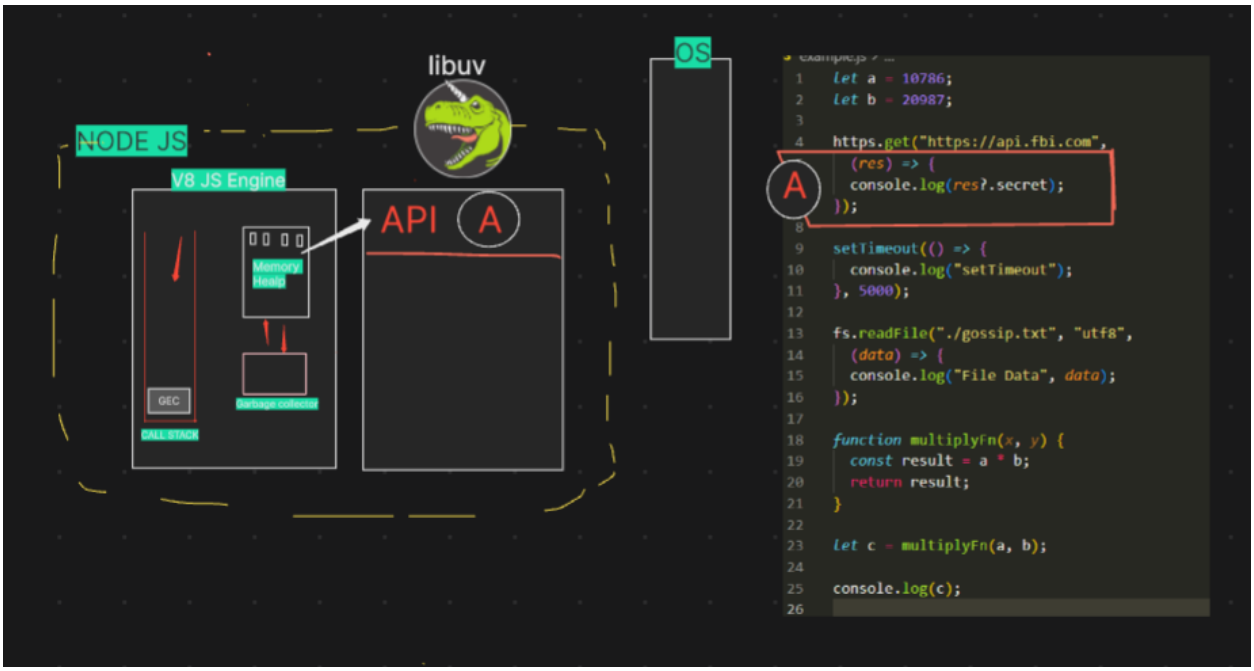


The JS engine cannot directly access OS files, so it calls on Libuv. Libuv, being very cool and full of superpowers, communicates with the OS, performs all the necessary tasks, and then returns the response to the JS engine. He offloads the work and does wonders behind the scene.

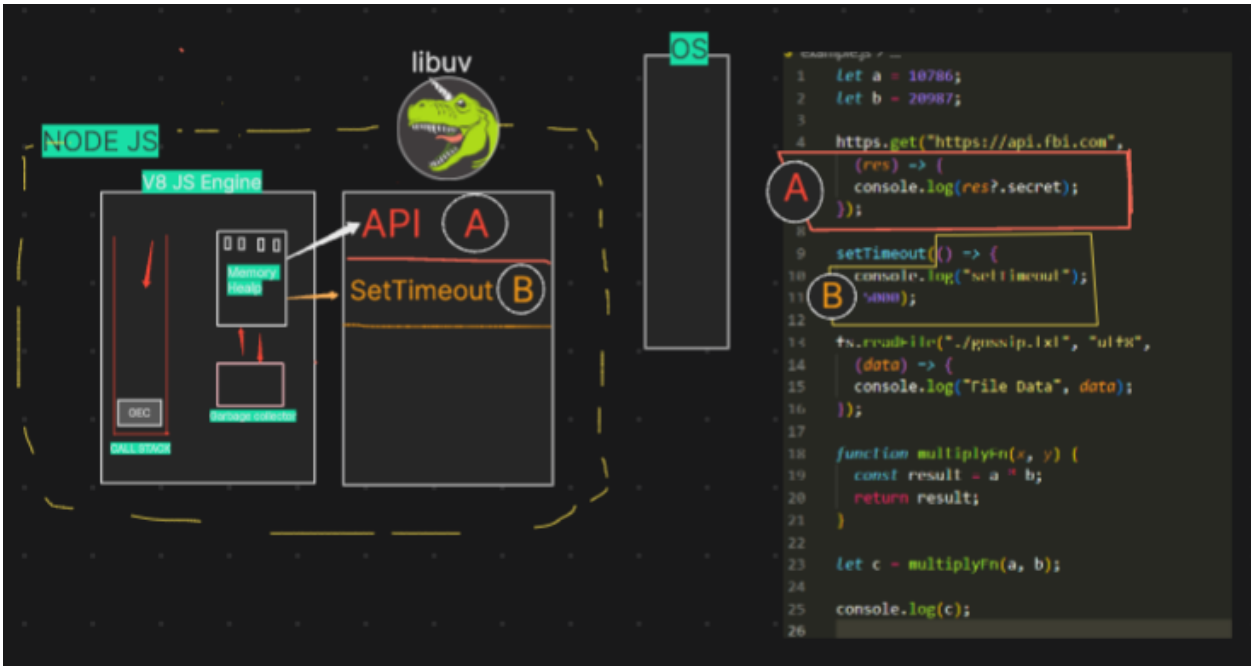

NODE JS
V8 JS Engine
Memory Heap
GEC
CALL STACK
Garbage collector
offLoads
LIBUV
OS
file
DB
timer

- The variables `let a` and `let b` are executed within the GEC (Global Execution Context) during the synchronous phase of the code execution process.
- However, when the code encounters an API call, the V8 engine, while still operating within the GEC, recognizes that it's dealing with an asynchronous operation. At this point, the V8 engine signals `libuv`—the superhero of Node.js—to handle this API call.

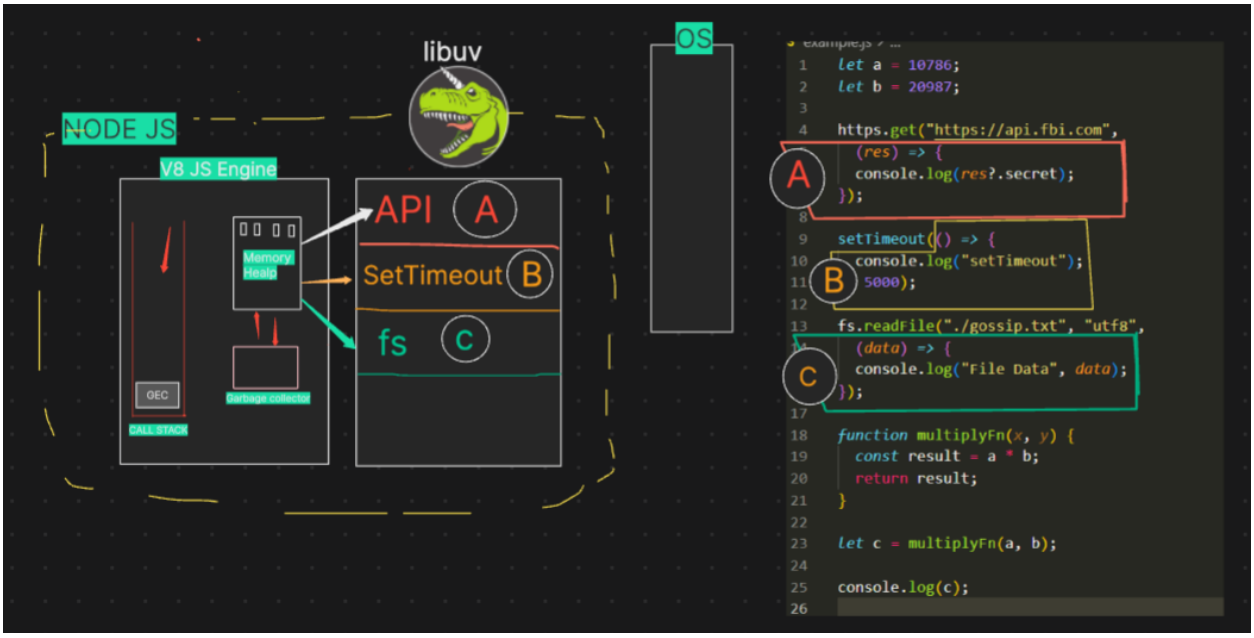
- What happens next is that `libuv` registers this API call, including its associated callback function (name - A), within its event loop, allowing the V8 engine to continue executing the rest of the code without waiting for the API call to complete.



- **Next, when the code encounters a `setTimeout` function, a similar process occurs.**
- **The V8 engine identifies this as another asynchronous operation and once again notifies `libuv`.**



- **Following this, when the code reaches a file operation (like reading or writing a file), the process is similar.**
- **The V8 engine recognizes this as another asynchronous task and alerts `libuv`.**
- **`libuv` then registers the file operation and its callback in the event loop.**



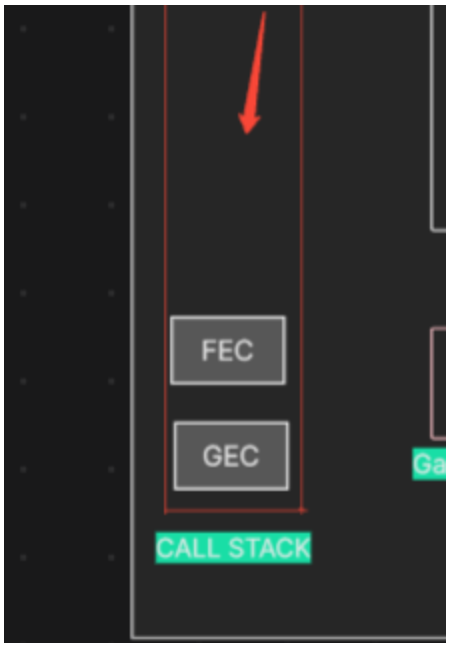
Next, when the code executes `let c = multiplyFn(a, b);`, the JavaScript engine creates a new function context for `multiplyFn` and pushes it onto the call stack.

The function takes two parameters, `x` and `y`, and within the function, the engine multiplies these values (`a * b`) and stores the result in the `result` variable.

The JavaScript engine handles this operation as part of the **synchronous code execution**

Next, when the code executes `let c = multiplyFn(a, b);`, the JavaScript engine creates a new function context for

`multiplyFn` and pushes it onto the call stack.



Once the
`multiplyFn` completes its execution and returns the result, the function context is popped off the call stack, and the result is assigned to the variable `c`

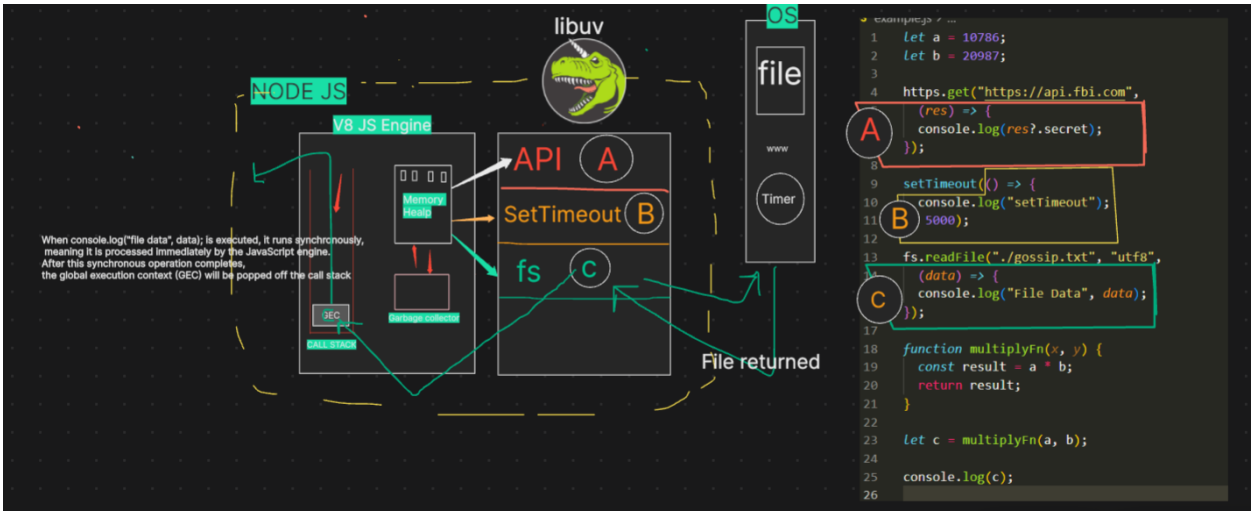
imp concept :

- **When the function execution context is popped off the call stack, the garbage collector may clear any memory allocated for that context in the memory heap, if it is no longer needed.**
- **After `console.log(c)` is executed and the value of `c` is printed to the console, the global execution context will also eventually be removed from the call stack if the code execution is complete.**
- **With the global context popped off the call stack, the JavaScript engine has finished processing, and the program ends.**

- **Now the call stack becomes empty, the JavaScript engine can relax, as there is no more code to execute.**

- **At this point, `libuv` takes over the major tasks. It handles operations such as processing timers, managing file system calls, and communicating with the operating system.**

- `libuv` performs these heavy tasks in the background, ensuring that asynchronous operations continue to be managed effectively.



In summary, Node.js excels in handling asynchronous I/O operations, thanks to its non-**blocking I/O model.**