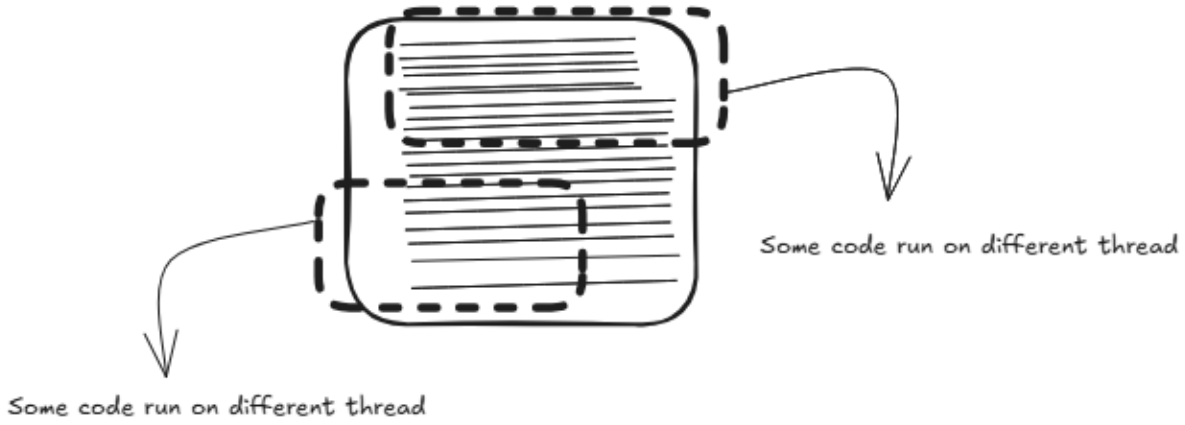
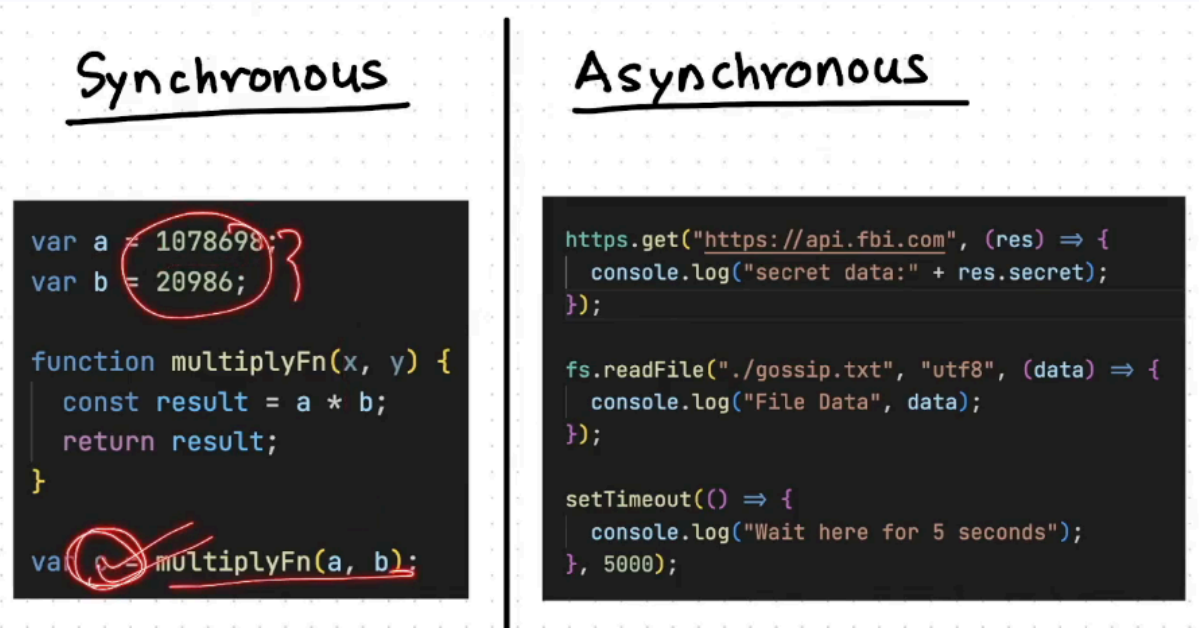
Episode-06-libuv & async IO

- this episode very important for fundamental of nodejs
- "Node.js has an event-driven architecture capable of asynchronous I/O.”
- js is synchronous single threaded Language
- thread means in whole program which many single single container who work within their space
- js in single thread synch → means line 3 executed after line no 2
- where other language some code runs on other thread some code work on different thread



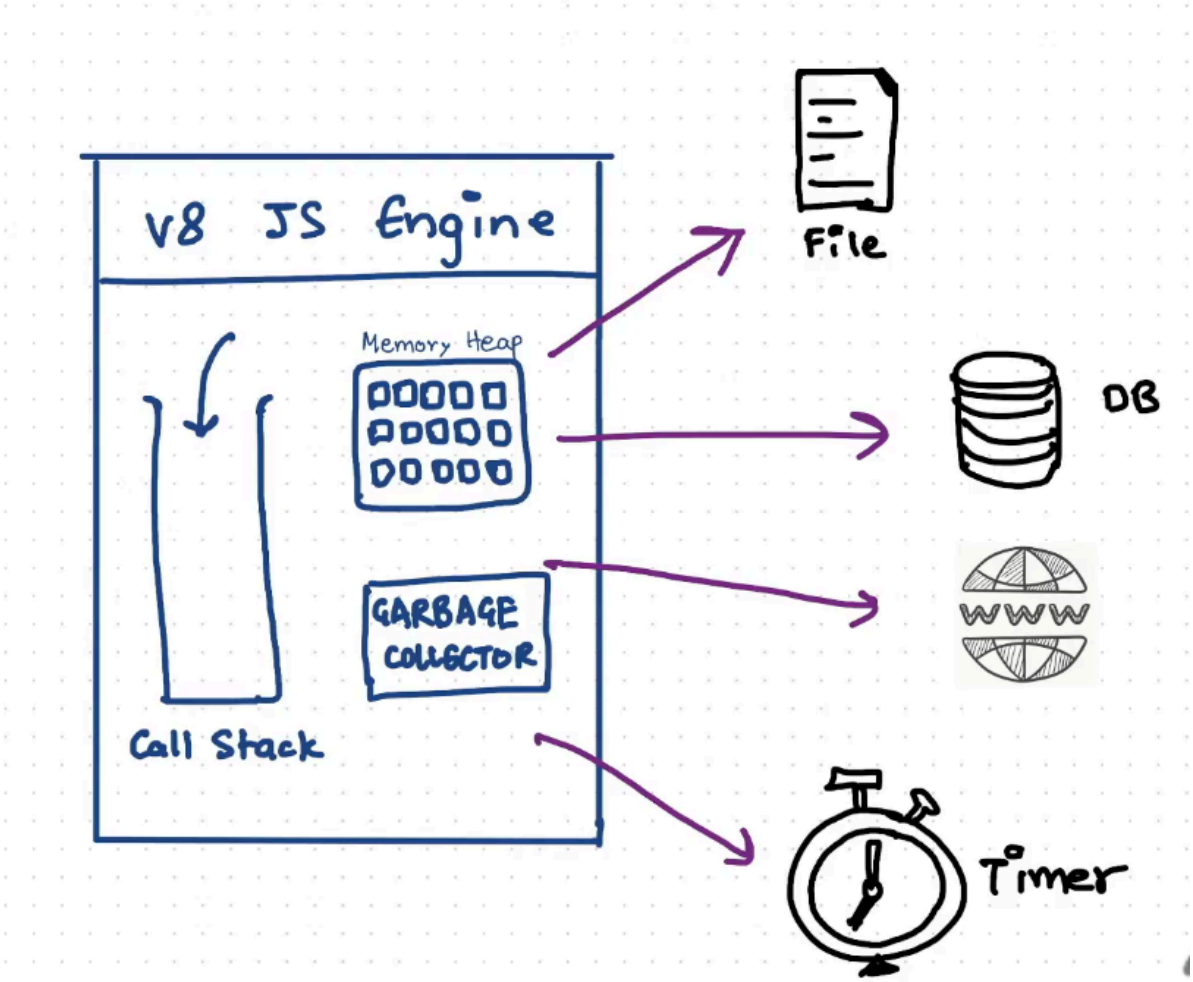
- if language is single thread and synchronous .then your thread is block if something longer program happen how js handle that and then nodejs comes to the picture



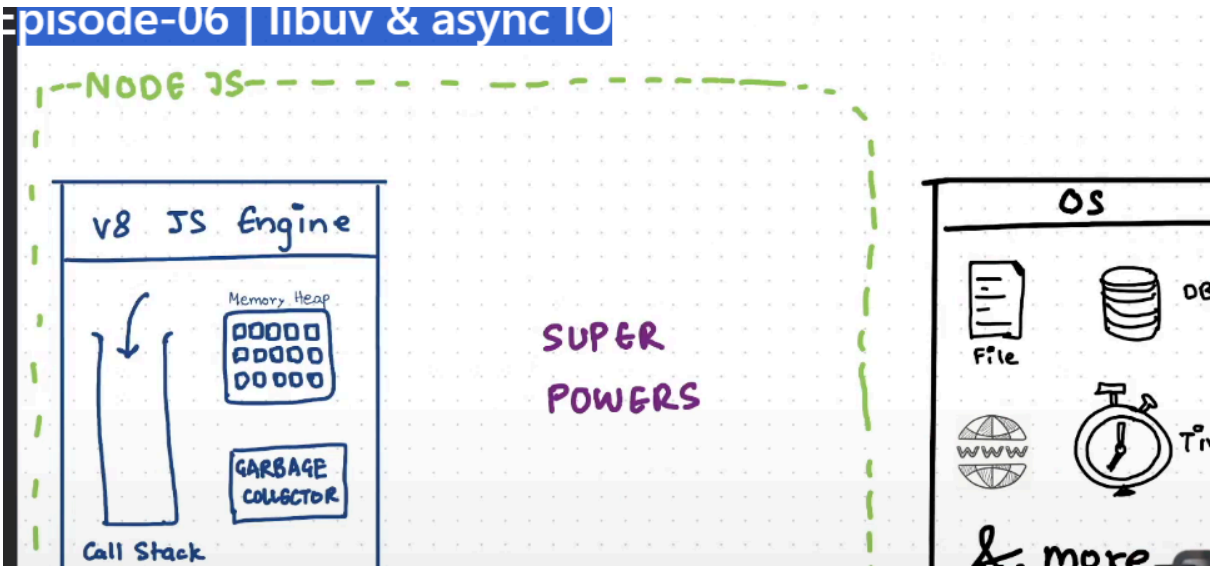
- js engine love the sync code. but don't like the async code because its block js engine so the with help of nodejs async code also work fast

How Async Code Work On JS?

- 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.



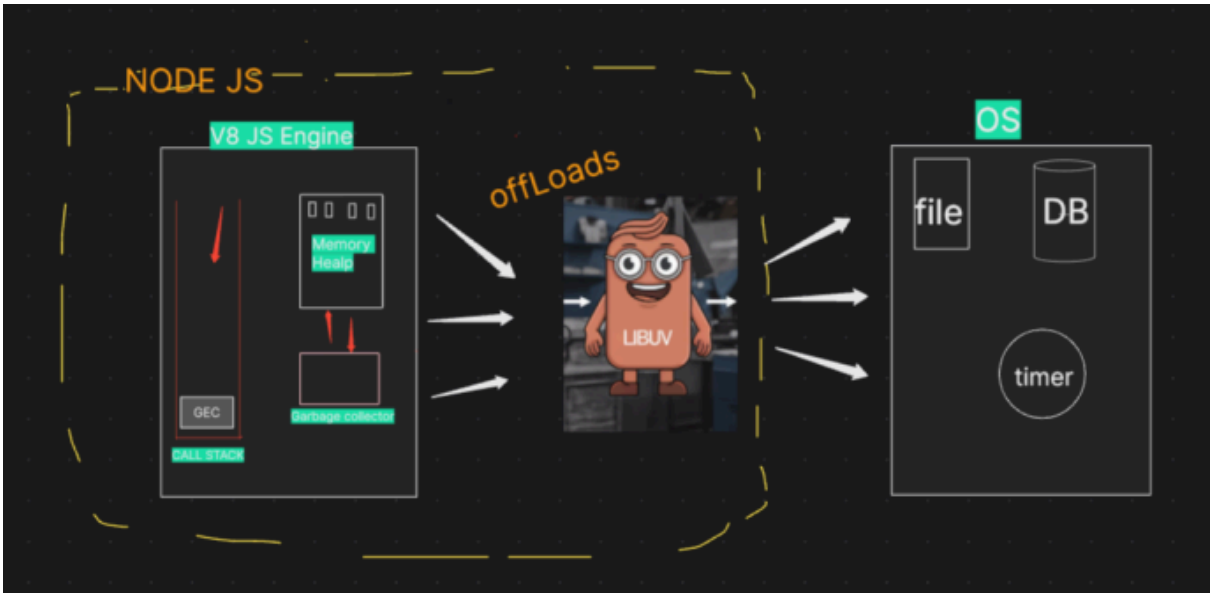
- js need to contact have an access to database or server filesystem etc etc for that js need superpower like nodjs and Nodejs Making that with Library called **LibUV**



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

LibUV

- 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.



- 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.

-Async i/o made Simple by libuv

- at end of c code program which is called libuv who run js Async Code in Server Side
- js is high level language to connect with os u need to low level language .that's why libuv write in c
- libuv is mid level architecture between js and os
- libuv is io callback Queue, timeouts, thread pool , event loop

IN BROWSER Async Code

- The browser provides Web APIs (like `setTimeout`, `fetch`, `XMLHttpRequest`, DOM manipulation methods) that allow the JavaScript engine to offload certain tasks to the browser's underlying multi-threaded environment.
- When these tasks are initiated, they are handled by the browser's internal mechanisms, freeing up the JavaScript main thread.

IN SERVER SIDE Async Code

- Node.js is a server-side runtime environment built on Chrome's V8 JavaScript engine. It uses `libuv`, a cross-platform C library, to handle asynchronous I/O operations (like file system access, network requests, and more).
- `libuv` provides the event loop and a thread pool, allowing Node.js to manage concurrent operations efficiently without blocking the single-threaded JavaScript execution. This is crucial for building scalable server applications.

- js runtime environment and its server-side applications, not directly to JavaScript code running within a web browser.

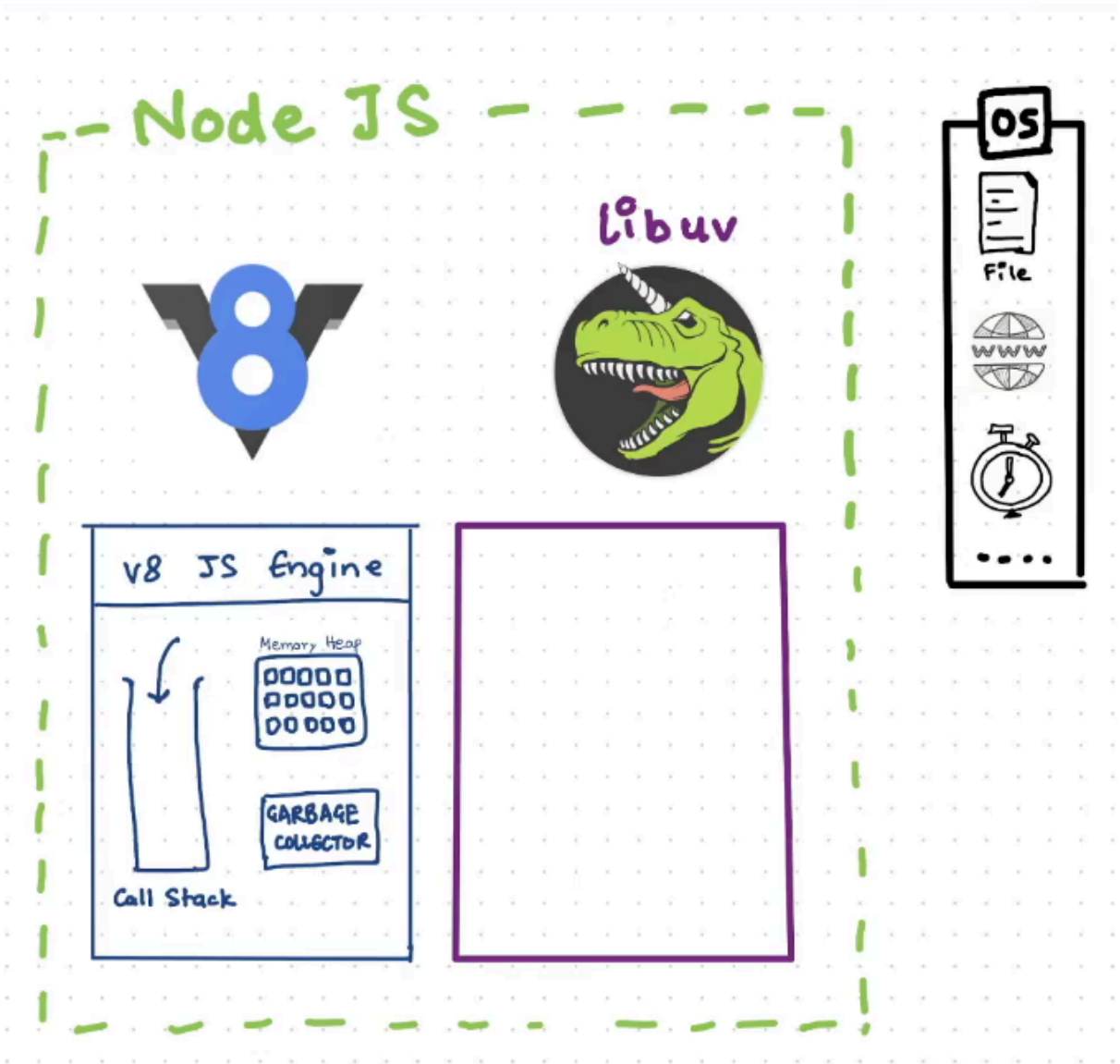
VS CODE Async With Node

- Visual Studio Code (VS Code) is a code editor. When developing Node.js applications in VS Code, you write JavaScript code that will be executed by the Node.js runtime.
- VS Code itself does not directly interact with `libuv`; rather, it provides an environment for writing, debugging, and running your Node.js code, which then leverages `libuv` under the hood.

In summary, `libuv` is an integral part of Node.js for enabling its asynchronous, non-blocking nature, which is a key feature for server-side development. Browser-based JavaScript, while also heavily relying on asynchronous patterns, uses the browser's own Web APIs to manage these operations, not `libuv`.

Libuv is in Node Github as Dependency

- https://github.com/nodejs/node/tree/main/deps/uv



Example-1

```
var a = 1078698;
var b = 20986;

https.get("https://api.fbi.com", (res) => { console.log(res?.secret) });

setTimeout(() => { console.log(setTimeout) }, 5000);

fs.readfile("./gossip.txt", "utf8", (data) => {
```

```
console.log("File Data", data)
});

function multiplyFn(x, y) {
    const result = a * b;
    return result
}
var c = multiplyFn(a, b)
console.log(c);
```

- what ever code you will write a GEC was created
- The variables let a and let b are executed within the GEC (Global Execution Context) during the synchronous phase of the code execution process.

```
https.get("https://api.fbi.com", (res) ⇒ {
console.log(res?.secret)
});

//callback fnA()
```

- 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

```
setTimeout(() ⇒ { console.log(setTimeout) }, 5000);
```

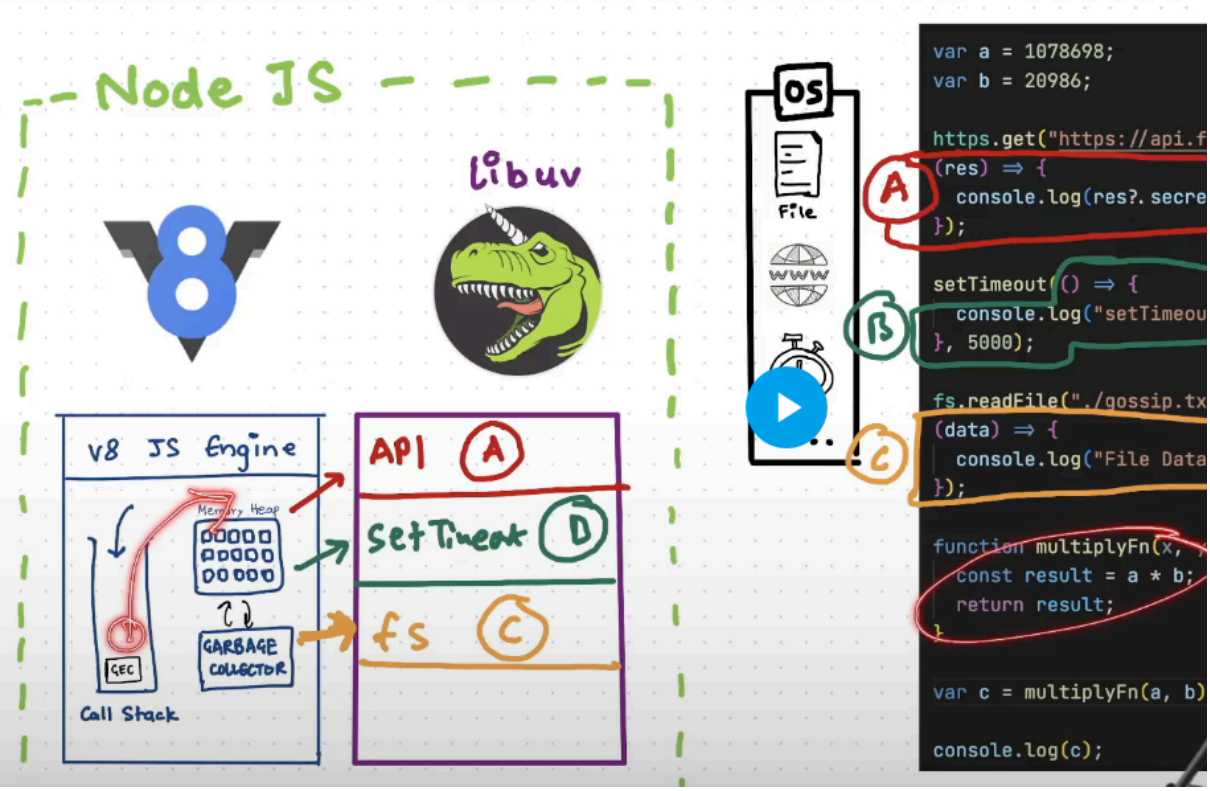
```
//callback fnB()
```

- 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 .

```
fs.readfile("./gossip.txt", "utf8", (data) ⇒ {
console.log("File Data", data)
});

// callBackFn C()
```

- 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
- here when the result variable store a memory in memory heap also remove after the FEC remove from call stack and it remove by garbage Collector

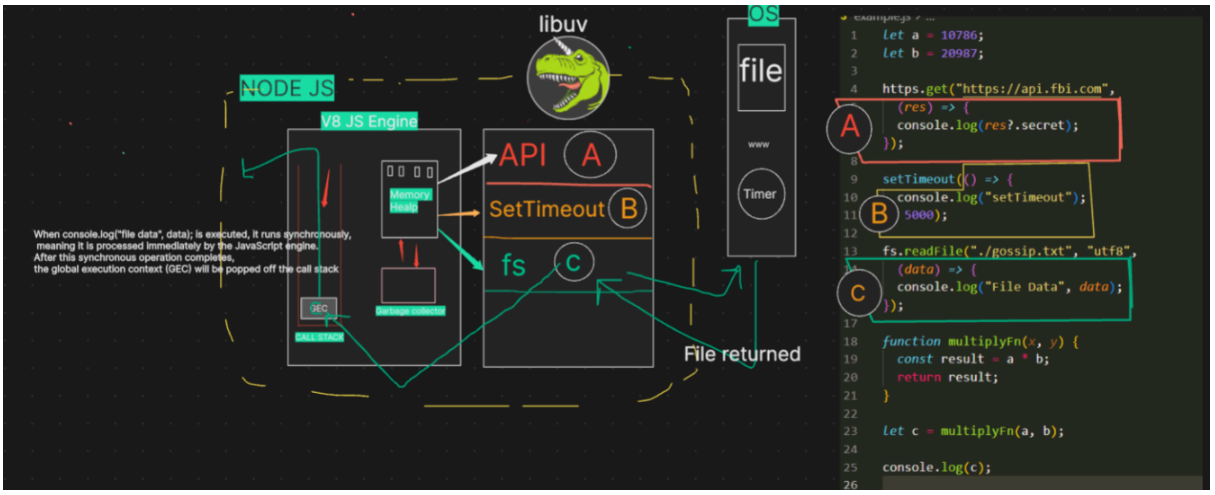
Important Notes While This

- 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.