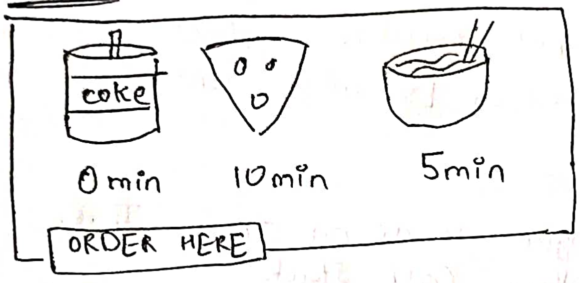
Episode-06 → LibUV and Async IO

- Javascript is a Single Threaded Language. It executes the code line by line in single thread.
- Thread? → It is like a container where we can run any process.
- MultiThread? → There will many containers which share same Memory.

Synchronous

example → Restaurant



- Ⓐ → coke (0 min)
- Ⓑ → Noodles (5min)
- Ⓒ → pizza (15min)
- Ⓓ → Coke (15 min)
- Ⓔ → Noodles (20 min)

Order of execution →

Ⓐ → Ⓑ → Ⓒ → Ⓓ → Ⓔ
coke Noodles Pizza coke Noodle

- Blocking Operation
- Javascript is Synchronous
- JS code example →

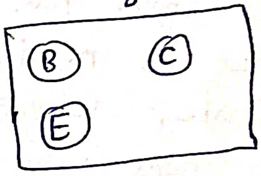
```
var a = 1078698
var b = 20986
function multiplyFn(x,y) {
        const result = a * b
        return result
  }
var c= multiplyFn(a,b)
```

- These task executes immediately

Asynchronous

- Ⓐ → coke (0 min)
- Ⓑ → Noodles (10 min)
- Ⓒ → Pizza (5 min)
- Ⓓ → Coke (0 min)
- Ⓔ → Noodle (10 min)

waiting Area



Order of execution

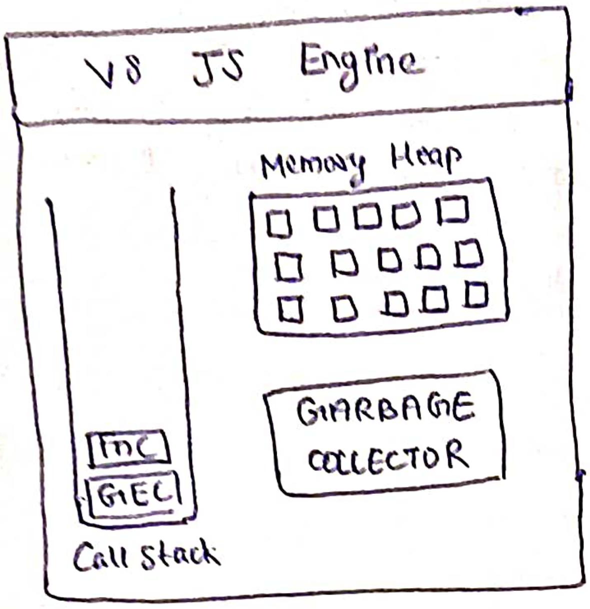
Ⓐ → Ⓓ → Ⓒ → Ⓑ → Ⓔ
Coke ~~Noodles~~ Coke Noodles Pizza Noodles

- Non Blocking Operation
- Javascript Becomes Asynchronous cause of super powers of Node
- JS code example:

```
https.get("https://api.fbi.com", (res)=>{
   console.log("secret data:" + res.secret);
 });
fs.readFile("./gossip.txt", "utf8", (data) => {
   console.log("File Data", data);
 });
setTimeOut(()=>{
        console.log("wait here for 5seconds");
 }, 5000);
```

- These tasks takes time to execute

Understanding Execution of Synchronous Code



executed in JS Engine

```
① var a = 1078698;
② var b = 20986
③ function multiplyFn (x,y){
    ⓘ const result = a*b
    ⓘⓘ return result
  }
④ var c= multiply Fn (a,b)
```

GEC → Global execution context
FnC → Function Execution Context.

- JS Engine has only one call stack and JS Engine runs on single Thread
- Every code that we write is executed in this Call stack.
- JS Engine has Memory Heap which contains numbers and functions, anything that is needed to be stored.
- Garbage Collector is responsible for deallocation of Memory which is no longer used. Responsible for Memory Management.
- Explanation → ① a will be stored in Memory
  - ② b will be stored in Memory
  - ③ function will also be stored in Memory but will not execute
  - ④ in line ④, multiply function will be called, then function execution context will be pushed in call stack
  - ⓘ value of a*b will be calculated and stored in memory created for result variable
  - ⓘⓘ result variable is returned which is eventually stored in memory created for variable c.

⇒ At first Global execution context was created which contains the whole code, after whole code runs, the Global execution will pop out of call stack and hence Call stack becomes empty.

⇒ Also whenever a function is called, A function execution context is pushed inside a call stack and after code of function runs completely, the FnC pops out of stack.

Understanding Execution of Asynchronous Code

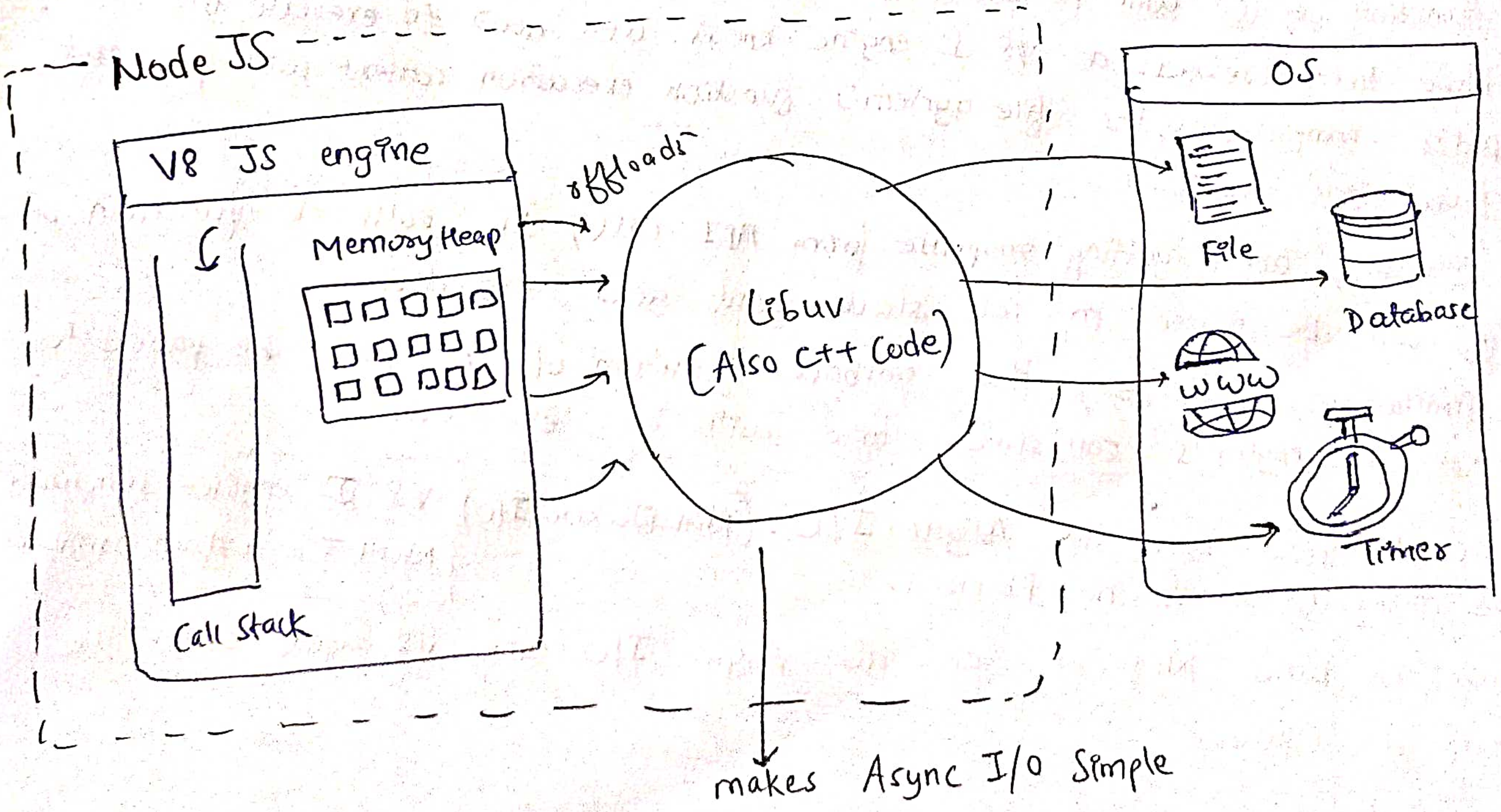
Why Asynchronous JS is needed?

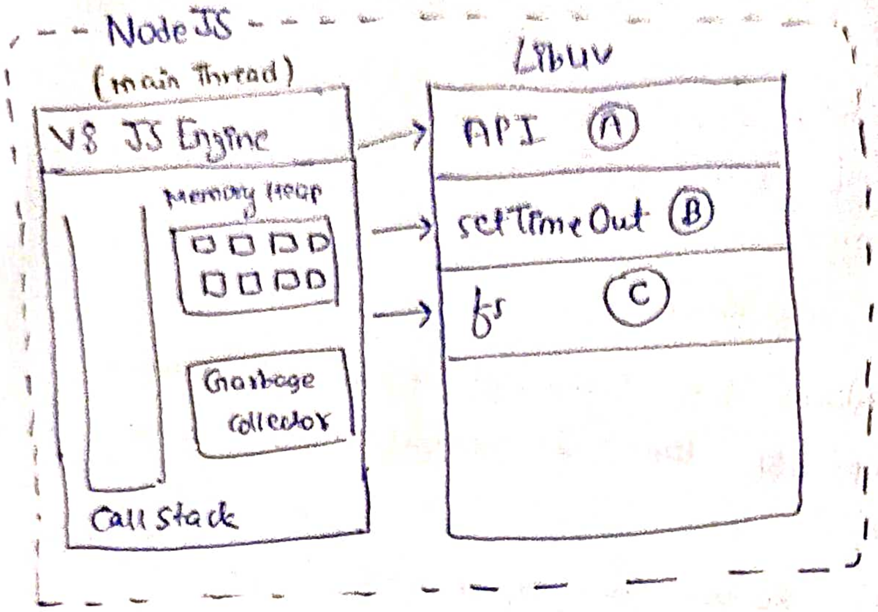
1. To fetch data from $\boxed{\text{file}}$ which may take Time.
2. " " " " $\boxed{\text{database}}$ which takes Time
3. " " " " $\boxed{\text{Web/Internet}}$ " " "
4. When we need a $\boxed{\text{Timer}}$ and more ....

Problem → Javascript waits for None i.e. javascript does not know how to wait. It has no concept of time It executes the code immediately when given to it.

Solution → NodeJs allows the JS to use the superpowers it packs within alongside V8 engine. NodeJs needs to contact the Operating System to Access those functions which it is not capable of like getting Time and fetching file from OS. $\boxed{\text{LibUV}}$ helps the V8 engine in there.

Libuv → when V8 JS engine needs any file, it tells to libuv, libuv talks to file ~~sest~~ system, gets response & gives it back to V8 engine. Also libuv can talk to timer, get database and can make API call. Whatever the responses come from them, it gives it back to V8 engine. JS Engine does synchronous and Most asynchronous things are given to Libuv that is it offloads those tasks to libuv.





```
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)
```

- Javascript executes the synchronous code and when it encounters the code which it doesn't know, gives it to libuv and it performs the task and return the response

Explanation →

- Javascript will finish executing the whole code and will give async tasks to libuv.
- Javascript after completing execution of code synchronously sits idle and waits ~~for~~ response of libuv
- Suppose file reading is completed and data is returned, then callback function of Ⓒ will be passed in call stack where execution context would have been created. as V8 JS engine knows well how to execute this code. After completion, the file system's function execution context will pop out from call stack.
- Similarly, after getting response from API call, the callback function of Ⓐ will be passed in call stack and gets executed.
- Similarly, after 5s, the callback function of Ⓑ will be passed to V8 JS engine's call stack and will be executed.

V8 JS engine → synchronou
NodeJS → Asynchronous

Hence NodeJS can do Async I/O. (Non-Blocking I/O) cuz main Thread is not Blocked.

This is how NodeJS can do Async I/O using V8 engine with the help of libuv.