Episode-06 → libuv & async I/O

- Node.js has an event driven architecture, capable of asynchronous I/O.

- JS is a synchronous single threaded language, it means that there is only one thread and on that thread we are running JS Engine.

- JS execute a piece of code line by line in a single thread, in OS we have lot of process, so inside the processors thread is a separate container where we can run any process, there is a concept of multithreading where we have multiple threads and they share the same memory.

  So, JS run the code in synchronous fashion (One after the Other).

- But if a language is synchronous single threaded it can't do a lot of things, suppose if a multiple request coming in, how would we handle those requests, will we block these thread? Something is taking a lot of time, how would we handle that?
  And JS is not capable of that.

- JS is capable of running fast, synchronous code. JS executes very fast.

Q:> what is the difference between synchronous & asynchronous?
A:> Suppose we are running a restaurant and in our restaurant we have a counter and there anyone can place order.
Menu has 3 options → Coke, pizza and Noodles.

- There is a huge queue, persons – A, B, C, D, E. They are waiting to place their order.

- It takes 5min to prepare the noodles, for pizza it takes 10min and 0 min for coke.

- A → Coke
  B → Noodles
  C → Pizza
  D → coke
  E → Noodles

- If restaurant is running in synchronous way, how these order be handled?

- Person A orders the coke, this order can be fulfilled in 0min, A is now gone from the queue.

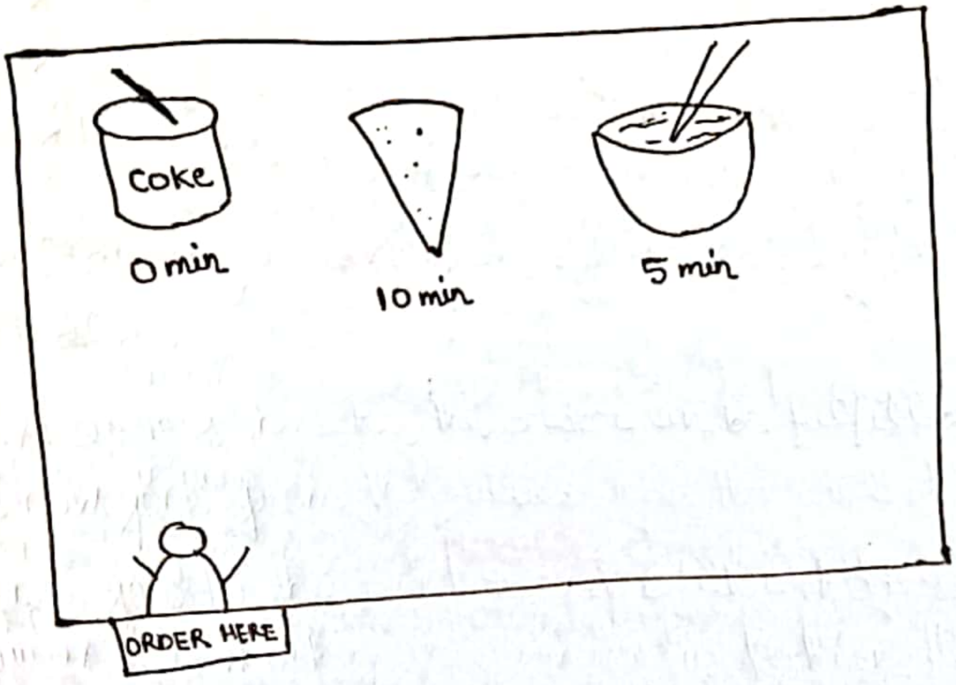
  Person B goes to the counter and order Noodles. Suppose this restaurant is working in synchronous fashion, it means it can only take the order of person C only once the order of person B is finished. So the person B is standing there in the queue and then waiting for noodles to be prepared and after 5 min the noodles is fulfilled.

- Person B is gone from the queue, now Person C is waiting for another 10min because he ordered Pizza. So it means after 15min he got the order. After receiving the order he has gone from the queue.

- Person D immediately got his order because he ordered coke, so this order again fulfilled in 15min.

- And for Person E, he has to wait 20min to get out from the queue and receive the order because he ordered noodles which took 5 min more.

meanwhile there would be so many more people who would have joined the queue.



A →	Coke →	0 min
B →	Noodles →	5 min
C →	Pizza →	15 min
D →	Coke →	15 min
E →	Noodles →	20 min

Now, Asynchronous fashion →

- Person A order fulfill in 0min

- Now Person B gives the order for Pizza, counter person will take the order, sent the message for order of Pizza, Person B goes away from queue and sits in waiting area.

meanwhile Person B gone away from queue B, now person C can Order, his order of noodles is taken, now person C also gone from the queue and sits in waiting area.

- Person D ordered coke, his order can fulfill very quickly. He doesn't even have to wait, his order fulfilled in 0 min.

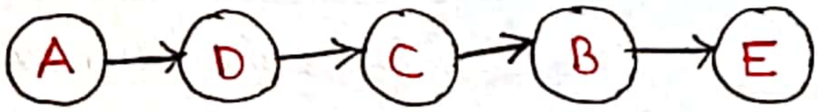
- Similarly Person E goes there and make order for noodles and sits in waiting area.

  Now, the queue is clear.

  If more people come in the queue, they all can be process very fast.

- After 5 min the order of noodles completed, the processing of another noodles order started happening. So after 10 min the Pizza was also ready and second noodle order was also ready, Person C & E has gone from the waiting area like Person C who left the waiting area before 5 min.

- The order of execution over here is First the order of A is fulfilled, then order of D was fulfilled, after D the order of C was fullfilled which was noodles and then later on at 10 min the order of B (Pizza) & order of Person E (noodles) was fulfilled.



$$\begin{tikzcd} \boxed{A} \arrow[r] & \boxed{D} \arrow[r] & \boxed{C} \arrow[r] & \boxed{B} \arrow[r] & \boxed{E} \end{tikzcd}$$

- But in the synchronous mode the order of execution will always be

$$\begin{tikzcd} \boxed{A} \arrow[r] & \boxed{B} \arrow[r] & \boxed{C} \arrow[r] & \boxed{D} \arrow[r] & \boxed{E} \end{tikzcd}$$

According to the queue

- So the best way of running restaurant is asynchronous and async is non-blocking queue.

That's how Node.js works.

- JS in itself is synchronous but with Node.js, superpowers it becomes asynchronous.

Synchronous →

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

- JS Engine loves this type of code because JS Engine can execute this code very fast even through the variable numbers are very high.

Asynchronous →

-
  ```
  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 5 seconds");
  }, 5000);
  ```

- But if we give this code to JS Engine, this code is basically an API Call, reading the file, waiting for some task for particular time, so JS Engine doesn't love this code. This type of code blocks the JS Engine.

- JS Engine wants to keep its queue very clear.

- For API call, waiting Task, reading the files computer has to do a lot of tasks.

Asynchronous task take time to execute.

Q.> How synchronous code is executed by JS Engine ?

A.>- JS Engine has a call stack and this whole thing running on a single thread so it has just one call stack and every code, we write executed in call stack over there.

- JS Engine also has a Memory Heap, variable allocation take place here and their value pushed inside them.
  Memory Heap contains variable, functions, etc.

- JS Engine also has a Garbage Collector, later in the programme if we don't use a particular variable, the garbage collector collects unused variable, unused functions.

- So whenever we run a code, a Global Execution Context (GEC) is created, all the code runs inside call stack inside GEC.

- GEC pushed inside call stack after all code wrapped and passed into GEC.

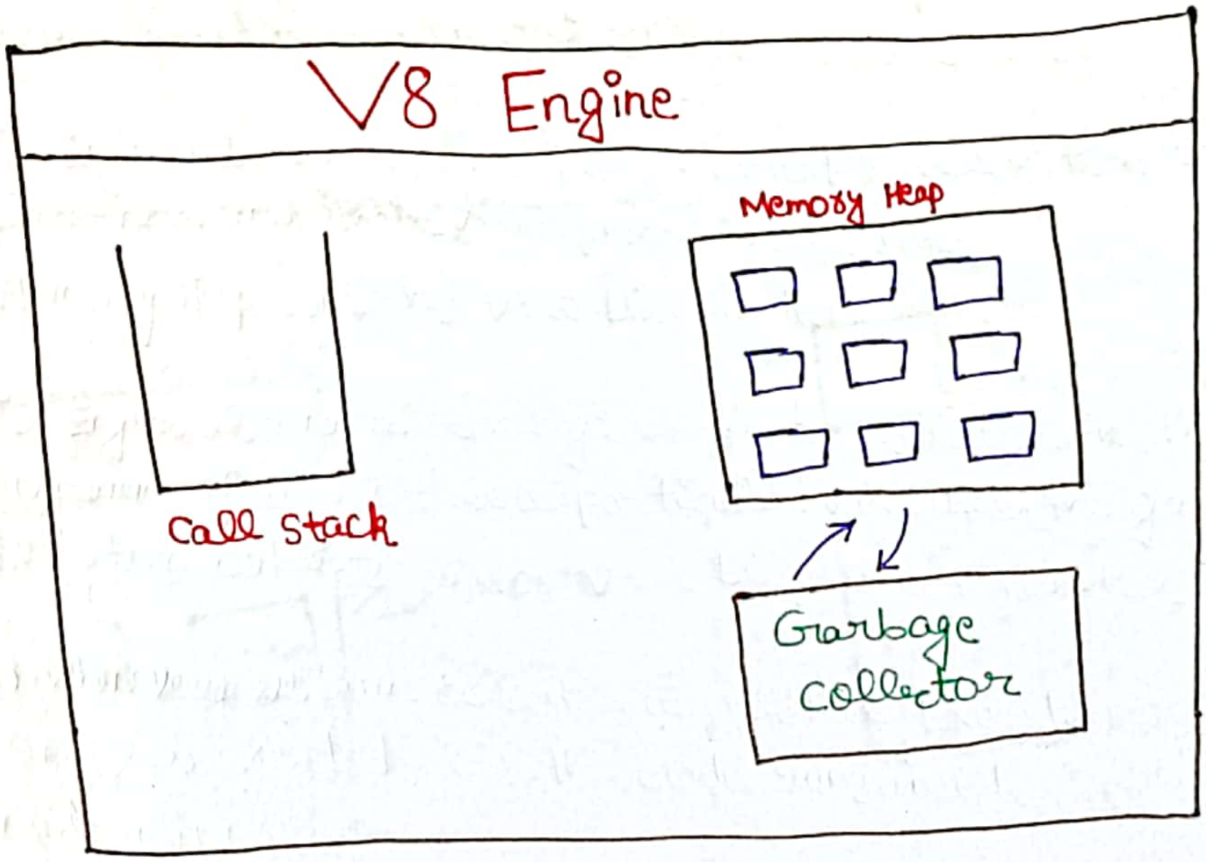
- Whenever we call a function, one more Execution Context is created and pushed inside call stack.

- call stack doesn't push out GEC because the code is still running.

- The code inside function will run in Functional Execution Context.

- FEC after finishing it's job running a function moves out of Call Stack and poped of from callstack.

- Once the whole code is executed, GEC is also gone away from the call stack.
  Call stack will become empty.



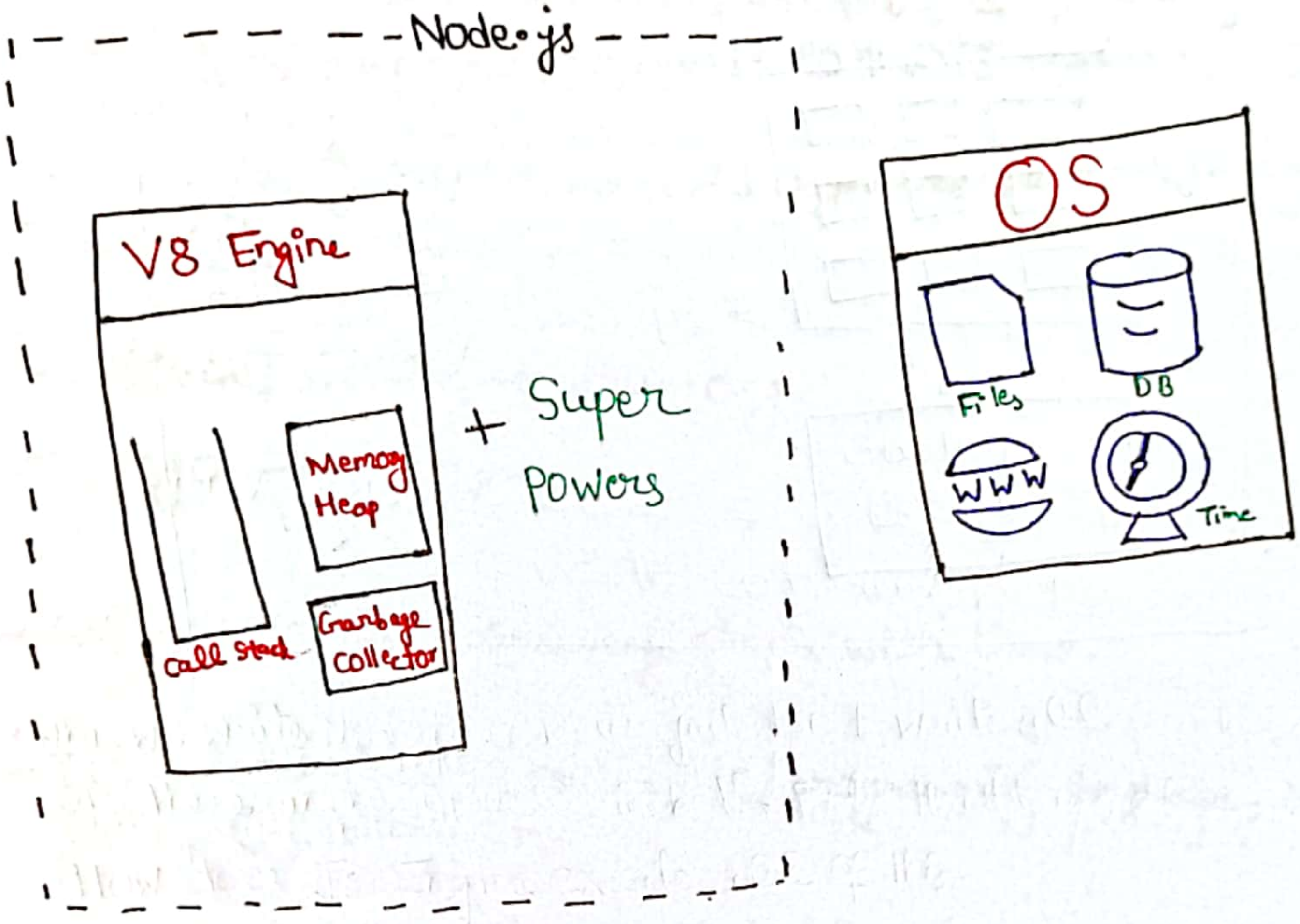
- When we write Node.js code, we just don't write all synchronous code, there can be API calls in betn, File Handling, etc in between.
  How does JS Engine execute all of this.

- Note → JS Engine in itself doesn't have a concept of time, we just give JS code it will execute it, it doesn't know how to wait.

  Time, Tide and JS waits for none.

- If we want that JS Engine can read File, DB & waits for some time So we need Some superpowers.

- JS Engine Can't do this that's why Node.js comes into the picture and Node.js give the superpowers to JS Engine to interact with Operating System and do that.



- Suppose if we want to read a file, so JS Engine needs to talk to Operating System (OS) who knows the location of the file.

- JS Engine doesn't has the capability to connect with Database, it need Some Superpower to connect with DB.

- Similarly, JS Engine needs to connect to time function inside OS because OS manges the time.

- These Superpowers are given to JS Engine by Node.js and Node.js is doing that by a library known as "libuv" (Super Hero).

- libuv is very core thing inside Node.js.

- JS Engine makes file access it can't directly do that, it tells libuv and libuv talks to the file system, get back the response and give it to the V8 Engine.

- V8 Engine just off loads (add asynchronous task) everything to libuv.

- official website → libuv.org
- Logo → Unicorn

- Async I/O has become very simple with libuv.

- It's just somepiece of code of 'C' programming language, it's just a C library.

- 97% of code of libuv written in C language.

- JS is High-level language, to connect with OS we need a low-level language which is 'C'.
C language is very efficient to talk to OS. That's why libuv written in C language.

- libuv acts as a middle layer between JS Engine & OS.

- Node.js github repository → deps → uv
  (This is libuv)

  It is existing as the dependency.

- Garbage Collection is important concept in C language.

- Apart from V8 & libuv, there are lot more modules & dependencies.

- Whenever JS engine sees an API call, File reading operation, setTimeout, it off loads all these codes to the libuv and then libuv manages it.

Q.> How mixture of Synchronous & asynchronous code work inside Node.js?

A.>
```
Var a = 1078698;
Var b = 20986;

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

setTimeout(() => {
    Console.log("setTimeout");         → (B) (callback fxn)
}, 5000);
```

```
fs.readFile("./gossip.txt", "utf8",
  (data) => {
    console.log("File Data", data);
  });
```
→ Ⓒ (Callback fxn)

```
function multiplyFn(x,y) {
    const result = x*y;
    return result;
}

var c = multiplyFn(a,b);

console.log(c);
```

- Whatever code we write in Node.js will run Call stack and there will be a GEC which will be created inside Call stack, the code will be executed line by line, variable memory is allocated at Memory Heap, Garbage collector is in Sync with the Memory Heap.

When the code of API call is executed inside GEC, JS Engine connects with libuv and handover the API call and ask for the data, libuv just register the API call and libuv will also take the callback.

- This callback function will be executed once the API call has been made.

But JS Engine can't wait.

- Meanwhile libuv managing the API call, JS Engine will move to the next line.

- Now, JS Engine moves to the setTimeout line, this timer again handled by libuv.

- JS moves to the File read line, it again call the libuv.

- When JS Engine move to the 'multiply Fn', it easily know how to deal with this function.

- It will give memory to this function to Memory Heap.

- Var C = multiplyFn(a, b);

  Whenever a function call is made, JS Engine will create a new Functional Execution Context.

- After finishing the function execution, FEC will moves out of the Callstack, so all the memory which was allocated to the 'result' is cleared by Garbage collector.

- After printing value of 'C' on console window, GEC moves out of the callstack.

- libuv manages multitask at a time.

- Suppose from the code of 'FileSystem' file is returned now, as soon as the file data has comeup, now Libuv knows the callback function i.e. C needs to be calculated & executed.

- Now libuv will just give the callback function to V8 engine and it will push it inside the callstack, now C (callback fxn) function is created and now C function will run line by line.

- And Execution Context will move out of the callstack after completing the job.

- Suppose meanwhile API call was being successful, now libuv will give callback function A to the V8 Engine and it will execute it line by line.
  Similarly with setTimeout & callback function B.

- Node.js is asynchronous & V8 engine is synchronous.

- Node.js can do Async I/O, that's why it is known as Non-Blocking I/O because it's not blocking the main thread.
  And even with single thread, it can do so many async operation very fast.

For more PDF, follow

gitHub → rajeshjha2000