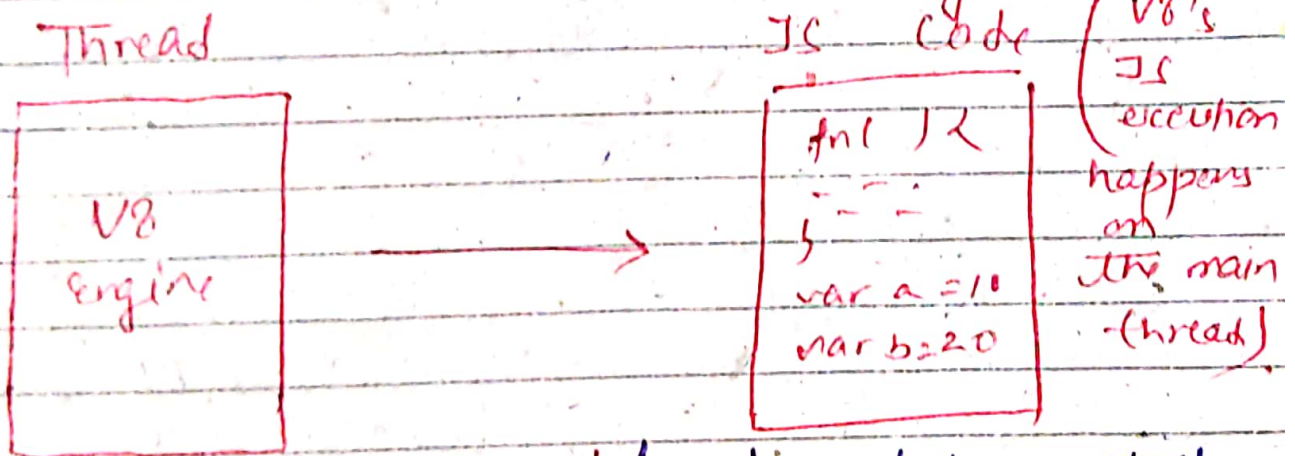# Namaste Node.js

Chapter -06 — Libuv & Asynchronous I/O

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

→ Javascript is a) synchronous → (line by line code execution.)

b) single threaded (only one thread on which V8 engine runs)

Thread

JS code (V8's JS execution happens on the main thread)

```
fn( ){
;---
;
var a =10
var b=20
```

→ A thread is the smallest unit of execution inside a process.

→ A task running inside the program / process.

→ lightweight than process.

(code gets executed in synchronous way in Javascript-i.e. before moving onto the second line, first line of code must be executed.)

→ and on single thread whole code runs.

Synchronous v/s Asynchronous

1. Execution Model

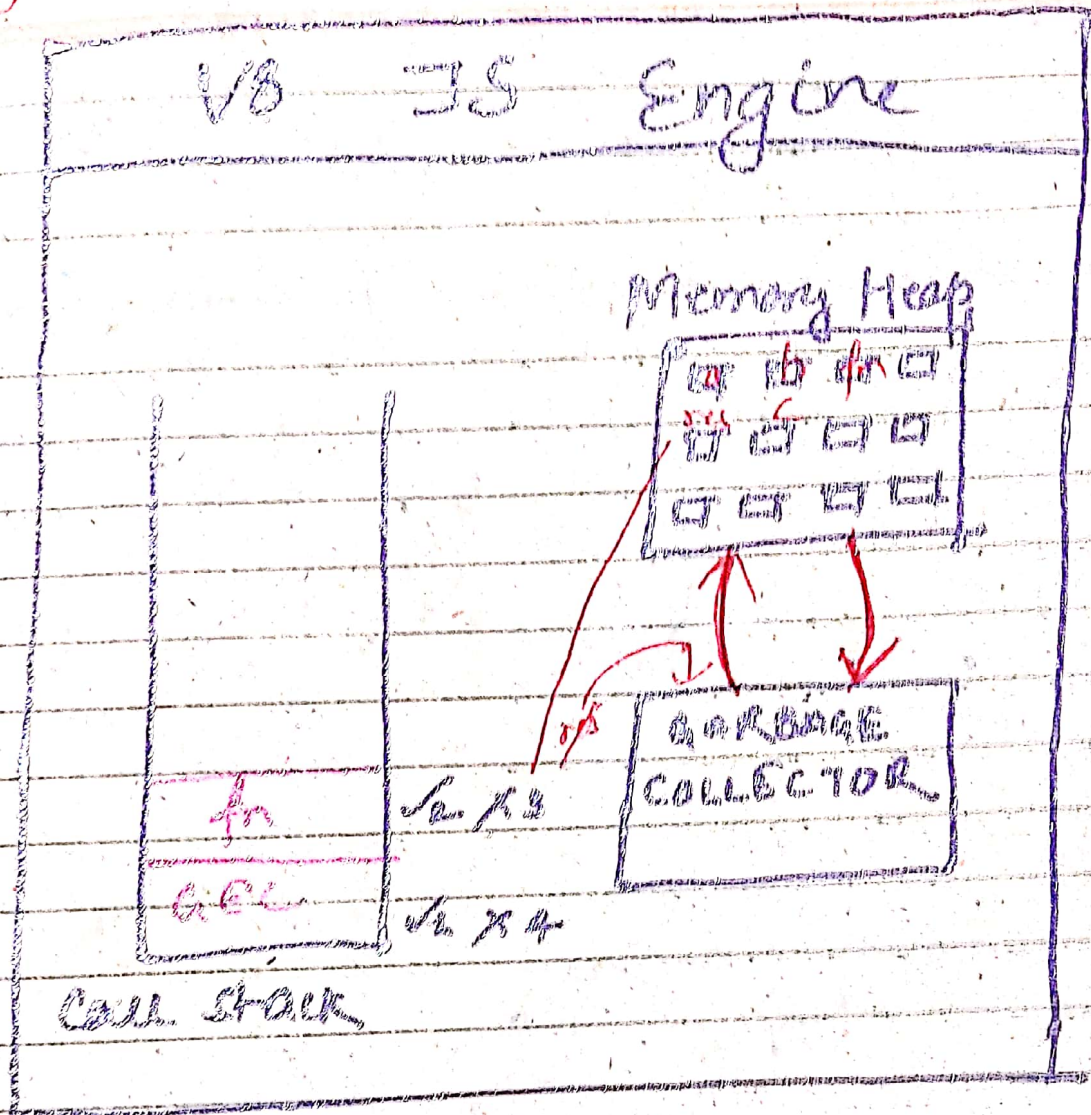| Feature	Synchronous	Asynchronous
a) Execution order	code runs line by line, blocking further execution until current task completes	code can start long tasks and move on without waiting
b) Blocking	Yes, blocks the call stays	No blocking
c) Example	CPU-heavy loops, calculations	API fetch, timers, file I/O

2) Impact on Performance

	Synchronous	Asynchronous
a) UI Responsiveness	→ can freeze UI	i) keeps UI smooth
b) Concurrency	→ None	→ Allows concurrency through callback, promises
c) wait times	→ High (because each task waits)	ii) Lowers (task overlap)

3) Where Each is used

Synchronous	Asynchronous
a) Pure computations	a) Network calls
b) Basic scripts	b) DB queries
c) Immediate operations	c) timers
d) Blocking loops	d) File system operations

Synchronous code Running in JS Engine

```
┌─────────────────────────────────────────────┐
│              V8   JS  Engine                  │
│                                               │
│                          Memory Heap          │
│                        ┌──────────────┐       │
│                        │ a  b  c  d   │       │
│                        │ e  f  g  h   │       │
│                        │ i  j  k  l   │       │
│                        └──────────────┘       │
│     ┌────┬────┬──────────────┐                │
│     │ Fn │ a,b│              │  Garbage        │
│     │    │    │  Collector   │                 │
│     │Gec │ x,y│              │                 │
│     └────┴────┴──────────────┘                │
│      Call Stack                               │
└─────────────────────────────────────────────┘
```

→ var a = 1078698;
var b = 20986;

function multiplyFn (x, y) {
const res = a*b;
return result;
}

var c = multiplyFn (a, b);

i) all the functions and variables get stored inside the memory heap.

ii) whenever JS file runs, a GEC gets created onto call stack.

iii) If a function gets called, its EEC gets pushed onto the stack.

iv) once everything gets executed, GEC get destroyed and get garbage collected.

Asynchronous Code Running in JS Engine

V8 JS Engine

Memory Heap

file system →

File

fetching data →

Database

Garbage Collector

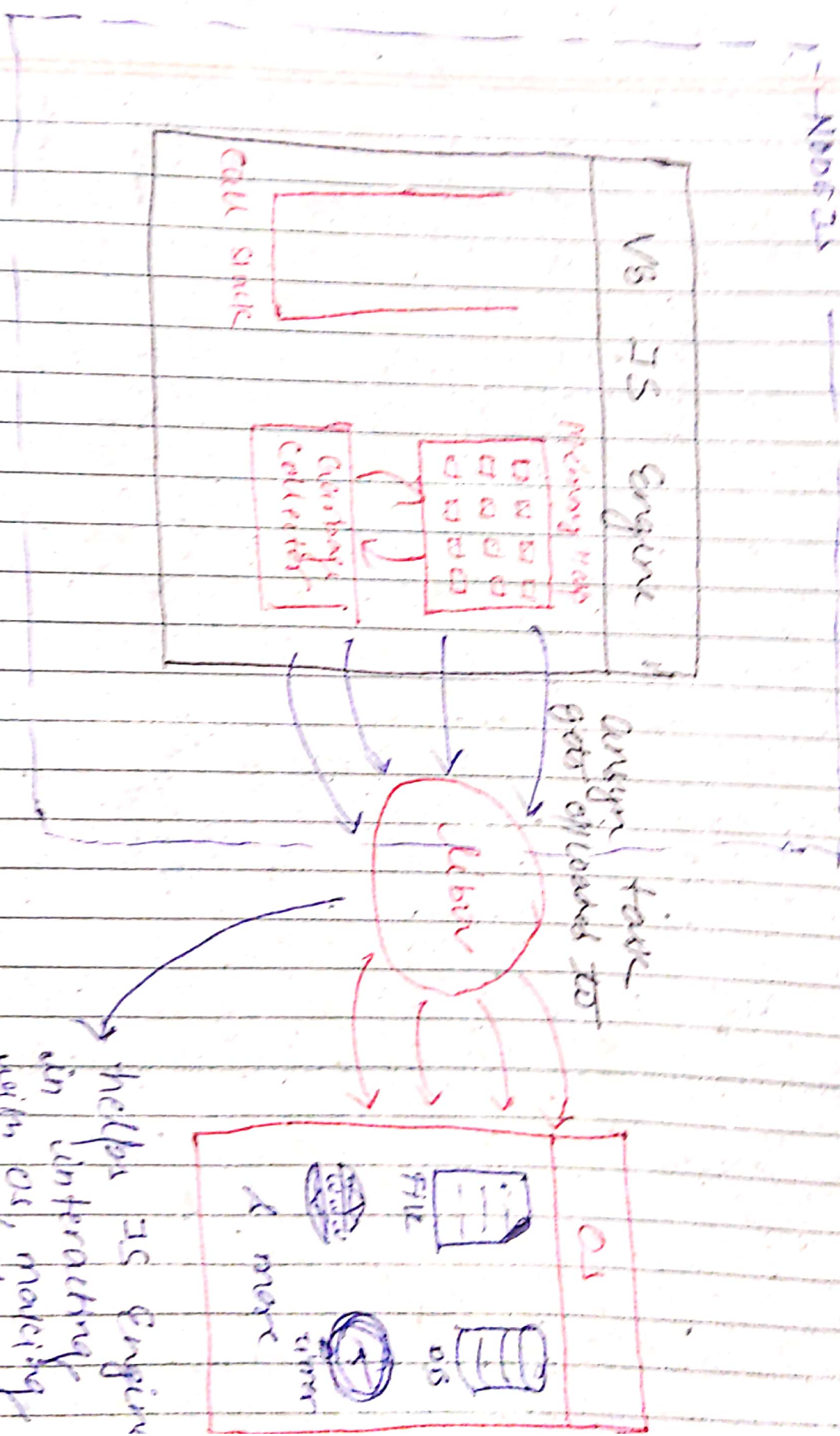API calls →

WWW

Call Stack
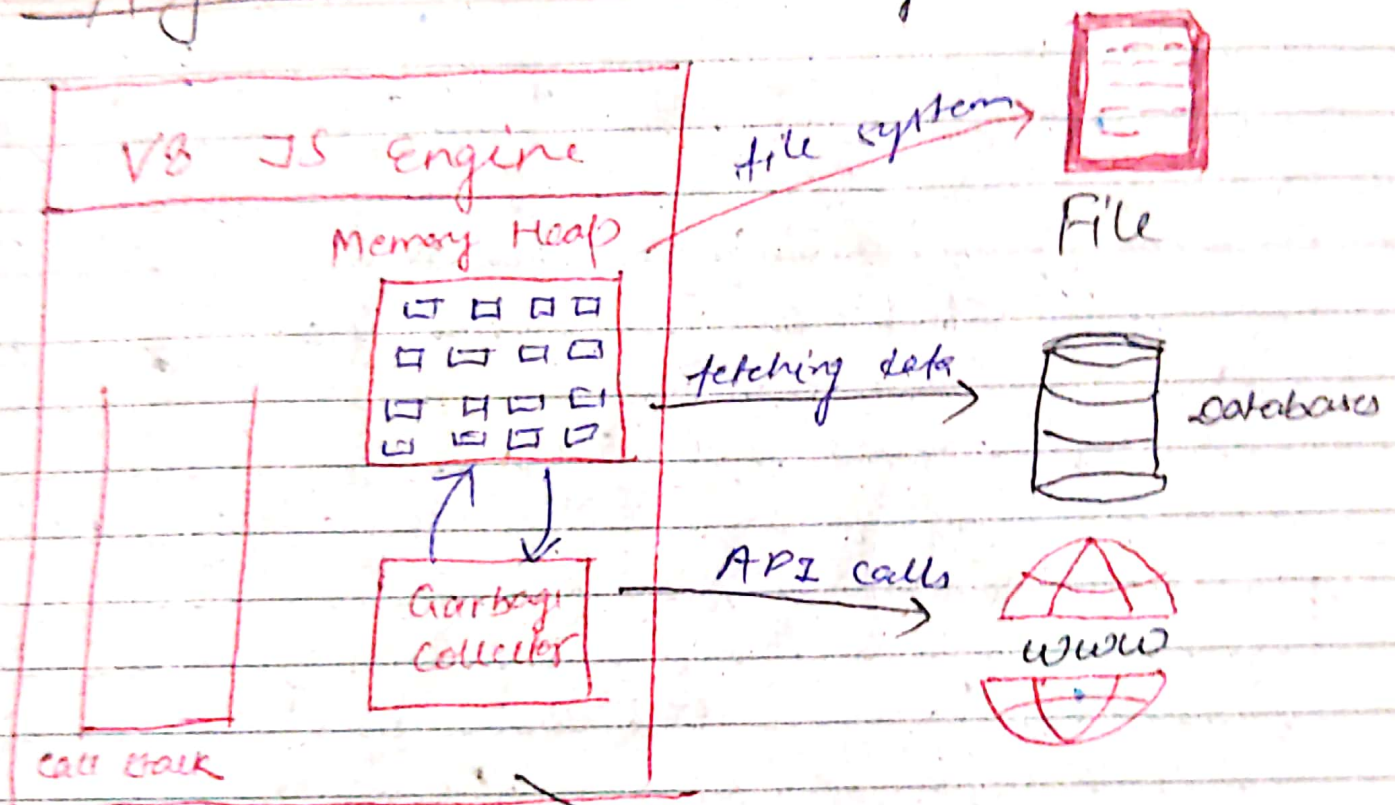
→ these all other functionalities are not part of core JS (e.g., timers), but provided by its runtime environment. → timers

→ all the asyn operation gets offloaded to libuv.

Q) What is libuv ?

A) i) libuv is a multi-platform c library that provides support for asynchronous $I/0$ based on event loops.

ii) It supports epoll(4), Kqueue(2), windows $IOCP$, Solaris event ports and linux $io-uring$.

iii) It is primarily designed for use in Node.js but it is also used by the other software projects.

iv) Node.js is an asynchronous because of libuv (because it contains Event Loop, Thread Pool ----).

v) It also interacts with operating systems.
(works as a communicator between Node.js and operating system).
(it part of Node.js).

Q)) How the Javascript file runs?

A)) i) A global Execution Context is created and pushed onto the call stack.. (all code runs inside it) (in single ~~sync~~ threaded sync manner.

ii) Variables and functions gets stored inside the GEC in < key, value > pairs and occupy space in memory heaps.

iii) Garbage collectors works in sync with the memory heap. It collects and clean the garbage. (destroy their (free memory from them) variables which doesn't have any references (they are not in use now).)

iv) whenever JS engine encounters any asynchronous operation (API calls, timers, file read/write, etc.) these thing happens a) it calls libuv to execute these asynchronous calls.
b) libuv registers this asynchronous call and it will also take the callback (store it)
c) JS engine moves to the next line and handles

synchronous (execute it itsey) and
asynchronous operations (offloads to libuv).
accordingly.   (main thread doesn't)
                                (get blocked.)

d) If it encounters any function calling,
   a functional execution context gets
   created and pushed onto the call
   stack. If the whole code (function)
   gets executed, it moves out of the
   call stack and garbage collector
   will free up all the memory
   occupied by this function. (Asynchronous)

e) when the whole JC file finished
   executing, GEC popped off the
   call stack & Garbage collector
   will free up all the memory spaces
   occupied by the whole JS file.
   (call stack becomes empty).

+) Behind these all things, libuv is
   managing and handling all the async
   operations,
   suppose an async operation (fileread)
   gets completed and it has acquired the
   required data then, it gives the

callback function (associated with fsreadFile)
and gives it to V8.

→ call stack

g) V8 engine pushes the callback
function to get it executed in
single threaded synchronous manner, gets
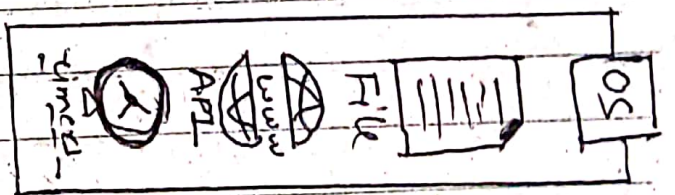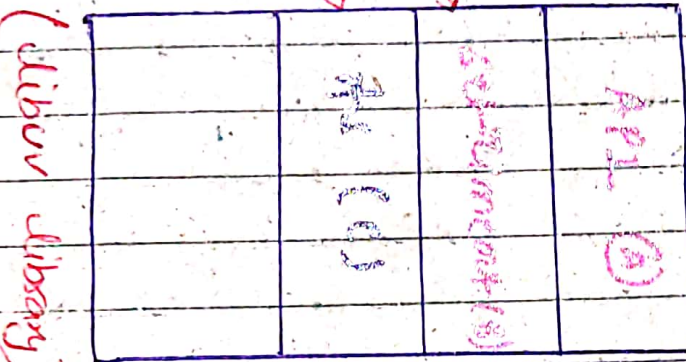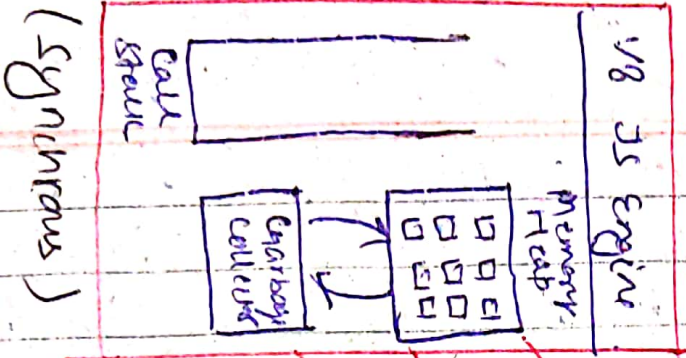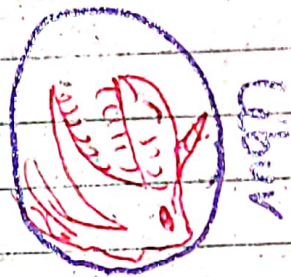executed and popped off the
call stack.

h) Suppose many other async operations
gets resolved by libuv, then same
thing will be done (passed to
JS engine and to get executed) for
them.

NOTE :: Node.js in Asynchronous
Javascript engine is synchronous.

• In these manner, Node.js able to
perform asynchronous I/0 without blocking
the main thread and being single
threaded.

NODE.js (Asynchronous)

V8 JS Engine

```
JS Engine
  Memory-
   Heap
  Garbage
  collector
Call
Stack
```

(Synchronous)

Library

Api

```
setTimeout(s)
readFile(s)
Api
```

(libuv /libsag)

All async
offloaded to
libuv

making main
thread free

OS

```
OS
File
www
API
timer
```

JS code

```
var a = 10;
var b = 20;
```

(A)
```
https.get ("https://api.mcn.com"
(res) =>
  console.log ("res! source").
)
```

(B)
```
setTimeout ((s) =>
  console.log ("setTimeout").
y 5000 )
console.log ("File data", data).
```

(C)
```
fs.readFile (",gossip.txt","utf-8"
(data) =>
  console.log ("File data", data).
y).

function multiplyFn (n, y) {
  const result = a * b;
  return result;
}

var c = multiply Fn (a, b);
console.log (c);
```