

Notes:

<https://petal-estimate-4e9.notion.site/Class-2-JS-101-2f17dfd107358074a477dc0f71bf9a11>

Synchronous code

Synchronous code is executed line by line, in the order it's written. Each operation waits for the previous one to complete before moving on to the next one.

For example

```
function sum(n) {  
    let ans = 0;  
    for (let i = 1; i <= n; i++) {  
        ans = ans + i  
    }  
    return ans;  
}  
  
const ans1 = sum(100);  
console.log(ans1);  
const ans2 = sum(1000);  
console.log(ans2);  
const ans3 = sum(10000);  
console.log(ans3);
```

JavaScript | ⚡ ...

I/O heavy operations

I/O (Input/Output) heavy operations refer to tasks in a computer program that involve a lot of data transfer between the program and external systems or devices. These operations usually require waiting for data to be read from or written to sources like disks, networks, databases, or other external devices, which can be time-consuming compared to in-memory computations.

Examples of I/O Heavy Operations:

1. Reading a file
2. Starting a clock
3. HTTP Requests

In this example, the `index.js` program is dependent on the Operating System because it needs the OS to read the file (`a.csv`) from disk.

<https://petal-estimate-4e9.notion.site/I-O-heavy-operations-2f17dfd107358038a458dea99e0f8ef4>

When we call something like:

```
read("a.csv")
```

Node.js does not read the file by itself using the CPU. Instead, it asks the Operating System to perform the file read operation.

So the total time taken depends on:

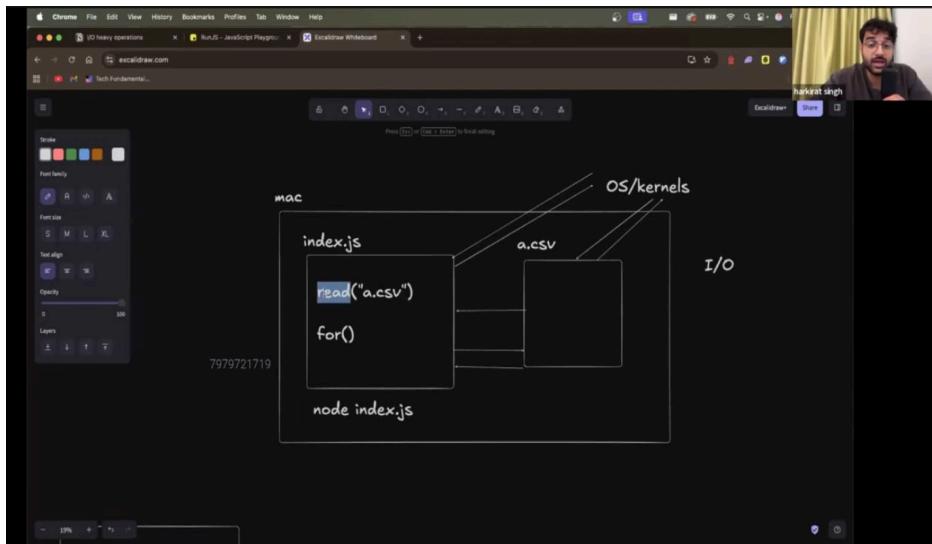
- How fast the OS can access the disk
- File size
- Disk speed (SSD vs HDD)
- System I/O performance

During this waiting time, the CPU is mostly idle (not doing heavy computation).

That's why this task is called:

👉 I/O Heavy (Input/Output Heavy)

because it depends on input/output operations handled by the Operating System, not on CPU processing power.



Let's try to write code to do an **I/O** heavy operation -

1. Open repl.it
2. Create a file in there (a.txt) with some text inside
3. Write the code to read a file **synchronously**

```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);
```

4. Create another file (b.txt)
5. Write the code to read the other file **synchronously**

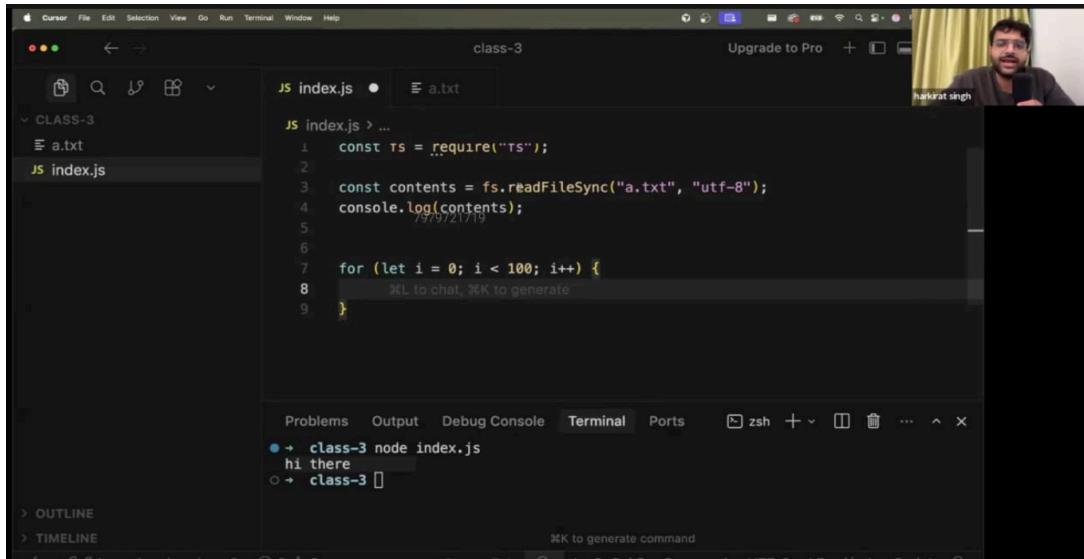
```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);

const contents2 = fs.readFileSync("b.txt", "utf-8");
console.log(contents2);
```

💡 What is wrong in this code above?

We are reading code synchronously it means we are waiting each time when we reading file because it depend on operating system so cpu heavy task also get delay like for loop



I/O bound tasks vs CPU bound tasks

CPU bound tasks

CPU-bound tasks are operations that are limited by the speed and power of the CPU. These tasks require significant computation and processing power, meaning that the performance bottleneck is the CPU itself.

```
let ans = 0;
for (let i = 1; i <= 1000000; i++) {
    ans = ans + i
}
console.log(ans);
```

 A real world example of a CPU intensive task is `running for 3 miles`. Your legs/brain have to constantly be engaged for 3 miles while you run.

I/O bound tasks

I/O-bound tasks are operations that are limited by the system's input/output capabilities, such as disk I/O, network I/O, or any other form of data transfer. These tasks spend most of their time waiting for I/O operations to complete.

```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);
```

💡 A real world example of an I/O bound task would be `Boiling water`. I don't have to do much, I just have to put the water on the kettle, and my brain can be occupied elsewhere.

Functional arguments -

<https://petal-estimate-4e9.notion.site/Functional-arguments-2f17dfd107358023ad2efc88a2edfc3f>



```
JS index.js 2 ● a.txt
JS index.js > doArithmetic
1
2   function sum(a, b) {
3     return a + b;
4   }
5
6   function sub(a, b) {
7     return a - b;
8   }
9
10  function doArithmetic(a, b, whatToDo) {
11    if (whatToDo == "sum") {
12      return sum(a, b)
13    }
14    if (whatToDo)
15  }
16
17  const ans1 = doArithmetic(1, 2, "sum");
18  const ans2 = doArithmetic(1, 2, "sub");
19
```



```
••• ← → class-3 Upgrade to Pro
JS index.js X a.txt
JS index.js > ans1
10  function doArithmetic(a, b, whatToDo) {
11    if (whatToDo == "sum") {
12      let s = sum(a, b)
13      return s;
14    }
15    if (whatToDo == "sub") {
16      let s = sub(a, b)
17      return s
18    }
19  }
20
21  const ans1 = doArithmetic(1, 2, "sum");
22  const ans2 = doArithmetic(1, 2, "sub");
23
```



A screenshot of a Mac OS X desktop showing a terminal window titled "index.js". The code in the terminal is:

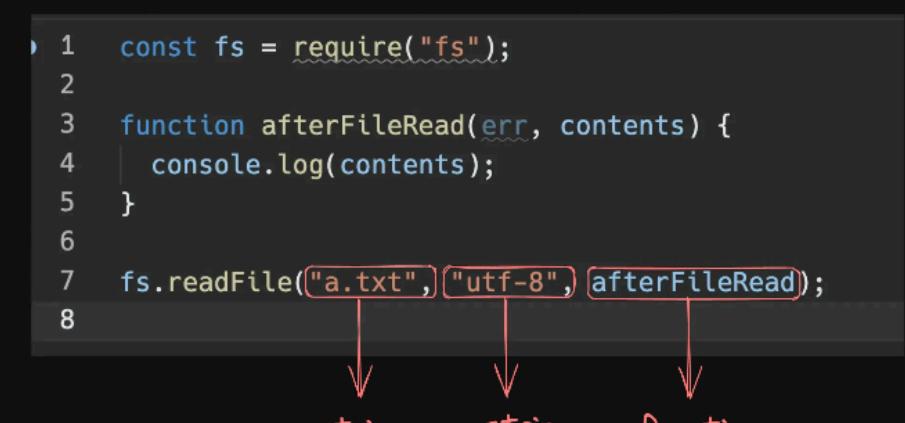
```
JS index.js > doArithmetic
1
2 function sum(a, b) {
3     return a + b;
4 }
5
6 function sub(a, b) {
7     return a - b;
8 }
9
10 function doArithmetic(a, b, fn) {
11 }
12
13
14 const ans1 = doArithmetic(1, 2, sum);
15 const ans2 = doArithmetic(1, 2, sub);
16
```

The status bar at the bottom shows "Launchpad class-3" and "Ln 10, Col 31". A video call interface is visible on the right, showing a man with a beard and glasses, identified as "harkirat singh".

Asynchronous code, callbacks

Let's look at the code to read from a file `asynchronously`. Here, we pass in a `function` as an `argument`. This function is called a `callback` since the function gets `called back` when the file is read

```
1 const fs = require("fs");
2
3 function afterFileRead(err, contents) {
4     console.log(contents);
5 }
6
7 fs.readFile("a.txt", "utf-8", afterFileRead);
8
```



Three red arrows point downwards from the three arguments in the `readFile` call to the words "string", "string", and "function" respectively, indicating their types.



A screenshot of a video conferencing interface. On the right side, there is a video window showing a man with glasses and a beard, identified as "harkirat singh". The video window has a dark background with yellow curtains visible behind him. The overall interface looks like a standard video call application.

```
JS index.js > ...
1 const fs = require("fs");
2
3 function fileReadCallback(err, contents) {
4     console.log(contents);
5     console.log(contents);
6     console.log(contents);
7 }
8
9 fs.readFile("a.txt", "utf-8", fileReadCallback);
10
11 let s = 0;
12 for(let i = 0; i < 100000; i++) {
13     s += i;
14 }
15
16 console.log(s);
17
```



A screenshot of a video conferencing interface, similar to the one above. On the right side, there is a video window showing the same man, "harkirat singh", with a dark background and yellow curtains. In the bottom right corner of the video window, there are several emoji icons: two thumbs up, a smiling face, and a laughing face.

```
JS index.js > ...
2 const a = 1;
3 const b = 2;
4
5 console.log(a);
6 console.log(b);
7
8 function callback() {
9     console.log(a + b);
10 }
11
12 setTimeout(callback, 1000)
13
14
15
```

JS Architecture for async code

How JS executes asynchronous code - <http://latentflip.com/loupe/>