# I/O Heavy Operations

I/O (Input/Output) heavy operations refer to tasks in a computer 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.

We're going to introduce imports/requires next. A require statement lets you import code/functions export from another file/module.

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

1. Open replit
2. Create a file in there (a.txt) with some text inside.
3. Write a code to read a file Synchronously.

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

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

1. Create another file (b.txt)
2. 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);
```

fs is a node.js module that allows us to work with external files.
The Node.js file system module allows you to work with the file system on your computer.
To include the File System module, use the require() method:
var fs = require('fs');

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

There are two methods, we can use:
readFile() -------→ Asynchronous
readFileSync ---→ Synchronous

# 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 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 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 operation 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 task is 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.

# Doing I/O bound tasks in the real world:

What if you were tasked with doing 3 things:

1. Boil some water.
2. Do some laundry.
3. Send a package via mail.

Would you do these -

1. One by one (Synchronously) → Worst approach as it takes long time.
2. Context switch between them (concurrently) →Better approach – running task asynchronously .
3. Start all 3 tasks together, and wait for them to finish. The first one that finishes gets catered to first.

 Good talk- Concurrency is not parallelism.

. Synchronously (one by one)

```
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);
const content3 = fs.readFileSync("b.txt", "utf-8");
console.log(content3);
```

. Start all 3 tasks together, and wait for them to finish

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

fs.readFile("a.txt","utf-8", function(err,contents)
{
console.log(contents)
});

fs.readFile("b.txt","utf-8",function(err,
contents){
console.log(contents2);
});
fs.readFile("b.txt","utf-8",function(err,
contents){
console.log(contents2);
});
```

Functional Arguments:

```
function sum(a,b){
    return a+b
}
function divide(a,b){
    return a/b
}
function subtract(a,b){
    return a-b
}

function doOperation(x, y, op){     // op is a functional argument
     let val = op(x, y)
     return val;
}


const ans = doOperation(1, 2, divide);
console.log(ans);
```

## 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.

```
const fs = require(“fs”);

function afterFileRead(err, contents){
      console.log(contents);
}



fs.readFile(“a.txt”, “utf-8”, afterFileRead);
```