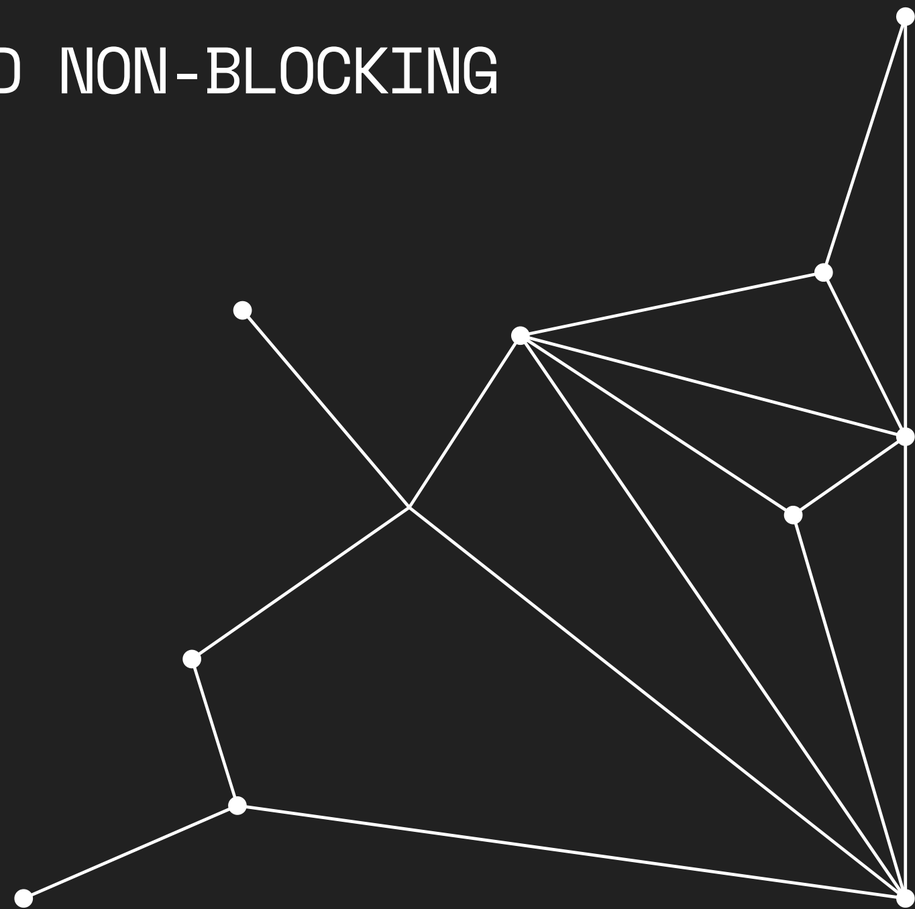
ECHO SERVER

BY : SAKSHAM KUMAR

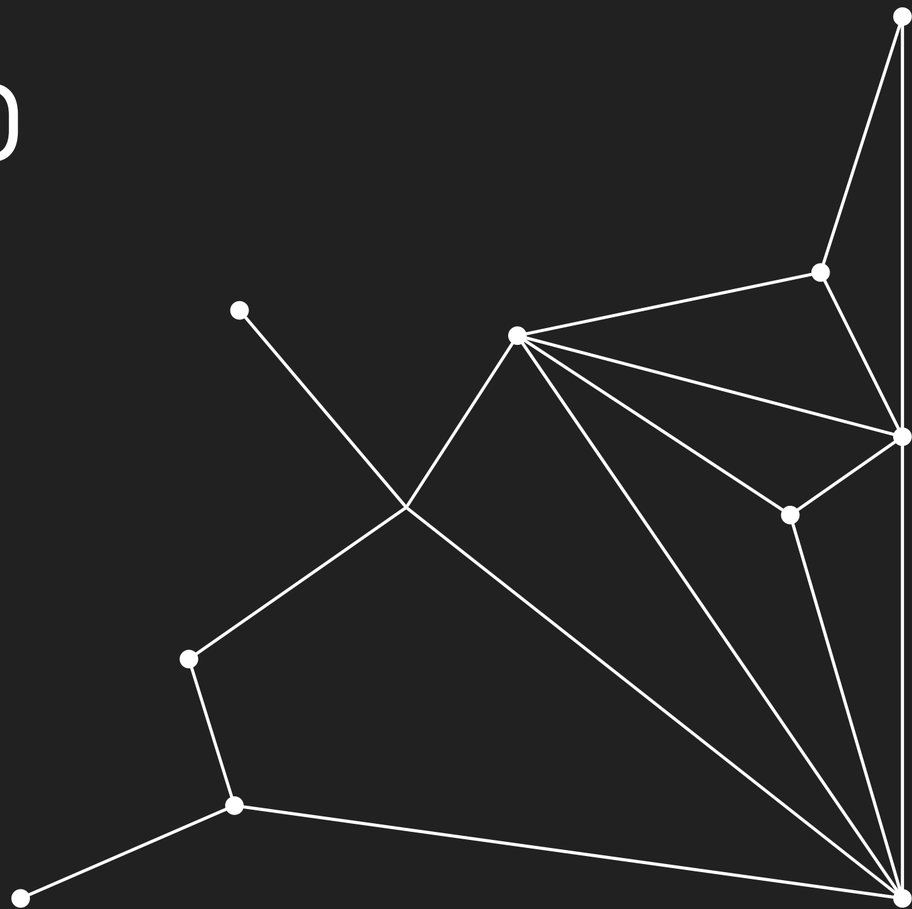
» ls Content

- DIFFERENT TYPES OF I/O : BLOCKING AND NON-BLOCKING
- EVENT DRIVEN I/O AND EVENT LOOP
- USE OF EVENT LOOP IN NODE JS



» cd Content/Types_Of_IO

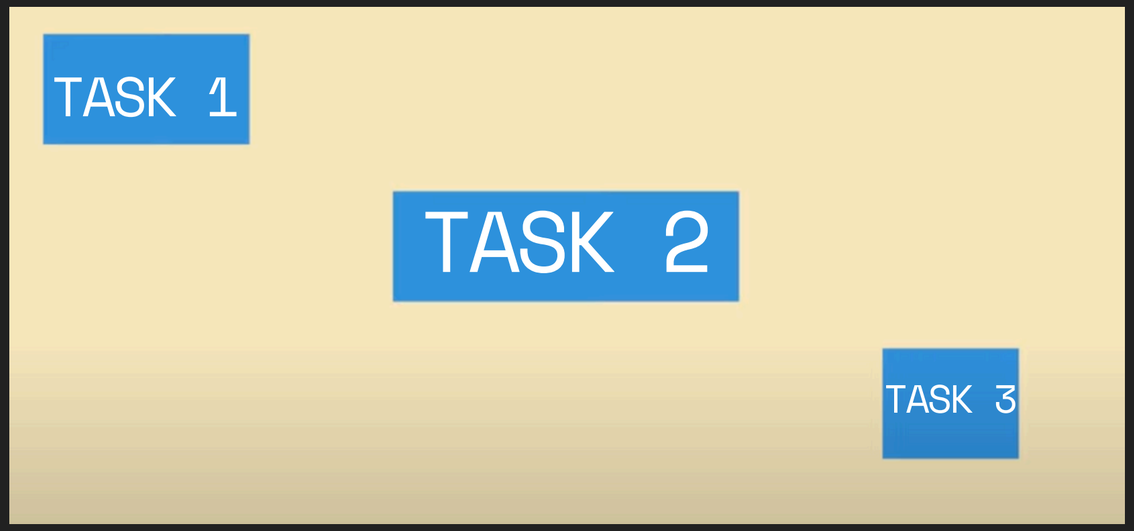
1. BLOCKING I/O
2. NON - BLOCKING I/0



- Blocking and non-blocking I/O refer to how input/output operations (like reading from a file, network, or console) are handled in a program.

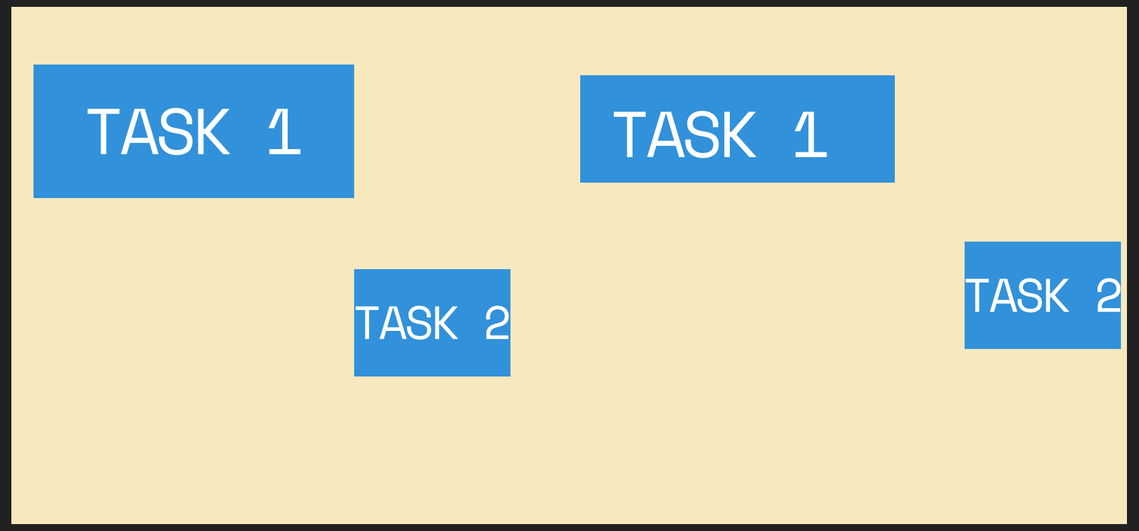
BLOCKING I/O

Execution is paused until the request is executed



NON - BLOCKING I/O

Program continues to execute other tasks while checking if the I/O operation is finished.



ANALOGY

In a restaurant, the waiter takes multiple orders without waiting for one to be completed before moving to the next table. **NON BLOCKING I/O** works in a similar way

Now, imagine a restaurant where a waiter takes one order at a time and does not serve anyone else until the current order is fully prepared. **BLOCKING I/O** works like this

A Good Server

- In technical terms, a server is a software, or device that provides services or data to other clients over a network

- There can be different type of servers - Web Servers, Database Servers, File Servers, SSH servers, etc

- A Good server must handle multiple client requests at the same time with less performance impact.

SINGLE THREAD VS MULTI THREADING

- In a process, a thread represents a unit of execution inside it.

1. One single thread is executed at a time.
2. Uses fewer system resources
3. Slower for complex tasks, might freeze if a task takes too long.
4. No concurrency.

1. Process executes multiples threads at a time
2. Uses more system resources
3. Can handle multiple requests at once.
4. Supports concurrency - multiple tasks run in parallel.

- Note that multi threading is not Non - Blocking I/O as tasks can be blocked if threads wait on resources

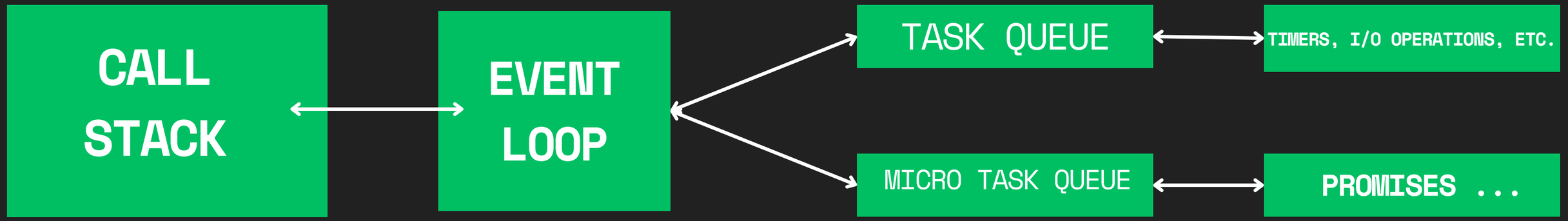
>> CAN A CONCURRENCY LIKE MULTITHREADING BE ACHIEVED WITH USING LESS RESOURCES ?

YES, Asynchronous I/O does that.

- It uses a single thread which switches between tasks without blocking.
- Concurrency is achieved via Non - Blocking I/O and **event loops**
- Uses less system resources as compared to multi-threading.

» cd ../Event_Driven_Architecture

- USED TO HANDLE MULTIPLE TASKS CONCURRENTLY USING A SINGLE THREAD



CALL STACK - It is a stack data structure (last in, first out) which keeps track of the function running and the functions to be executed next.

TASK QUEUE - It is a queue data structure (first in, last out). Functions regarding web APIs such as timers (eg, setTimeout(), etc.), I/O operations, etc.

MICRO TASK QUEUE - It is also a queue, but more prioritized than task queue. Functions regarding Promises (then(), catch(), finally()), the code after `await` keyword, queueMicroTask().

- When a promise or queueMicroTask() occurs in the call stack, it is pushed to micro tasks queue.
- While if a Web API function occurs in the call stack, it is pushed to tasks queue.
- If the call stack is empty, first the functions in the micro tasks queue are executed in the call stack.
- When the micro tasks queue gets empty, the functions in the tasks queue are executed in the call stack.
- In this way, the thread doesn’t have to wait for a function to finish, it instead can proceed to run other functions in the call stack.

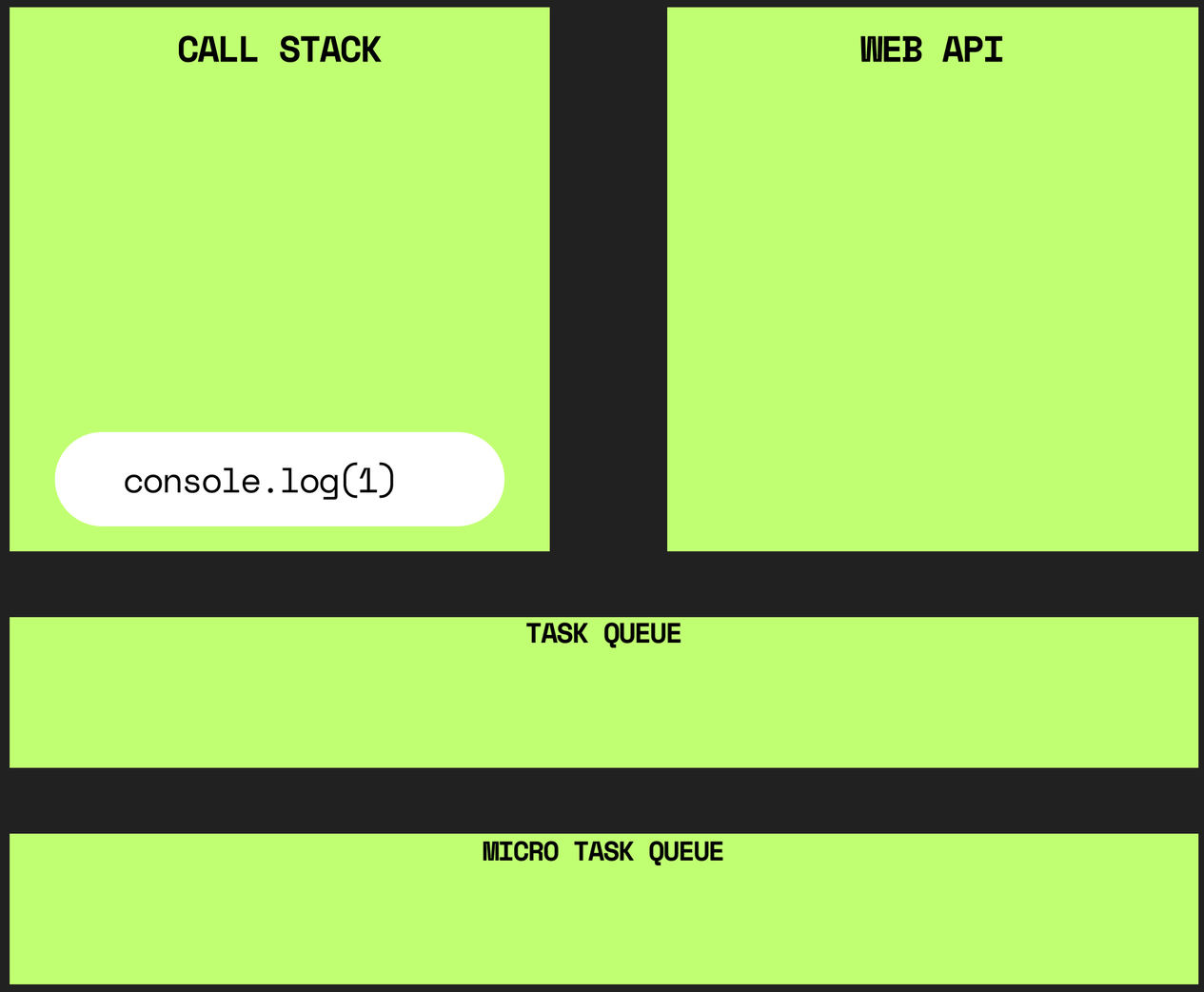
» cd ../Event_Loop_In_Node_js

```
console.log(1);

setTimeout(() => {
  console.log(4);
}, 100);

Promise.resolve()
    .then(() => {
      console.log(3);
    });

console.log(2);
```

INPUT

```
❯ node test.js
1
2
3
4
```

OUTPUT



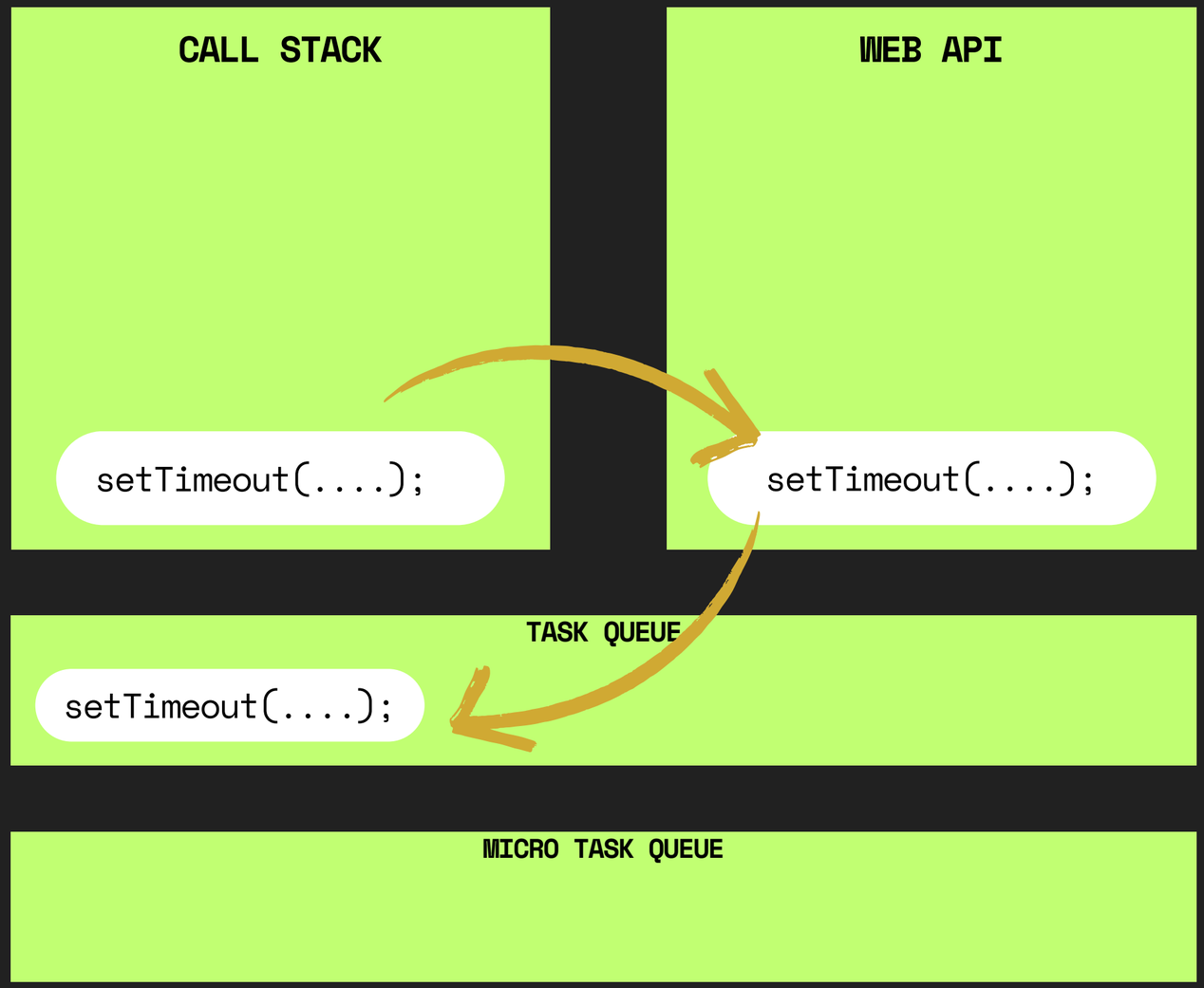
setTimeout is a web API function so, it is moved to Task Queue.

```
console.log(1);

setTimeout(() => {
  console.log(4);
}, 100);

Promise.resolve()
    .then(() => {
      console.log(3);
    });

console.log(2);
```

INPUT

```
❯ node test.js
1
2
3
4
```

OUTPUT



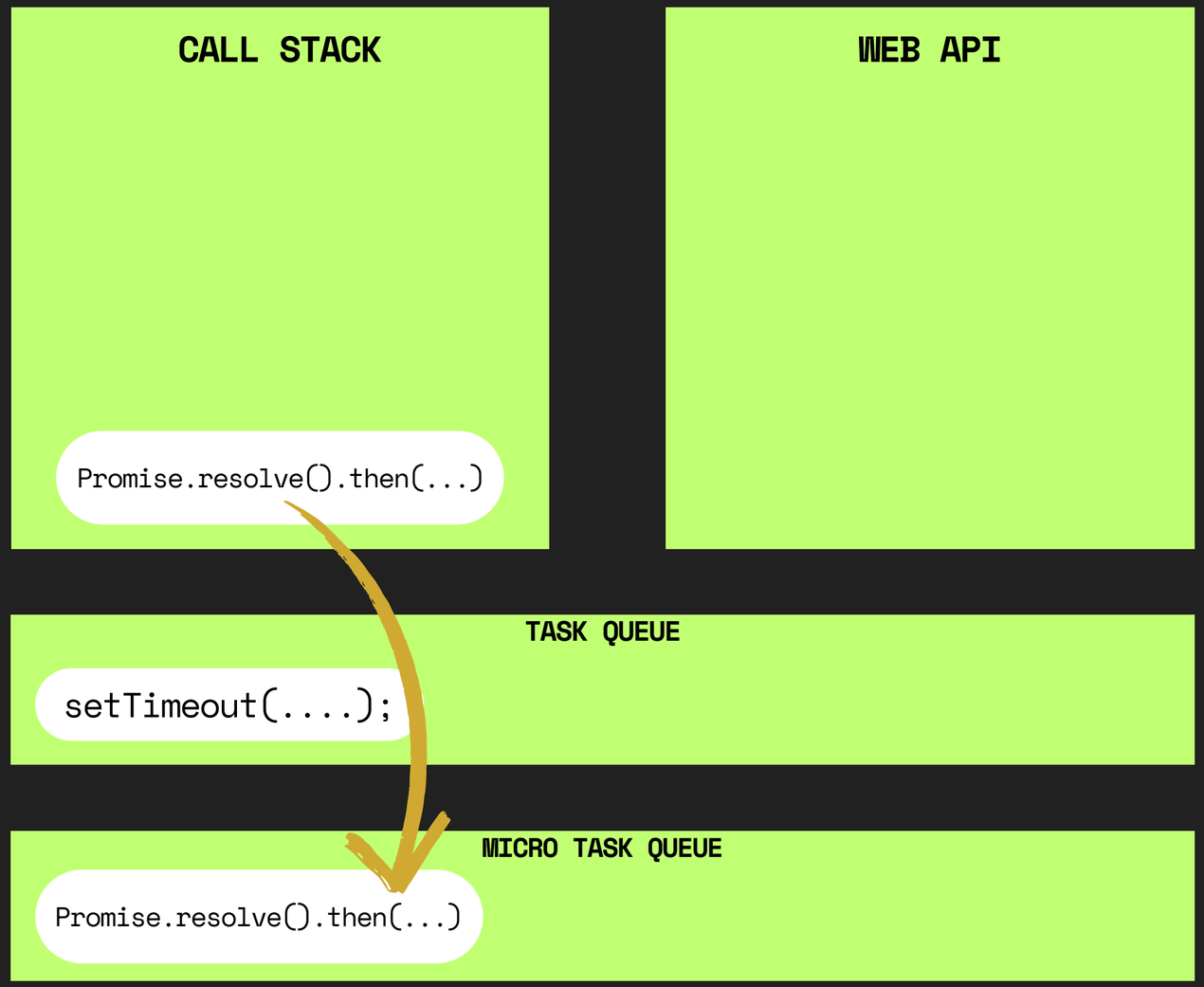
Promise is moved to Micro Task Queue (given higher priority than setTimeout)

```
console.log(1);

setTimeout(() => {
  console.log(4);
}, 100);

Promise.resolve()
    .then(() => {
      console.log(3);
    });

console.log(2);
```

INPUT

```
❯ node test.js
1
2
3
4
```

OUTPUT

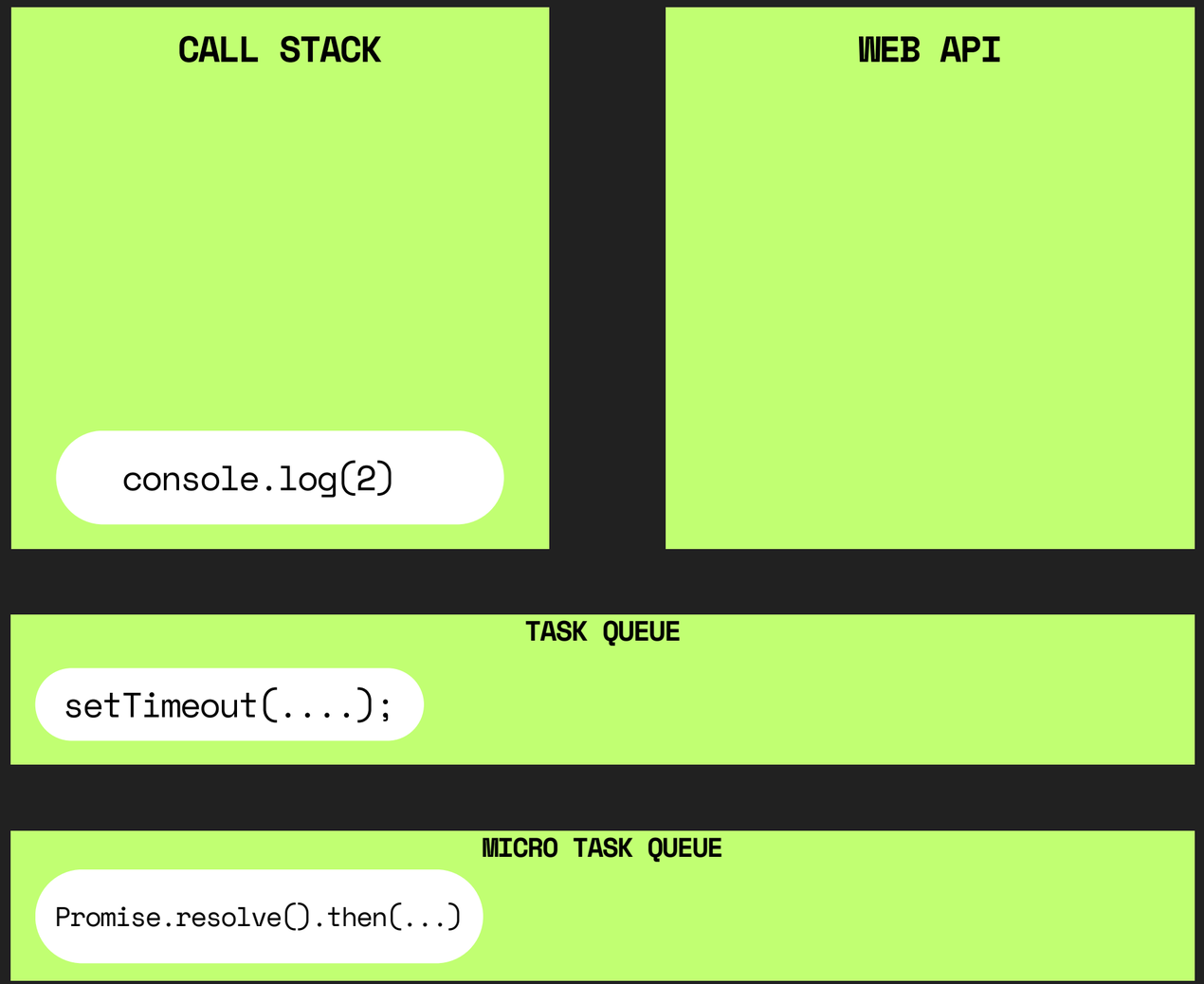


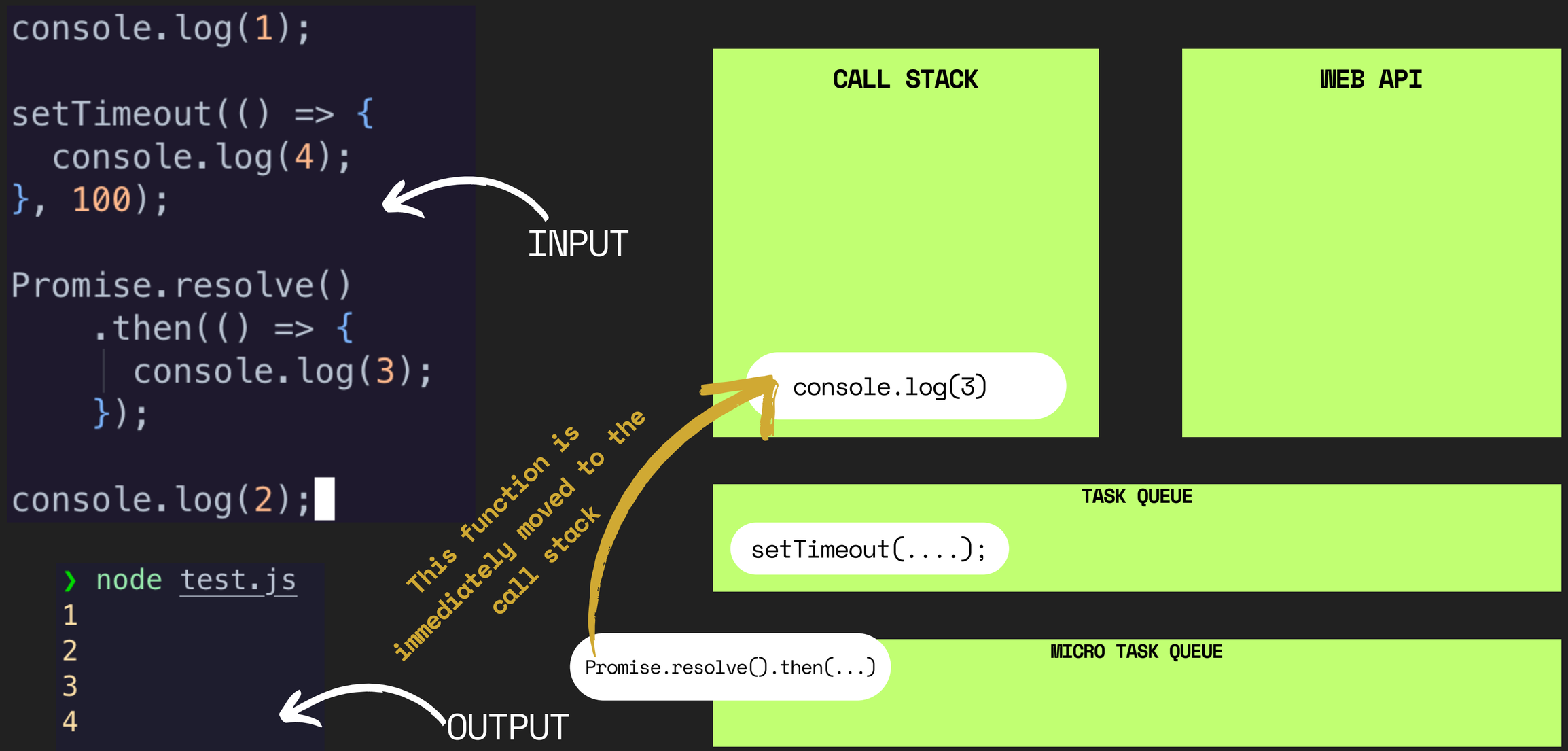
```
console.log(1);

setTimeout(() => {
  console.log(4);
}, 100);

Promise.resolve()
    .then(() => {
      console.log(3);
    });

console.log(2);
```

INPUT

```
❯ node test.js
1
2
3
4
```

OUTPUT



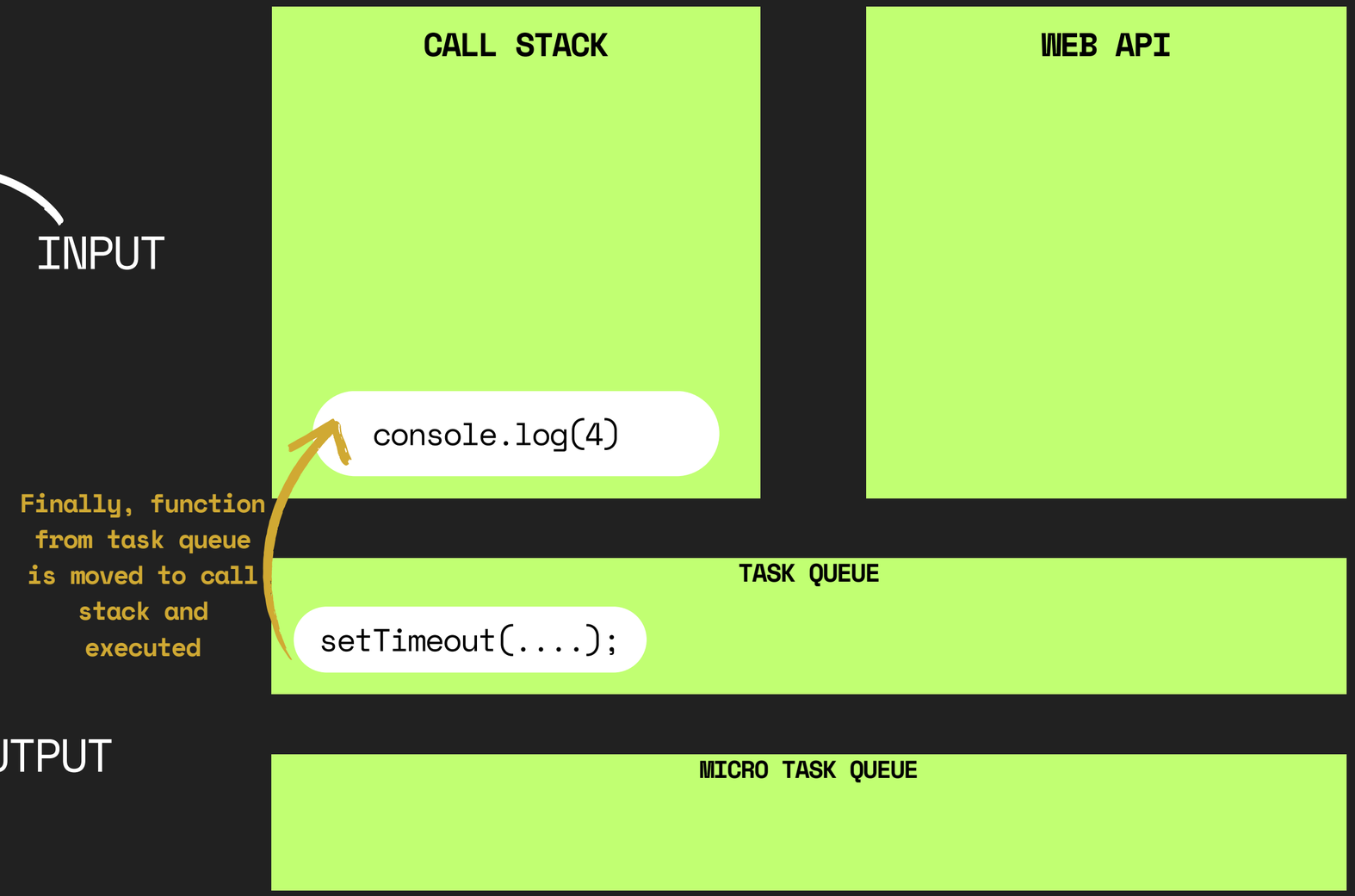

console.log(1);
setTimeout(() => {
console.log(4);
}, 100);
Promise.resolve()
.then(() => {
console.log(3);
});
console.log(2);
INPUT
CALL STACK
WEB API
console.log(3)
This function is immediately moved to the call stack
TASK QUEUE
setTimeout(....);
MICRO TASK QUEUE
Promise.resolve().then(...)
❯ node test.js
1
2
3
4
OUTPUT

```
console.log(1);

setTimeout(() => {
  console.log(4);
}, 100);

Promise.resolve()
    .then(() => {
      console.log(3);
    });

console.log(2);
```

INPUT

```
❯ node test.js
1
2
3
4
```

OUTPUT



» cd ../Challenges

1 » Predict_The_Output

```
1  // Challenge : Predict the output
2
3  console.log("Start Loop");
4
5  setTimeout(() => {
6    console.log("Checkpoint 1");
7  }, 2000);
8
9  setImmediate(() => {
10   console.log("Checkpoint 2");
11 })
12
13 process.nextTick(() => {
14   console.log("Checkpoint 3");
15 })
16
17 console.log("End Loop");
```

1 ›› Predict_The_Output

```
1  // Challenge : Predict the output
2
3  console.log("Start Loop");
4
5  setTimeout(() => {
6    console.log("Checkpoint 1");
7  }, 2000);
8
9  setImmediate(() => {
10   console.log("Checkpoint 2");
11 })
12
13 process.nextTick(() => {
14   console.log("Checkpoint 3");
15 })
16
17 console.log("End Loop");
```

```
❯ node challenge1.js
Start Loop
End Loop
Checkpoint 3
Checkpoint 2
Checkpoint 1
```

- setTimeout(..) being a Web API is executed at last.
- setIntemediate(..) belong to Macro tasks queue which are given more priority than Tasks queue and less priority than Micro Tasks queue
- process.nextTick(..) being in Micro tasks queue is executed first.

1 ›› Predict_The_Output

```
1  // Challenge : Predict the output
2
3  console.log("Start Loop");
4
5  setTimeout(() => {
6    console.log("Checkpoint 1");
7  }, 2000);
8
9  setImmediate(() => {
10   console.log("Checkpoint 2");
11 })
12
13 process.nextTick(() => {
14   console.log("Checkpoint 3");
15 })
16
17 console.log("End Loop");
```

```
❯ node challenge1.js
Start Loop
End Loop
Checkpoint 3
Checkpoint 2
Checkpoint 1
```

- setTimeout(..) being a Web API is executed at last.
- setIntemediate(..) belong to Macro tasks queue which are given more priority than Tasks queue and less priority than Micro Tasks queue
- process.nextTick(..) being in Micro tasks queue is executed first.

2 ›› Debug_Challenge

```
// Challenge : Debug the code.
// You are required to find out why "End of script" logs before "Timeout executed" and fix this.

const fs = require('fs');

fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error('Error reading file');
  }
});

setTimeout(() => {
  console.log('Timeout executed');
}, 0);

console.log('End of script');

```

```
❯ node challenge2.js
End of script
Timeout executed
```

2 ›› Debug_Challenge

```
// Challenge : Debug the code.
// You are required to find out why "End of script" logs before "Timeout executed" and fix this.

const fs = require('fs');

fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error('Error reading file');
  }
});

setTimeout(() => {
  console.log('Timeout executed');
}, 0);

console.log('End of script');

```

```
❯ node challenge2.js
End of script
Timeout executed
```

FIXED CODE :

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

async function read(){
  try{
    const data = await fs.readFile("file.txt");
    return data;
  }catch (err) {
    console.log("Error : ", err);
    return null;
  }
}

async function main(){

  // Handle with async operations
  await read();

  // wrapping timer in promise
  await new Promise(resolve => {
    setTimeout(() => {
      console.log("Timeout executed");
      resolve();
    }, 0);
  });

  console.log("End of script");
}
main();
```

```
❯ node challenge2.js
Timeout executed
End of script
```

2 ›› Fix_The_Code

```
1 // Challenge: Identify why setTimeout is delayed
2 //            depsite being set with a delay value 0.
3 //            And how to solve it.
4
5 function executeMe(){
6   const start = Date.now();
7   while(Date.now() - start < 2000) {}
8 }
9
10 executeMe();
11
12 setTimeout(() => {
13   console.log("Why am I being delayed TwT")
14 }, 0);
```

2 » Fix_The_Code

```
// Challenge: Identify why setTimeout is delayed
//            depsite being set with a delay value 0.
//            And how to solve it.

function executeMe(){
  const start = Date.now();
  while(Date.now() - start < 2000) {}
}

executeMe();

setTimeout(() => {
  console.log("Why am I being delayed TwT")
}, 0);
```

- Split blocking operation into chunks
- So, even after a 2 min dealy, timeout executes because the event loop is controlled.
- setIntermmediate queues chucks in the MacroTasks Queue, which prevents the blocking of other functions

Improved Code

```
async function executeMe(){
  const CHUNK = 100;
  let elapsed = 0;

  while(elapsed < 2000){
    await new Promise(resolve => {
      setImmediate(() => {
        const start = Date.now();
        while (Date.now() - start < CHUNK) {}
        elapsed += CHUNK;
        resolve();
      })
    })
  }

}

async function main(){
  executeMe();

  // Now the function can execute between chunks
  setTimeout(() => {
    console.log(`Yay :)`)
  }, 0);
}

main();
```

THANK

YOU