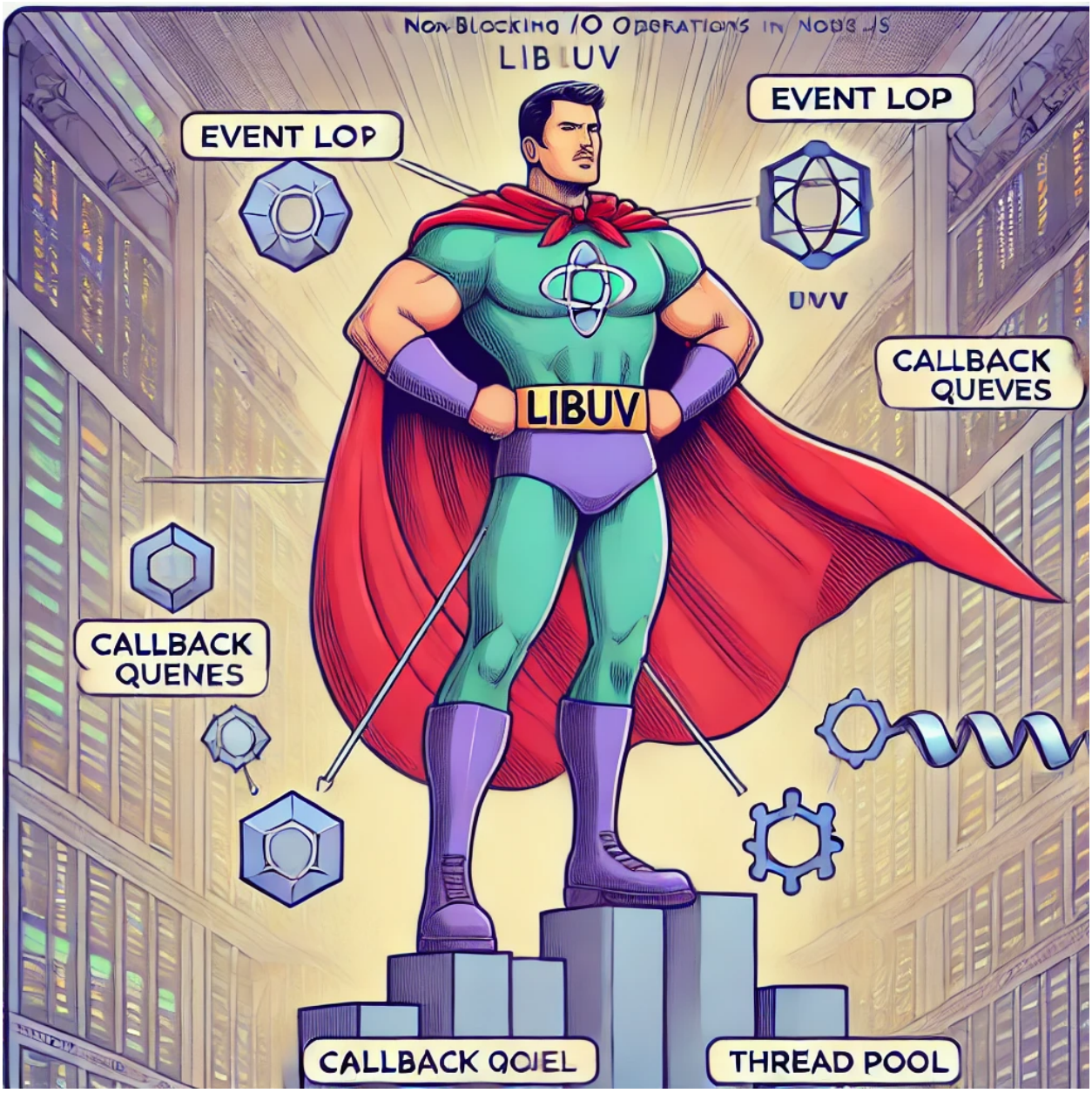
Episode-09 | libuv & Event Loop



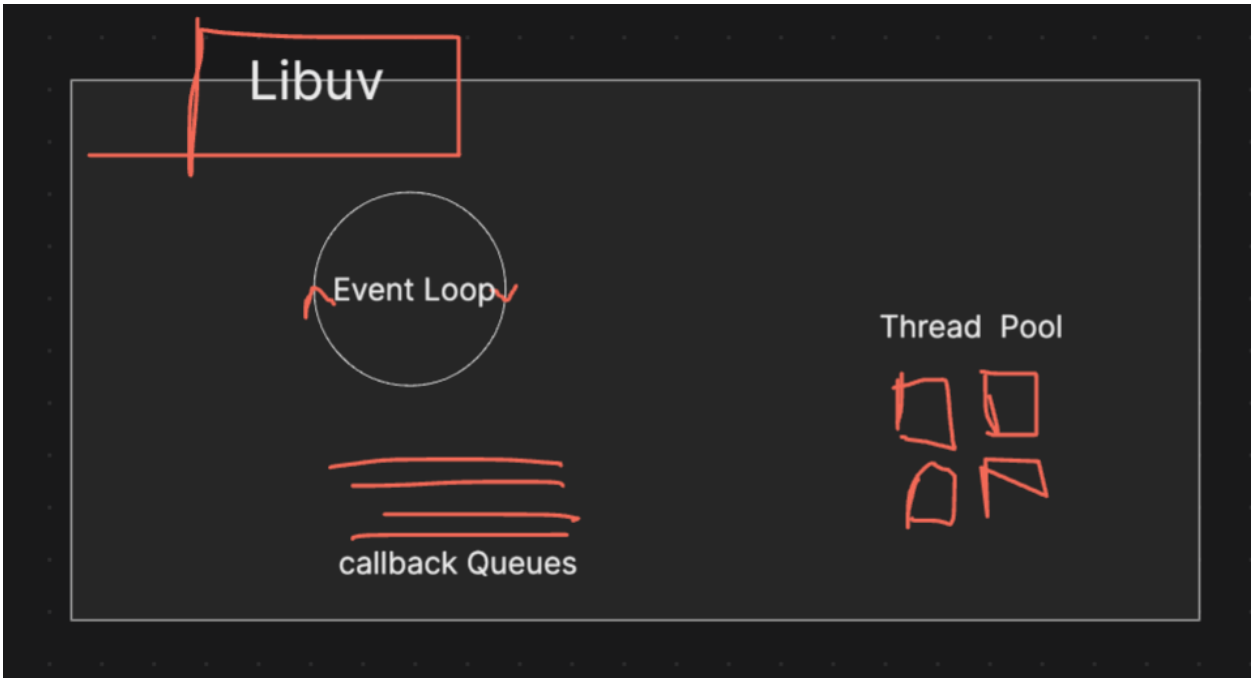
You're diving into the core of Node.js's asynchronous handling with `libuv`.

Understanding how `libuv` manages the event loop, callback queues, and thread pools is crucial, especially for non-blocking I/O tasks.

The **event loop** in `libuv` is the heart of how Node.js handles asynchronous operations. It allows Node.js to perform non-blocking I/O operations, even though JavaScript is single-threaded. Tasks that are offloaded to `libuv` include file system operations, DNS lookups, and network requests, among others.

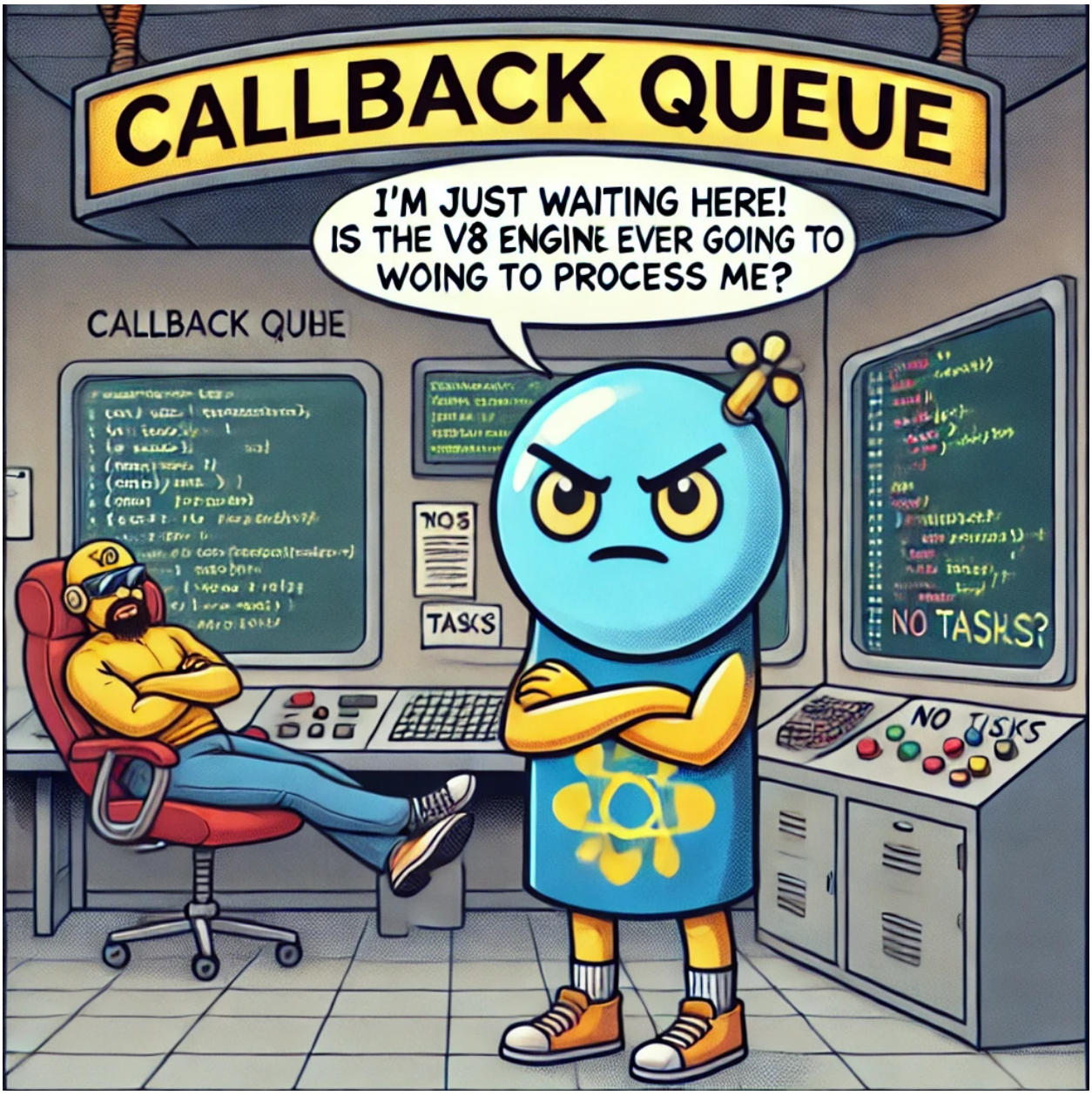
The **callback queue** is where callbacks are stored after an asynchronous operation is completed. The event loop processes this queue to execute the callbacks when the call stack is empty.

The **thread pool**, on the other hand, is used for handling more time-consuming tasks that cannot be handled within the event loop without blocking it, such as file system operations or cryptographic functions.



When a task is offloaded to `libuv`, `libuv` internally performs several operations. For instance, if you initiate a file read operation, once the data is received back from the operating system (OS), it is `libuv`'s responsibility to handle the callback function and eventually send it to the call stack for execution.

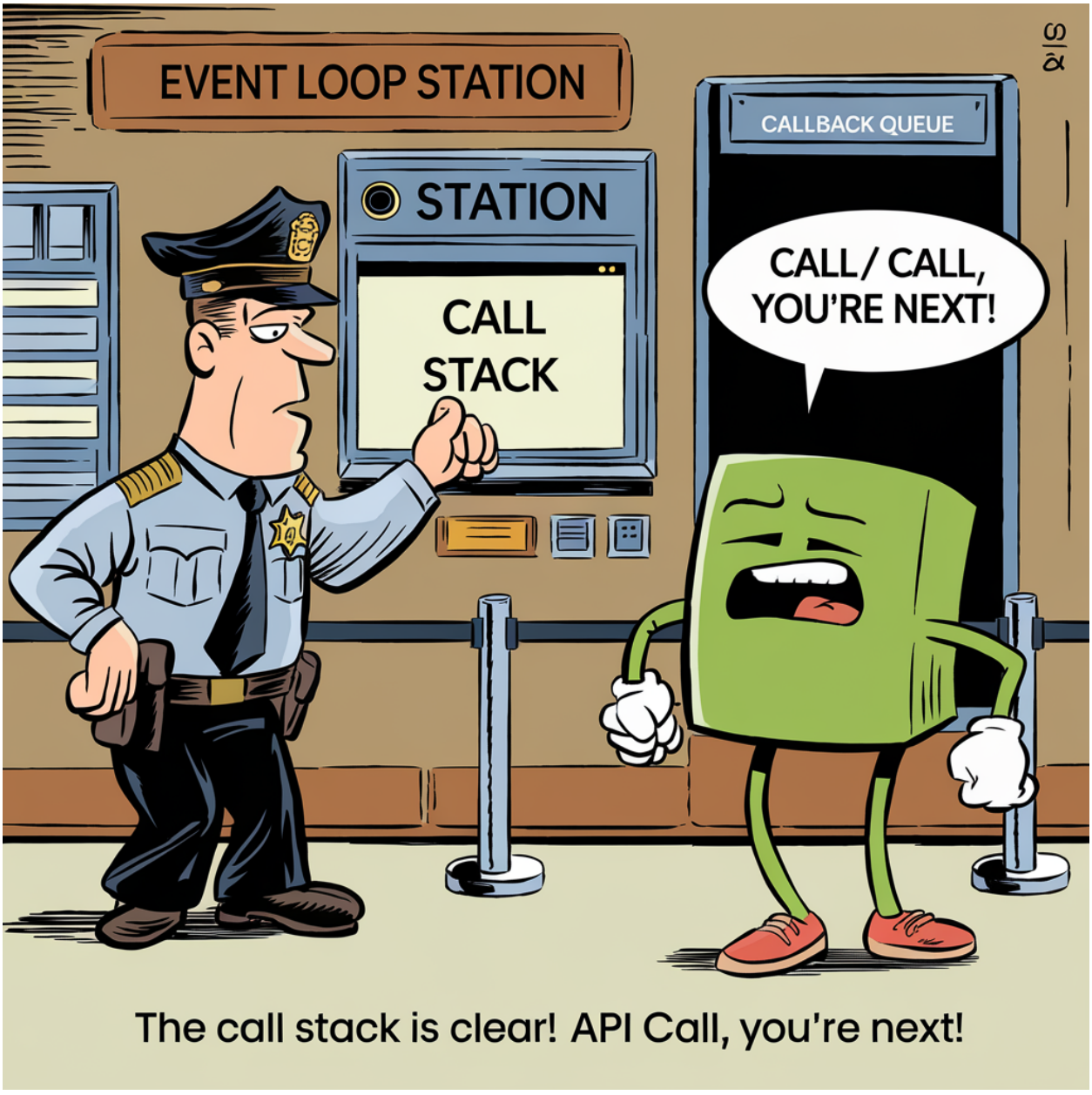
Now, imagine there are millions of lines of JavaScript code running within the JavaScript engine. If an asynchronous task like an API call returns data very quickly from the OS to `libuv`, that API call must wait in the callback queue within `libuv` until the V8 engine is free to process it. This ensures that the non-blocking nature of Node.js is maintained, as the V8 engine can continue executing other code without being blocked by these asynchronous operations.



So, let's say multiple asynchronous tasks, like an API call returning results, a `setTimeout`, and a file read operation, are completed simultaneously. To manage this, `libuv` maintains separate callback queues for different types of tasks, such as timers, API calls, and file reads.

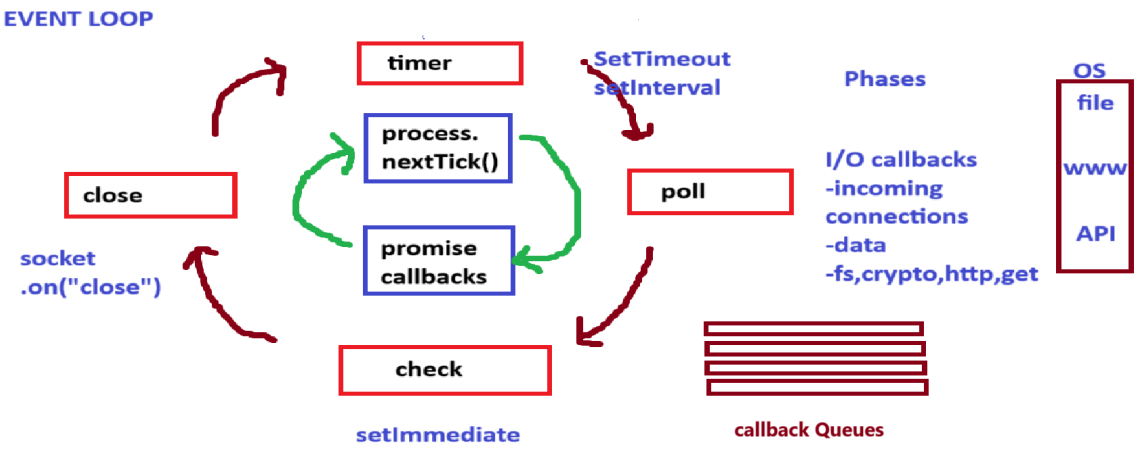
This is where the event loop comes into play. The event loop continuously monitors the call stack, checking if it's empty. If the stack is empty, the event loop

takes tasks from the callback queues and pushes them onto the call stack for execution.



The event loop's main responsibility is to ensure that all pending tasks in the callback queues are executed at the appropriate time and in the correct order of priority. But how does the event loop prioritize these tasks? How does it manage the timing and efficient execution of tasks behind the scenes?

Let's take a closer look at what happens internally inside the event loop



The event loop in LIBUV operates in four major phases:

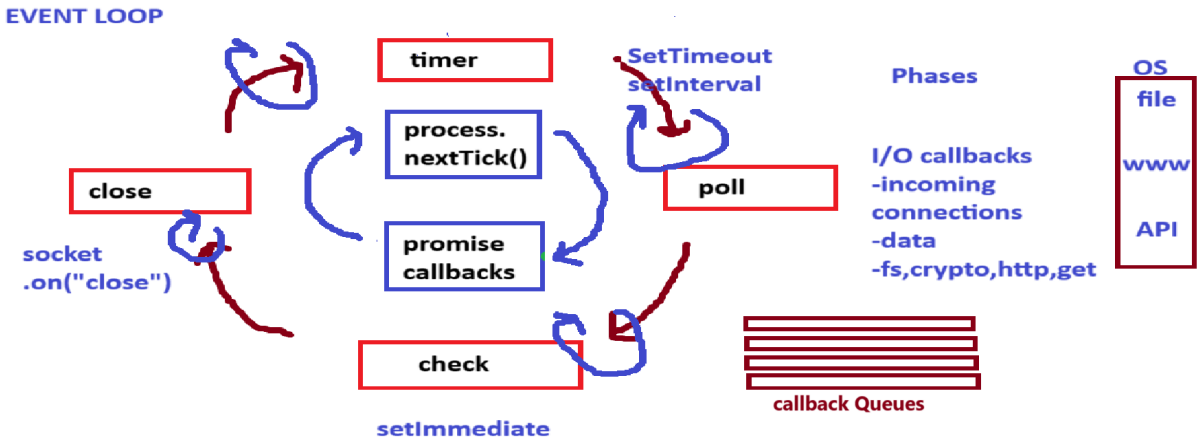
1. **Timers Phase**: In this phase, all callbacks that were set using `setTimeout` or `setInterval` are executed. These timers are checked, and if their time has expired, their corresponding callbacks are added to the callback queue for execution.
2. **Poll Phase**: After timers, the event loop enters the Poll phase, which is crucial because it handles I/O callbacks. For instance, when you perform a file read operation using `fs.readFile`, the callback associated with this I/O operation will be executed in this phase. The Poll phase is responsible for handling all I/O-related tasks, making it one of the most important phases in the event loop.
3. **Check Phase**: Next is the Check phase, where callbacks scheduled by the `setImmediate` function are executed. This utility API allows you to execute callbacks immediately after the Poll phase, giving you more control over the order of operations.
4. **Close Callbacks Phase**: Finally, in the Close Callbacks phase, any callbacks associated with closing operations, such as socket closures, are handled. This phase is typically used for cleanup tasks, ensuring that resources are properly released.

Event Loop Cycle with `process.nextTick()` and Promises [very important]

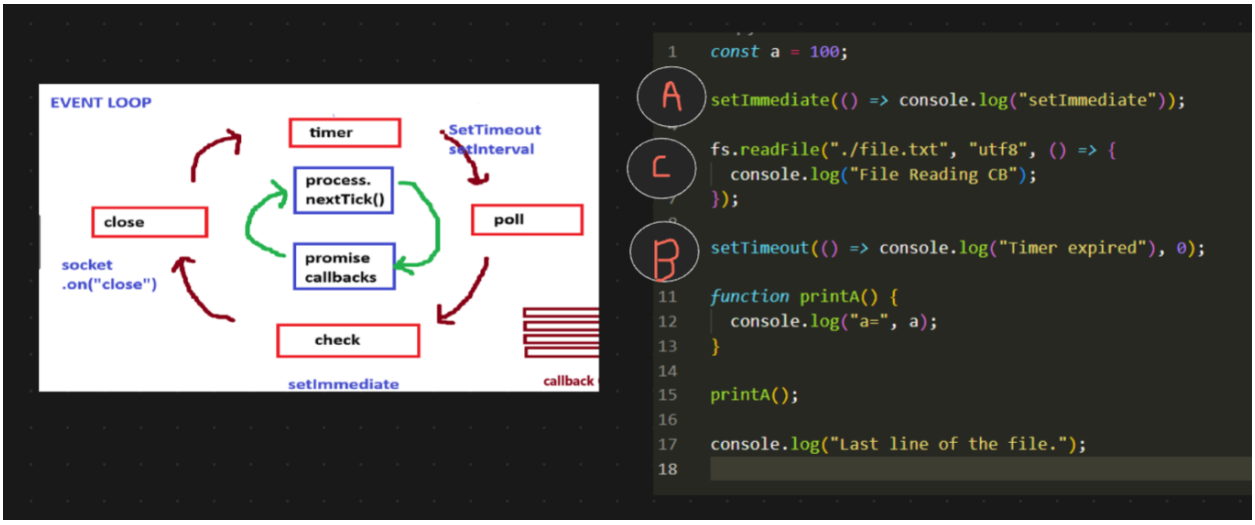
Before the event loop moves to each of its main phases (Timers, I/O Callbacks,

Poll, Check, and Close Callbacks), it first processes any pending microtasks. Microtasks include tasks scheduled using `process.nextTick()` and Promise callbacks. This ensures that these tasks are handled promptly before moving on to the next phase.

1. Start Event Loop Cycle
2. Process `process.nextTick()` Callbacks
   - Execute all `process.nextTick()` callbacks
3. Process Promise Callbacks
   - Execute all Promise callbacks (microtasks)
4. Move to Timers Phase
   - Execute timer callbacks
5. Move to I/O Callbacks Phase
   - Execute I/O callbacks
6. Move to Poll Phase
   - Execute I/O events
7. Move to Check Phase
   - Execute `setImmediate` callbacks
8. Move to Close Callbacks Phase
   - Execute close callbacks
9. Repeat Event Loop Cycle



Q 1 : What is the output?



The code begins by declaring a constant `a` with a value of 100:

```
const a = 100;
```

Next, the `setImmediate` function is encountered. This function is asynchronous, so the V8 engine offloads it to the `libuv` library. The associated callback function (let's call it **A**) is placed into the `setImmediate` callback queue, where it waits for execution because the V8 engine is currently busy.

The code then encounters the `fs.readFile` function. `libuv` handles this operation by calling the OS to start reading the file. Meanwhile, V8 continues processing the remaining code.

Moving on, the code encounters the `setTimeout` function with callback function **B** and a delay of 0 milliseconds. Function **B** is added to the timers queue, where it waits for execution, as V8 is still busy with the current task.

The function `printA()` is called next, which outputs `a = 100` to the console:

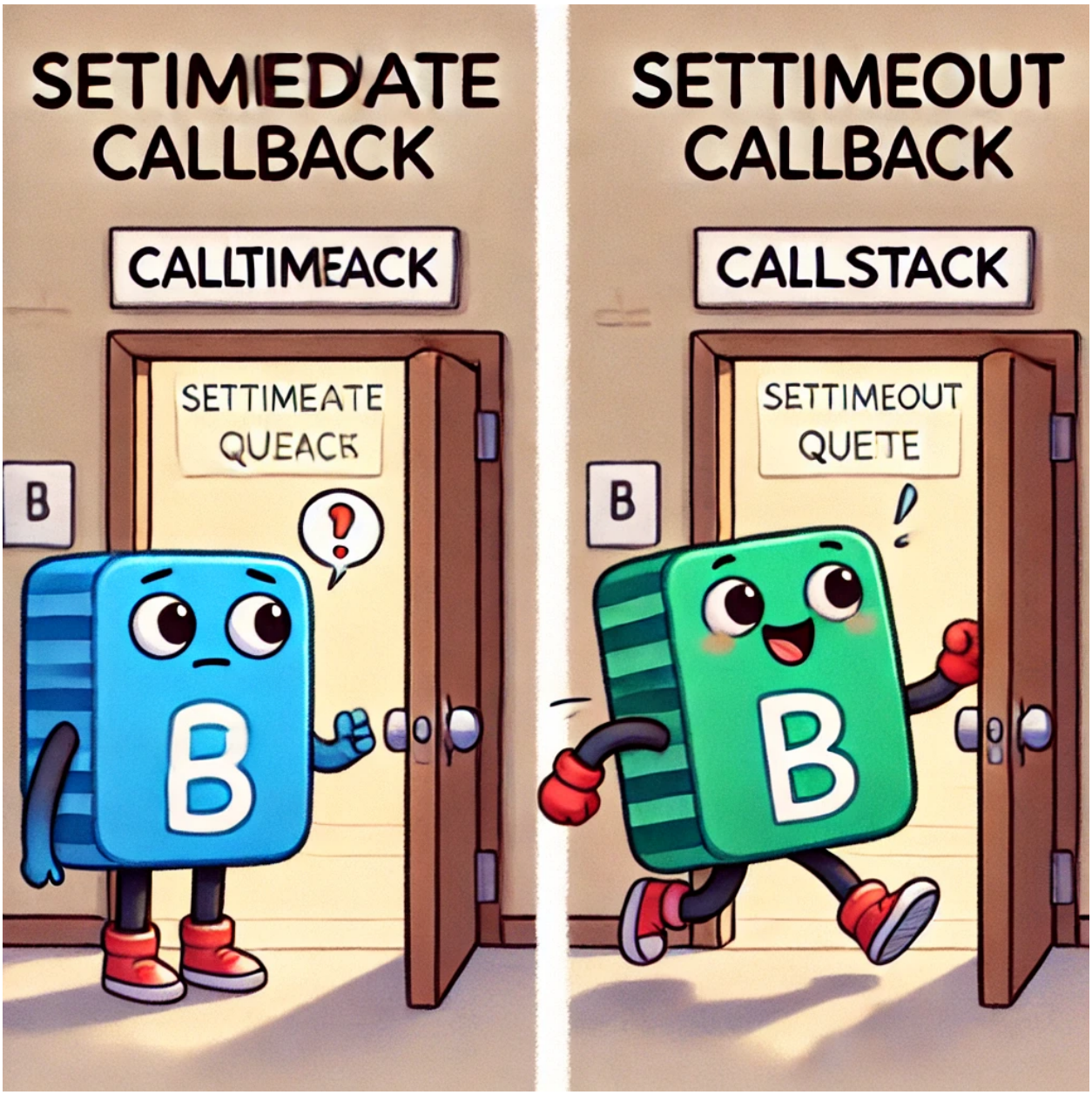
```
function printA() {
  console.log("a=", a);
}
printA();
```

Following this, the console logs "Last line of the file":

```
console.log("Last line of the file.");
```

At this point, all synchronous code has been executed, and the call stack is now empty.

Now, the event loop begins its cycle. It first checks for any pending `process.nextTick` callbacks, but since there are none, it moves to the timers phase. Here, it finds **B** in the timers queue and executes it, printing "Timer expired" to the console. **B** is then removed from the call stack.



Next, the event loop moves to the poll phase. Finding nothing in the poll queue, it proceeds to the check phase, where **A** is waiting. The event loop then pushes **A** to the call stack, and it is executed, printing "setImmediate" to the console.

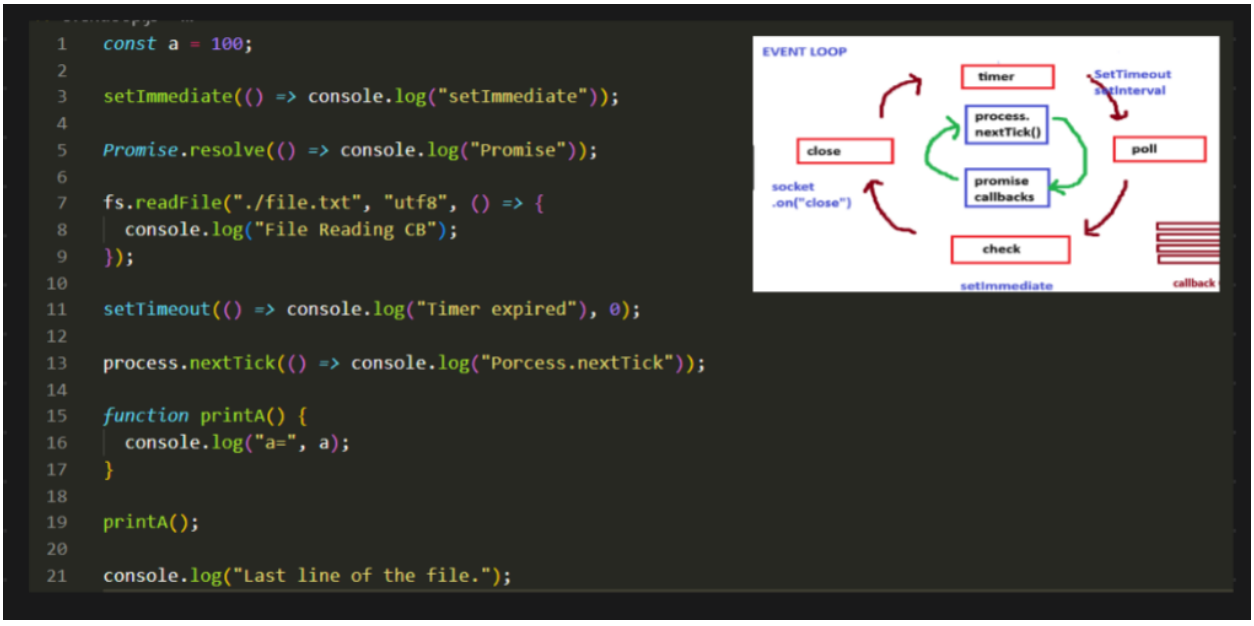
Meanwhile, `libuv` finishes reading the file, and the associated callback function **C** waits in the poll queue. The event loop eventually picks up **C**, executes it, and "File Reading CB" is printed to the console. **C** is then removed from the call stack, leaving the stack empty once more.

Thus, the final output of the code is:

```
a = 100
Last line of the file.
Timer expired
setImmediate
File Reading CB
```

This demonstrates how asynchronous tasks are managed in Node.js, with the event loop ensuring they are executed at the appropriate time.

Q2: What is the output?



The code starts by importing the `fs` module and declaring a constant `a` with a value of 100:

```
const fs = require("fs");
const a = 100;
```

Next, the `setImmediate` function is encountered. This function is asynchronous, so the V8 engine offloads it to the `libuv` library. The associated callback function (let's call it **A**) is placed into the `setImmediate` callback queue, where it waits for execution.

The code then processes a `Promise.resolve` function with a resolved value of `"promise"`. The `.then` method is used to log this value. Since promises are handled as microtasks, the `.then` callback is placed in the microtask queue and will be executed after the current synchronous code and `process.nextTick` callbacks.

Following this, the `fs.readFile` function is encountered. `libuv` handles this operation by calling the OS to start reading the file asynchronously. The associated callback function (let's call it **B**) will be placed into the poll queue once the file read operation is complete.

The `setTimeout` function is encountered next, with a delay of 0 milliseconds and an associated callback function **C**. This function is placed into the timers queue, where it will wait for execution once the V8 engine is free.

Then, the `process.nextTick` function is processed. `process.nextTick` callbacks are also microtasks and are executed before Promises in the microtask queue. The callback function (let's call it **D**) is placed into the microtask queue and will be executed soon after the current execution context is complete.

Next, the `printA()` function is called, which outputs `a = 100` to the console:

```
function printA() {
  console.log("a=", a);
}
printA();
```

Following this, the console logs "Last line of the file":

```
console.log("Last line of the file.");
```

At this point, all synchronous code has been executed, and the call stack is now empty.

The event loop begins its cycle by processing the microtasks. The first microtask to be executed is the `process.nextTick` callback **D**, which prints "Process.nextTick" to the console. Next, the `Promise` callback **E** logs the resolved value `"promise"` to the console.

The event loop then moves on to the timers phase, where it finds the `setTimeout` callback **C** waiting in the timers queue. It executes **C**, printing "Timer expired" to the console. **C** is then removed from the call stack.

Next, the event loop moves to the poll phase. If any I/O callbacks were ready, they would be executed here. The file read operation is complete, so the associated callback function **B** is executed, printing "File Reading CB" to the console. **B** is then removed from the call stack.

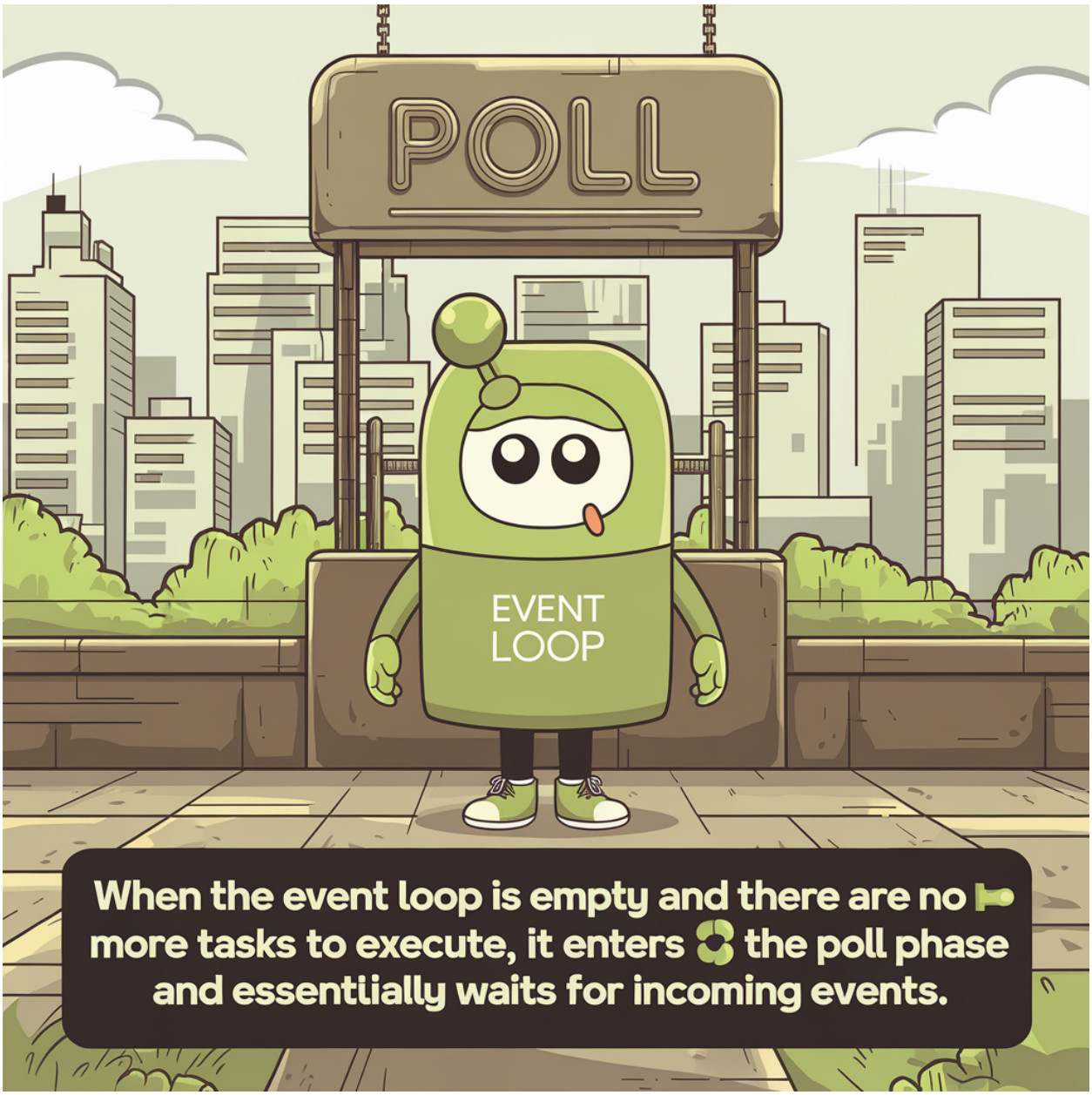
Finally, the event loop moves to the check phase, where it finds the `setImmediate` callback **A** waiting. It executes **A**, printing "setImmediate" to the console.

Thus, the final output of the code is:

```
a = 100
Last line of the file.
```

```
Process.nextTick
promise
Timer expired
setImmediate
File Reading CB
```

ONE MORE IMP CONCEPT



When the event loop is empty and there are no more tasks to execute, it enters the **poll phase** and essentially waits for incoming events

Q3: What is the output?

```js
setImmediate(() => console.log("setImmediate"));

setTimeout(() => console.log("Timer expired"), 0);

Promise.resolve("promise").then(console.log);

fs.readFile("./file.txt", "utf8", () => {
  setTimeout(() => console.log("2nd timer"), 0);

  process.nextTick(() => console.log("2nd nextTick"));

  setImmediate(() => console.log("2nd setImmediate"));

  console.log("File reading CB");
});

process.nextTick(() => console.log("Porcess.nextTick"));

console.log("Last line of the file.");

```

1. **Synchronous Code Execution:**
   - `const fs = require("fs");` — Imports the `fs` module.
   - `setImmediate(() => console.log("setImmediate"));` — Schedules callback **A**.
   - `setTimeout(() => console.log("Timer expired"), 0);` — Schedules callback **B**.
   - `Promise.resolve("promise").then(console.log);` — Schedules Promise callback.
   - `fs.readFile(...)` — Starts asynchronous file read operation. Inside the callback, the following occurs:
     - `setTimeout(() => console.log("2nd timer"), 0);` — Schedules callback **C**.
     - `process.nextTick(() => console.log("2nd nextTick"));` — Schedules callback **D**.
     - `setImmediate(() => console.log("2nd setImmediate"));` — Schedules callback **E**.
     - `console.log("File reading CB");` — Logs `"File reading CB"`.
   - `process.nextTick(() => console.log("Porcess.nextTick"));` — Schedules callback **F**.
   - `console.log("Last line of the file.");` — Logs `"Last line of the file."`.
2. **Microtasks:**

- The microtasks queue is processed first, so callbacks **F** (from `process.nextTick`) and the Promise callback are executed in this order.
    - `"Porcess.nextTick"` is logged.
    - `"promise"` is logged from the Promise callback.

3. **Timers Phase:**
    - The event loop processes the timers queue next, executing callback **B** (from `setTimeout` with 0 milliseconds delay).
        - `"Timer expired"` is logged.
4. `setImmediate` **Phase:**
    - The event loop then processes the `setImmediate` callbacks. Both **A** and **E** are executed:
        - `"setImmediate"` is logged.
        - `"2nd setImmediate"` is logged.
5. **Poll Phase:**
    - During the poll phase, any remaining I/O callbacks are processed. In this case, callback **C** (from `fs.readFile`) is executed, which includes:
        - Logging `"2nd timer"` from callback **C**.
        - Logging `"File reading CB"` from the `fs.readFile` callback.

Final Output:

The final output of the code is:

```
Last line of the file.
Porcess.nextTick
promise
Timer expired
setImmediate
2nd setImmediate
```

```
2nd timer
File reading CB
```

Q4: What is the output?

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

setImmediate(() => console.log("setImmediate"));

setTimeout(() => console.log("Timer expired"), 0);

Promise.resolve("promise").then(console.log);

fs.readFile("./file.txt", "utf8", () => {
  console.log("File reading CB");
});

process.nextTick(() => {
  process.nextTick(() => console.log("inner nextTick"));
  console.log("Porcess.nextTick");
});

console.log("Last line of the file.");
```

`process.nextTick` callbacks have a higher priority than other asynchronous operations. This means that if you have nested `process.nextTick` callbacks, the inner `process.nextTick` callback will be executed before the outer one.

A:

The code begins by importing the `fs` module:

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

Next, it schedules several asynchronous tasks:

1. `setImmediate`: This function schedules a callback (let's call it **A**) to be executed in the next iteration of the event loop. The callback prints "setImmediate" to the console.

2. `setTimeout`: This function schedules a callback (let's call it **B**) to be executed after a delay of 0 milliseconds. The callback prints "Timer expired" to the console.
3. `Promise.resolve`: This creates a promise that is immediately resolved with the value "promise". The `.then` method schedules a callback (let's call it **C**) to be executed once the promise is resolved, which prints the resolved value to the console.
4. `fs.readFile`: This function initiates an asynchronous file read operation. Once the file is read, its callback (let's call it **D**) prints "File reading CB" to the console.
5. `process.nextTick`: This function schedules a callback (let's call it **E**) to be executed on the next iteration of the event loop, before any I/O tasks. The callback itself schedules another `process.nextTick` callback (let's call it **F**) and prints "Process.nextTick" to the console. The second `process.nextTick` callback prints "inner nextTick" to the console.
6. `console.log("Last line of the file.")`: This line of code prints "Last line of the file." to the console.

Execution Flow

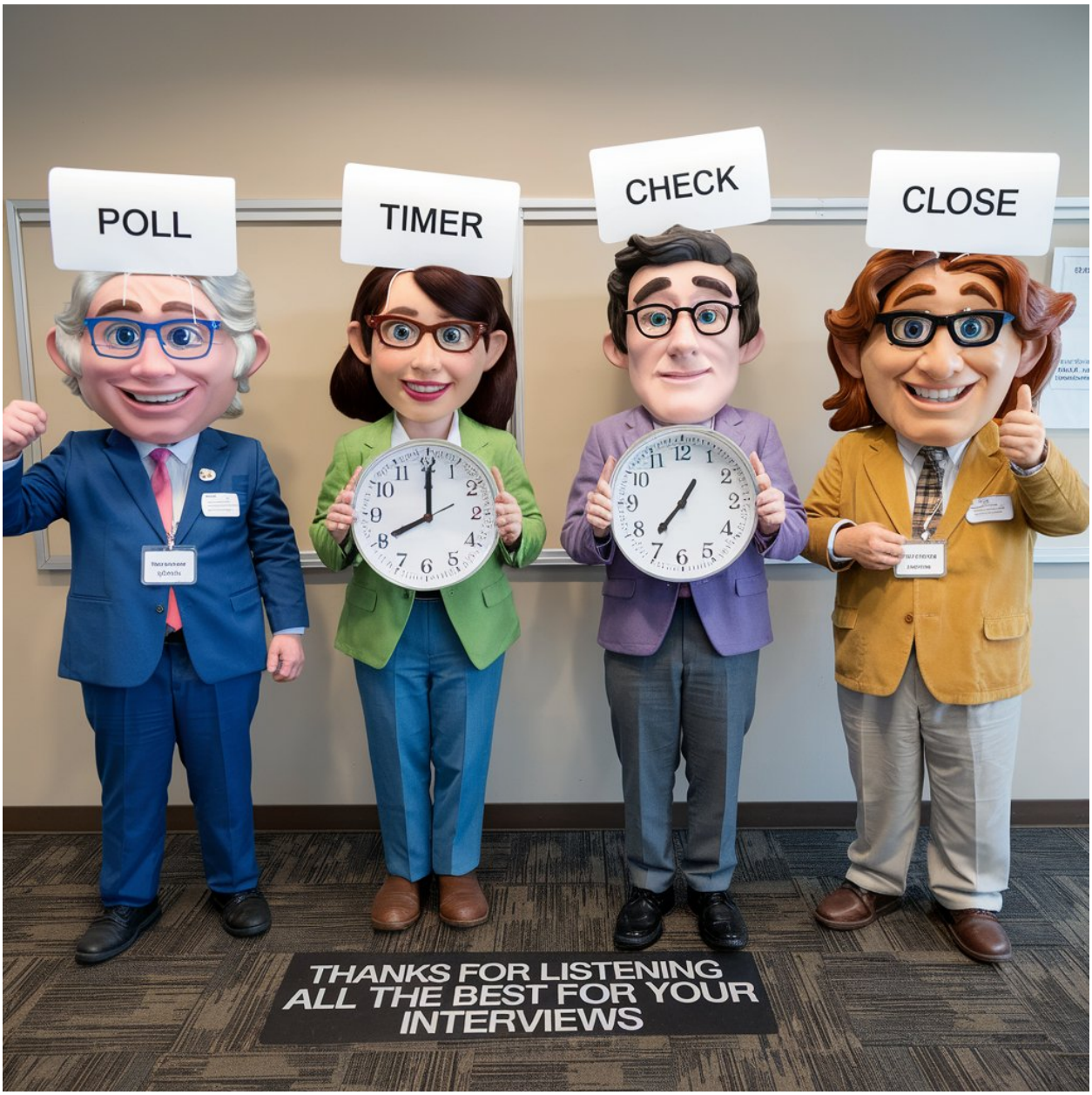
1. **Synchronous Code Execution**:
   - The constant `fs` is assigned the `fs` module.
   - The `setImmediate`, `setTimeout`, and `Promise.resolve` functions are scheduled.
   - `process.nextTick` schedules its callbacks.
   - `console.log("Last line of the file.")` is executed and printed to the console.
2. **Event Loop Cycle**:
   - **Microtasks Phase**:
     - The event loop first processes `process.nextTick` callbacks. It executes **E**, which schedules **F** and prints "Process.nextTick". Then, **F** is executed, printing "inner nextTick".
     - Next, the promise callback (**C**) is executed, printing "promise".

- **Timers Phase**:
  - The event loop moves to the timers phase and executes the `setTimeout` callback (**B**), printing "Timer expired".
- **Poll Phase**:
  - In the poll phase, the event loop handles I/O operations. It finds that the file read operation is complete and executes the `fs.readFile` callback (**D**), printing "File reading CB".
- **Check Phase**:
  - The event loop then moves to the check phase and executes the `setImmediate` callback (**A**), printing "setImmediate".

Final Output:

The final output of the code is:

```
Last line of the file.
Process.nextTick
inner nextTick
promise
Timer expired
setImmediate
File reading CB
```


POLL
TIMER
CHECK
CLOSE
THANKS FOR LISTENING
ALL THE BEST FOR YOUR
INTERVIEWS