Non-Blocking I/O in Node.js

1. What is Non-Blocking I/O?

- **Non-blocking I/O** means that operations (like reading a file, making a network request, or querying a database) **do not block** the execution of other code.
- Instead of waiting for an operation to complete, Node.js **delegates** the task and moves to the next operation.
- When the operation is done, Node.js uses **callbacks, promises, or async/await** to handle the result.

---

2. How Node.js Handles Non-Blocking I/O?

Node.js is **single-threaded** and uses an **event loop** to handle I/O operations asynchronously.

Event Loop & Asynchronous Callbacks

1. **I/O operation starts** (e.g., reading a file).
2. Instead of waiting, Node.js **delegates** the task to the OS.
3. The **event loop** continues executing other tasks.
4. When the I/O operation completes, **callback functions** are executed.

---

3. Example: Blocking vs. Non-Blocking Code

Blocking (Synchronous) Code

```
const fs = require('fs');
console.log('Start');
const data = fs.readFileSync('file.txt', 'utf8');  // Blocks execution
console.log(data);
console.log('End');
//  Output (Execution Order):
// Start
// (file contents)
// End
```

The program **waits** for the file to be read before moving on.

Non-Blocking (Asynchronous) Code

```
const fs = require('fs');
console.log('Start');
fs.readFile('file.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
});
console.log('End');
// ✅ Output (Execution Order):
// Start
// End
// (file contents)
```

The file is read **asynchronously**, and execution continues without waiting.

---

4. How Non-Blocking I/O Works with APIs

Node.js **delegates** tasks like:

- **File System Operations** (fs.readFile, fs.writeFile)
- **Database Queries** (MongoDB, MySQL)
- **Network Requests** (http, axios, fetch)
- **Timers** (setTimeout, setInterval)

Example: **Asynchronous HTTP request**

```
const https = require('https');
console.log('Start');
https.get('https://jsonplaceholder.typicode.com/todos/1', (res) => {
    let data = '';
    res.on('data', chunk => data += chunk);
    res.on('end', () => console.log(JSON.parse(data)));
});
console.log('End');
// ✅ Execution Order:
// Start
// End
// { JSON response from API }
```

The HTTP request is **asynchronous**, so console.log('End') executes before the response arrives.

---

5. Handling Non-Blocking I/O with Promises & Async/Await

Using **callbacks** can lead to "callback hell." Instead, use **Promises** or async/await for cleaner code.

Using Promises

```
const fs = require('fs').promises;
console.log('Start');
fs.readFile('file.txt', 'utf8')
    .then(data => console.log(data))
    .catch(err => console.error(err));
console.log('End');
// □ Using Async/Await
const fs = require('fs').promises;
async function readFileAsync() {
    console.log('Start');
    try {
        const data = await fs.readFile('file.txt', 'utf8');
        console.log(data);
    } catch (err) {
        console.error(err);
    }
    console.log('End');
}
readFileAsync();
// ✅ Output Order (Both Promises & Async/Await):
// Start
// End
// (file contents)
```

Clean and readable!

---

6. Key Takeaways

✔ Node.js uses **non-blocking I/O** to handle multiple operations efficiently.
✔ The **event loop** ensures that Node.js remains responsive.
✔ **Callbacks, Promises, and Async/Await** are used to manage asynchronous tasks.
✔ Always prefer **async/await** for better code readability.

Want to Learn More?

Would you like a deeper dive into:

- Event Loop & Call Stack
- Asynchronous Patterns (Callbacks vs. Promises vs. Async/Await)
- Performance Optimization with Non-Blocking I/O

Let me know what interests you!