08 NodeJS: Blocking VS Non Blocking (o síncrono vs asíncrono)

Uno de los conceptos centrales en Node.js, que deriva directamente de su naturaleza asíncrona y orientada a eventos, es la distinción entre operaciones **bloqueantes (blocking)** y **no bloqueantes (non-blocking)**. Esto se relaciona estrechamente con la diferencia entre código **síncrono** y **asíncrono**.

Recordemos que Node.js, en su modelo básico, utiliza un único hilo principal para ejecutar nuestro código JavaScript (el Event Loop).

- **Operación Bloqueante (Síncrona):** Una operación es bloqueante si impide que otras operaciones se ejecuten en el hilo principal hasta que ella misma haya finalizado. Cuando se encuentra una operación bloqueante, el hilo de Node.js se detiene y espera. Solo cuando la operación concluye, el hilo puede continuar con la siguiente instrucción. Las funciones síncronas realizan su tarea completa y devuelven el resultado antes de pasar el control.
- **Operación No Bloqueante (Asíncrona):** Una operación es no bloqueante si permite que el hilo principal continúe ejecutando otras tareas mientras ella se procesa en segundo plano (generalmente operaciones de Entrada/Salida - I/O). Cuando se inicia una operación no bloqueante, se le indica a Node.js qué hacer cuando la operación finalice (normalmente mediante una función de *callback*, una Promesa o `async/await`). El hilo principal queda libre inmediatamente para ejecutar el código siguiente. Cuando la operación en segundo plano termina, se notifica al Event Loop, que ejecutará la función de *callback* (o resolverá la Promesa) tan pronto como el hilo principal esté desocupado.

La mejor manera de entender esto es con un ejemplo práctico, como la lectura de archivos que has propuesto usando el módulo `fs` (File System).

Ejemplo 1: Código Bloqueante (Síncrono)

```
// bloqueante.js
const fs = require("fs"); // Importamos el módulo File System

console.log("Inicio del programa");

// fs.readFileSync es la versión SÍNCRONA (bloqueante) de leer un archivo.
// La 'Sync' al final del nombre es la clave.
// El programa se DETENDRÁ aquí hasta que el archivo 'archivo.txt' se lea completamente.
const data = fs.readFileSync("archivo.txt", "utf8");

// Esta línea SOLO se ejecuta DESPUÉS de que readFileSync haya terminado.
console.log(data);

// Esta línea también espera a que readFileSync y el console.log anterior terminen.
console.log("Fin del programa");
```

Archivo `archivo.txt` (contenido de ejemplo):

```
Este es el contenido del archivo de texto.
```

Ejecución y Salida Esperada (`node bloqueante.js`):

```
Inicio del programa
Este es el contenido del archivo de texto.
Fin del programa
```

Análisis (`bloqueante.js`):

1. Se imprime "Inicio del programa".
2. Se llama a `fs.readFileSync`. El programa **se detiene**. Node.js espera activamente a que el sistema operativo lea el archivo completo del disco y devuelva su contenido. Durante este tiempo (que puede ser corto o largo dependiendo del tamaño del archivo y la velocidad del disco), el hilo principal de Node.js está **bloqueado** y no puede hacer nada más.
3. Una vez que `readFileSync` termina, el contenido del archivo se almacena en la variable `data`.
4. Se imprime el contenido de `data`.
5. Se imprime "Fin del programa".

La ejecución es estrictamente secuencial y predecible, pero a costa de bloquear el hilo. Si esta lectura de archivo fuera lenta y Node.js estuviera ejecutando un servidor web, ninguna otra solicitud podría ser atendida durante ese tiempo.

Ejemplo 2: Código No Bloqueante (Asíncrono)

```
// nobloqueante.js
const fs = require("fs"); // Importamos el módulo File System

console.log("Inicio del programa");

// fs.readFile es la versión ASÍNCRONA (no bloqueante) por defecto.
// No devuelve el contenido directamente. En su lugar, acepta una función de 'callback'.
// Node.js INICIA la operación de lectura y SIGUE EJECUTANDO el código siguiente INMEDIATA
MENTE.
fs.readFile("archivo.txt", "utf8", (err, data) => {
  // Esta función de callback se ejecutará MÁS TARDE, cuando la lectura del archivo termine.
  if (err) {
    // Es crucial manejar posibles errores en operaciones asíncronas.
    console.error("Error al leer el archivo:", err);
    return; // O manejar el error de otra forma
    // throw err; // Lanzar el error detendría el programa si no se captura.
  }
  // Si no hubo error, 'data' contiene el contenido del archivo.
  console.log(data);
});

// Esta línea se ejecuta INMEDIATAMENTE después de INICIAR readFile,
```

```
// SIN esperar a que la lectura del archivo termine.
console.log("Fin del programa");
```

Archivo `archivo.txt` (mismo contenido):

```
Este es el contenido del archivo de texto.
```

Ejecución y Salida Esperada (`node nobloqueante.js`):

```
Inicio del programa
Fin del programa
Este es el contenido del archivo de texto.
```

Análisis (`nobloqueante.js`):

1. Se imprime "Inicio del programa".
2. Se llama a `fs.readFile`. Node.js **inicia** la operación de lectura del archivo en segundo plano (delegándola al sistema operativo a través de mecanismos internos como libuv). **No espera**.
3. La función de *callback* `(err, data) => {...}` se registra para ser ejecutada cuando la operación de lectura finalice.
4. Node.js continúa **inmediatamente** con la siguiente línea de código.
5. Se imprime "Fin del programa".
6. En algún momento posterior (cuando el sistema operativo termina de leer el archivo y notifica a Node.js), el Event Loop ve que la operación ha finalizado y que el hilo principal está libre.
7. El Event Loop ejecuta la función de *callback* registrada.
8. Dentro del callback, se comprueba si hubo un error (`err`). Si no, se imprime el contenido del archivo (`data`).

Observen el orden diferente de la salida. "Fin del programa" aparece *antes* del contenido del archivo. Esto demuestra que Node.js no esperó a `readFile` y continuó ejecutando el resto del script.

¿Por qué preferir lo No Bloqueante en Node.js?

El modelo no bloqueante es la piedra angular de la eficiencia de Node.js para aplicaciones con muchas operaciones de I/O (servidores web, APIs, etc.). Permite que el único hilo principal maneje muchas tareas concurrentemente. Mientras una operación de I/O (leer archivo, consulta a BD, petición de red) está esperando en segundo plano, el hilo principal puede atender otras solicitudes o ejecutar otro código.

Las versiones síncronas (`readFileSync`, `writeFileSync`, etc.) existen y pueden ser útiles en algunos casos específicos (por ejemplo, leer un archivo de configuración al inicio del programa antes de que necesite hacer cualquier otra cosa, o en scripts simples de línea de comandos donde la concurrencia no es una preocupación), pero deben usarse con precaución, especialmente en aplicaciones de servidor, ya que pueden degradar severamente el rendimiento y la capacidad de respuesta. La convención en Node.js es favorecer siempre las alternativas asíncronas (con callbacks, Promises o async/await) para las operaciones de I/O.