21. Asincronía

La asincronía es uno de los pilares fundamentales de Javascript. La **asincronía** es una técnica que permite que tu programa inicie una **tarea de larga duración** y siga respondiendo a otros eventos mientras esa tarea se ejecuta, en lugar de tener que esperar hasta que esa tarea haya terminado.

Existen varias formas de gestionar la asincronía en JavaScript:

- **Mediante callbacks**: Es probablemente la forma más clásica de gestionar la asincronía. Sin embargo, el código basado en callbacks puede volverse difícil de entender cuando el propio callback tiene que llamar a funciones que aceptan un callback.🙄
- **Mediante promesas**: Este es un mecanismo moderno para gestionar la asincronía de forma **no bloqueante**. Las promesas supusieron un gran salto en JavaScript al introducir una mejora sustancial sobre los callbacks y un manejo más elegante de nuestras tareas asíncronas.
- **Mediante** `async/await`: Es una forma simplificada de manejar promesas, pero **bloqueante**. Las palabras clave `async` y `await` surgieron para simplificar el manejo de las promesas, escribir código más sencillo, reducir el anidamiento y mejorar la trazabilidad al depurar.

Cada uno de estos mecanismos tiene sus propias ventajas y desventajas, y la elección de uno u otro depende de las necesidades específicas de cada proyecto y la familiaridad del desarrollador con cada uno de ellos.

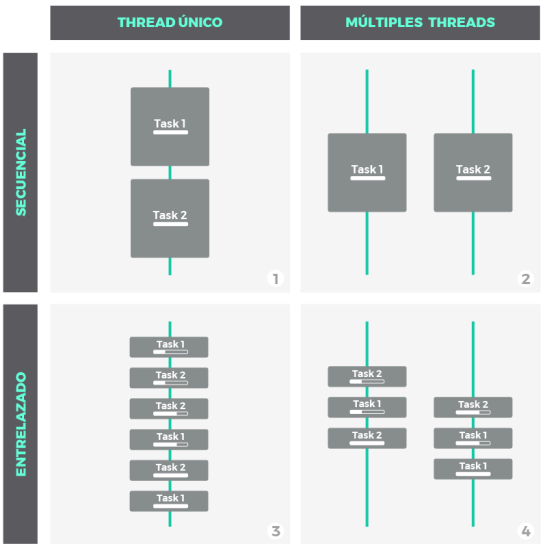
Concurrencia y Paralelismo

Concurrencia y paralelismo son conceptos relacionados pero con un importante matiz de diferencia entre ellos. Es por esto que muy a menudo se confunden y se utilizan erróneamente. Vayamos al grano:

- **Concurrencia**: cuando dos o mas tareas progresan simultáneamente.
- **Paralelismo**: cuando dos o mas tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.

Nótese la diferencia: que varias tareas **progresen** simultáneamente no tiene porque significar que sucedan al mismo tiempo. Mientras que la concurrencia aborda un problema más general, el paralelismo es un sub-caso de la concurrencia donde las cosas suceden exactamente al mismo tiempo.

Mucha gente aún sigue creyendo que la concurrencia implica necesariamente más de un *thread*. **Esto no es cierto**. El entrelazado (o multiplexado), por ejemplo, es un mecanismo común para implementar concurrencia en escenarios donde los recursos son limitados. Piensa en cualquier sistema operativo moderno haciendo multitarea con un único *core*. Simplemente trocea las tareas en tareas más pequeñas y las entrelaza, de modo que cada una de ellas se ejecutará durante un breve instante. Sin embargo, a largo plazo, la impresión es que todas progresan a la vez.



- **Escenario 1**: no es ni concurrente ni paralelo. Es simplemente una ejecución secuencial, primero una tarea, después la siguiente.
- **Escenario 3**: muestra como la concurrencia puede conseguirse con un único *thread*. Pequeñas porciones de cada tarea se entrelazan para que ambas mantengan un progreso constante. Esto es posible siempre y cuando las tareas puedan descompuestas en subtareas mas simples.

  Escenario 2 y **4**: ilustran paralelismo, utilizando multiples *threads* donde las tareas o subtareas corren en paralelo exactamente al mismo tiempo. A nivel de *thread*, el escenario **2** es secuencial, mientras que **4** aplica entrelazado.

JavaScript (en su entorno de ejecución tradicional, como el navegador o Node.js con el modelo de evento único) puede lograr **concurrencia pero no paralelismo** en la ejecución de su código propio (el código JavaScript que escribes). Esto se debe a su modelo de **event loop** y **single-threaded**. Vamos a desglosarlo:

---

1. Concurrencia en JavaScript

JavaScript usa un **único hilo principal** (main thread) para ejecutar tu código, pero puede manejar múltiples tareas de manera **concurrente** gracias al **event loop** y a las **colas de tareas** (task queues). Aquí, las tareas **progresan simultáneamente** pero no se ejecutan al mismo tiempo (no hay paralelismo en el hilo de JS). Ejemplos:

- **Entrelazado (Interleaving)**: Si tienes tareas asíncronas como `setTimeout`, `Promises` o `async/await`, el event loop las maneja intercalando su ejecución sin bloquear el hilo principal.
- **E/S no bloqueante**: Operaciones como llamadas HTTP, lectura de archivos (en Node.js) o eventos del DOM se delegan a APIs del entorno (navegador o sistema operativo), y cuando terminan, su callback se encola para ejecutarse en el hilo principal.

👉 **Esto es concurrencia**: Las tareas avanzan "a la vez" (en términos de progreso), pero no se ejecutan en paralelo en el hilo de JS.

---

2. Paralelismo en JavaScript

JavaScript **no puede ejecutar tu código en paralelo en múltiples hilos** (en el entorno tradicional), ya que solo hay **un hilo para tu código**. Sin embargo:

- **Web Workers**: En navegadores y Node.js, existen los **Web Workers**, que permiten ejecutar scripts en hilos separados (real paralelismo), pero no comparten memoria con el hilo principal (se comunican por mensajes). Esto sí es paralelismo, pero no es parte del modelo tradicional de JS.
- **Operaciones externas**: Las APIs del entorno (como el motor de Chrome o libuv en Node.js) pueden usar múltiples hilos del sistema operativo para tareas como E/S o cálculos pesados (ej: `fs.readFile` en Node.js), pero esto ocurre **fuera del hilo de JS**.

👉 **El código JS propio (tu lógica) no se ejecuta en paralelo**, pero el entorno puede usar paralelismo para otras cosas (sin afectar el single-threaded de JS).

Conclusión

- ✅ **JavaScript es concurrente**: Maneja múltiples tareas simultáneamente (con el event loop), pero en un único hilo.
- ❌ **JavaScript no es paralelo (por defecto)**: Tu código JS no se ejecuta en paralelo en múltiples hilos (a menos que uses Web Workers o APIs externas).
- ⚠️ **El entorno puede ser paralelo**: Las APIs del navegador o Node.js sí pueden usar paralelismo (pero no es "JavaScript" en sí, sino su entorno de ejecución).

Ejemplo gráfico

Escenario	Concurrencia	Paralelismo
JS + Event Loop	✅ Sí	❌ No
JS + Web Workers	✅ Sí	✅ Sí
Llamadas a APIs externas	✅ Sí (JS)	✅ Sí (Entorno)

▼ Hilos ≠ Núcleos

No necesariamente. La relación entre **hilos** (threads) y **núcleos de CPU** (cores) es un tema clave en computación, pero no hay una correspondencia directa 1:1. Vamos a aclararlo:

---

1. Hilos vs. Núcleos

- **Núcleo (CPU Core)**:
  - Es una unidad física de procesamiento en la CPU.
  - Cada núcleo puede ejecutar instrucciones de forma independiente (real paralelismo físico).
  - Ejemplo: Una CPU con 4 núcleos puede ejecutar 4 tareas **verdaderamente en paralelo** (una por núcleo).
- **Hilo (Thread)**:
  - Es una secuencia de instrucciones que puede ser gestionada por el sistema operativo (SO).
  - **Un solo núcleo puede manejar múltiples hilos** mediante técnicas como:
    - **Multithreading simultáneo (SMT)**: Ejecuta partes de diferentes hilos en el mismo núcleo (ej: Hyper-Threading en Intel).
    - **Conmutación de contexto (context switching)**: El SO alterna rápidamente entre hilos, dando la ilusión de paralelismo (concurrencia).

---

2. ¿3 hilos = 3 núcleos?

No necesariamente. Depende de la CPU y el SO:

- **Caso 1**: Si tienes **3 núcleos físicos** (o 3 núcleos lógicos con SMT), 3 hilos pueden ejecutarse **en paralelo** (uno por núcleo).
- **Caso 2**: Si tienes **1 núcleo**, 3 hilos se ejecutarán **concurrentemente** (no en paralelo), compartiendo el mismo núcleo mediante conmutación de contexto.
- **Caso 3**: Si tienes **2 núcleos con Hyper-Threading** (4 hilos lógicos), 3 hilos podrían distribuirse en 2 núcleos físicos (2 en paralelo + 1 esperando).

👉 **Los hilos son una abstracción del SO**, no siempre implican núcleos físicos adicionales.

---

3. Ejemplo práctico

Configuración CPU	Hilos activos	¿Paralelismo real?	Explicación
1 núcleo (sin SMT)	3 hilos	❌ No	Concurrencia por conmutación de contexto.
4 núcleos físicos	3 hilos	✅ Sí (3 en paralelo)	Cada hilo ocupa un núcleo libre.
2 núcleos + Hyper-Threading (4 hilos lógicos)	3 hilos	✅ Parcial (2 en paralelo + 1 en espera)	2 núcleos físicos ejecutan 2 hilos; el tercero compite por recursos.

---

4. JavaScript y los hilos

- **JavaScript (sin Web Workers)**: Corre en **un único hilo**, sin importar cuántos núcleos tenga la CPU.
- **Con Web Workers**: Se crean hilos adicionales, pero:

- Cada Worker es un hilo separado gestionado por el navegador/Node.js.
- Si la CPU tiene múltiples núcleos, los Workers pueden distribuirse entre ellos (paralelismo real).

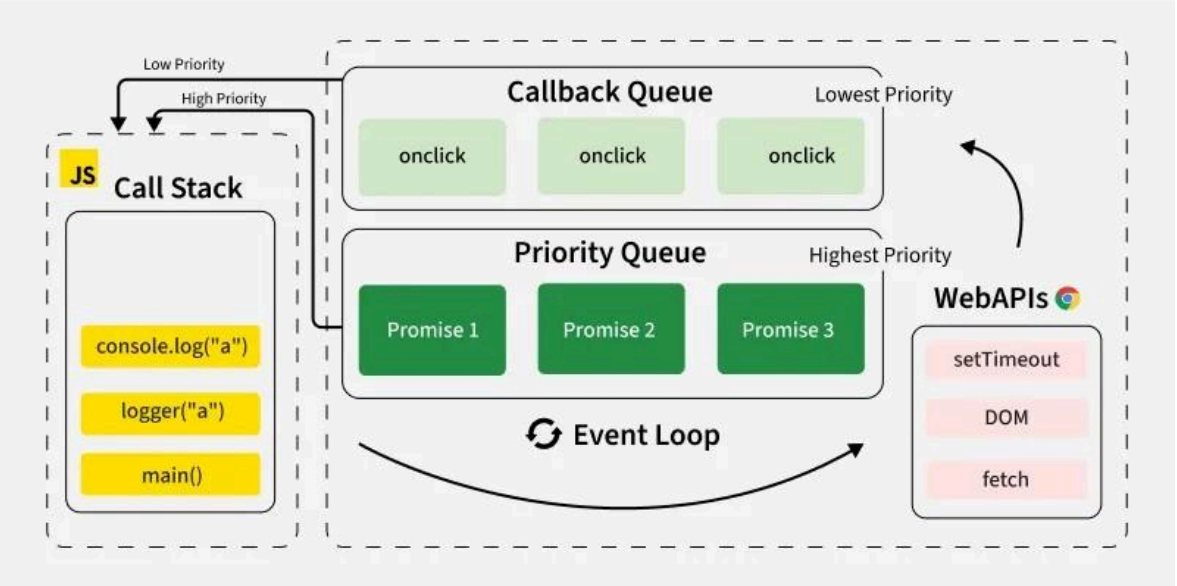
Conclusión

- **Hilos ≠ Núcleos**: Los hilos son unidades lógicas; los núcleos son unidades físicas.
- **Paralelismo requiere núcleos**: Para que hilos se ejecuten **al mismo tiempo**, se necesitan núcleos físicos suficientes.
- **JavaScript**: Por defecto, no usa paralelismo (single-threaded), pero puede aprovecharlo con Workers o APIs externas.

Respuesta final:

No, 3 hilos no siempre significan 3 núcleos trabajando. Depende de la arquitectura de la CPU y de cómo el SO asigna los hilos a los núcleos disponibles. El paralelismo real solo ocurre si hay suficientes núcleos físicos (o lógicos con SMT).

▼ Event loop



Pila de llamadas

http://latentflip.com/loupe/?code=Cgpjb25zb2xlLmxvZygiQ29kaWdvIFNpbmNyb25vIik7CmNvbnNvbGUubG9nKCJFbXBlemFtb3MiKQoKbGV0IGRvcyA9IGZ1bmN0aW9uKCl7CiAgIGNvbnNvbGUubG9nKCJFbnRybyBlbiBkb3MiKTsKICAgY29uc29sZS5sb2coIkVzdG95IGVuIGRvcyIpOwogICBjb25zb2xlLmxvZygiU2FsZ28gZGUgZG9zIik7Cn07CgpsZXQgdW5vID0gZnVuY3Rpb24oKXsKICAgY29uc29sZS5sb2coIkVudHJvIGVuIHVubyIpOwogICBkb3MoKTsKICAgY29uc29sZS5sb2coIlNhbGdvIGRlIHVubyIpOwp9OwoKdW5vKCkKCmNvbnNvbGUubG9nKCJIZW1vcyB0ZXJtaW5hZG8iKTs%3D!

```
console.log("Codigo Sincrono");
console.log("Empezamos")

let dos = function(){
   console.log("Entro en dos");
   console.log("Estoy en dos");
   console.log("Salgo de dos");
};

let uno = function(){
   console.log("Entro en uno");
   dos();
   console.log("Salgo de uno");
};

uno()

console.log("Hemos terminado");
```

http://latentflip.com/loupe/?code=Cgpjb25zb2xlLmxvZygiQ29kaWdvIEFzaW5jcm9ubyIpOwpjb25zb2xlLmxvZygiRW1wZXphbW9zIik7CmxldCBkb3MgPSBmdW5jdGlvbigpewogICBjb25zb2xlLmxvZygiRW50cm8gZW4gZG9zIik7CiAgIHNldFRpbWVvdXQoZnVuY3Rpb24oKXsKICAgICAgY29uc29sZS5sb2coInRhcmRvIG1pbCIpOwogICB9LDAwMDApOwogICBjb25zb2xlLmxvZygiRXN0b3kgZW4gZG9zIik7CiAgIGNvbnNvbGUubG9nKCJTYWxnbyBkZSBkb3MiKTsKfTsKCmxldCB1bm8gPSBmdW5jdGlvbigpewogICBjb25zb2xlLmxvZygiRW50cm8gZW4gdW5vIik7CiAgIGRvcygpOwogICBjb25zb2xlLmxvZygiU2FsZ28gZGUgdW5vIik7Cn07CgpjbGVhclRpbWVvdXQoKTsKCnVubygpCgpjb25zb2xlLmxvZygiSGVtb3MgdGVybWluYWRvIik7

```
console.log("Codigo Asincrono");
console.log("Empezamos")

let dos = function(){
   console.log("Entro en dos");
   setTimeout(function(){
      console.log("tardo mil");
```

```
    }, 1000);
    console.log("No pienso esperarte");
    console.log("Salgo de dos");
};

let uno = function(){
    console.log("Entro en uno");
    dos();
    console.log("Salgo de uno");
};

uno()

console.log("Hemos terminado");
```

Los `console.log` son bloqueantes. Incluso si ponemos un `setTimeout` de 0 segundos en uno se va a bloquear hasta que se hayan terminado las ejecuciones bloqueantes.

http://latentflip.com/loupe/?code=Cgpjb25zb2xlLmxvZygiQ29kaWdvIEFzaW5jcm9ubyIpOwpjb25zb2xlLmxvZygiRW1wZXphbW9zIikKCmxldCBkb3MgPSBmdW5jdGlvbigpewogICAgY

```
console.log("Codigo Asincrono");
console.log("Empezamos")

let dos = function(){
    console.log("Entro en dos");
    setTimeout(function(){
        console.log("tardo mil");
    }, 1000);
    console.log("No pienso esperarte");
    console.log("Salgo de dos");
};

let uno = function(){
    console.log("Entro en uno");
    setTimeout(function(){
        console.log("tardo cero");
    }, 0);
    dos();
    console.log("Salgo de uno");
};

uno()

console.log("Hemos terminado");
```

Si modificamos los tiempos del `setTimeout` veremos que las tareas se despachan por su orden de finalización, perosiempre después de todo el código bloqueante.