09 NodeJS: Single Thread

Hemos mencionado que Node.js opera fundamentalmente sobre un **único hilo principal** para ejecutar el código JavaScript del usuario. A primera vista, esto podría parecer una limitación severa para manejar múltiples tareas simultáneamente, como atender a varios usuarios en un servidor web. Sin embargo, Node.js logra la concurrencia (manejar múltiples cosas al mismo tiempo, aunque no necesariamente en paralelo) gracias a su arquitectura basada en el **Event Loop (Bucle de Eventos)** y las operaciones **asíncronas no bloqueantes**.

El código `single-thread.js` nos ayuda a visualizar este proceso:

Código de Ejemplo:

```
// single-thread.js
console.log("Inicio"); // 1. Síncrono

setTimeout(() => { // 2. Asíncrono (Timer 1)
  console.log("Uno");
}, 3000); // Se ejecutará después de ~3000ms

setTimeout(() => { // 3. Asíncrono (Timer 2)
  console.log("Dos");
}, 0); // Se ejecutará "lo antes posible", pero después del script actual

setTimeout(() => { // 4. Asíncrono (Timer 3)
  console.log("Tres");
}, 0); // Igual que el anterior, se encola después del Timer 2

console.log("Fin"); // 5. Síncrono
```

Salida Esperada:

```
Inicio
Fin
Dos
Tres
Uno
```

Desglose del Proceso (Simplificado):

1. **Call Stack (Pila de Llamadas):** Imaginen una pila donde se colocan las tareas que se están ejecutando *ahora mismo*. Solo se puede ejecutar lo que está en la cima de la pila.
2. **Node APIs / Web APIs:** Son funcionalidades proporcionadas por el entorno (Node.js o el navegador) que pueden manejar operaciones en segundo plano (como timers, I/O de red, lectura de archivos).
3. **Callback Queue (Cola de Callbacks) / Task Queue:** Una cola donde se colocan las funciones de *callback* que están listas para ejecutarse porque su operación asíncrona asociada (como un timer o una lectura de archivo) ha finalizado.

4. **Event Loop:** Es el "director de orquesta". Su trabajo es constantemente verificar: "¿Está la Call Stack vacía?". Si lo está, toma la primera función de la Callback Queue (si hay alguna) y la empuja a la Call Stack para que se ejecute.

Recorrido del Código `single-thread.js`:

1. `console.log("Inicio");` **(Línea 1):**
   - `console.log("Inicio")` se añade a la Call Stack.
   - Se ejecuta, imprime "Inicio" en la consola.
   - Se elimina de la Call Stack.
2. `setTimeout(/* Uno */, 3000);` **(Línea 3):**
   - `setTimeout` se añade a la Call Stack.
   - Node.js (a través de sus APIs internas) inicia un temporizador de 3000ms. La función callback `() => console.log("Uno")` se asocia a este timer.
   - `setTimeout` **retorna inmediatamente** y se elimina de la Call Stack. *No espera los 3000ms*.
3. `setTimeout(/* Dos */, 0);` **(Línea 7):**
   - `setTimeout` se añade a la Call Stack.
   - Node.js inicia un temporizador de ~0ms. La función callback `() => console.log("Dos")` se asocia a este timer.
   - `setTimeout` **retorna inmediatamente** y se elimina de la Call Stack.
   - El temporizador de 0ms finaliza casi instantáneamente. La función `() => console.log("Dos")` se coloca en la **Callback Queue**, esperando su turno.
4. `setTimeout(/* Tres */, 0);` **(Línea 12):**
   - `setTimeout` se añade a la Call Stack.
   - Node.js inicia otro temporizador de ~0ms. La función callback `() => console.log("Tres")` se asocia a este timer.
   - `setTimeout` **retorna inmediatamente** y se elimina de la Call Stack.
   - El temporizador de 0ms finaliza casi instantáneamente. La función `() => console.log("Tres")` se coloca en la **Callback Queue**, detrás de la función para "Dos".
5. `console.log("Fin");` **(Línea 16):**
   - `console.log("Fin")` se añade a la Call Stack.
   - Se ejecuta, imprime "Fin" en la consola.
   - Se elimina de la Call Stack.
6. **Fin del Script Síncrono:** Todo el código principal del archivo se ha ejecutado. La Call Stack está ahora **vacía**.
7. **El Event Loop entra en acción:**
   - Verifica la Call Stack: Está vacía.
   - Verifica la Callback Queue: Encuentra `() => console.log("Dos")`.
   - Mueve `() => console.log("Dos")` de la Queue a la Call Stack.

- Se ejecuta `console.log("Dos")`, imprime "Dos". Se elimina de la Call Stack.

8. **Siguiente Tick del Event Loop:**
   - Verifica la Call Stack: Está vacía.
   - Verifica la Callback Queue: Encuentra `() => console.log("Tres")`.
   - Mueve `() => console.log("Tres")` de la Queue a la Call Stack.
   - Se ejecuta `console.log("Tres")`, imprime "Tres". Se elimina de la Call Stack.
9. **El Event Loop sigue esperando:** La Callback Queue está vacía por ahora. El timer de 3000ms sigue corriendo en segundo plano.
10. **Después de ~3000ms:** El primer temporizador (para "Uno") finaliza. La función `() => console.log("Uno")` se coloca en la **Callback Queue**.
11. **Siguiente Tick del Event Loop (cuando ocurra):**
    - Verifica la Call Stack: Está vacía.
    - Verifica la Callback Queue: Encuentra `() => console.log("Uno")`.
    - Mueve `() => console.log("Uno")` de la Queue a la Call Stack.
    - Se ejecuta `console.log("Uno")`, imprime "Uno". Se elimina de la Call Stack.

Conclusiones Clave:

- **Ejecución Síncrona Primero:** Todo el código síncrono del script principal se ejecuta de principio a fin antes de que cualquier callback asíncrono (como los de `setTimeout`) tenga la oportunidad de ejecutarse. Por eso "Inicio" y "Fin" aparecen primero.
- `setTimeout(..., 0)` **No es Inmediato:** No significa "ejecutar ahora", sino "ejecutar tan pronto como sea posible *después* de que la pila de llamadas actual esté vacía". Por eso "Dos" y "Tres" aparecen después de "Fin".
- **Orden en la Cola:** Los callbacks se colocan en la cola a medida que sus operaciones asíncronas finalizan. Los callbacks de los timers de 0ms finalizan antes que el de 3000ms. Como se programaron en el orden "Dos" luego "Tres", sus callbacks entran en la cola y se ejecutan en ese mismo orden.
- **No Bloqueo:** Mientras los timers esperaban, Node.js pudo ejecutar otras partes del código (`console.log("Fin")`) y luego procesar los callbacks una vez que el script principal terminó. El hilo no se quedó bloqueado esperando los timers.

Este modelo permite a Node.js ser muy eficiente manejando operaciones de I/O (que son inherentemente lentas y asíncronas, como esperar respuestas de red o leer discos) sin necesidad de la complejidad de la gestión manual de múltiples hilos para cada conexión o tarea. Sin embargo, también implica que una operación síncrona muy larga y que consuma mucha CPU *sí* puede bloquear el Event Loop, impidiendo que se procesen otros eventos.
