REPÚBLICA DE VENEZUELA

UNIVERSIDAD RAFAEL URDANETA

CÁTEDRA: INTRODUCCIÓN A SISTEMAS DISTRIBUIDOS

URU 2022 A

EVALUACIÓN NÚMERO 1

INTRODUCCIÓN A SISTEMAS DISTRIBUIDOS

Estudiante:

Gabriela Lobo C.I.: 25.483.621

Maracaibo, 12 de marzo de 2022

1. Para esta evaluación, lea el código en tres archivos: ‘test1.js’, ‘test2.js’ y ‘test3.mjs’, analícelos e intente adivinar el orden de las líneas que imprime en pantalla, luego, para cada uno, realice un texto descriptivo que explique el porqué del orden de ejecución observado.

RESPUESTA

- **Test1.js**

```
-   new Promise(function (resolve) {
-       console.log('new promise')
-       resolve()
-     }).then(() => {
-       console.log('then 1')
-     })
-
-     async function foo () {
-       console.log('async function')
-     }
-
-     foo().then(() => {
-       console.log('then 2')
-     })
-
-     setImmediate(() => {
-       console.log('immediate 1')
-     })
-
-     setTimeout(() => {
-       console.log('timeout 1')
-     })
-
-     process.nextTick(() => {
-       console.log('nextTick 1')
-     })
-
-     queueMicrotask(() => {
-       console.log('microtask 1')
-     })
-
-     setTimeout(() => {
```

```
-       console.log('timeout 2')
-     })
-
-     setImmediate(() => {
-       console.log('immediate 2')
-     })
-
-     process.nextTick(() => {
-       console.log('nextTick 2')
-     })
-
-     process.nextTick(() => {
-       console.log('nextTick 3')
-     })
-
-     queueMicrotask(() => {
-       console.log('microtask 2')
-     })
```

Para el archivo test1.js, se observa que se trata de un ejemplo del *Event Loop* de NodeJS. Esto es lo que permite a NodeJS ejecutar Operaciones de I/O no bloqueantes. Cuando NodeJS inicia, el mismo arranca el Event Loop, procesa el script de entrada provisto, el cual podría hacer llamadas API asíncronas, planificar Timers, o llamar a *process.nextTick()*, luego comienza a procesar el Event Loop. El siguiente diagrama busca mostrar una vista general del orden de operación del Event Loop de NodeJS.

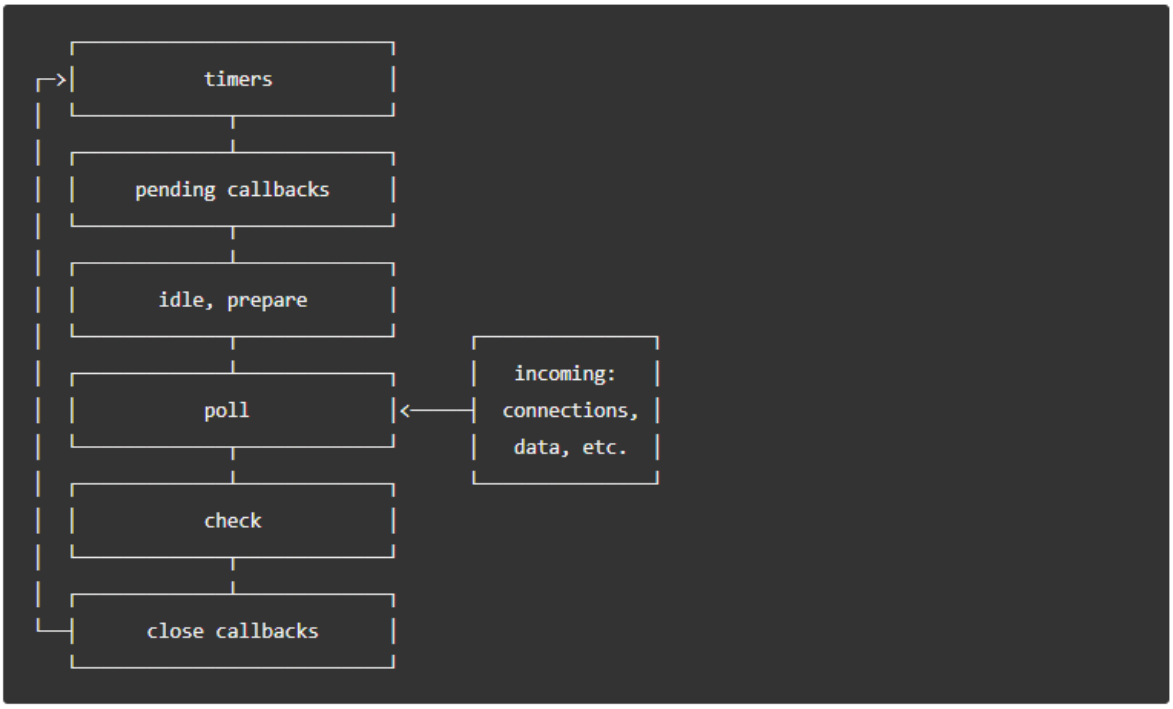

Diagrama de operaciones del Event Loop de NodeJS. *Fuente: Documentación de NodeJS*.

En el código en cuestión, podemos observar que tenemos diferentes definiciones, así que se buscará ir explicando las líneas que se imprimen en consola y el porqué de ese orden. Primeramente, comenzamos con la promesa, esta es la primera línea que se imprime en pantalla, mostrando ***new promise***, producto de que Timeout() no posee una ejecución planificada, y la cola de la poll no está vacía, por ello, ésta comienza a iterar dentro de su cola de callbacks, ejecutándolas sincrónicamente. Luego de que se imprime el mensaje de new promise, se imprime el mensaje de la función asíncrona ***async function***, ya que al ser una función asíncrona, no tiene que esperar a que se termine el proceso anterior para ella comenzar a ejecutarse, sin embargo, contrariamente a lo que se podría suponer, la línea que se imprime luego de esta última es el callback

de la promesa, ***then 1***, que se ejecutó inicialmente, puesto que como se están manejando colas de callbacks, éstas son ejecutadas bajo la política de procesamiento Primero adentro Primero afuera, (FIFO por sus siglas en inglés), y el primer callback en registrarse dentro de la cola fue el de la promesa, así, las impresión que sigue es el callback de la función asíncrona ***then 2***, que fue la segunda en iniciar.

Tras esas impresiones, lo que sigue es la impresión de los process.nextTick(), debido a que este no es técnicamente parte del event loop, en vez de ello, la cola de nextTick será procesada luego de que la operación actual sea completada, sin importar la etapa actual del event loop, quedando las impresiones ***nextTick 1***, ***nextTick 2*** y ***nextTick 3*** respectivamente.

Seguido de estos 3, se ejecutan las líneas de ***microTask 1*** y seguido a ella ***microTask 2***, siendo las últimas en imprimirse las líneas correspondientes a ***timeOut 1***, ***timeOut 2***, ***immediate 1***, e ***immidiate 2***.

Un punto importante a tomar en consideración es que la ejecución entre setTimeout y setImmediate cuando están dentro del módulo principal será trazado por el rendimiento del proceso (que podría ser afectado por otras aplicaciones corriendo en el equipo).

- **Test2.js**

```
- const { readFile } = require('fs')
-
- readFile(__filename, () => {
-   new Promise(function (resolve) {
-     console.log('new promise')
-     resolve()
-   }).then(() => {
-     console.log('then 1')
-   })
-
-   async function foo () {
-     console.log('async function')
-   }
-
-   foo().then(() => {
-     console.log('then 2')
-   })
-
-   setImmediate(() => {
-     console.log('immediate 1')
-   })
-
-   setTimeout(() => {
-     console.log('timeout 1')
-   })
-
-   process.nextTick(() => {
-     console.log('nextTick 1')
-   })
-
-   queueMicrotask(() => {
-     console.log('microtask 1')
-   })
-
-   setTimeout(() => {
-     console.log('timeout 2')
-   })
-
-   setImmediate(() => {
-     console.log('immediate 2')
-   })
-
-   process.nextTick(() => {
```

```
-       console.log('nextTick 2')
-     })
-
-     process.nextTick(() => {
-       console.log('nextTick 3')
-     })
-
-     queueMicrotask(() => {
-       console.log('microtask 2')
-     })
-   })
```

Para este otro archivo, correspondiente a test2.js, podemos observar que tenemos un proceso de I/O dentro del módulo principal, por ello, el orden de impresión del código varía un poco respecto al ejemplo anterior. Primeramente, se imprime la línea de la promesa ***new promise***, y seguido a ella la línea de la función asíncrona foo ***async function***, sus callbacks están guardados en la cola de callbacks, sin embargo deben esperar a que el proceso de I/O finalice antes de imprimir la línea del callback de la promesa, es por ello que las líneas de la cola de process.nextTick() son impresas en pantalla antes de dicho callback, mostrando en pantalla ***nextTick 1***, ***nextTick 2*** y ***nextTick 3***, ya que su cola se ejecuta sin importar en qué fase del event loop se encuentre.

Una vez que los callbacks de process.nectTickqueue son ejecutadas, se ejecuta el callback de la promesa que es la primera que entró en la cola, seguido del callback de la función asíncrona foo, mostrando en pantalla ***then 1*** y ***then 2*** respectivamente, dando paso así a la impresión de las líneas correspondientes a ***microTack 1*** y ***microTask 2***.

Finalmente, se imprimen las líneas referentes a los timers, siendo mostradas en pantalla primero las líneas de los setImmediate() antes que los setTimout() debido a que se cuenta con un proceso de I/O, ejecutándose primero el timer de setImmediate sobre setTimeout, siendo ésta su principal ventaja sobre setTimeout.

- **Test3.mjs**

```
-  new Promise(function (resolve) {
-      console.log('new promise')
-      resolve()
-    }).then(() => {
-      console.log('then 1')
-    })
-
-    async function foo () {
-      console.log('async function')
-    }
-
-    foo().then(() => {
-      console.log('then 2')
-    })
-
-    setImmediate(() => {
-      console.log('immediate 1')
-    })
-
-    setTimeout(() => {
-      console.log('timeout 1')
-    })
-
-    process.nextTick(() => {
-      console.log('nextTick 1')
-    })
-
-    queueMicrotask(() => {
-      console.log('microtask 1')
-    })
-
-    setTimeout(() => {
-      console.log('timeout 2')
```

```
-     })
-
-     setImmediate(() => {
-       console.log('immediate 2')
-     })
-
-     process.nextTick(() => {
-       console.log('nextTick 2')
-     })
-
-     process.nextTick(() => {
-       console.log('nextTick 3')
-     })
-
-     queueMicrotask(() => {
-       console.log('microtask 2')
-     })
```

Este último archivo test3.mjs posee el mismo código que el primer archivo test1.js, con la diferencia de que es un archivo con el formato de módulo ES6, lo que influye un poco en el orden de impresión de las líneas en pantalla. Igual que en el ejemplo uno, la primera y segunda línea que se imprime son las de ***new promise*** y ***async function*** respectivamente, continuando esta vez con sus callbacks, mientras que los callbacks de la cola de nextTick() se ejecuta un poco después (recordando que esta cola es asíncrona y se ejecuta sin importar en qué fase del event loop se encuentre), imprimiendo así en pantalla ***then 1*** seguido del callback de la función foo ***then 2***.

Luego de lo anterior, se imprime en pantalla las líneas que competen a las ***microTask 1*** y ***microTask 2***, siendo después de estos dos últimos que se ejecutan la cola de callbacks de nextTickqueue, mostrando entonces ***nextTick 1, nextTick 2*** y ***nextTick 3***, acabando con la impresión de los timers, que en este caso respecto

al anterior se ejecuta primero el setImmediate que el setTimeout debido al trazado del rendimiento de la máquina, quedando al final ***immediate 1, immediate 2, timeout 1*** y ***timeout 2***.