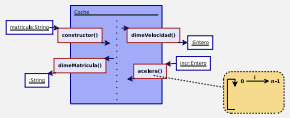
Grado Tecnologías Interactivas

Programación 2




# Práctica 4

UNIVERSIDAD POLITECNICA DE VALENCIA

Escola Politècnica Superior de Gandia

DSIC

Departament de Sistemes Informàtics i Computació

# Práctica 4

## ¡ Atención !

▷ Se recuerda que las prácticas deben prepararse antes de acudir al aula informática, anotando en el enunciado las dudas que se tengan.

▷ Los diseños y algortimos que se piden en esta práctica deben escribirse en la libreta de apuntes para poder ser revisados.

▷ Utiliza *git* en cada ejercicio haciendo *commit* cada vez que consigas un "hito".

▷ La realización de las prácticas es un trabajo individual y original. En caso de plagio se excluirá al alumno de la asignatura. Por tanto, es preferible presentar el trabajo realizado por uno mismo aunque éste tenga errores.

1

## Condiciones de Carrera

Una condición de carrera ("race condition" en wikipedia) es una inconsitencia en los datos de un programa provocada porque dichos datos son manipulados *simultáneamente* por dos funciones del programa (o por una misma función ejecutada dos veces de forma concurrente).

La ejecución simultánea de una o varias funciones de un programa ocurre cuando

- el programa tiene varios hilos de ejecución ("thread" en wikipedia)
- el programa tiene un único hilo de ejecución, pero éste es capaz alternar la ejecución de dos funciones (en instantes distintos ejecutar código de funciones diferentes) por lo que *parece* que hay varios hilos.

El segundo caso (un sólo hilo capaz de ejecutar funciones alternativamente) es lo que ocurre en *node.js*. Tradicionalmente, las condiciones de carrera han sido un problema típico de programas con varios hilos. Dado que *node.js* sólo tiene un hilo, se cree erróneamente que no pueden darse condiciones de carrera. Pero esto no es cierto, como vamos a ver.

Primero veamos un trozo de código de un hipotético programa de reserva de un único asiento (por simplificar).

```
1 var elAsiento = "libre"
2
3 function reservarAsiento( nombre ) {
4  if( elAsiento == "libre" ) {
5    elAsiento = nombre
6  }
7 }
```

A "simple vista" no parece haber ningún problema porque pensamos la función `reservarAsiento()` se ejecuta toda *atómicamente*: sin interrupciones. Pero en un programa con dos hilos, el primero de ellos podría ejecutar la función para hacer una reserva para "juan", llegar hasta la línea 4 viendo que el asiento está libre y justo en ese punto detenerse, antes de anotar el nombre al asiento. Este primer hilo, cuando continúe, ejecutará la línea 5 y anotará "juan". Sin embargo, mientras el primer hilo "duerme", un segundo hilo puede ejecutar la misma función para realizar una reserva a nombre de "pepe" y terminar sin detenerse. Aquí es donde se produce el desastre, porque el segundo hilo no debería haber anotado el nombre de "pepe"; pero lo ha hecho y cree que la reserva es de "pepe". Sin embargo, cuando el primer hilo despierte, anotará el nombre de "juan" y ambos hilos creerán que han hecho la reserva correctamente.

En *node.js* (que tiene sólo un hilo), el anterior código tal cuál está, no producirá nunca ninguna condición de carrera porque no hay nada en ese código que detenga el hilo. Pero el hilo de *node.js*, alterna la ejecución de distintas funciones cada vez que realizamos una operación de entrada/salida o lo suspendemos con `setTimeout()` (para no estar ocioso esperando).

## Ejercicio.

Prueba el siguiente código. Comprueba que efectivamente ocurre una "condición de carrera" y busca el lugar donde sucede.

```
// --------------------------------
// --------------------------------
var elAsiento = "nadie" // variable global (mala idea)

// --------------------------------
// Texto -> cambiarNombre()
//
// Tras un intervalo, cambia el valor de la variable
// global (mala idea) elAsiento.
// --------------------------------
function cambiarNombre( nombre ) {
```

```
 setTimeout( function () {
  console.log( " *** elAsiento es para: " + nombre + "***" )
  elAsiento = nombre
 }, 100 )
} // ()

// --------------------------------
// Texto -> hacerReserva()
// colateralmente, cambia la variable global (mala idea)
// --------------------------------
function hacerReserva( nombre ) {
 if ( elAsiento == "nadie" ) {
  cambiarNombre( nombre )
  return
 }
} // ()

// --------------------------------
// main
// --------------------------------
console.log( "Intento reservar para juan." )
console.log( "Como es el primero en reservar, el asiento ")
console.log( "debería ser para él" )
hacerReserva( "juan" )

console.log( "Intento reservar para pepe." )
hacerReserva( "pepe" )

//
// Sin embargo, al final, ¿qué vale elAsiento?
```

```
    //
    setTimeout( ()=>console.log("elAsiento finalmente es para "+elAsiento), 1000)
```

El trozo de código donde pueden ocurrir condiciones de carrera se llama *sección crítica* ("sección crítica" en wikipedia). Para evitar condiciones de carrera hay que conseguir que una sección crítica no se ejecute dos veces simultáneamente. Es decir, si ha comenzado a ejecutarse una sección crítica, no puede volver a ser ejecutada antes que termine la ejecución actual. Esta propiedad se conoce como *exclusión mútua*, porque la ejecución de la sección crítica *excluye* una ejecución simultánea.

Para conseguir exclusión mútua, hay distintos mecanismos como *cerrojos* ("cerrojos" en wikipedia) o *semáforos* ("semáforos" en wikipedia).

Un tipo de aplicaciones especialmente sensible a las condiciones de carreras son las que usan bases de datos. Por ello, ofrecen una herramienta llamada *transacción* ("transacción" en wikipedia) que permite agrupar varios accesos a la base de datos (consultas, modificaciones, borrados, inserciones) para que sean ejecutados *atómicamente*. En cualquier aplicación que use una base de datos, deberemos usar transacciones.

## 2 Promesas

Como sabemos, una función asíncrona termina inmediatamente, pero recibe un *callback* que será llamado cuando la tarea que debe realizar la función, efectivamente, se ha completado.

Un sencillo algoritmo que tenga tres pasos secuenciales en los que cada uno depende del anterior, implica que hayan de ir anindándose las llamadas en el callback del paso anterior. Por ejemplo:

```
// --------------------------------
// R -> porDos() -> R (versión con callback)
// --------------------------------
function porDos( n, callback ) {
 setTimeout( function () {
  callback( n*2 )
 }, 1000 )
} // ()
// --------------------------------
// main()
// --------------------------------
//  Algoritmo:
//       a <- calcular el doble 3
//       b <- calcular el doble 4
//       c <- calcular el doble 5
//       Sumar a+b+c
porDos( 3, function( a ) {
 porDos( 4, function( b ) {
  porDos( 5, function( c ) {
   console.log( a+b+c )
  })
 })
})
```

Este anidamiento produce el efecto conocido por el nombre de "callback hell" o "pyramid of doom" porque puede llegar a ser un "infierno" y la apariencia de dicho código es de una "pirámide".

Una forma de solucionar esto es utilizar una clase de biblioteca llamada Promise (promesa), que se utiliza de siguiente forma:

1. La función asíncrona debe crear un objeto de tipo Promise y devolverlo.
2. El constructor del objeto promesa debe recibir una función anónima como ésta:

```
function ( resolver, rechazar ) {
   // código que queremos escribir realmente
  ...
}
```

3. Dentro de la anterior función hemos de escribir el código. Para devolver el resultado que calculemos llamaremos a la función resolver(). Si quisiéramos avisar de un error, llamaríamos a la función rechazar().
4. Cuando llamemos a la función asíncrona, obtendremos una promesa. Para sacar de ella el resultado, hemos de llamar al método then() dándole un callback.

Veamos un primer ejemplo con la función porDos() "promisificada".

```
// --------------------------------
// R -> porDos() -> R (versión con promesa)
// --------------------------------
function porDos( n ) {
 var prom = new Promise( function( resolver, rechazar ) {
  setTimeout( function () {
   resolver( n*2 )
  }, 1000 )
 })

 return prom
} // ()
```

```
// --------------------------------
// main()
// --------------------------------
var p = porDos( 3 )

p.then( function( a ) {
 console.log( "el resultado de 2*3 es " + a )
})
```

Una ventaja de las promesas es que se pueden encadenar los then(). Por tanto, podemos escribir el siguiente código que no tiene ninguna pirámide y por tanto es más legible aunque dista de ser perfecto.

```
// --------------------------------
// R -> porDos() -> R (versión con promesa)
// --------------------------------
function porDos( n ) {
 var prom = new Promise( function( resolver, rechazar ) {
  setTimeout( function () {
   resolver( n*2 )
  }, 300 )
 })

 return prom
} // ()
// --------------------------------
// main()
// --------------------------------
var a
var b
var c
porDos( 3 ) // pido calcular 2*3
```

```
    .then( function( r ) { // cuando esté, entonces ...
     a = r // guardo el resultado
     return porDos( 4 )  // pido calcular 2*4
    })
    .then( function( r ) { // cuando esté, entonces ...
     b = r // guardo el resultado
     return porDos( 5 )  // pido calcular 2*5
    })
    .then( function( r ) { // cuando esté, entonces ...
     c = r // guardo el resulatdo
     return (a+b+c) // hago la suma de todo y devuelvo el valor
    })
    .then( function( total ) { // cuando esté, entonces ...
     console.log( "total = " + total  )
    })
```

En el código anterior, véase que el código de cada .then() puede devolver una promesa o directamente un valor. Si se devuelve una promesa, la cadena no continúa hasta que ésta se resuelve.

En general, muchos programadores no usan promesas porque si se organiza bien el código con callbacks, resulta igual de legible que el equivalente usando promesas.

Sin embargo, las últimas versiones de JavaScript han ampliado el lenguaje añadiendo funciones async que esperan con await la resolución de promesas. Esto permite escribir este programa, que es prerfectamente legible y claro:

```
// --------------------------------
// R -> porDos() -> R (versión con promesa)
// --------------------------------
function porDos( n ) {
 var prom = new Promise( function( resolver, rechazar ) {
  setTimeout( function () {
   resolver( n*2 )
  }, 300 )
 })

 return prom
} // ()
// --------------------------------
// --------------------------------
async function hacerUnaSuma () {
 var a = await porDos( 3 ) // llamo a calcular 2*3 y espero
 var b = await porDos( 4 ) // llamo a calcular 2*4 y espero
 var c = await porDos( 5 ) // llamo a calcular 2*5 y espero

 // Hago la suma de todo y devuelvo el valor.
 // En realidad devuelve una promesa con el valor
 // resuelto y guardado dentro
 return (a+b+c)
} // ()
// --------------------------------
// main()
// --------------------------------
hacerUnaSuma().then( function( total ) {
 console.log( " total = " + total )
})
```

## Ejercicios

1. Prueba detallamadamente cada uno de los anteriores ejemplos.
2. Escribe un programa que
   - Tenga una función `leerFichero()` con este diseño

$$\boxed{\texttt{nombreFichero:Texto}} \rightarrow leerFichero() \rightarrow \boxed{\texttt{contenidoFichero:Texto | Error}}$$

     y esté implementada mediante una promesa.
   - Tenga una función `escribirFichero()` con este diseño

$$\boxed{\begin{array}{l}\texttt{nombreFichero:Texto}\\ \texttt{contenido:Texto}\end{array}} \rightarrow escribirFichero() \rightarrow \boxed{\emptyset \texttt{ | Error}}$$

     y esté implementada mediante una promesa.
   - Tenga una función `concatenarFicheros()` con este diseño

$$\boxed{\begin{array}{l}\texttt{nombreOrigen1:Texto}\\ \texttt{nombreOrigen2:Texto}\\ \texttt{nombreDestino:Texto}\end{array}} \rightarrow concatenarFicheros() \leftarrow \boxed{\emptyset \texttt{ | Error}}$$

     que cree un nuevo fichero uniendo el contenido de otros dos ficheros. Esta función debe ser `async` e implementarse usando `await` con las funciones `leerFichero()` y `escribirFichero()`.

12 abril 2019