# Programación concurrente

Universitat d'Alacant
Universidad de Alicante


# Concurrencia en PHP, Javascript y otros lenguajes de Internet

Francisco Joaquín Murcia Gómez

25 de febrero de 2022

# Índice

# 1. PHP

PHP es un lenguaje interpretado influenciado por Perl, C, C++, Java, Tcl, se adapta especialmente al desarrollo web (lógica de negocio, animaciones...), aunque también tiene otros usos aplicaciones gráficas autónomas y el control de drones. Es muy usado hoy en día en frameworks de desarrollo web como Laravel o Symfony.

## 1.1. Hilos en PHP

Para la programación con hilos, PHP hace uso de pthreads, es una API orientada a objetos que proporciona todas aplicaciones PHP pueden crear, leer, escribir, ejecutar y sincronizar con subprocesos, trabajadores y objetos subprocesados. Cabe destacar que no es válida para un entorno de servidor, por lo tanto, queda restringida solo para las aplicaciones basadas en cliente.

La clase Thread es la clase encargada de crear hilos básicos en PHP, su funcionamiento es similar a su homóloga de Java (Thread), es una clase que se hereda, de tal modo que si se desea que la clase con la que se está trabajando use hilos, esta extenderá de la clase Thread, la programación del hilo se hará en una función run(). Un ejemplo sería el siguiente:

```php
<?php
//Este código imprime por pantalla el id de los 50 hilos creados
class workerThread extends Thread {
public function __construct($i){//constructor
  $this->i=$i;
}

public function run(){//función del hilo
  while(true){
   echo $this->i;
   sleep(1);
  }
}
}

for($i=0;$i<50;$i++){
$workers[$i]=new workerThread($i);//creamos un hilo
$workers[$i]->start();//ejecutamos un hilo
}

?>
```

## 1.2. Semáforos en PHP

PHP hace uso del modulo Semaphore, este módulo proporciona envolturas para la familia de funciones de IPC de System V, estas funciones son:

- sem_get: Devuelve un id que se puede usar para acceder al semáforo.
- sem_adquire: Bloquea el semáforo.
- sem_release: Desbloquea el semáforo.
- sem_remove: Borra el semáforo

Un ejemplo sería el siguiente:

```php
<?php

print "<pre>\n";

// Clave de ejemplo para el semáforo
$clave_semaforo = 33;

$sem_id = sem_get ($clave_semaforo, 1);
print "Se obtuvo el ID de semaforo $sem_id\n";

print "Esperando al semaforo...\n";

//bloquea el semáforo
if (! sem_acquire ($sem_id))
    die ('Ocurrio un fallo esperando al semáforo.');

//******************************************************
//aqui se hacen las sentencias protegidas por los semáforos
//******************************************************

//se desbloquea el semáforo
if (! sem_release ($sem_id))
    die ('Ocurrio un fallo liberando la via');
?>
```

## 1.3. Asincronía

Realmente, no existe la asincronía en php, de hecho, para laravel es recomendable usar colas. La idea básica es que en un script PHP coloca las tareas en una cola. Luego, se tienen una cola de tareas que se ejecutan en en otro momento, esta cola saca tareas y se procesa independientemente del PHP original. Básicamente va generando un proceso para cada tarea de la cola.

No obstante existen librerías de la comunidad para trabajar con asincronía, como "Asynchronous and parallel PHP" o "Amp"

# 2. Javascript

JavaScript (JS) es un lenguaje de programación interpretado que proviene de Java y Self. Se emplea para la lógica de negocio de entornos web, creación de juegos en la web, aplicaciones móviles... La mayoría de aplicaciones de JS son para el lado del cliente, no obstante, también se usa en el lado del servidor para permitir desarrollos web completos, para ello se usan herramientas como Node.js.


Figura 1: El juego del dinosaurio de Google esta hecho en JS

JavaScript utiliza una arquitectura basada en eventos que se ejecuta en un único hilo. A continuación veremos las dos versiones de concurrencia, la programación basada en eventos y la "programación multihilo".

## 2.1. Asincronía

Como ya hemos dicho, JS usa una arquitectura basada en eventos, estos eventos pueden ser desde un click de un botón hasta un error en una base de datos. Son tareas asíncronas, es decir que no sucede al mismo tiempo, los eventos son gestionados por un loop de eventos llamado "EventLoop". Cabe destacar que la programación basada en eventos no es concurrencia, de hecho a esta se le llama la falsa concurrencia ya que "emula una programación concurrente", dicho esto, a continuación explicaremos el loop de eventos y la sincronía en JS.

### 2.1.1. Funcionamiento de EventLoop

Event loop funciona en conjunto a una pila donde guarda las operaciones a realizar (call stack) y una cola donde se almacenan las funciones asíncronas para ejecutarse usando FIFO (callback queue).

Poniendo un ejemplo, en un código EventLoop pondría las funciones síncronas en la pila de llamadas y las irá ejecutando. Si ocurre un evento, se coloca la función en la callback queue y esperará a que sea el primero de la cola y que no haya ninguna función en la pila. Una vez terminada las funciones de la pila, EventLoop irá seleccionando la primera función de la cola y la pasará a la pila, repitiéndose así el proceso hasta ejecutar todas las secuencias.

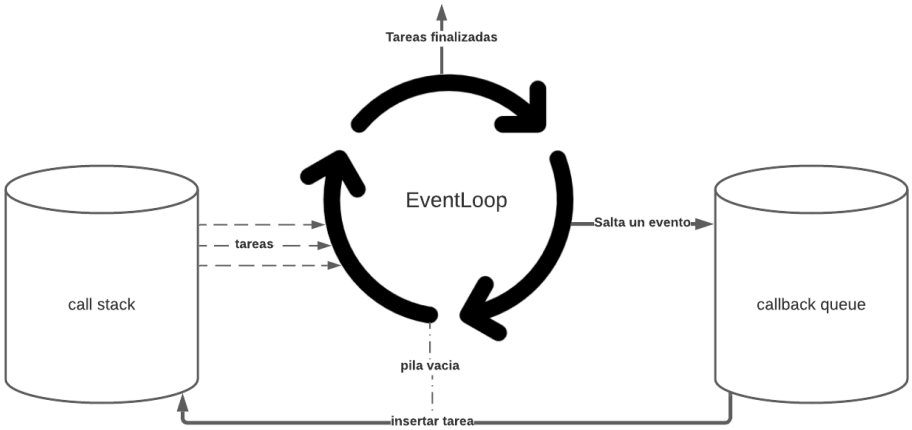


Figura 2: Esquema del funcionamiento de EventLoop

### 2.1.2. Promesas

Una promesa es una promesa (como su nombre indica) de que una acción se va a realizar, esta puede dar como resalado que se realice o que no.

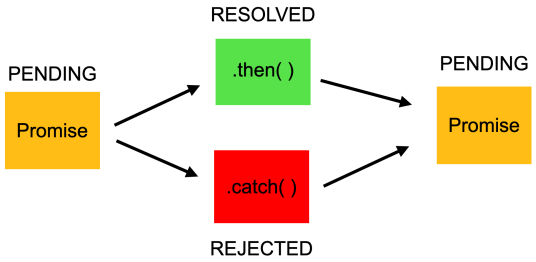


Figura 3: Esquema de una promesa

Como se observa en la figura 3, .then(función)acción se usaría para escribir la premisa si se cumple la promesa, si no se usaría .catch(función)acción.

```
fetch("/robots.txt")
  .then(function(response) {
    /* Código a realizar cuando se cumpla la promesa */
  })
  .catch(function(error) {
    /* Código a realizar cuando se rechaza la promesa */
  });
```

También, es posible generar nuestras propias promesas con new Promise(), a continuación veremos un ejemplo del uso de una prosa.

```
/* Creación de la promesa */
const doTask = (iterations) => new Promise((resolve, reject) => {
  const numbers = [];
```

```
  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      reject({//rechazo de la promesa
        error: true,
        message: "Se ha sacado un 6"
      });
    }
  }
  resolve({//acierto de la promesa
    error: false,
    value: numbers
  });
});

/* Llamada a la funcion */
doTask(10)
  .then(result => console.log("Tiradas correctas: ", result.value))
  .catch(err => console.error("Ha ocurrido algo: ", err.message));
```

En este caso la función doTask, genera n números aleatorios y si es 6 uno de ellos, salta un error. Hace uso de la sentencia reject para rechazar la promesa y la sentencia resolve para decir que la promesa es valida.

### 2.1.3. Async/await

Async/await llevan el uso de las promesas a un nivel superior, ya que nos las permiten controlar a nuestro gusto y hace más intuitivo el uso de promesas.

La palabra async sirve para definir una función como asíncrona:

```
/* Funcion asincrona */
async function funcion_asincrona1() {
  return 42;
}
/* Funcion en flecha asincrona */
const funcion_asincrona2 = async () => 42;
```

La palabra await hace que continúe ejecutándose otra tarea mientras espera a que se cumpla la promesa. Por ejemplo, si el ejemplo anterior lo implementamos como async/await quedaría de la siguente forma:

```
/* Llamada con async */
const doTask = async (iterations) => {
  const numbers = [];
  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      return {
        error: true,
        message: "Se ha sacado un 6"
      };
    }
  }
```

```
  return {
    error: false,
    value: numbers
  };
}

/* Llamada a la funcion con await */
const resultado = await doTask(10);   // Devuelve un objeto, no una promesa
```

Cabe destacar que solo se puede usar await dentro de funciones asíncronas, a continuación un código de ejemplo que obtiene la imagen del avatar de githup:

```
async function showAvatar() {

  // leer nuestro JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // leer usuario github
  let githubResponse = await fetch('https://api.github.com/users/${user.name}');
  let githubUser = await githubResponse.json();

  // muestra el avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // espera 3 segundos
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}
```

Por ultimo y como dato curioso en los navegadores modernos si que podemos usar await fuera de una función asíncrona siempre y cuando usemos módulos (una forma de importar scipts), si no tendríamos que abrazarlo dentro de un async.

```
//ejemplo de un navegador nuevo
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
. . .

console.log(user);

//**********************************************************************

//ejemplo de un navegador antiguo
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  . . .
})();
```

```
16
17 console.log(user);
```

## 2.2. Hilos en JS

JavaScript no usa programación multihilo, el motor dedica un único hilo para cada script que se ejecute, es decir, dos scripts dos hilos y así sucesivamente. Esto es así, ya que como ya hemos mencionado JS utiliza un arquitectura basada en eventos que emplea un único hilo. Sin embargo, debido a las web cada vez más dinámicas y todas las llamadas a servidor que se requieren hoy en día se necesitaría muchas llamadas de eventos, por esto, existen librerías públicas creadas por la comunidad como "Concurrent.Thread" y adaptaciones en el navegador como "Web Workers" para simular una programación multihilo en JS. algunas de ellas son:

### 2.2.1. La librería Concurrent.Thread

En la web de código abierto `https://sourceforge.net` el usuario Daisuke Maki publico en 2015 la librería Concurrent.Thread que permite simular la ejecución en múltiples hilos. Su funcionamiento es simple:

El siguiente código, la función imprimir, incrementa e imprime en número i que se pasa por parámetro. Esta función dejara colgado al navegador mientras ejecuta esta función.

```
1 function imprimir (i){
2     while (1) {
3         console.log(i);
4         i++;
5     }
6 }
```

Para solucionar esto, se declara la función como hilo con "Concurrent.Thread.create", así que cada vez que se ejecute la función, esta lo hará en otro hilo.

```
1 Concurrent.Thread.create(function(){
2     var i = 0;
3     while ( 1 ) {
4        document.body.innerHTML += i + "<br />";
5        i++;
6     }
7 });
```

# 3. C#

C# es un lenguaje de programación orientado a objetos desarrollado Microsoft como parte de su plataforma .NET. Esa influido principalmente por la familia de C/C++ aunque utiliza el modelo de objetos similar al de Java.

Para el desarrollo web se usa en ASP.NET que es un framework de Microsoft para el desarrollo web este usa C#. La concurrencia en este framework es amuy usada sobretodo para accesos a bases de datos o para ejecuciones de funciones con alto coste en la web.

## 3.1. Hilos en C#

Para trabajar con hilos, necesitamos hacer uso de la librería System.Threading. Para crearlos crearíamos un objeto de tipo Thread pasándole la función del hilo, luego con la función start() empezaría la ejecución del hilo. En es siguiente código podemos observar un ejemplo.

```
1  using System;
2  using System.Threading;
3
4  class PruebaHilos{
5
6      //codigo del hilo
7      static void EscribePalabra(string palabra){
8          while (true) Console.WriteLine(palabra);
9      }
10
11     static void Main(){
12         Thread t = new Thread(EscribePalabra);//creamos un nuevo hilo con la funcion EscribeY
13         t.Start("hilo"); // elecuta un hilo nuevo con el parametro de tipo string.
14         while (true) Console.WriteLine("hilo principal");
15     }
16
17 }
```

## 3.2. Sincronización

Para sincronizar hilos para acceder a una sección crítica, existen dos formas de hacerlas en C#, con un semáforo o con un mutex.

### 3.2.1. Mutex

Mutex se usa cuando solo se requiere a un hilo en la sección crítica. Haríamos uso de la librería System.Threading

```
1  using System;
2  using System.Threading;
3
4  class Example{
5      private static Mutex mut = new Mutex();
6      private const int numIterations = 1;
7      private const int numThreads = 3;
8
9      static void Main(){
10
11         for(int i = 0; i < numThreads; i++){ //creación de hilos
```

```
            Thread newThread = new Thread(new ThreadStart(ThreadProc));
            newThread.Name = String.Format("Thread{0}", i + 1);
            newThread.Start();
        }
    }

    private static void ThreadProc(){//funcion del hilo
        for(int i = 0; i < numIterations; i++){
            UseResource();
        }
    }

    private static void UseResource(){//metodo que simula el uso de un recurso
        Console.WriteLine("hilo {0} esta solicitando permiso al mutex", Thread.CurrentThread.Name);

        mut.WaitOne();//esperamos a que el recurso este libre

        Console.WriteLine("hilo {0} ha entrado en el área protegida", Thread.CurrentThread.Name);

        Thread.Sleep(500);// Simulamos un trabajo

        Console.WriteLine("hilo {0} ha salido de área protegida", Thread.CurrentThread.Name);

        mut.ReleaseMutex();//liberamos el recurso
        Console.WriteLine("hilo {0} ha liberado el recurso", Thread.CurrentThread.Name);

    }
}
```

En el código de ejemplo se hace uso de la función del objeto mutex WaitOne(), esta actuaría como una barrera que en el caso de no haber nadie en la sección crítica permitiría el paso, en caso contrario se pondrían a la cola los hilos esperando a que el hilo de la sección crítica libere el mutex con la función ReleaseMutex().

También cabe destacar que es posible hacer que el hilo se espere solo una cierta cantidad de tiempo, esto se hace pasándole los segundos como parámetro a WaitOne(). En el siguente ejemplo veremos una modificación de la función UseResource() del ejemplo anterior pero con un encolamiento de 1 segundo.

```
private static void UseResource(){
        // esperamos un segundo para entrar en la sección critica, si supera la espera, no accede
        Console.WriteLine("hilo {0} esta solicitando permiso al mutex", Thread.CurrentThread.Name);
        if (mut.WaitOne(1000)) {
           Console.WriteLine("hilo {0} ha entrado en el área protegida", Thread.CurrentThread.Name);

           Thread.Sleep(5000);// Simulamos un trabajo

           Console.WriteLine("hilo {0} ha salido de área protegida", Thread.CurrentThread.Name);

              mut.ReleaseMutex();//liberamos el recurso
           Console.WriteLine("hilo {0} ha liberado el recurso", Thread.CurrentThread.Name);
        }
        else {
           Console.WriteLine("hilo {0} no ha podido entra en la sección protegida", Thread.CurrentThread.Name);
        }
    }
```

### 3.2.2. Semáforo

Si se desea que varios hilos accedan a la sección crítica haremos uso del objeto Semaphore también de System.Threading, este semáforo se le indica por parámetro el número inicial de solicitudes que se pueden conceder simultáneamente y el número máximo de solicitudes que se pueden conceder simultáneamente Semaphore(Int32, Int32). El funcionamiento es igual que el mutex, pero la diferencia es que WaitOne() decrementa el contador interno de los recursos en la sección crítica, si estos es 0 no deja pasar.

El siguente ejemplo es una simulación de una tienda con tres dependientes, de tal modo que tres serán el máximo de hilos en la sección critica

```
using System;
using System.Threading;
{
    public class Program
    {
        static Thread[] threads = new Thread[10];
        static Semaphore sem = new Semaphore(3, 3);

        public static void Main()
            for (int i = 0; i < 10; i++){//creación de los hilos

                threads[i] = new Thread(mostrador);
                threads[i].Name = "Persona" + i;
                threads[i].Start();
            }
            Console.Read();
        }

        public static void mostrador()
        {
            Console.WriteLine("{0} en Fila", Thread.CurrentThread.Name);
            sem.WaitOne();
            Console.WriteLine("{0} en atencion", Thread.CurrentThread.Name);
            Thread.Sleep(300);//sección critica
            Console.WriteLine("{0} Saliendo", Thread.CurrentThread.Name);
            sem.Release();
        }
    }
}
```

## 3.3. Asincronía

La librería System.Threading.Tasks es una clase que nos permite ejecutar tareas de manera asíncrona. el funcionamiento es muy similar a el de JavaScript, usa async y await, ya que las funciones asíncronas se encabezarían como async y los elementos de espera se colocaría await, pero la diferencia está en que las funciones asíncronas se declaran con el tipo Task, y que en c# al no tener la restricción como en JS de usar un único hilo, cada tarea invoca a un hilo. A continuación, tenemos un ejemplo que emula a una consulta en una base de datos.

```
static async Task consulta_BD()
{
    Console.WriteLine("conectando a la BD");
    await Task.Delay(3000);
    Console.WriteLine("Hacieno consulta");
```

```
    await Task.Delay(3000);
    Console.WriteLine("Consulta hecha");
}
```

También la librería System.Threading.Tasks nos facilita el trabajar con mas de una función asincrona, ya que nos permite crear listas de tipo tarea (List <Task>) esa lista de tareas la podemos usar con la función WhenAny() que inicia todas las tareas y luego espera que se completen todas recopilando todos los resultados en orden de finalización, por lo que no se conservará el orden original. También esta WhenAll() que es como al anterior pero recopila todos los resultados en el orden correcto es decir, en el de las llamadas.

A continuación veremos un código de como se prepara unos huevos con bacón de manera asincrona, hacemos uso de una lista de tareas y WhenAny() para que las preparaciones finalicen cuando tengan que finalizar.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncLunch
{
    class Program
    {
        static async Task Main(string[] args)
        {

            var eggsTask = FryEggsAsync();
            var baconTask = FryBaconAsync();

            var breakfastTasks = new List<Task> { eggsTask, baconTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("los huevos estan listos");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("el bacon esta listo");
                }
                breakfastTasks.Remove(finishedTask);
            }

        }

        private static async Task<Bacon> FryBaconAsync()
        {
            Console.WriteLine("calentando la sarten...");
            await Task.Delay(3000);
            Console.WriteLine("cocinando bacon");
            await Task.Delay(3000);
            Console.WriteLine("poniendo bacon en el plato");

            return new Bacon();
        }
```

```
42
43          private static async Task<Egg> FryEggsAsync()
44          {
45              Console.WriteLine("calentando la sarten...");
46              await Task.Delay(3000);
47              Console.WriteLine("rompiendo los huevos");
48              await Task.Delay(3000);
49              Console.WriteLine("cocinando los huevos");
50              await Task.Delay(3000);
51              Console.WriteLine("poniendo los huevos en el plato");
52
53              return new Egg();
54          }
55      }
56  }
```

# 4. Conclusiones

Para finalizar el documento me gustaría dar una opinión personal de estos tres lenguages. Para el desarrollo de web, he podido observar que como elementos de concurrencia se usa más la asíncronia (pese a no ser concurrencia) que la programación multihilo, tanto JavaScript como C# al ser lenguages mas modernos facilita esta asíncronía, PHP sin embargo se queda un poco atrás en este aspecto, de hecho el que para sus semáforos use la familia de System V demuestra esto. También puedo concretar que C# seria el lenguage que facilita mas la programación concurrente.