

MARC ALCÓN MELIÁ
CRISTINA GARCÍA ARMIJO

CONCURRENCIA EN UNITY



ÍNDICE

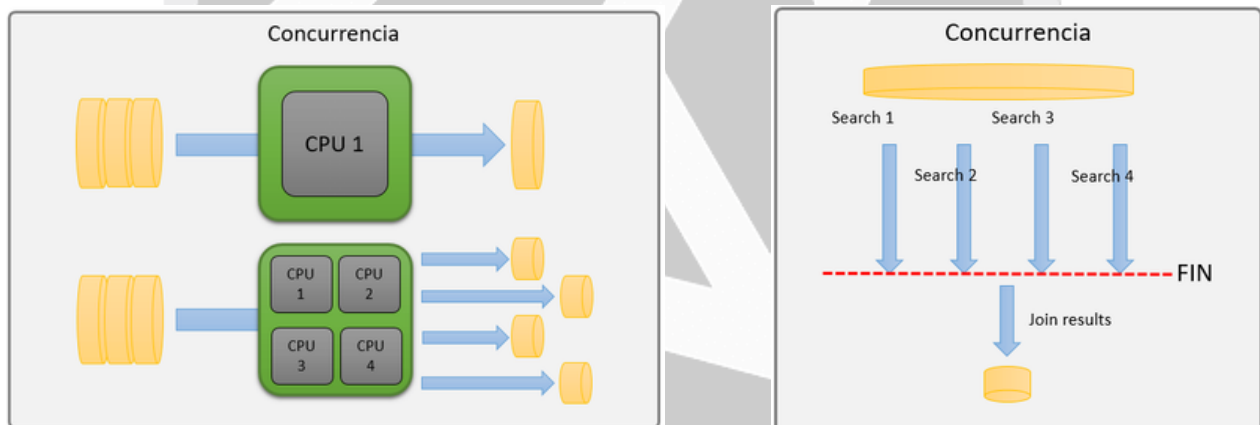
¿QUÉ ES LA CONCURRENCIA?	3
CORRUTINAS	5
¿QUÉ SON?	5
¿CÓMO SE IMPLEMENTAN?	7
THREADS	11
¿QUÉ SON?	11
¿CÓMO SE IMPLEMENTAN?	13
JOBS	20
¿QUÉ SON?	20
¿CÓMO SE IMPLEMENTAN?	23
¿QUÉ MECANISMO ES MEJOR?	28

¿QUÉ ES LA CONCURRENCIA?

Como punto de partida para este documento es importante conocer que la **concurrency** es la **habilidad que tienen las distintas partes de un programa o algoritmo para ser ejecutado en desorden o en orden parcial, pero sin afectar de ninguna forma al resultado final de éste.**

Dichos cálculos se ejecutarán en los diferentes procesadores del ordenador mediante el uso de **distintos hilos de ejecución.**

De esta manera se consigue que los distintos hilos de ejecución interactúen entre ellos mientras se están ejecutando.



Pero esto tiene un problema, ya que que haya diferentes rutas de ejecución tiene como consecuencia que el resultado final del programa será completamente indeterminado.

Esto quiere decir que **no se podrá saber con certeza el orden de ejecución de estos hilos** (en qué orden actúan, cuales acaban antes, cual es el último hilo en actuar...) produciendo errores como condiciones de carrera, bloqueos mutuos o problemas de inanición los cuales han de resolverse por el programador para el correcto funcionamiento del código implementado.

CONCURRENCIA EN UNITY

Durante este documento se tratará en profundidad las diferentes formas de aplicar la concurrencia en Unity, sus puntos a favor y en contra partiendo de un nivel básico-medio de Unity.

Tras aplicar la concurrencia en Unity (cuando sea necesaria, ya que utilizarla en zonas en las que el tratamiento de datos sea muy bajo conseguirá lo contrario al objetivo ya que el coste de crear varios procesos será mayor que el beneficio obtenido por tratarlos concurrentemente) se obtendrá una gran **optimización del código** propuesto por medio de diferentes estructuras y/o métodos, los cuales **serán procesados en CPUs diferentes** para conseguir ser ejecutados en menor tiempo.

Para ello, Unity dispone de tres mecanismos diferentes que cada programador puede utilizar en su proyecto:

1-. CORRUTINAS

2-. THREADS

3-. JOBS

CORRUTINAS

¿QUÉ SON?

En primer lugar se empezará hablando de las corrutinas, cómo funcionan, cómo se aplican y qué beneficio pueden llegar a aportar al código una vez implementadas de forma correcta y eficaz.

Normalmente, cuando se ejecuta una función, todo lo que halla en su interior ocurrirá dentro de un mismo "frame update", ocasionando en diferentes ocasiones, un resultado diferente al esperado, ya que no podrán ocurrir una secuencia de eventos durante el tiempo, sino que ocurrirán dentro de un mismo frame de la forma más veloz posible.

Pongamos por ejemplo que lo que el usuario quiere es reducir la opacidad (el alfa) de un objeto gradualmente hasta que sea totalmente transparente, para eso se usaría una función como la siguiente:

```
void Fade()  
{  
    for (float ft = 1f; ft >= 0; ft -= 0.1f)  
    {  
        Color clr = gameObject.material.color;  
        clr.a = ft;  
        gameObject.material.color = c;  
    }  
}
```

El resultado esperado es la reducción del alfa del material del objeto gradualmente, pero tras ejecutar esta función el objeto pasará de ser visible a ser transparente en un solo frame.

Aquí es donde entran en juego las corrutinas, aportando una solución sencilla al problema.

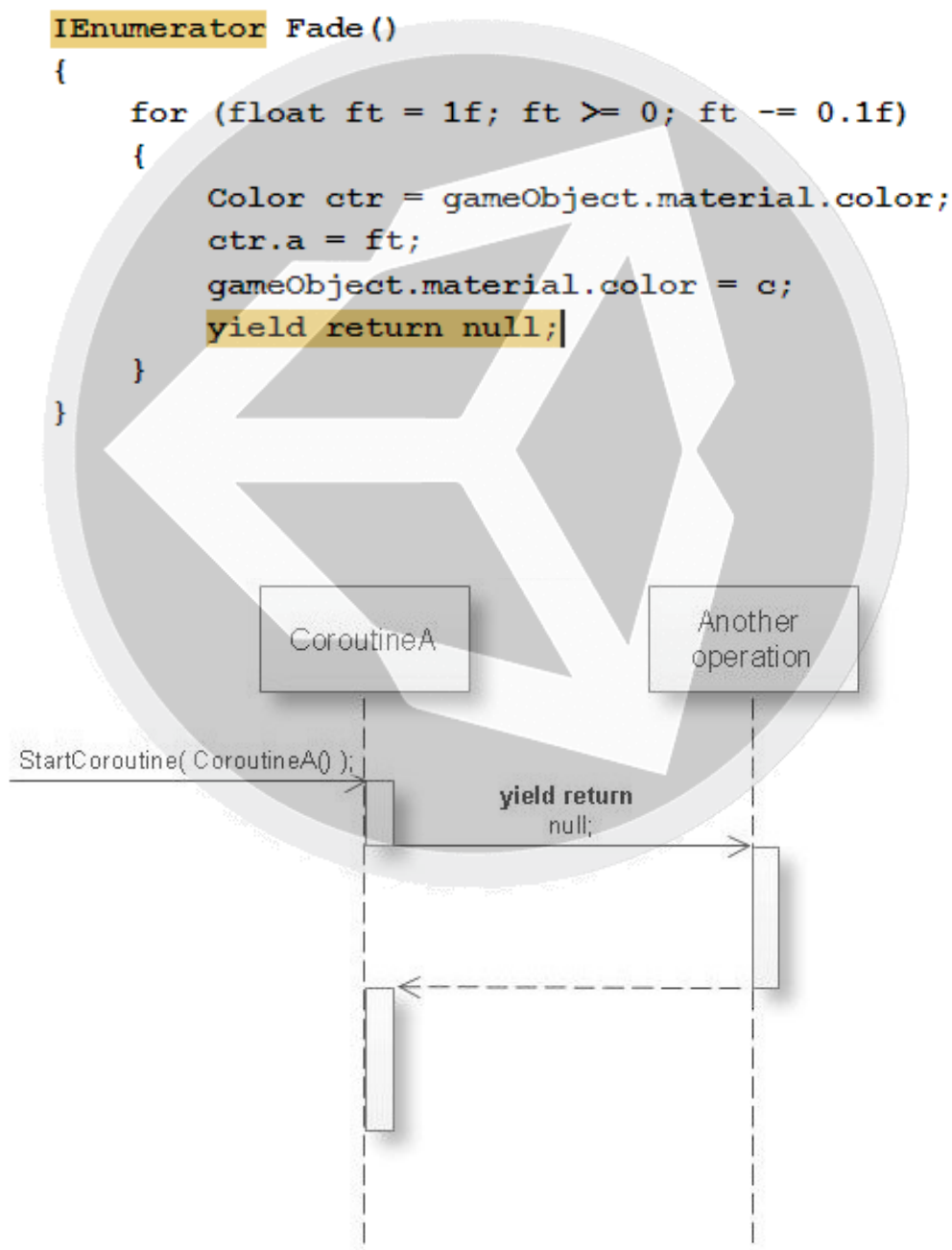
Una corrutina es una función que puede parar su ejecución y devolverle a Unity el control antes de acabar, consiguiendo por ello actuar en segundo plano mientras el programa sigue ejecutando el método principal.

CONCURRENCIA EN UNITY

La forma correcta de aplicar una corrutina al código descrito anteriormente es:

- Cambiar el tipo de función a corrutina cambiando void por IEnumerator.
- Utilizar yield return y la cantidad de tiempo que se quiere que transcurra.

Obteniendo como resultado :



CORRUTINAS

¿CÓMO SE IMPLEMENTAN?

Es importante conocer que las **corrutinas no funcionan como las funciones normales**, ya que estas no devuelven floats, strings, booleanos... **devuelven un IEnumerator**, una variable de tipo .NET para pausar la iteración.

```
IEnumerator Fade()  
{  
    for (float ft = 1f; ft >= 0; ft -= 0.1f)  
    {  
        Color c = renderer.material.color;  
        c.a = ft;  
        renderer.material.color = c;  
        yield return null;  
    }  
}
```

Y ahora se van a explicar las diferentes funciones que caracterizan a las corrutinas:

StartCoroutine(): Una vez creada la corrutina se deberá llamar desde el código principal para que funcione, utilizando StartCoroutine.

```
void Update()  
{  
    if (Input.GetKeyDown("f"))  
    {  
        StartCoroutine("Fade");  
  
        // resto del código...  
    }  
}
```

CONCURRENCIA EN UNITY

De esta manera se consigue que la corrutina "Fade" sea activada tras pulsar la tecla F la cual, al llegar a "yield return null", devolverá el control al programa principal pero ejecutándose como un bucle for con una iteración por frame (sin ocasionar errores debido al tratamiento de los parámetros de ese "bucle for" de forma segura y correcta entre "yields").

WaitForSeconds: Suspende la corrutina durante una cantidad determinada de tiempo. Se aplica cuando queremos alargar el efecto de una corrutina en el tiempo.

```
IEnumerator Fade()  
{  
    for (float ft = 1f; ft >= 0; ft -= 0.1f)  
    {  
        Color c = renderer.material.color;  
        c.a = ft;  
        renderer.material.color = c;  
        yield return new WaitForSeconds(.1f);  
    }  
}
```

Ahora el objeto tardará más tiempo en pasar a ser transparente, ya que se esperará 0.1 segundos entre cada una de las iteraciones del bucle en vez de ejecutarse a cada frame, mientras el código principal se sigue ejecutando con normalidad.

De esta forma se puede observar como las corrutinas tienen un uso primordial ya que se encargarán de optimizar el código de forma eficiente.

Aplicado al ámbito de los videojuegos, hay numerosas tareas que deben llevarse a cabo de forma periódica o continua por lo que deberían incluirse en Update() (que se ejecuta varias veces por segundo), pero en ocasiones, a pesar de que se buscan tareas que se ejecuten periódicamente, es necesario que lo hagan menos frecuentemente o incluso en tiempos diferentes entre ellas.

CONCURRENCIA EN UNITY

Una tarea que cumple estos requisitos podría ser la siguiente, que verifica si el jugador ha entrado en la distancia de peligro con un enemigo:

```
function ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
    {
        if (Vector3.Distance(transform.position,
enemies[i].transform.position) < dangerDistance) {
            return true;
        }
    }
    return false;
}
```

Como se trata de una función que no se quiere actualizar a cada frame, ya que el jugador no se mueve tan rápido, en vez de añadirla en la función Update se creará una corrutina para llevarla a cabo:

```
IEnumerator DoCheck()
{
    for(;;)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}
```

Al llamar a la corrutina "DoCheck()" en el código, se irá comprobando el ProximityCheck() cada 0.1s, **liberando así al juego de hacer una gran cantidad de checks cada frame sin que haya ningún impacto significativo en el videojuego**, ya que todas las comprobaciones que hubiera hecho si hubiera estado en el Update en vez de ser una corrutina no hubieran hecho más que ralentizar el juego, ya que eran comprobaciones que no iban a afectar de manera efectiva en el juego ya que el jugador no tienen tanta velocidad como para que esa comprobación se tenga que realizar a cada frame.

CONCURRENCIA EN UNITY

Otras funciones referentes a las corrutinas que el programador debería saber o conocer a la hora de aplicarlas son estas:

StopCoroutine(): Desactiva una corrutina en concreto. Se utiliza de igual forma que StartCoroutine(), siendo incluida en el programa principal.-

StopAllCoroutines(): Desactivar todas las corrutinas de un GameObject. Si se desactiva un GameObject directamente, todas sus corrutinas quedan desactivadas también, pero al hacerlo de esa forma puede dar lugar a errores, así que mejor usar este método.

Yield return null: Suspende la corrutina, para reanudarla en el siguiente frame. Esto tiene la misma funcionalidad que WaitForSeconds solo que aquí no se esperará una cantidad determinada de tiempo entre iteración e iteración, sino que cuando acabe una se continuará con la siguiente sin ningún espacio entre iteraciones.

THREADS

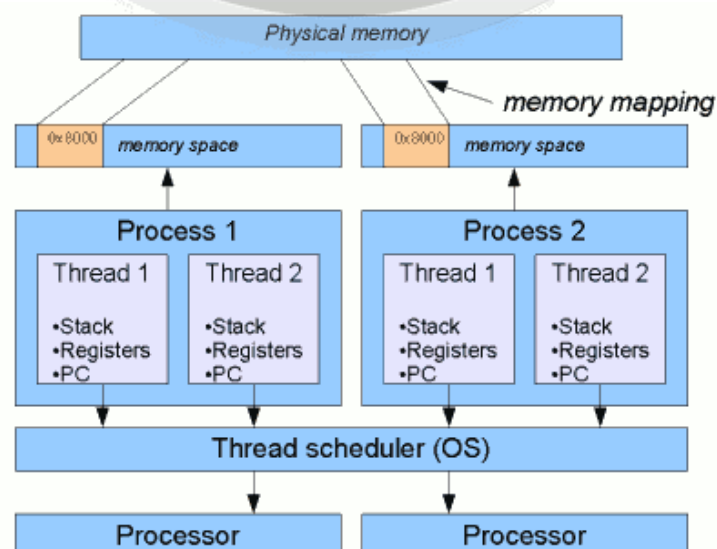
¿QUÉ SON?

Al igual que las corrutinas, los threads son otra forma que tiene Unity de realizar concurrencia, se va a proceder a explicar cómo funcionan y qué beneficios tienen.

En primer lugar y al contrario que las corrutinas, el threading (usar threads) toma ventaja de la estructura **multicore** que tienen hoy en día las CPUs más modernas de **realizar dos o más tareas de forma concurrente y eficaz**.

Hacer uso del threading en Unity acarrea el uso de características tales como poder **derivar cálculos muy complejos mediante threads a otras CPUs** para que se siga realizando el cálculo en segundo plano mientras se mantiene la Interfaz del Usuario activa y que responda al jugador. También se hace uso del threading para **hacer la misma tarea mediante diferentes algoritmos y comprobar su eficacia**.

Y finalmente, otra de las características del threading es **dividir una tarea muy larga en diferentes tareas más pequeñas, ejecutarlas por separado con diferentes threads en diferentes CPUs y usando memoria compartida, y finalmente juntarlas todas al acabar mostrando así el resultado total**.



CONCURRENCIA EN UNITY

¿Y qué aporta utilizar threading o multithreading en Unity? Gracias a los threads, **un videojuego puede aumentar de manera significativa sus FPS**, ya que los threads dan lugar a un cálculo de operaciones de manera más estructurada, consiguiendo así un beneficio muy significativo en la fluidez del juego, lo cual debe ser un objetivo primordial para todo programador a la hora de programar un videojuego.

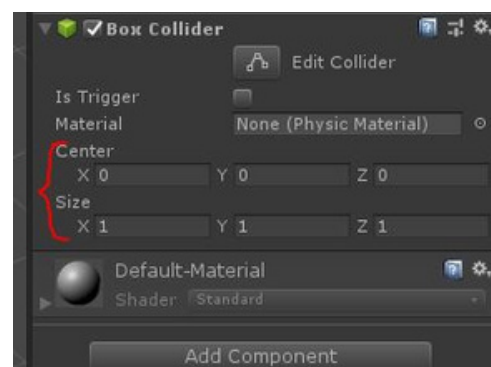
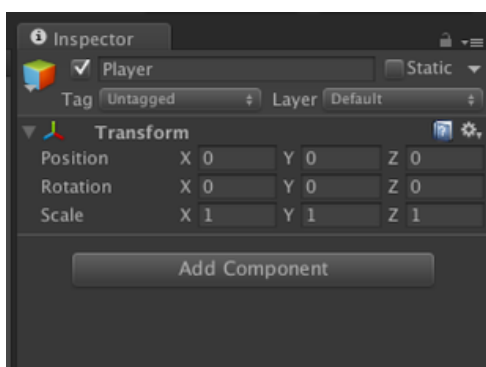
Aún así, usar threads y hacer uso del threading puede comprometer el buen funcionamiento del programa, por lo que es necesario tener muy claro su uso y su aplicación.

Una de las propiedades de los threads es que utilizan **memoria compartida** para todos los threads, y eso puede dar lugar a comportamientos inesperados si no se toman las precauciones adecuadas al acceder o modificar dicha memoria, las llamadas condiciones de carrera.

Por eso, toda operación de threading debe estar bien protegida y asegurada.

Y antes de proseguir y para finalizar este apartado es necesario hablar sobre la API de Unity.

La API de Unity recoge todos aquellos objetos o métodos que son importados mediante Unity Engine, Unity Editor, Unity o Other, que son los espacios que provee Unity para poder trabajar con su motor. Una de sus características principales es que no es Thread safe, significando que no es seguro aplicar Threads en ella, por tanto, **referencias como transform.position o usar los diferentes componentes que puede tener un GameObject en Unity (Colliders, Sprite Renderers...) no pueden ser usados para aplicar multithreading**, al no ser esos parámetros accesibles por los diferentes threads (solo por el "main thread") y por tanto no pueden ser usados para llevar a cabo cálculos ni operaciones.



THREADS

¿CÓMO SE IMPLEMENTAN?

Para aplicar threading en Unity, al contrario que las corrutinas que no necesitan ningún "namespace" (colección de clases importadas al usar un prefijo escogido), el uso de threads y multithreading en Unity necesita en primer lugar:

Threading namespace: para usarlo lo importamos al principio del script usando el siguiente fragmento de código, pudiendo así hacer uso de todas las funciones propias de los threads.

```
using System;  
using System.Threading;
```

Crear un thread: es importante que los threads se creen todos desde el main thread para que no se origine ningún error.

En este ejemplo se crean dos threads y a cada uno le asignaremos un trabajo diferente (que será una función) para que lleven a cabo cada uno por separado, como podemos ver en el fragmento siguiente:

```
//Este trozo se añadirá en el main thread  
Thread ThreadOne = new Thread(Work1);  
Thread ThreadTwo = new Thread(Work2);  
  
void Work1()  
{  
    for(int i = 1; i <=10; i++)  
    {  
  
        Debug.Log("Work 1 is called " + i.ToString());  
  
    }  
}
```

CONCURRENCIA EN UNITY

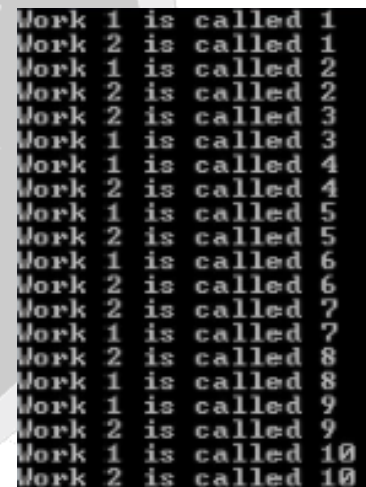
```
void Work2()  
{  
    for (int i = 1; i <= 10; i++)  
    {  
  
        Debug.Log("Work 2 is called " + i.ToString());  
  
    }  
  
}
```

Si llamamos a las funciones por separado la salida que tendrá el programa sería primero el work 1 imprime los números del 1 al 10 y luego el work 2 hace lo mismo.

Thread.Start(): sirve para iniciar el trabajo de un thread una vez éste ha sido creado.

```
Thread ThreadOne = new Thread(Work1)  
Thread ThreadTwo = new Thread(Work2)  
  
ThreadOne.Start();  
ThreadTwo.Start();
```

Este es el resultado que se obtendría si se usan threads en vez de hacer las funciones de forma iterativa:



The screenshot shows the Unity console output where the logs for Work 1 and Work 2 are interleaved, demonstrating concurrent execution. The logs are as follows:

Thread	Iteration	Log Message
Work 1	1	Work 1 is called 1
Work 2	1	Work 2 is called 1
Work 1	2	Work 1 is called 2
Work 2	2	Work 2 is called 2
Work 2	3	Work 2 is called 3
Work 1	3	Work 1 is called 3
Work 1	4	Work 1 is called 4
Work 2	4	Work 2 is called 4
Work 1	5	Work 1 is called 5
Work 2	5	Work 2 is called 5
Work 1	6	Work 1 is called 6
Work 2	6	Work 2 is called 6
Work 2	7	Work 2 is called 7
Work 1	7	Work 1 is called 7
Work 2	8	Work 2 is called 8
Work 1	8	Work 1 is called 8
Work 1	9	Work 1 is called 9
Work 2	9	Work 2 is called 9
Work 1	10	Work 1 is called 10
Work 2	10	Work 2 is called 10

Pero no basta solo con iniciar un thread y darle un objetivo, también hace falta saber usar ciertos métodos para manejar dichos threads y que funcionen de forma correcta.

También se puede forzar un thread a que finalice de ejecutarse de forma rápida o abortar la ejecución de un hilo porque ya no se necesita que se siga ejecutando, a continuación se encuentran los métodos que permiten al programador poder realizar esas funcionalidades.

CONCURRENCIA EN UNITY

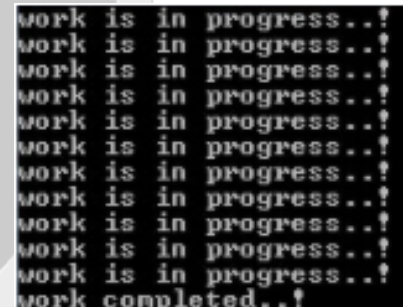
Thread.Join(): es un método que se usa para hacer que un determinado thread finalice su trabajo o para hacer que el resto de los threads que estén activos en ese momento se detengan hasta que el thread en cuestión haya finalizado su trabajo.

También hace que cuando se adjunta a un thread, este se ejecute primero, antes que los demás.

```
// Creamos el thread, lo ejecutamos y hacemos join
Thread ThreadOne = new Thread(MethodJoin);
ThreadOne.Start();
ThreadOne.Join();
Debug.Log("Work completed!");

static void MethodJoin()
{
    for (int i = 0; i <= 10; i++)
    {
        Debug.Log("Work in progress...");
    }
}
```

Y el resultado final obtenido será esta imagen que hay a la derecha:



```
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work is in progress..?
work completed..?
```

De esta forma se puede comprobar cómo, al hacer ThreadOne.Join(), el thread principal se detiene (ya que no imprime "Work completed"), ya que, como se ha explicado anteriormente, al hacer Thread.Join() se le dará prioridad a dicho thread y hasta que este no acabe el resto se quedarán paralizados sin avanzar, y una vez finalice este thread se reanudará la ejecución del resto de threads lo más rápido posible.

CONCURRENCIA EN UNITY

Thread.Sleep(): como su propio nombre indica, dormirá (suspenderá) el thread durante un intervalo específico de tiempo, que puede ser especificado en milisegundos o en intervalo de tiempo.

Al suspender un thread, este no consumirá recursos de la CPU, así que de forma indirecta se guarda memoria para utilizarla en los otros threads que se estén procesando.

```
Stopwatch stWatch = new Stopwatch();
stWatch.Start();

Thread ThreadOne = new Thread(ProcessSleep);
ThreadOne.Start();
ThreadOne.Join();

stWatch.Stop();
TimeSpan ts = stWatch.Elapsed;

string elapsed Time = String.Format("{0:00}:{1:00}:{2:00}",
ts.Hours, ts.Minutes, ts.Seconds);

Console.WriteLine("TotalTime " + elapsedTime);

Console.WriteLine("work completed..!");

static void ProcessSleep()
{
    for (int i = 0; i <= 5; i++)
    {
        Console.WriteLine("work is in progress..!");
        Thread.Sleep(4000); //Sleep for 4 seconds
    }
}
```

Y la salida del programa sería esta que se puede ver aquí:

```
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
TotalTime 00:00:24
work completed..!
```

Y así, cada vez que se imprima el mensaje de “work in progress..!” se esperará 4 segundos hasta que se realiza la siguiente iteración del bucle, ya que tras imprimir el mensaje el thread se duerme 4 segundos; y hasta que no finaliza este hilo, no se reanuda el principal por lo que no aparece el mensaje “work completed..!”.

CONCURRENCIA EN UNITY

Thread.Abort(): acaba o elimina un thread para que no pueda continuar ejecutándose. Lo que hace es que produce una excepción "ThreadAbortException" en el thread que se ha abortado para que se inicie en el thread principal la finalización del thread que se ha abortado.

```
TreadOne.Abort () ;
```

También es importante saber qué es y cómo controlar una **threadpool**.

Lo que hace una threadpool es que **mantiene numerosos threads en espera a una asignación de tarea** para que sean colocados por el programa y trabajen de forma concurrente.

De esta forma, se consigue incrementar la capacidad de ejecución y evitar la latencia en ejecución gracias a la creación y destrucción de threads para tareas cortas y de poca duración.

Se aplica de la siguiente manera; en primer lugar, se guardan en una variable todos los "inputs" o variables que necesitará la función a ejecutar en los múltiples threads; y en segundo lugar se llama a la threadpool.

```
var threadInput = ...; // Variables para la función
```

```
ThreadPool.QueueUserWorkItem(ThreadFuncion, threadInput);
```

Como se puede observar, al llamar a la threadpool no hay que crear los diferentes threads uno por uno, lo único que hay que hacer es encolar la función que quieres ejecutar con threads dentro de la threadpool con las variables que necesite y ella misma ya se encargará de usar los threads necesarios para llevar dicha función a cabo.

Por ello podemos concluir que es **más eficiente** usar una threadpool que usar threads por separado, ya que **no se gasta tiempo de ejecución creando los threads uno por uno**, y permite reutilizarlos si fuera necesario, haciendo que nunca haya límite de threads disponibles.

CONCURRENCIA EN UNITY

Ahora al lector le puede asaltar la duda de, **¿cómo podemos ayudar a la seguridad del threading?**

La solución viene de mano de una herramienta llamada **cerrojos**, que asegura que la información compartida entre threads no sea alterada erróneamente.

Aún así, el simple hecho de usar un cerrojo no va a garantizar la seguridad del threading; hay que identificar la información crítica que va a ser accedida y/o modificada por los diferentes threads y una vez conocida, se activará el cerrojo antes de acceder o modificarla para ser desactivado tras salir de la zona crítica.

De esta forma se garantiza que al mismo tiempo **solo un thread acceda o modifique dicha información** y evitando las condiciones de carrera, por lo que se necesita una idea clara del funcionamiento del código.

En este fragmento de código podemos observar un uso **incorrecto** de los cerrojos, ya que sigue dando lugar a una condición de carrera, ya en algunos casos se imprimirá "x times y is 1" y en otros casos "x times y is 0", debido a que lo conseguido ahora es que el thread principal y el nuevo thread "compitan" el uno con el otro para ver quién llega antes al cerrojo, pero no asegura que los resultados obtenidos sean fieles.

```
public class NotLockedThread : MonoBehaviour {  
  
    float x = 1f;  
    float y = 0f;  
  
    private static readonly object Lock = new object();  
  
    void Start () {  
        new Thread (Multiply).Start();  
  
        lock(Lock) {  
            Debug.Log("Main thread gets first");  
            y += 1f;  
        }  
        Multiply();  
    }  
  
    void Multiply(){  
        lock(Lock) {  
            Debug.Log("Secondary thread gets first");  
  
            Debug.Log("x times y is " + x*y);  
            y = x*y;  
        }  
    }  
}
```

CONCURRENCIA EN UNITY

Para implementar los cerrojos se debe comenzar **creando una variable especial de tipo readonly** (nos interesa que sea solo de lectura) para controlar cuándo está o no bloqueada una zona de código para el resto de threads.

Tras eso, hay que valorar cuál va a ser la **información crítica** a la que diferentes threads van a acceder para así usar la variable cerrojo bloqueando su entrada cuando un thread haga uso de ella, y desbloqueando su uso una vez el thread deje de acceder o modificar dicha información.

```
class Program
{
    static readonly object locked = new object();

    static void PrintInfo()
    {
        lock (locked)
        {
            for (int i = 1; i <= 4; i++)
            {
                Debug.Log("i value: {0}", i);
                Thread.Sleep(1000);
            }
        }
    }

    void Start
    {
        Thread t1 = new Thread(new ThreadStart(PrintInfo));
        Thread t2 = new Thread(new ThreadStart(PrintInfo));

        t1.Start();
        t2.Start();
    }
}
```

CON CERROJO:

```
i value: 1
i value: 2
i value: 3
i value: 4
i value: 1
i value: 2
i value: 3
i value: 4
```

SIN CERROJO:

```
i value: 1
i value: 1
i value: 2
i value: 2
i value: 3
i value: 3
i value: 4
i value: 4
```

Y la salida que proporciona el programa será que está arriba al lado del código. Efectivamente no se han imprimido los dos threads a la vez, ya que gracias al lock **primero ha accedido un thread completo y luego ha accedido el otro**.

Si no hubiera habido lock, la salida hubiera sido la que hay debajo, ya que se hubiera **realizado la ejecución de ambos al mismo tiempo**, por lo que ambos hubieran accedido a la misma función sin esperar a que el otro acabe.

JOBS

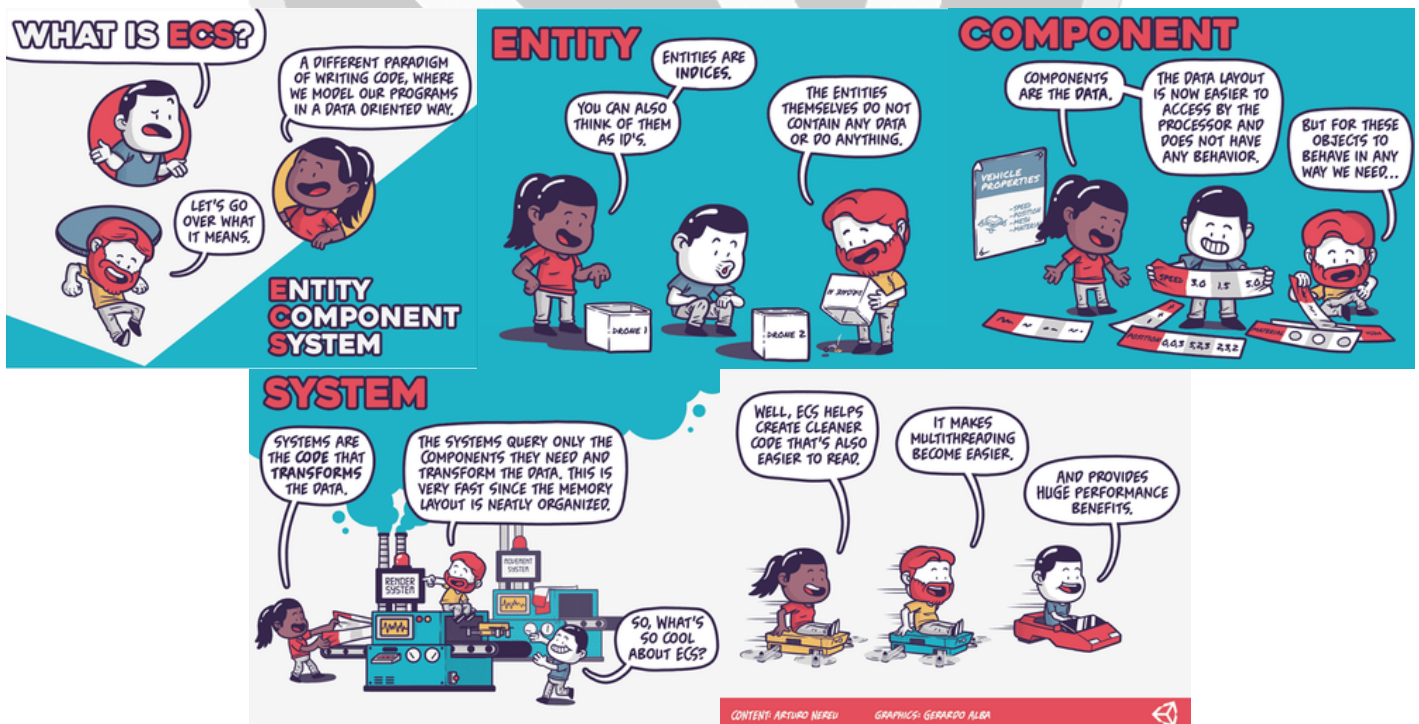
¿QUÉ SON?

Y para finalizar, se va a explicar y profundizar en los trabajos (jobs), qué son y cómo funciona el Job System en C# y por tanto en Unity.

El C# Job System de Unity permite al programador **escribir y aplicar multithreading de forma segura** y que interactúe con el Unity Engine para una **calidad de juego superior**.

El Job System es un sistema muy utilizado junto con el ECS (Entity Component System), una arquitectura que hace que sea más fácil escribir código para que funcione en todas las plataformas.

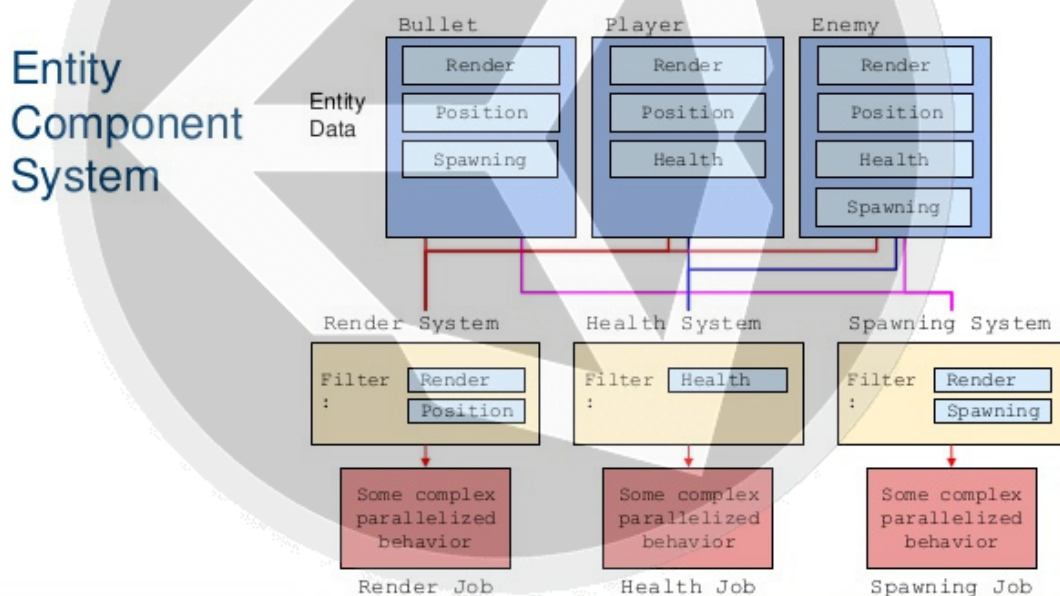
En **ECS**, la **E** se refiere a las entidades, las cosas que forman parte del juego, la **C** se refiere a los **componentes**, las variables asociadas a las entidades, y finalmente la **S** se refiere a los **sistemas**, la lógica que transforma los datos de los componentes de un estado al siguiente (mover entidades, cambiar su velocidad...).



CONCURRENCIA EN UNITY

El Job System se encarga de **utilizar multithreading usando jobs en vez de threads**. Un **job** es una estructura, una **unidad pequeña de trabajo** que **hace una tarea específica, recibe unos determinados parámetros y hace operaciones con ellos**, parecido a como funcionan las funciones estándar.

La diferencia importante no es lo que son los jobs en sí, sino como los dirige el Job System. El Job System dirige grupos de jobs en los diferentes cores del ordenador, y junto con un worker thread por cada CPU lógica que existe, encargado de evitar los famosos problemas de “context switching”. A parte de tener **un worker thread por cada CPU**, el **Job System encola todos los jobs para que se ejecuten**, y los **worker threads de cada CPU cogen los jobs que se encuentran en la cola y los ejecutan**, dirigiendo las diferentes dependencias de jobs y asegurando que se ejecuten en el orden correcto.



Las dependencias de jobs se dan cuando un job depende de otro para realizarse o para completarse.

De esta forma, si por ejemplo se tiene un jobA que tiene dependencia en un jobB para finalizar, el Job System se asegura de que **ese jobA no se empiece a ejecutar hasta que no haya terminado el jobB**, evitando una gran cantidad de errores comunes que puede original el multithreading.

CONCURRENCIA EN UNITY

Y como se ha comentado ya con anterioridad, otro de los problemas principales del uso del multithreading son las condiciones de carrera, que suelen causar bugs en muchos videojuegos. Son problemas difíciles de arreglar ya que dependen del código y en qué orden se ejecute cada thread, haciendo de las condiciones de carrera, como se ha comentado anteriormente, un problema muy significativo.

Por ejemplo, pongamos que un job manda una referencia de unos datos comunes del thread principal al thread actual; cómo se tratan de unos datos comunes no se puede verificar si cuando se está leyendo los datos necesarios, otro thread los está modificando en ese momento, ocasionando una condición de carrera al proporcionar a veces que los resultados no sean los esperados.

El Unity C# Job System detecta todas las potenciales condiciones de carrera y protege al programador y al programa de los posibles bugs que estas pudieran ocasionar. Lo que hace el C# Job System para solucionarlo es, en vez de mandar la referencia de los datos del main thread a los diferentes jobs, manda una copia de esos datos, de forma que los datos quedan totalmente aislados y no van a sufrir ninguna modificación y la condición de carrera queda eliminada.

JOBS

¿CÓMO SE IMPLEMENTAN?

Los jobs, al igual que los threads, también necesitan que se **importe su propio namespace** al principio del script para poder usarlos y usar sus métodos característicos.

```
using UnityEngine;  
using UnityEngine.Jobs;
```

Los jobs son estructuras, y como estructuras, heredan de una interfaz, en este caso de `IJob`. Al crear una estructura y hacerla heredar de `IJob` (creando así un job) se consigue que ese job individual se ejecute al mismo tiempo de forma concurrente con otros jobs y con el “main thread”.

Primero que todo, para crear un job se necesita:

- **Crear una estructura que implemente o herede de `IJob`.**
- **Añadir las variables que el job va a hacer servir** (hay que guardar las variables resultado en variables de tipo `NativeContainer`, a continuación se explicará porque).
- **Y finalmente crear un método dentro de dicha estructura job llamado “Execute” con el trabajo que llevará a cabo el job al ejecutarlo.**

Se debe tener en cuenta que cuando se ejecute un job, el método “Execute” se llevará a cabo una vez en una sola CPU.

```
// Job que realiza la suma de dos floats  
public struct MyJob : IJob  
{  
    public float a;  
    public float b;  
    public NativeArray<float> result;  
  
    public void Execute()  
    {  
        result[0] = a + b;  
    }  
}
```

CONCURRENCIA EN UNITY

La parte mala de la forma de la forma de solucionar las condiciones de carrera es que no solo **se usa más memoria que la que usan los threads**, sino que también, al copiar variables para cada job eso hace que se queden en cierta manera asilados, y aquí es donde **entran las variables NativeContainer**, que son un tipo de memoria compartida. Una variable NativeContainer es una variable que proporciona una relativa seguridad en C# para ser un tipo de memoria compartida. Cuando lo usamos con el C# Job System lo que hace la variable NativeContainer es permitir al job acceder a información compartida con el main thread en vez de trabajar con una copia. Hay diferentes tipos de variables NativeContainer (NativeArray, NativeList, NativeQueue...) pero todas tienen la misma finalidad.

La forma en la que se garantiza la seguridad de dichas variables que usan memoria compartida es que **se registratodo lo que se lee y escribe de un NativeContainer y se evalúa con las clases DisposeSentinel y AtomicSafetyHandle**, dos clases que se usan de forma automática por las variables NativeContainers.

DisposeSentinel se encarga de detectar de forma automática cualquier fuga de memoria que ocurra en cualquier momento, por tanto estas variables estarán siempre controlando que no ocurra ninguna fuga de memoria, y si ocurre mostrará un error al programador.

AtomicSafetyHandle verifica cuando dos jobs están accediendo a la misma variable NativeContainer al mismo tiempo y muestra una excepción con un mensaje de error claro en el que se menciona por qué ha ocurrido el error y cómo solucionarlo, proporcionando así una validación del sistema y una seguridad completa.

También cabe comentar que, cuando un job tiene acceso a una variable NativeContainer, se le da acceso tanto de lectura como de escritura, lo que puede hacer que se ralentice la ejecución. Además, el C# Job System no deja al programador planificar un job que tiene permisos de escritura de una variable NativeContainer mientras otro job está escribiendo en dicha variable.

CONCURRENCIA EN UNITY

Por eso mismo, si un job no va a necesitar escribir o modificar una variable NativeContainer lo mejor es marcarla con el atributo **ReadOnly** :

```
[ReadOnly]
public NativeArray<int> input;
```

De esta forma, el C# Job System permite que dicho job se ejecute al mismo tiempo que otros jobs que tienen también un acceso solo de lectura a esa misma variable NativeContainer (en este caso NativeArray).

Tras crear el job (mencionado arriba), hay que seguir tres pasos:

- Instanciar el job** (lo que significa crear la estructura de un nuevo job).
- Popular las variables del job**, es decir, darles valor para que el job pueda ejecutarse.
- Y finalmente llamar al método Schedule()**.

Schedule(): añade el job a una cola de jobs que esperan para ejecutarse cuando sea su turno; una vez un job ha sido programado no se puede interrumpir de ninguna manera. Hay que tener en cuenta que solo se puede llamar al método Schedule() desde el "main thread", ya que es el encargado de organizar la ejecución de todos los jobs.

```
// Aquí se almacenará el resultado del job
// El 1 hace referencia a la longitud del array
// Será 1 ya que solo almacenamos un resultado aquí
NativeArray<float> resultado =
    new NativeArray<float>(1, Allocator.TempJob);

// Creamos y preparamos el job para su ejecución
MyJob jobData = new MyJob();
jobData.a = 10;

jobData.b = 10;
jobData.resultado = result;

// Lo programamos para que se ejecute
JobHandle handle = jobData.Schedule();

// Esperamos a que se complete el job
// Complete() es una función similar a Join() en los
threads
handle.Complete();

// Le asignamos el resultado a una variable nueva
float aPlusB = resultado[0];

// Y liberamos la memoria usada por el NativeArray
resultado.Dispose();
```

CONCURRENCIA EN UNITY

Allocator: cuando el programador crea una variable `NativeContainer`, hay que especificar el tipo de asignación de memoria que se va a necesitar, cuyo tipo depende de la longitud de tiempo que tarde a ejecutarse el job, de forma que se puede ajustar la asignación para que conseguir la mejor actuación en cada caso. Hay tres clases de `Allocator` para la asignación de memoria y liberación de variables `NativeContainer`, simplemente hay que especificar que tipo de las tres se va a hacer servir cuando se cree la variable.

- En primer lugar está **`Allocator.Temp`**, la cual se trata de la asignación más rápida y con menor tiempo de vida (un frame o menos).

- En segundo lugar está **`Allocator.TempJob`**, cuya asignación es más lenta que `Temp` pero más rápida que `Persistent`, y se usa para asignaciones de un tiempo de vida aproximado a cuatro frames (además de ser thread-safe). Si no se ejecuta `Dispose()` para liberar la memoria en esos cuatro frames la consola imprime un warning. Se trata de la asignación más usada por los jobs pequeños o simples.

- En tercer y último lugar tenemos **`Allocator.Persistent`**, y se trata de la asignación más lenta pero que puede durar todo el tiempo de vida que tiene el programa si fuera necesario. Se trata de la asignación más usada por los jobs más largos y que no debe ser utilizada cuando la actuación y ejecución es esencial.

JobHandle: Manejan la ejecución de jobs, y hacen que unos dependan de otros a la hora de ejecutarse.

Tiene la propiedad `IsCompleted` que devolverá `false` si el job indicado sigue en ejecución o por el contrario `true` si dicho job ya ha acabado su ejecución. De esta manera se consigue, que un job dependa de los resultados de otro, pudiendo pasar así como parámetro los resultados del primero al segundo job sin que ocurra ningún tipo de error ni problema. La forma de hacerlo sería la siguiente:

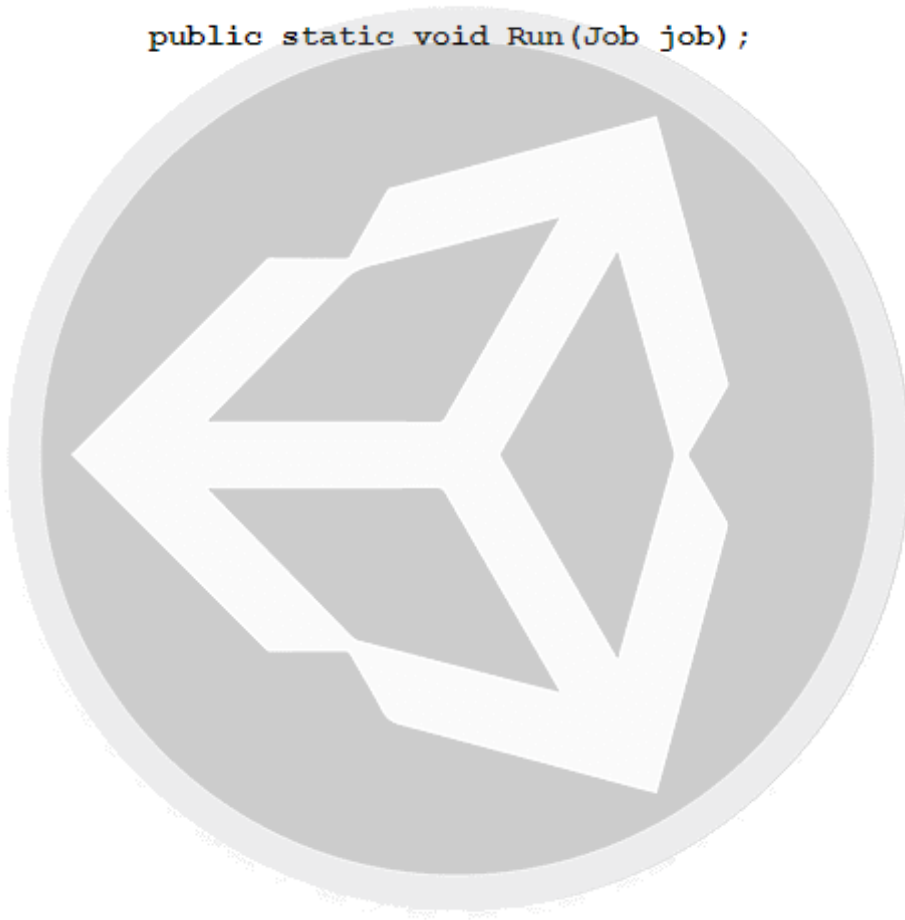
```
JobHandle firstJobHandle = firstJob.Schedule();  
secondJob.Schedule(firstJobHandle);
```

CONCURRENCIA EN UNITY

Así se crea el primer job y se programama y tras eso se programa el segundo job en función de cuando haya terminado el primer job.

Run(): Sería el equivalente de Debug.Log. Sirve para depurar y verificar que todo funciona correctamente usando la consola. Al sustituir Run() donde debería ir Schedule(), el job se ejecuta inmediatamente en el thread principal, y de esta manera se puede depurar el código del job y verificar que funciona de forma correcta.

```
public static void Run(Job job);
```



¿QUÉ MECANISMO ES MEJOR?

CORRUTINAS VENTAJAS

En primer lugar se han mostrado y explicado las corrutinas, y un punto a favor de estas es su **rápida comprensión y aplicación**, ya que no hay que entender nada más allá que un par de conceptos nuevas y su aplicación es extremadamente sencilla.

Se trata de una estructura **muy utilizada** sobre todo para hacer transiciones en los juegos que duran más de un frame, como ir cambiando un color de un objeto o una imagen, aunque también son útiles para optimizar el programa haciendo que haya menos comprobaciones de funciones a cada frame como hemos explicado.

CORRUTINAS DESVENTAJAS

En su contra tenemos que las corrutinas emplean una **clase de control de flujo del programa que se puede llegar a confundir con la concurrencia sin serlo**, ya que todo lo que hacen es suspender o reanudar el control del "main thread", pero **no toma ventaja de las diferentes CPUs** que tiene un ordenador ni realiza ningún trabajo en ellas, Por lo que en conclusión si queremos aplicar concurrencia las corrutinas no son la opción a seguir.

THREADS VENTAJAS

En segundo lugar, los threads **muestran su mayor potencial cuando son utilizados para realizar cálculos que tienen un gran coste temporal**, dividiendo el algoritmo en diferentes partes y ejecutando cada una en un procesador, reduciendo así el tiempo total de ejecución.

Además de esto, utilizan **memoria compartida**, lo cual aunque puede dar lugar a algunos errores, también es positivo ya que **consumen menos recursos y gastan poca memoria**. También, como punto a favor, se puede destacar que no son únicos a Unity como las corrutinas y los jobs que si que son exclusivos de Unity, lo cual hace **que programadores experimentados en otros ámbitos o lenguajes les sea sencillo de utilizar o de aprender a implementarlos**.

THREADS DESVENTAJAS

En contra de los threads se puede mencionar que **no son igual de sencillos de implementar** que las corrutinas o que tenga **problemas importantes con las condiciones de carrera**, pero como hemos mostrado, las condiciones de carrera **se pueden solucionar** haciendo un correcto uso de los cerrojos y su aplicación a pesar de ser algo **más compleja** no supone un gran esfuerzo para un programador experimentado.

JOBS VENTAJAS

Finalmente también se ha hablado de los jobs, los cuales se han visto como **una solución a los problemas principales que plantean los threads** (condiciones de carrera y uso de memoria compartida) y que además se **ejecutan de forma más optimizada**. También **destacan por su fácil uso en combinación con ECS**, el cual es un sistema muy visto y utilizado en los videojuegos actuales, y por eso los jobs, al considerarse una “mejor versión” de los threads son muy utilizados y útiles en los videojuegos de hoy en día.

JOBS DESVENTAJAS

Pero en su contra también tienen que, como su mayor potencial se obtiene al combinarlo con el ECS, **si no se va a utilizar ECS puede resultar más óptimo usar threads en lugar de jobs**. Esto es debido que al usar jobs tenemos que usar variables de tipo NativeContainer y aprender a cómo usar los diferentes tipos de Allocator, por tanto al tener que ir también con cuidado y aprender como funcionan esos tipos de variables, **para un programador experimentado puede resultar más intuitivo el uso de threads** ya que esos no son únicos a Unity y se usan para múltiples ámbitos fuera de la programación de videojuegos.

CONCURRENCIA EN UNITY

	FACILIDAD DE APLICACIÓN DESDE CERO	APLICAN CONCURRENCIA	USO DE MEMORIA COMPARTIDA
CORRUTINAS	Alta	No realmente	No
THREADS	Media	Sí	Sí
JOBS	Media-Baja	Sí	Sí

	USO EXCLUSIVO DE UNITY	NIVEL DE SEGURIDAD	UTILIDAD DENTRO DE UNITY
CORRUTINAS	Sí	(No necesitan ser seguras, no aplican concurrencia real)	Controlar acciones de más de un frame de duración
THREADS	No	Alto aplicando correctamente cerrojos	Dividir cálculos de larga duración para reducir así su tiempo de ejecución
JOBS	Sí	Alto sin necesidad de cerrojos	Misma utilidad que los threads pero junto con ECS son más eficientes

Por esto, y gracias a los numerosos beneficios que proporcionan los threads a los videojuegos programados en Unity (aumento de FPS, rendimiento más óptimo, facilidad de control y aplicación, uso óptimo de memoria compartida...) concluimos que **a la hora de usar concurrencia en Unity a nivel general (sin usar ECS) la mejor opción es el uso de threads.**