

Unity

CONCURRENCY IN UNITY



INDEX

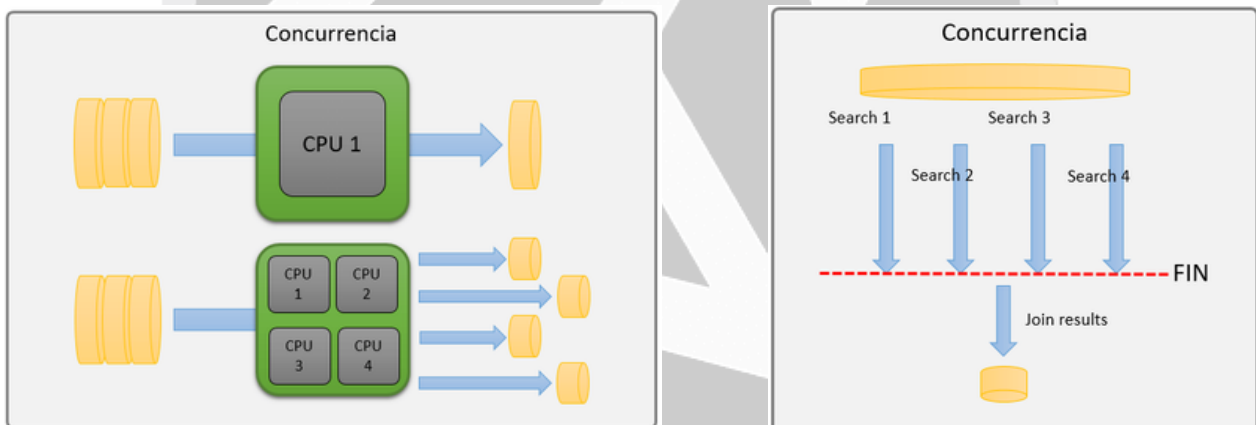
CONCURRENCY, WHAT IS IT?	3
COROUTINES	5
WHAT ARE THEY?	5
HOW TO IMPLEMENT THEM	7
THREADS	11
WHAT ARE THEY?	11
HOW TO IMPLEMENT THEM	13
JOBS	20
WHAT ARE THEY?	20
HOW TO IMPLEMENT THEM	23
WHICH OPTION IS BEST?	28

CONCURRENCY, WHAT IS IT?

As a starting point for this document, it is important to know that **concurrency is the ability of different parts of a program or algorithm to be executed in a disorderly or partially ordered way, but without affecting the final result of the program.**

These calculations will be executed on the different processors of the computer by using different threads of execution.

In this way, it is possible for the different threads of execution to interact with each other while they are running.



However, this poses a problem, as having different paths of execution results in the final outcome of the program being completely indeterminate.

This means that **the execution order of these threads cannot be determined with certainty** (in what order they act, which ones finish first, which is the last thread to act, etc.), leading to errors such as race conditions, mutual deadlocks, or starvation problems that must be resolved by the programmer to ensure proper functionality of the implemented code.

CONCURRENCY IN UNITY

Throughout this document, the different ways of applying concurrency in Unity, their advantages and disadvantages, will be thoroughly discussed starting from a basic to intermediate level of Unity.

After applying concurrency in Unity (only when necessary, as using it in areas where the data processing is very low will achieve the opposite of the objective since the cost of creating multiple processes will be greater than the benefit obtained by treating them concurrently), a great **optimization of the proposed code** will be obtained through different structures and/or methods, which will be processed **on different CPUs** to achieve faster execution.

To achieve this, Unity provides three different mechanisms that each programmer can use in their project:

- 1-. **COROUTINES**
- 2-. **THREADS**
- 3-. **JOBS**

COROUTINES

WHAT ARE THEY?

Firstly, we will start by discussing coroutines, how they work, how they are applied, and what benefits they can bring to the code once implemented correctly and efficiently.

Normally, when a function is executed, everything that happens inside it will occur within the same "frame update," causing different outcomes than expected on various occasions since a sequence of events cannot occur over time, but will occur within the same frame as quickly as possible.

Let's say, for example, that the user wants to gradually reduce the opacity (alpha) of an object until it is completely transparent. For that, a function like the following would be used:

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

The expected result is the gradual reduction of the material's alpha of the object. However, after executing this function, the object will go from being visible to being transparent in a single frame.

This is where coroutines come into play, providing a simple solution to the problem.

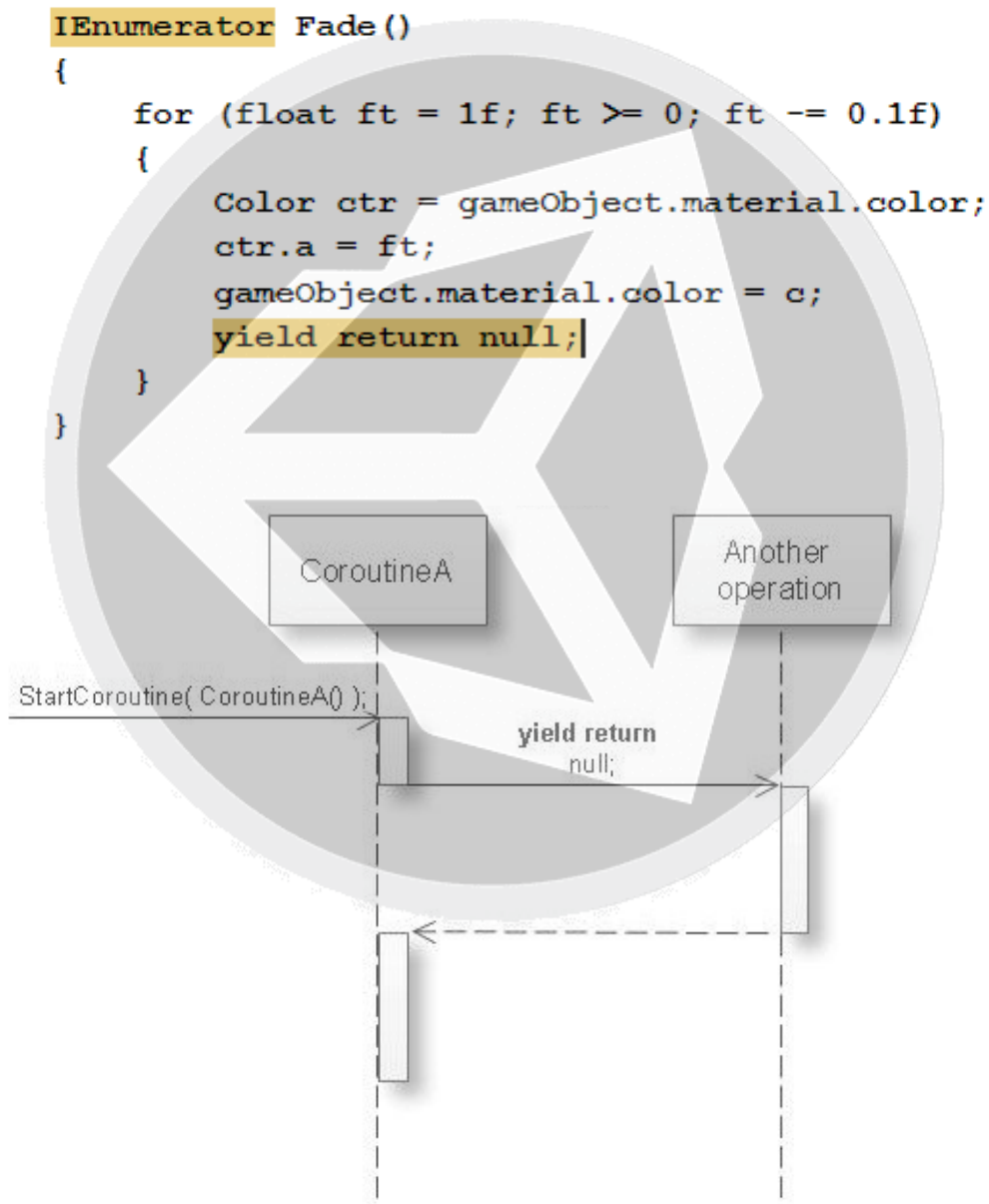
A coroutine is a function that can pause its execution and return control back to Unity before finishing, thereby allowing it to act in the background while the program continues to execute the main method.

CONCURRENCY IN UNITY

The correct way to apply a coroutine to the code described above is:

- Change the function type to a coroutine by replacing "void" with "IEnumerator".
- Use "yield return" and the amount of time you want to pass.

The resulting code would be:



COROUTINES

HOW TO IMPLEMENT THEM

It's important to know that **coroutines don't work like normal functions** because they don't return floats, strings, booleans, etc.

Instead, **they return an IEnumerator, a .NET type variable used to pause iteration.**

```
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;  
    }  
}
```

And now the different functions that can be used with coroutines are going to be explained:

:

- **StartCoroutine():** Once the coroutine has been created, it must be called from the main code using `StartCoroutine` to work.

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

CONCURRENCY IN UNITY

This way, the "Fade" coroutine will be activated after pressing the F key, which, upon reaching "yield return null", will return control to the main program while executing as a for loop with one iteration per frame (without causing errors due to the safe and correct handling of the parameters of that "for loop" between yields).

- **WaitForSeconds:** Suspends the coroutine for a specified amount of time. It is applied when we want to extend the effect of a coroutine over time.

```
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);  
    }  
}
```

In this way, the object will take longer to become transparent, as it will wait 0.1 seconds between each iteration of the loop instead of executing on every frame, while the main code continues to run normally.

This shows how coroutines have a fundamental use in optimizing code efficiently.

Applied to the realm of videogames, there are numerous tasks that must be carried out periodically or continuously, and they should be included in Update() (which runs several times per second). However, sometimes, even though periodic execution is desired, it is necessary that it occurs less frequently or even at different times.

CONCURRENCY IN UNITY

A task that meets these requirements could be the following, which checks if the player has entered the danger distance with an enemy:

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

As it is a function that doesn't need to be updated every frame since the player doesn't move so fast, instead of adding it to the Update function, a coroutine will be created to carry it out:

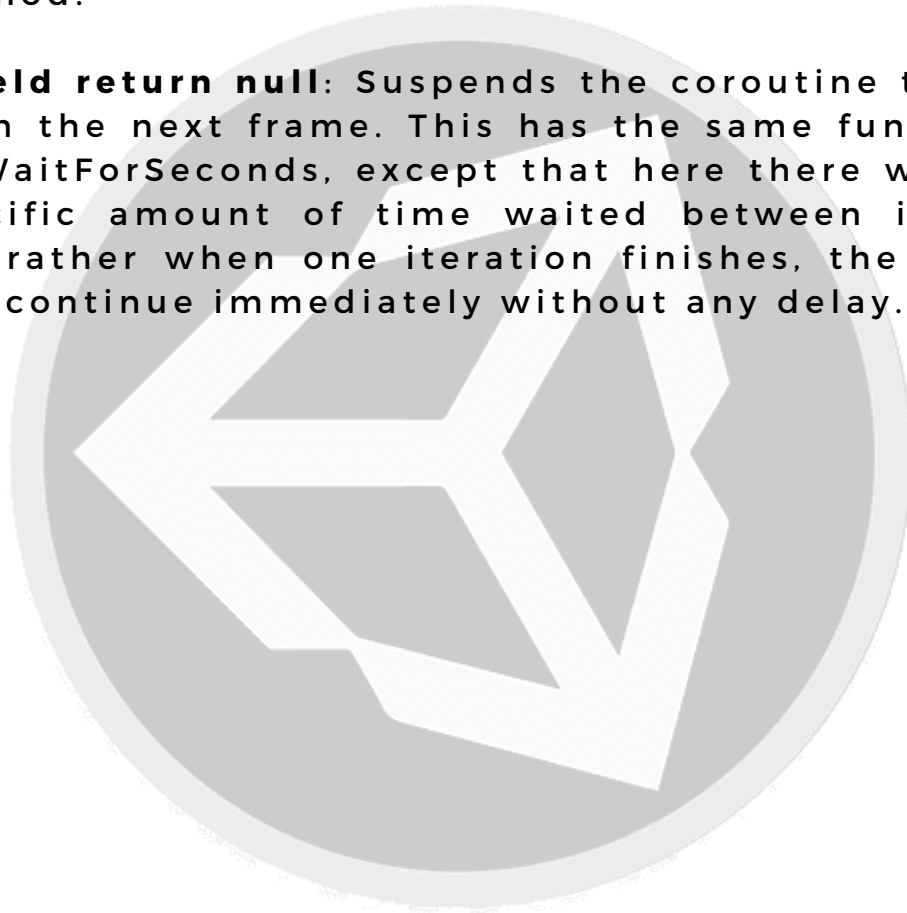
```
IEnumerator DoCheck()
{
    for(;;)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}
```

By calling the "DoCheck()" coroutine in the code, the ProximityCheck() will be checked every 0.1s, **freeing the game from doing a large amount of checks every frame without any significant impact on the game.** If it had been in the Update function instead of being a coroutine, all those checks would have only slowed down the game, since they were checks that wouldn't have had an effective impact on the game, as the player doesn't move fast enough for that check to be performed every frame.

CONCURRENCY IN UNITY

Other coroutine-related functions that the programmer should know or be aware of when applying them are:

- **StopCoroutine():** Stops a specific coroutine. It is used in the same way as StartCoroutine(), being included in the main program.
- **StopAllCoroutines():** Disabling all coroutines of a GameObject. If a GameObject is directly disabled, all its coroutines will also be disabled, but doing it that way can lead to errors, so it is better to use this method.
- **Yield return null:** Suspends the coroutine to resume it on the next frame. This has the same functionality as WaitForSeconds, except that here there won't be a specific amount of time waited between iterations, but rather when one iteration finishes, the next one will continue immediately without any delay.



THREADS

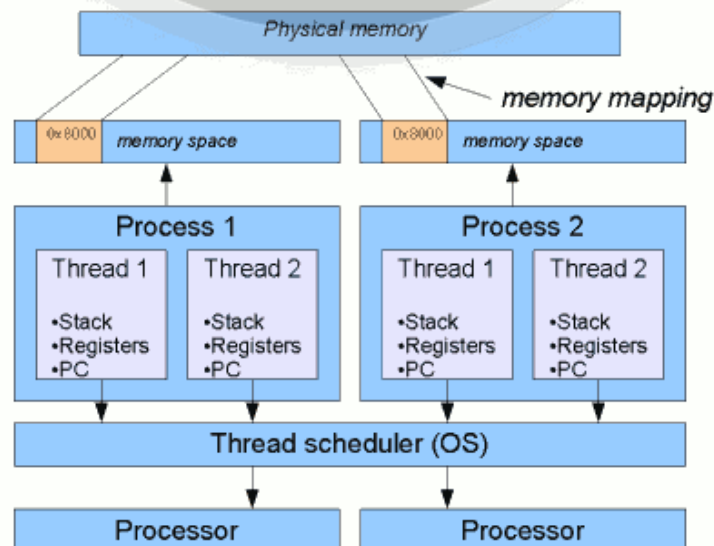
WHAT ARE THEY?

Like coroutines, threads are another way that Unity has to achieve concurrency. We will proceed to explain how they work and what benefits they have.

Firstly, and unlike coroutines, threading takes advantage of the **multicore** structure that modern CPUs have by which they can perform **two or more tasks concurrently and efficiently**.

Using threading in Unity allows for taking advantage of multicore CPUs to perform two or more tasks concurrently and efficiently, **which is unlike using coroutines**. Threading allows for performing very complex calculations by offloading them to other CPUs in the background while keeping the user interface active and responsive to the player. Threading is also used to **perform the same task using different algorithms and compare their effectiveness**.

Another characteristic of threading is dividing a very long task into smaller tasks, executing them separately with different threads on different CPUs and using shared memory, and finally merging them all together to display the final result.



CONCURRENCY IN UNITY

And what does using threading or multithreading bring to Unity? Thanks to threads, a video game can significantly increase its FPS, since threads allow for a more structured calculation of operations, thus achieving a very significant benefit in the smoothness of the game, which should be a primary objective for every programmer when programming a video game.

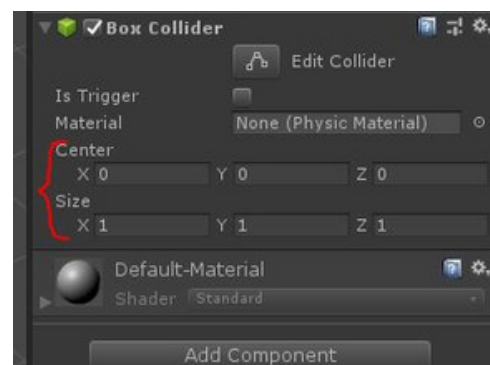
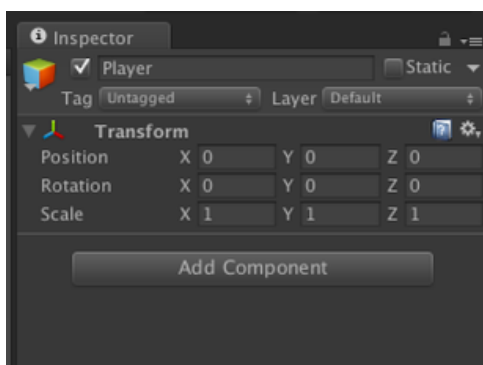
Despite of that, using threads and threading can compromise the proper functioning of the program, so it is necessary to have a clear understanding of their use and application.

One of the properties of threads is that they use **shared memory** for all threads, and that can lead to unexpected behaviors if the appropriate precautions are not taken when accessing or modifying that memory, known as race conditions.

Therefore, any threading operation must be well protected and secured.

And before continuing and to conclude this section, it is necessary to talk about the Unity API.

The Unity API collects all those objects or methods that are imported through Unity Engine, Unity Editor, Unity or Other, which are the spaces provided by Unity to work with its engine. One of its main characteristics is that it is not Thread safe, meaning that it is not safe to apply Threads in it. Therefore, **references** such as transform.position or **using the different components that a GameObject can have in Unity** (Colliders, Sprite Renderers...) **cannot be used for multithreading**, as those parameters are not accessible by different threads (only by the "main thread") and therefore cannot be used to carry out calculations or operations.



THREADS

HOW TO IMPLEMENT THEM

To apply threading in Unity, unlike coroutines that do not require any namespace, the use of threads and multithreading in Unity requires first:

- **Threading namespace:** to use it we import it at the beginning of the script using the following code snippet, thus being able to use all the functions specific to threads.

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

- **Creating a thread:** It is important that threads are all created from the main thread to avoid any errors. In this example, two threads are created and each one is assigned a different task (which will be a function) to carry out separately, as we can see in the following code fragment:

```
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());  
  
    }  
}
```

CONCURRENCY IN UNITY

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

If we call the functions separately, the output of the program would be the work 1 function printing numbers from 1 to 10 first and then the work 2 function doing the same.

- **Thread.Start()**: used to initiate the work of a thread once it has been created:

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

This is the result that would be obtained if threads are used instead of making the functions iteratively:

```
Work 1 is called 1  
Work 2 is called 1  
Work 1 is called 2  
Work 2 is called 2  
Work 2 is called 3  
Work 1 is called 3  
Work 1 is called 4  
Work 2 is called 4  
Work 1 is called 5  
Work 2 is called 5  
Work 1 is called 6  
Work 2 is called 6  
Work 2 is called 7  
Work 1 is called 7  
Work 2 is called 8  
Work 1 is called 8  
Work 1 is called 9  
Work 2 is called 9  
Work 1 is called 10  
Work 2 is called 10
```

But it's not enough to just start a thread and give it a task, you also need to know how to use certain methods to handle those threads and make them work properly.

You can also **force a thread to finish** executing quickly or abort the execution of a thread because it is no longer needed to continue running. Below are the methods that allow the programmer to perform these functionalities.

CONCURRENCY IN UNITY

- **Thread.Join():** method used to make a particular thread finish its work or to make the rest of the active threads stop until the thread in question has finished its work.

It also causes the thread to which it is attached to run first, before the others.

```
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...");  
    }  
}
```

And the result obtained will be this:

```
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..!
```

This way you can check how, by doing `ThreadOne.Join()`, the main thread stops (since it does not print "Work completed"), since, as explained earlier, by doing `Thread.Join()` priority will be given to that thread and until it finishes the rest will be frozen without advancing, and once this thread finishes, the execution of the rest of the threads will resume as quickly as possible.

CONCURRENCY IN UNITY

- **Thread.Sleep()**: as its name suggests, it will sleep (suspend) the thread for a specific interval of time, which can be specified in milliseconds or as a time interval. By suspending a thread, it will not consume CPU resources, so indirectly, memory is saved to be used in other threads being processed.

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

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

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

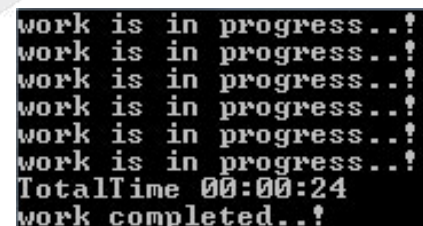
string elapsed Time = String.Format("{0:00}:{1:00}:{2:00}",
ts.Hours, ts.Minutes, ts.Seconds);

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
    }
}
```

And the output of the program would be as seen in this image:



```
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..!
```

And so, every time the "work in progress..!" message is printed, it will wait for 4 seconds until the next iteration of the loop is performed, since after printing the message the thread sleeps for 4 seconds; and until this thread finishes, the main thread won't resume, so the "work completed..!" message doesn't appear.

CONCURRENCY IN UNITY

- **Thread.Abort():** it terminates or aborts a thread so that it cannot continue to execute. What it does is to produce a "ThreadAbortException" exception in the thread that has been aborted so that the main thread initiates the termination of the aborted thread.

```
ThreadOne.Abort();
```

The **threadpool** is also important to know and how to control it.

What a threadpool does is **keep numerous threads waiting for a task assignment to be placed by the program** and work concurrently.

This way, it increases the execution capacity and avoids latency in execution thanks to the creation and destruction of threads for short and low-duration tasks.

It is applied as follows; first, all the inputs or variables needed by the function to be executed in the multiple threads are saved into a variable; and second, the threadpool is called.

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

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

As can be seen, when calling the threadpool, it is not necessary to create the different threads one by one. The only thing that needs to be done is to enqueue the function to be executed with threads within the threadpool along with the necessary variables, and the threadpool itself will take care of using the necessary threads to carry out that function.

Therefore, we can conclude that using a threadpool is more efficient than using threads separately since **there is no execution time spent creating threads one by one**, and it allows for reuse of threads if necessary, ensuring that there is never a limit on the available threads.

CONCURRENCY IN UNITY

Now the reader may wonder, **how can we improve threading security?**

The solution comes from a tool called **locks**, which ensures that the shared information between threads is not altered erroneously.

However, **just using a lock will not guarantee threading security**; it is necessary to identify the critical information that will be accessed and/or modified by the different threads and, once known, activate the lock before accessing or modifying it and then disable it after leaving the critical zone.

In this way, it is ensured that **only one thread accesses or modifies such information at the same time**, avoiding race conditions, so a clear idea of the code's functioning is needed.

In this code snippet we can observe an **incorrect** use of locks, as it still leads to a race condition, where in some cases "x times y is 1" will be printed and in other cases "x times y is 0", because what is achieved now is that the main thread and the new thread "compete" with each other to see who reaches the lock first, but it does not ensure that the results obtained are accurate.

```
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;  
        }  
    }  
}
```

CONCURRENCY IN UNITY

To implement locks, you should start by creating a special variable of type **readonly** (we are interested on it being read-only) to control when a code zone is or is not blocked for the rest of the threads.

After that, it is necessary to evaluate which will be the critical information that different threads will access in order to use the lock variable, blocking its entry when a thread makes use of it, and releasing its use once the thread stops accessing or modifying said information.

```
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();
    }
}
```

USING LOCKS:

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

WITHOUT LOCKS:

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

And the output provided by the program will be above next the code. Indeed, the two threads have not been printed at the same time, since thanks to the lock, **one thread has accessed the entire block of code first and then the other.**

If there had not been a lock, the output would have been as shown below, since **both threads would have been executed at the same time**, and they would have accessed the same function without waiting for the other to finish.

JOBS

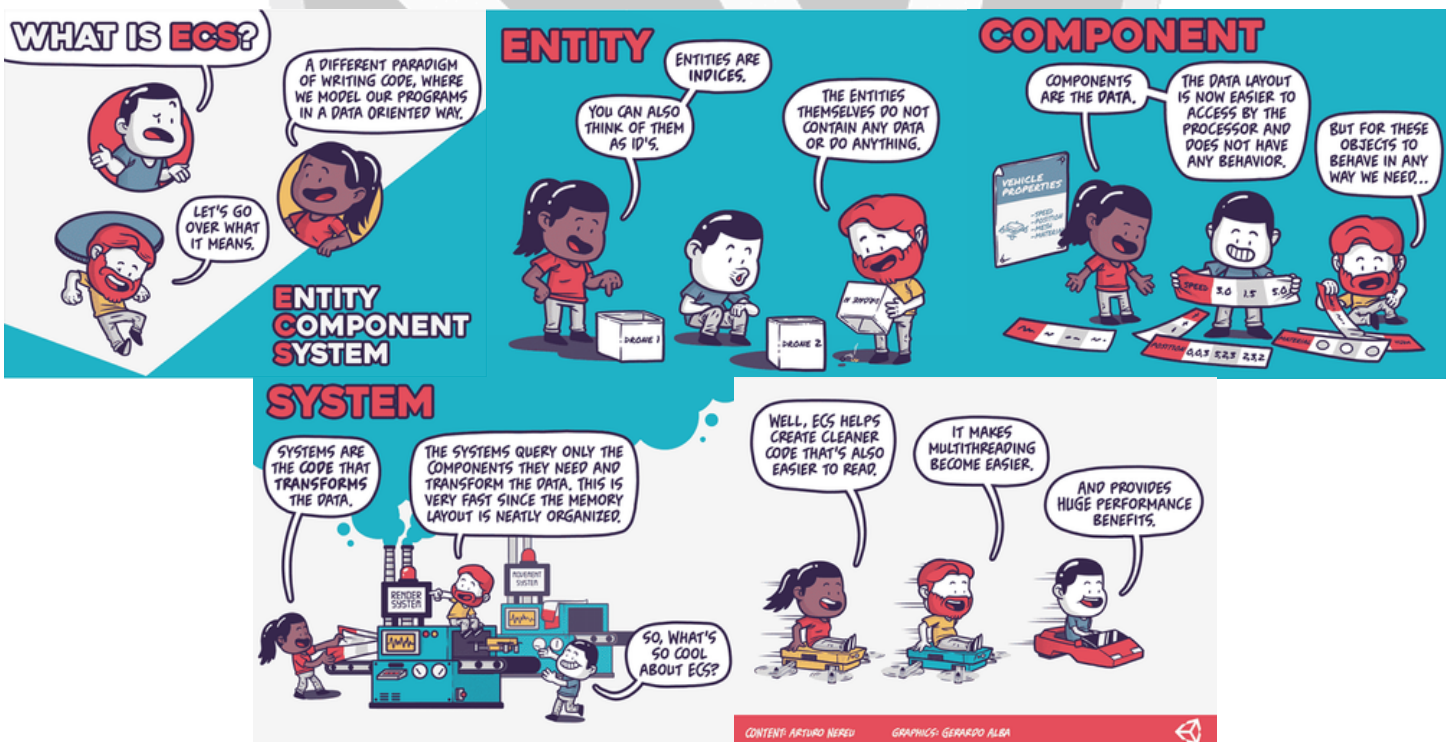
WHAT ARE THEY?

And finally, we are going to explain and delve into jobs, what they are, and how the Job System works in C# and therefore in Unity.

The Unity C# Job System allows the programmer to **write and safely apply multithreading** that interacts with the Unity Engine for **superior game quality**.

The Job System is commonly used together with **ECS (Entity Component System)**, an architecture that makes it easier to write code that works across all platforms.

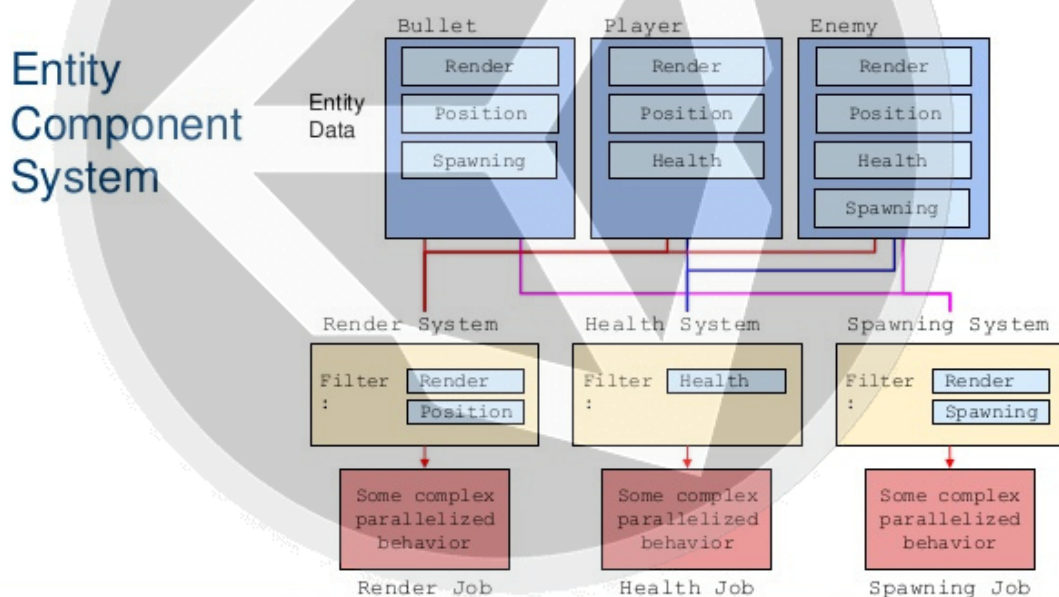
In **ECS**, the **E** refers to **entities**, the objects that make up the game, the **C** refers to **components**, the variables associated with entities, and finally the **S** refers to **systems**, the logic that transforms component data from one state to the next (moving entities, changing their speed, etc.).



CONCURRENCY IN UNITY

The Unity Job System **manages multithreading using jobs instead of threads**. A job is a structure, a small unit of work that performs a specific task, receives certain parameters, and performs operations with them, similar to how standard functions work.

The important difference is not what jobs are per se, but how the Job System directs them. The Job System directs groups of jobs to the different cores of the computer, along with a worker thread for each logical CPU that exists, responsible for avoiding the famous "context switching" problems. **In addition to having a worker thread for each CPU, the Job System queues all jobs to be executed, and the worker threads of each CPU pick up the jobs that are in the queue and execute them, directing the different job dependencies and ensuring that they are executed in the correct order.**



Dependencies between jobs occur when one job depends on another to be executed or completed.

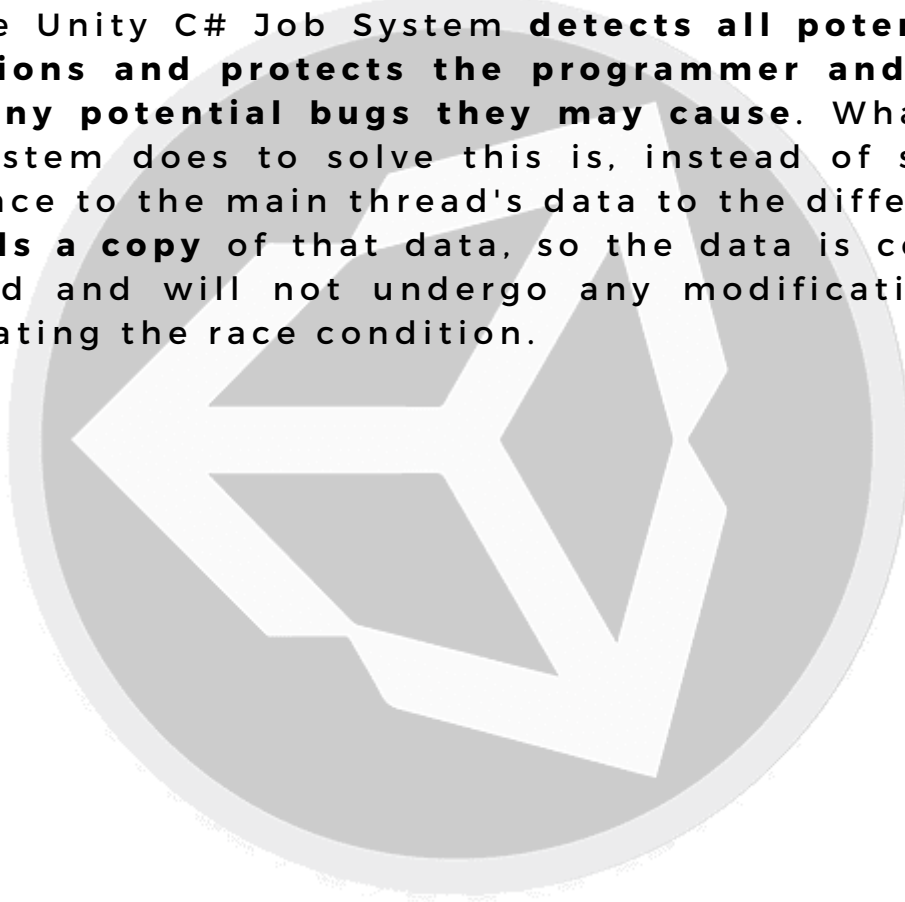
In this way, for example, if there is a jobA that depends on jobB to finish, the Job System ensures that jobA is not started until jobB has finished, **avoiding many common errors that can arise from multithreading.**

CONCURRENCY IN UNITY

As previously mentioned, another major problem with using multithreading is race conditions, which often cause bugs in many video games. These are difficult problems to fix because they depend on the code and the order in which each thread executes, making race conditions, as mentioned earlier, a significant problem.

For example, suppose a job sends a reference to some common data from the main thread to the current thread. Since it is common data, it cannot be verified if when reading the necessary data, another thread is modifying it at that moment, causing a race condition and sometimes resulting in unexpected results.

The Unity C# Job System **detects all potential race conditions and protects the programmer and program from any potential bugs they may cause.** What the C# Job System does to solve this is, instead of sending a reference to the main thread's data to the different jobs, **it sends a copy** of that data, so the data is completely isolated and will not undergo any modifications, thus eliminating the race condition.



JOBS

HOW TO IMPLEMENT THEM

Jobs, just like threads, also need their own namespace to be imported at the beginning of the script in order to use them and their characteristic methods.

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

Jobs are structures, and as structures, they inherit from an interface, in this case `IJob`. By creating a structure and making it inherit from `IJob` (thus creating a job), that individual job can be executed concurrently with other jobs and the main thread.

First of all, to create a job you need to:

- **Create a structure that implements or inherits from `IJob`.**
- **Add the variables that the job will use** (the result variables must be stored in variables of type `NativeContainer`, which will be explained below).
- **And finally, create a method within that job structure called "Execute" with the task that the job will carry out when executed.**

That being said, it should be noted that when a job is executed, the "Execute" method will be carried out once on a single CPU.

```
public struct MyJob : IJob  
{  
    public float a;  
    public float b;  
    public NativeArray<float> result;  
  
    public void Execute()  
    {  
        result[0] = a + b;  
    }  
}
```

CONCURRENCY IN UNITY

It should be noted that the downside of solving race conditions in this way is that not only does **it use more memory than threads**, but also, by copying variables for each job, it causes them to be somewhat isolated. **This is where NativeContainer variables come in**, which are a type of shared memory. A NativeContainer variable is a variable that provides relative safety in C# for being a type of shared memory. When used with the C# Job System, what the NativeContainer variable does is allow the job to access shared information with the main thread instead of working with a copy. There are different types of NativeContainer variables (NativeArray, NativeList, NativeQueue, etc.), but they all have the same purpose.

The way in which the safety of these variables that use shared memory is guaranteed is that **everything that is read and written from a NativeContainer is registered and evaluated with the DisposeSentinel and AtomicSafetyHandle classes**, two classes that are automatically used by NativeContainer variables.

DisposeSentinel automatically detects any memory leaks that occur at any time, so these variables will always be monitoring to ensure that no memory leaks occur, and if they do occur, it will display an error to the programmer.

AtomicSafetyHandle verifies when two jobs are accessing the same NativeContainer variable at the same time and throws an exception with a clear error message stating why the error occurred and how to fix it, thus providing a validation of the system and complete safety.

It should also be noted that when a job has access to a NativeContainer variable, it is given both read and write access, which can slow down the execution. Additionally, the C# Job System does not allow the programmer to schedule a job that has write permissions to a NativeContainer variable while another job is writing to that variable.

That is why, if a job is not going to need to write or modify a NativeContainer variable, **it is best to mark it with the ReadOnly attribute**:

CONCURRENCY IN UNITY

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

In this way, the C# Job System allows that job to execute at the same time as other jobs that also have a read-only access to that same NativeContainer variable (in this case NativeArray).

After creating the job (mentioned above), you need to follow three steps:

- **Instantiate the job** (meaning creating a new job structure).
- **Populate the job variables**, meaning give them values so that the job can execute.
- **And finally call the Schedule() method.**

- **Schedule()**: adds the job to a queue of jobs that are waiting to be executed when it is their turn; once a job has been scheduled it cannot be interrupted in any way. It should be noted that the Schedule() method can only be called from the main thread, as it is responsible for organizing the execution of all 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();
```

CONCURRENCY IN UNITY

- **Allocator:** when a programmer creates a new `NativeContainer` variable, they need to specify the type of memory allocation they will need, which depends on the length of time the job takes to execute, so that the allocation can be adjusted to achieve the best performance in each case. There are three types of `Allocator` for allocating and releasing memory for `NativeContainer` variables, and the programmer simply needs to specify which type of `Allocator` to use when creating the variable.

- First, there is **`Allocator.Temp`**, which is the fastest and shortest-lived allocation (one frame or less).

- Secondly, there is **`Allocator.TempJob`**, whose allocation is slower than `Temp` but faster than `Persistent`, and is used for allocations with a lifetime of approximately four frames (in addition to being thread-safe). If `Dispose()` is not executed to release memory within those four frames, a warning is printed in the console. It is the most commonly used allocation for small or simple jobs.

- Third and lastly, we have **`Allocator.Persistent`**, which is the slowest allocation but can last the entire lifetime of the program if necessary. It is the most commonly used allocation for longer jobs and should not be used when performance and execution are critical.

- **`JobHandle`:** They manage the execution of jobs and make them depend on each other when it comes to running.

It has the property **`IsCompleted`** which will return **false** if the indicated job is still running or **true** if that job has already finished its execution. This allows one job to depend on the results of another, thus being able to pass the results of the first job as a parameter to the second job without any errors or issues. The way to do this would be as follows:

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

CONCURRENCY IN UNITY

This is how the first job is created and scheduled, and then the second job is scheduled based on when the first job has finished.

- **Run()**: This is similar to `Debug.Log`. It is used for debugging and verifying that everything is working correctly using the console. By replacing `Run()` where `Schedule()` should go, the job is executed immediately on the main thread, allowing the code of the job to be debugged and verified to work correctly.

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



WHICH OPTION IS BEST?

COROUTINES ADVANTAGES

On the one hand, coroutines have been shown and explained, and a point in favor of these is their **quick understanding and application**, since nothing beyond a couple of new concepts needs to be understood and their application is extremely simple.

It is a structure that is **widely used**, especially for making transitions in games that last more than one frame, such as changing the color of an object or an image, although they are also useful for optimizing the program by reducing the number of function checks per frame, as we have explained.

COROUTINES DISADVANTAGES

On the other hand, **coroutines use a program flow control class that can be confusing with concurrency without actually being it**, as all they do is suspend or resume control of the "main thread," but they do not take advantage of the different CPUs that a computer has or perform any work on them. Therefore, in conclusion, **if we want to apply concurrency, coroutines are not the option to follow.**

THREADS ADVANTAGES

It has been shown the threads' greatest potential is when **used to perform computations that have a significant time cost**, dividing the algorithm into different parts and executing each one on a processor, thereby reducing the total execution time.

In addition, one advantage of using threads is that they use **shared memory**, which although it can lead to some errors, is also positive as **they consume fewer resources and use little memory.**

CONCURRENCY IN UNITY

Another good thing about them is that they are **not unique to Unity**, unlike coroutines and jobs which are exclusive to Unity. This makes it easier for experienced programmers from other fields or languages to use or learn how to implement them.

THREADS DISADVANTAGES

Against threads, it can be mentioned that they are **not as easy to implement** as coroutines, and that they can have significant **problems with race conditions**. However, as we have shown, race conditions can be solved by making **correct use of locks** and their application, which, despite being somewhat more complex, does not require a great effort for an experienced programmer.

JOB ADVANTAGES

Finally, we have also talked about jobs, which have been seen as a solution to the main problems posed by threads (race conditions and use of shared memory) and which also **run more optimally**. They also stand out for their **ease of use in combination with ECS**, which is a widely seen and used system in modern video games, and for this reason **jobs, considered a "better version" of threads, are very useful and commonly used** in today's video games.

JOB DISADVANTAGES

However, a drawback of jobs is that **their full potential is achieved when combined with ECS**, and if ECS is not going to be used, it may be more optimal to use threads instead of jobs. This is because when using jobs, we have to use variables of type NativeContainer and learn how to use the different types of Allocator. Therefore, for an experienced programmer, **the use of threads may be more intuitive**, as they are **not unique to Unity** and are used for multiple areas outside of game programming.

CONCURRENCY IN UNITY

	IMPLEMENTATION DIFFICULTY	ARE THEY REALLY USING CONCURRENCY?	USE OF SHARED MEMORY
COROUTINES	LOW	NO	NO
THREADS	MEDIUM	YES	YES
JOBS	MEDIUM - HIGH	YES	YES

	USE EXCLUSIVE IN UNITY?	SAFETY LEVEL	UTILITY INSIDE UNITY
COROUTINES	YES	THEY DO NOT NEED TO BE SAFE, NO TRULY CONCURRENCY	MANAGE ACTIONS THAT TAKE MORE THAN A FRAME
THREADS	NO	HIGH BY CORRECTLY APPLYING LOCKS, LOW WITHOUT THEM	DIVIDE LONG CALCULATIONS SO COMPILE TIME IS GREATLY REDUCED
JOBS	YES	HIGH WITHOUT THE NEED OF LOCKS	SAME AS THREADS, BUT MORE EFFICIENT TOGETHER WITH ECS

Therefore, thanks to the numerous benefits that threads provide to Unity game development (increased FPS, optimal performance, ease of control and implementation, optimal use of shared memory...), we conclude that when using concurrency in Unity at a general level (without using ECS), **the best option is to use threads.**