Про багатопоточність в Unity.

В unity раніше можна було використовувати багатопотокові обчислення, але все це потрібно було створювати розробнику самостійно, самому розв'язувати проблеми, що виникають, і обходити підводні камені.

Раніше необхідно було працювати безпосередньо з такими речами, як створення потоків, закриття потоків, пули, синхронізація і т.д. Але з версією unity 2018 року ситуація швидко змінюється, оскільки з’являється стек технологія, орієнтована на дані або Data-Oriented Technology Stack (DOTS). Тепер уся робота лягла на плечі unity, а від самого розробника вимагають тільки створення завдань та їх виконання.

Багатопоточність в **unity**

Unity використовує власну систему завдань (**Job**) для обробки власного коду у декількох робочих потоках, які залежать від кількості ядер процесора, доступних на пристрої, на якому запускається ваша програма. Зазвичай Unity виконує ваш код у одному потоці, який запускається за замовчуванням на початку програми і називається головним потоком (**Main**). Однак, коли ви використовуєте систему завдань (**Job System**), Unity виконує ваш код у робочих потоках, що називається багатопотоковістю.

Багатопоточність використовує здатність процесора обробляти багато потоків одночасно на декількох ядрах. Замість того, щоб завдання або інструкції виконувалися одна за одною, вони виконуються одночасно. Робочі потоки виконуються паралельно один одному і після завершення синхронізують свої результати з основним потоком.

Система завдань (**Job System**) гарантує, що кількість потоків відповідає продуктивності ядер процесора, а це означає, що ви можете запланувати стільки завдань, скільки вам потрібно, не знаючи точно, скільки ядер процесора доступно. Це відрізняється від інших систем завдань, які покладаються на такі методи, як об'єднання потоків, де легше неефективно створити більше потоків, ніж ядер процесора.

Завдання (**Job**) - це невелика одиниця роботи, яка виконує одне конкретне завдання (справу). Завдання отримує параметри та оперує з даними, подібно до того, як це відбувається під час виклику методу. Завдання можуть бути самодостатніми або залежати від інших завдань, які мають бути завершені до того, як їх можна буде запустити. У Unity завданням називається будь-яка структура, що реалізує інтерфейс **IJob**.

Лише основний потік може планувати та виконувати завдання. Він не може отримати доступ до вмісту інших поточних завдань (**Job**), і два завдання (**Job**) не можуть отримати доступ до вмісту одного завдання (**Job**) одночасно. Щоб забезпечити ефективне виконання завдань, можна зробити їх залежними одне від одного. Система завдань у Unity дозволяє створювати складні ланцюжки залежностей, щоб гарантувати, що завдання завершуються у правильному порядку.

Завдання (**Job**) поділяються на 4 типи. Кожне для своєї задачі:

- **IJob**: Запускає одну справу у потоці.

- **IJobParallelFor**: Запускає справи паралельно. Кожен робочий потік, який виконується паралельно, має ексклюзивний індекс для безпечного доступу до спільних даних між робочими потоками.

- **IJobParallelForTransform**: Запускає справи паралельно. Кожен робочий потік, що виконується паралельно, має ексклюзивне перетворення з ієрархії перетворень для роботи з ним.

- **IJobFor**: Те саме, що й **IJobParallelFor**, але дозволяє запланувати роботу так, щоб вона не виконувалася паралельно. Через це, я не буду в подальшому розповідати про **IJobParallelFor**, оскільки **IJobFor** більш “прокачена” версія і майже все, що стосується другого також и стосується першого.

Далі код з офіційної документації unity.

Завдання (**IJob**) виглядає так:

```
1. using UnityEngine;
2. using Unity.Collections;
3. using Unity.Jobs;
4.
5. class ApplyVelocitySample : MonoBehaviour
6. {
7.     struct VelocityJob : IJob
8.     {
9.         // У завданні оголошуються всі дані, до яких буде здійснюватись
   доступ у завданні (Job)
10.        // Оголошуючи його тільки для читання, можна дозволити паралельний
   доступ до даних декільком робочим місцям
11.        [ReadOnly]
12.        public NativeArray<Vector3> velocity;
13.
14.        // За замовчуванням контейнери вважаються доступними для читання та
   запису
15.        public NativeArray<Vector3> position;
16.
17.        // Дельта-час має бути скопійовано до завдання, оскільки завдання
   (Job) зазвичай не мають поняття фрейму.
18.        // Основний потік чекає на завдання на тому ж або наступному кадрі,
   але завдання повинно
19.        // виконувати роботу у детермінований та незалежний спосіб при
   виконанні у робочих потоках.
20.        public float deltaTime;
21.
22.        // Код, що фактично виконується під час роботи
23.        public void Execute()
24.        {
25.            // Переміщення позицій на основі дельта-часу та швидкості
26.            for (var i = 0; i < position.Length; i++)
27.                position[i] = position[i] + velocity[i] * deltaTime;
28.        }
29.    }
```

```
1.   public void Update()
2.     {
3.         var position = new NativeArray<Vector3>(500, Allocator.Persistent);
4.
5.         var velocity = new NativeArray<Vector3>(500, Allocator.Persistent);
6.         for (var i = 0; i < velocity.Length; i++)
7.             velocity[i] = new Vector3(0, 10, 0);
8.
9.
10.        // Ініціалізуваня даних завдання (Job)
11.        var job = new VelocityJob()
12.        {
13.            deltaTime = Time.deltaTime,
14.            position = position,
15.            velocity = velocity
16.        };
17.
18.        // Запланування роботи, повертає JobHandle, за допомогою якого
   можна потім завершити завдання (Job)
19.        JobHandle jobHandle = job.Schedule();
20.
21.        /// Переконайтеся, що завдання завершено
22.        // Не рекомендується завершувати роботу негайно,
23.        // оскільки це не дає вам фактичного паралелізму.
24.        // Оптимально запланувати завдання (Job) на початку кадру, а потім
   зачекати на нього пізніше у цьому ж кадрі.
25.        jobHandle.Complete();
26.
27.        Debug.Log(job.position[0]);
28.
29.        // Native масиви мають бути видалені вручну
30.        position.Dispose();
31.        velocity.Dispose();
32.    }
33.}
```

В цілому так виглядають приклади використання завдання (**Job**).

```
1. struct VelocityJob : IJob
2.     {  [ReadOnly]
3.         public NativeArray<Vector3> velocity;
4.
5.         public NativeArray<Vector3> position;
6.
7.         public float deltaTime;
8.
9.         public void Execute()
10.             {
11.                 for (var i = 0; i < position.Length; i++)
12.                     position[i] = position[i] + velocity[i] *
    deltaTime;
13.             }
14.         }
```

Це основна частина завдання (**Job**), та частина, де проходить вся основна логіка.

Завдання (**Job**) працюють в іншому потоці, через що вони не можуть взаємодіяти з даними з основного потоку напряму. Для їх синхронізації використовуються контейнери NativeArray, вони синхронізують дані в всіх потоках з головним потоком (Main).

Змінні типу deltaTime не синхронізуються, а тільки клонуються в додаткові потоки.

Основна відміна **IJob** від інших завдань полягає в тому, що він виконує завдання в одному потоці. Тобто він виділяє 1 потік для нашого використання.

IJobFor або **IJobParallelFor** вони якби є циклами, де багато разів використовується **IJob**.

Тобто якщо б **IJob** був би змінною:

```
1.   IJob = 5;
```

То **IJobFor** або **IJobParallelFor** мали вигляд такий:

```
1. for (int i = 0; i < 5; i++)
2.                     {
3.                         IJob = 5;
4.                     }
```

Ці типи завдань (**Job**) використовуються коли потрібно зробити щось паралельно з великою кількістю об’єктів (або зробити паралельно велику кількість обчислень).

Щодо **IJobParallelForTransform**, то це по факту **IJobFor** або **IJobParallelFor** з можливістю змінювати позицію, поворот, розмір якогось об’єкта. Тобто робити трансформацію над об’єктами.

Дуже корисно коли потрібно, наприклад, перемістити велику кількість об’єктів з точки А в точку В. Перемістити їх паралельно буде значно швидше, ніж переміщати їх по одному (хоч і дуже швидко) в одному потоці.

Ось так працює багатопоточність в unity, в цілому все зроблено доволі легко для зрозуміння. Звісно багатопоточність в unity, як і всюди, **не** є панацеєю, вона **не** дає приросту в продуктивності всюди, але лише в конкретних випадках. Але там де вона потрібна, система виконує свою роботу на всі 100, на мій погляд використання її в великих проектах дає змогу авторам забезпечити гарну продуктивність та оптимізувати складні процеси.

Усі джерела використанні мною:

https://docs.unity3d.com/Manual/JobSystemOverview.html

https://www.kodeco.com/7880445-unity-job-system-and-burst-compiler-getting-started

https://docs.unity3d.com/Manual/job-system-jobs.html

https://docs.unity3d.com/Manual/JobSystemCreatingJobs.html

https://blog.unity.com/engine-platform/what-is-a-job-system

https://docs.unity3d.com/Manual/JobSystem.html