Unity C# Job System

2023년 2월 21일 화요일 오후 1:32

Job System

사용하면 얻을 수 있는 이점

1. 기존 Unity 기능과 잘 연동하는 **멀티 스레드 코드 작성**을 **용이**하게함.
2. 버스트 컴파일러와 함께 사용 시 **코드 생성 품질 개선** 및 **모바일 디바이스 배터리 소모량 감소**
3. Unity 내부 기능과 통합 및 워커 스레드 공유를 통한 **CPU리소스 경쟁 감소 및 회피**

Job System : 스레드를 대신하여 Job을 만들어 멀티스레드 코드를 관리하는 시스템

Job : 특정한 단일 작업을 수행하는 작은 작업 단위. 메서드 호출 동작과 유사한 방식으로 파라미터를 수신하고 데이터 작업을 수행한다. Job은 **독립적**일 수도 있고, **종속적**일 수도 있다.

Job 종속성 : 특정 Job이 완료될 때 까지 대기해야 하는 경우에 종속성이 있다고 한다.
Job System은 이를 인식하고 지원하며, JobA가 JobB에 종속된 경우, 시스템은 JobB가 완료될 때 까지 JobA를 실행시키지 않는다.

C# Job System의 안전 시스템
: 경쟁 상태를 예방하고 감지할 수 있는 시스템을 지원한다.
*경쟁 상태(Race Condition): 둘 이상의 입력 또는 조작의 타이밍(순서)에 따라 결과 값에 영향을 줄 수 있는 상태

C# Job System은 각 잡에 메인 스레드의 데이터 레퍼런스를 보내지 않고, 필요한 **데이터**의 **복사본**을 보내 경쟁 상태를 예방합니다. 복사본으로 인해 **데이터**가 **격리**되어 문제가 발생하지 않습니다.

위의 데이터를 복사하는 방봅으로 인해 Job은 **Blittable** 데이터 타입에만 액세스 할 수 있습니다.
*Blittable: 블록 전송 또는 개체를 복사하는 것에 적합한지 여부를 나타내는 .NET 관련 용어.

C# 잡 시스템은 memcpy를 사용하여 Blittable 타입을 복사하고 Unity의 관리되는 파트와 네이티브 파트 간에 데이터를 전송할 수 있습니다. 시스템은 잡을 예약할 때 memcpy를 사용하여 데이터를 네이티브 메모리에 저장하고, 잡을 실행할 때 이 복사본에 액세스할 수 있는 권한을 관리되는 파트에 할당합니다.

하지만 **NativeContainer**를 사용하면 Job이 항상 복사본으로 작업하는 것이 아니라 메인 스레드와 공유되는 데이터에 액세스 할 수 있습니다.

NativeContainer: burst때 잠깐 나왔던 저장소 타입. 네이티브 메모리에 상대적으로 안전한 C# 래퍼를 제공하는 관리되는 값 타입입니다.

위에 말한 안전 시스템이 NativeContainer에 대한 **읽기/쓰기 작업**을 **추적**하며, 이 시스템은 모든

NativeContainer에 내장되어 있습니다. **안전 검사(한도 검사, 할당 취소 검사, 경쟁 상태 검사)**는 Unity **Editor Mode** 및 **Play Mode**에서 이용할 수 있습니다.

*Job 내 Static Data에 대한 액세스는 보호되지 않습니다! 이 방법은 모든 안전 시스템을 우회합니다!
*잡을 실행할 때 Execute 메서드는 단일 코어에서 실행됩니다.

[Unity Document] NativeContainer 할당자

NativeContainer를 만들 때는 필요한 메모리 할당 타입을 지정해야 합니다. 할당 타입은 잡 실행 시간에 따라 다릅니다. 이렇게 하면 할당을 맞춤 설정하여 각 상황에서 최고의 성능을 끌어낼 수 있습니다.

NativeContainer 메모리 할당 및 릴리스에는 세 가지의 할당자 타입을 이용할 수 있습니다. NativeContainer를 인스턴스화할 때 적절한 타입을 지정해야 합니다.

- **Allocator.Temp**는 가장 빠른 할당입니다. 수명이 1프레임 이하인 할당에 적합합니다. Temp를 사용하여 잡에 NativeContainer 할당을 전달하면 안 됩니다. 또한 메서드 호출(예: MonoBehaviour.Update 또는 네이티브에서 관리되는 코드로의 기타 다른 콜백)을 반환하기 전에 Dispose 메서드를 호출해야 합니다.

- **Allocator.TempJob**은 Temp보다는 느리지만 Persistent보다는 빠른 할당입니다. 수명이 4프레임 이하인 할당에 적합하며 스레드 세이프 기능을 지원합니다. 4프레임 내에서 Dispose 메서드를 호출하지 않으면 콘솔은 네이티브 코드로 생성된 경고를 출력합니다. 대부분의 소규모 잡은 이 NativeContainer 할당 타입을 사용합니다.

- **Allocator.Persistent**는 가장 느린 할당이지만, 애플리케이션의 주기에 걸쳐 필요한 만큼 오래 지속됩니다. malloc에 대한 직접 호출을 위한 래퍼입니다. 오래 걸리는 잡은 이 NativeContainer 할당 타입을 사용할 수 있습니다. 성능이 중요한 상황에서는 Persistent를 사용하지 않아야 합니다.

ParallelFor Job

본디 Job은 하나의 작업만 수행 할 수 있습니다. 그러나 ParallelFor Job은 수많은 오브젝트에 동일한 작업을 할 수 있도록 별도의 기능이 제공됩니다.

[**ParallelFor Job 정의 예시: Unity Document**]

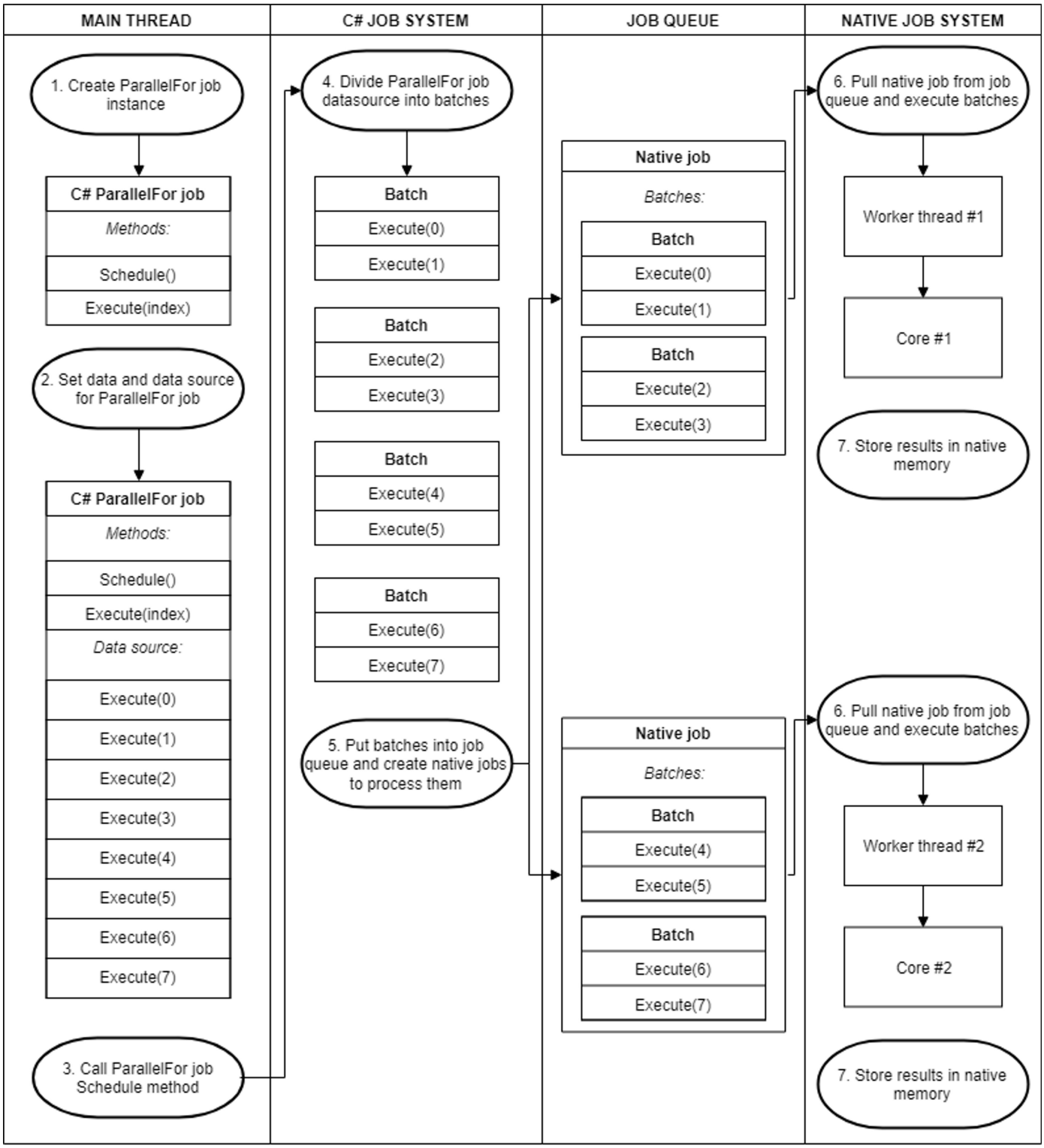
```
struct IncrementByDeltaTimeJob: IJobParallelFor
{
    public NativeArray<float> values;
    public float deltaTime;

    public void Execute (int index)
    {
        float temp = values[index];
        temp += deltaTime;
        values[index] = temp;
    }
```

}

ParallelFor Job 특징

- NativeArray를 사용하여 데이터 소스로 동작합니다.
- ParallelFor Job은 여러 개의 코어에서 동작합니다. 코어당 하나의 잡을 가지며, 각각 일정량의 작업을 처리합니다.
- 데이터 소스 항목당 하나의 Execute 메서드를 호출하며, Execute 메서드엔 정수 파라미터가 있습니다.

ParallelFor Job 예약



ParallelFor Job을 예약할 때 작업을 Batch로 나누어 코어 간에 배포합니다. 각 Batch에는 Execute 메서드의 하위 집합이 포함됩니다.

가장 중요한 C# Job System Tips

- **Static Data 액세스 피하기**: 잡 시스템에서 제공하는 안전 시스템을 모두 회피하게 됩니다. 한마디로 위험!
- **NativeContainer 컨텐츠를 직접 업데이트 하지 않기**: NativeContainer는 복사된 데이터를 넘겨주므로 직접 값을 변경하는 것은 의미가 없다. (예: NativeContainer[0]++;) 값을 변경후 다시 덧씌워야 변경 사항이 적용된다.
  (예: NativeContainer[0] = 10;)
- **JobHandle.Complete를 호출하기:** 이를 호출해야만 메인 스레드가 네이티브 컨테이너 타입의 소유권을 다시 사용할 수 있으며, 종속성이 완료됩니다. 또 Complete를 호출해야 안전 시스템의 상태도 정리되며, 하지 않으면 메모리 누수가 발생합니다.
- **메인스레드에서 예약 및 완료**: 이는 "Job에서 Job을 예약하지 않기"와 같습니다. 메인 스레드에서 JobHandle을 이용하여 종속성 관리를 해야합니다.
- **[ReadOnly] 속성 적극 활용하기:** 쓰기 권한에 대한 구분을 명확히 하는것은 쓰레드 성능 향상의 기본이됩니다.
- **Job에서 메모리 할당하지 않기:** Burst에서 언급했던 것과 같은 맥락인 것 같습니다. Job 내에서 관리되는 메모리 할당은 퍼포먼스를 크게 저하시킵니다.

적당한 Batch Size: CPU의 가장 빠른 **L1 캐시 메모리**는 보통 **32KB**이므로, Batch의 크기는 32KB이하로 설정하는 것이 좋습니다. 따라서, **[32KB / 요소의 크기]**가 적당한 Batch Size라고 생각됩니다.

적당한 Batch 개수: [총 원소 수 / BatchSize] - 소수점 올림, 정수 (단, Batch 개수는 CPU 개수를 초과할 수 없다.)

[배치를 할당하는 예시]

```
float[] data = new float[10000];
NativeArray<float> result = new NativeArray<float>(10000, Allocator.TempJob);

var job = new MyJob
{
   data = new NativeSlice<float>(data),
   result = result
};

var scheduleParams = new JobsUtility.JobScheduleParameters(
   UnsafeUtility.AddressOf(ref job),
   MyJob.JobStructSize,
   2, // Batch Num
   new JobHandle()
);
```

```
JobHandle jobHandle = JobsUtility.ScheduleParallelFor(ref scheduleParams, job.data.Length, 8000);
```

[이해하지 못한 팁]

- 예약된 배치를 플러시하기
- 적시에 예약 및 완료 사용
- 데이터 종속성 검사