비동기 파일 로딩

목차

시작하며...

개인 프로젝트에서 구현해본 비동기 파일 로딩 방식에 대해서 이야기 하고자 합니다.

주 스레드가 멈추지 않고 백그라운드에서 에셋 파일을 읽고 가공하여 전달하는 일련의 과정을 다룹니다.

비동기 파일 로딩 과정

에셋을 불러들이는 과정을 살펴보겠습니다.

주로 2개의 스레드가 사용되며 데이터 가공 과정에서 분산 처리를 위해 4개의 스레드가 사용됩니다.

비동기 파일 로딩 과정

IOCP를 사용한 비동기 파일 I/O를 사용해서 구현했기 때문에 프로그램이 실행되면 파일시스템 스레드는 자신을 대기 큐에 등록합니다.

비동기 파일 로딩 과정

메인 스레드에서 에셋을 로딩하기 위해서 비동기 파일 읽기 함수를 호출하고
2차 저장장치에서 파일을 읽어 메인 메모리에 적재하게 됩니다.

비동기 파일 로딩 과정

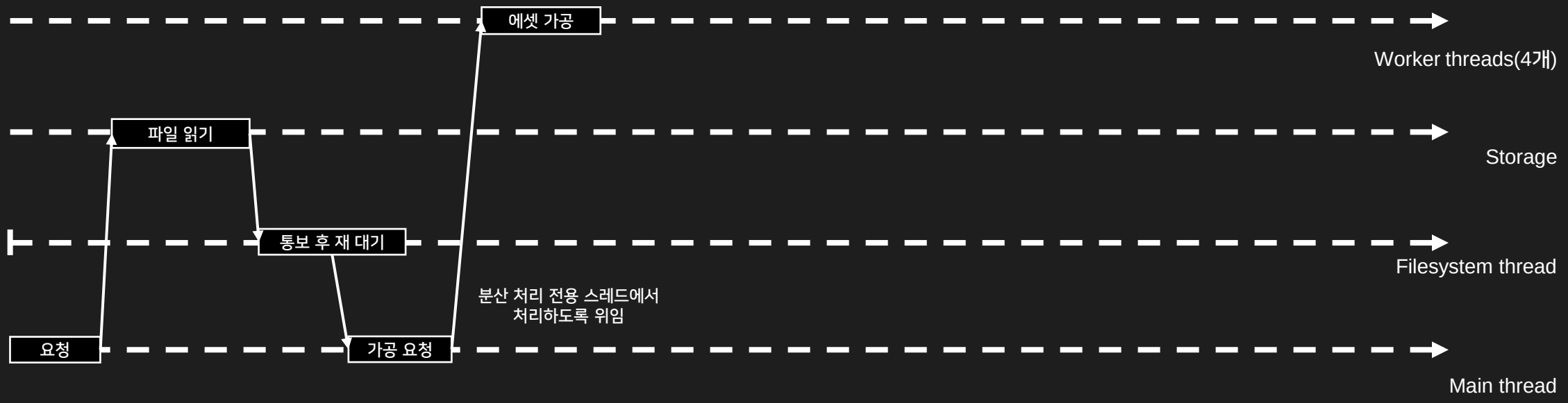
읽기 작업이 완료되면 대기하고 있던 파일시스템 스레드는 완료를 통보 받아 깨어납니다.

비동기 파일 로딩 과정

깨어난 파일시스템 스레드는 데이터의 버퍼와 크기를 메인 스레드로 전달한 다음 다시 자신을 대기 큐에 등록합니다.

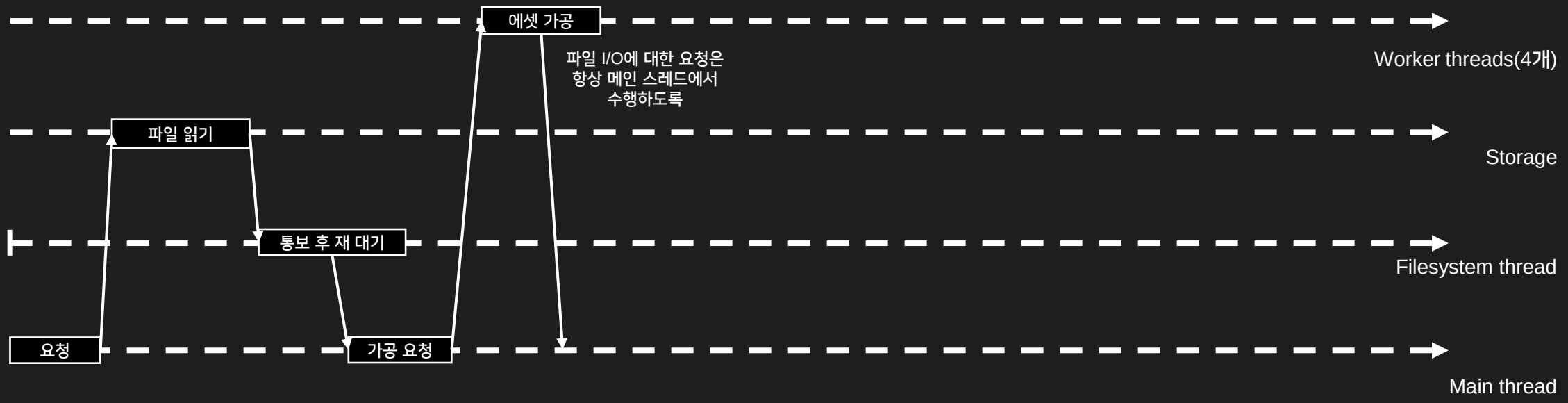
비동기 파일 로딩 과정

메인 스레드는 전달받은 데이터의 가공을 다른 워커 스레드로 위임합니다.



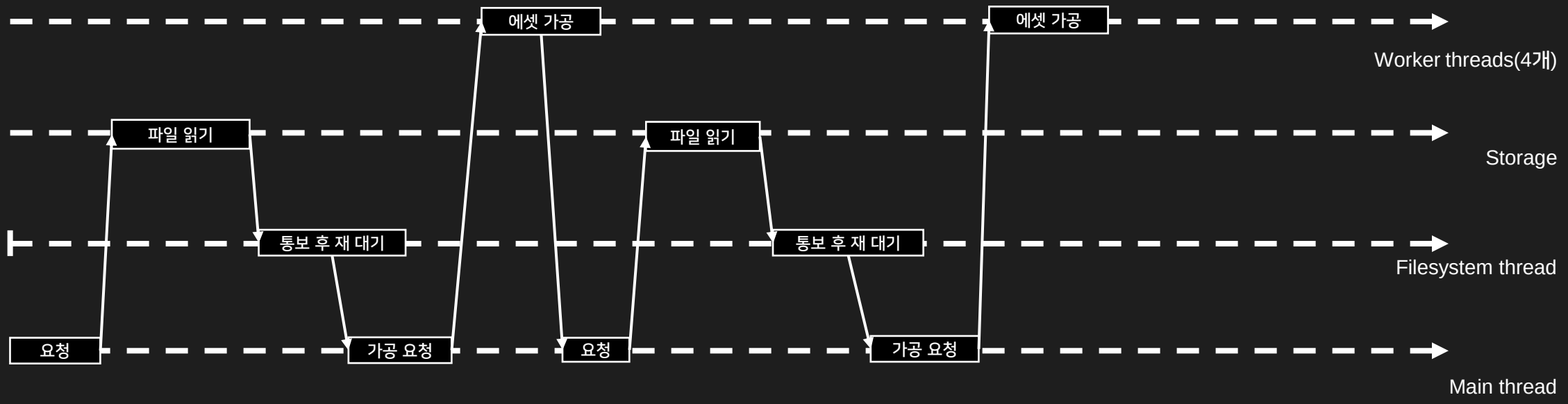
비동기 파일 로딩 과정

워커 스레드는 데이터를 게임에서 사용할 형태로 가공하면서 추가로 읽어야 할 파일이 있다면 이를 메인 스레드에 요청합니다.



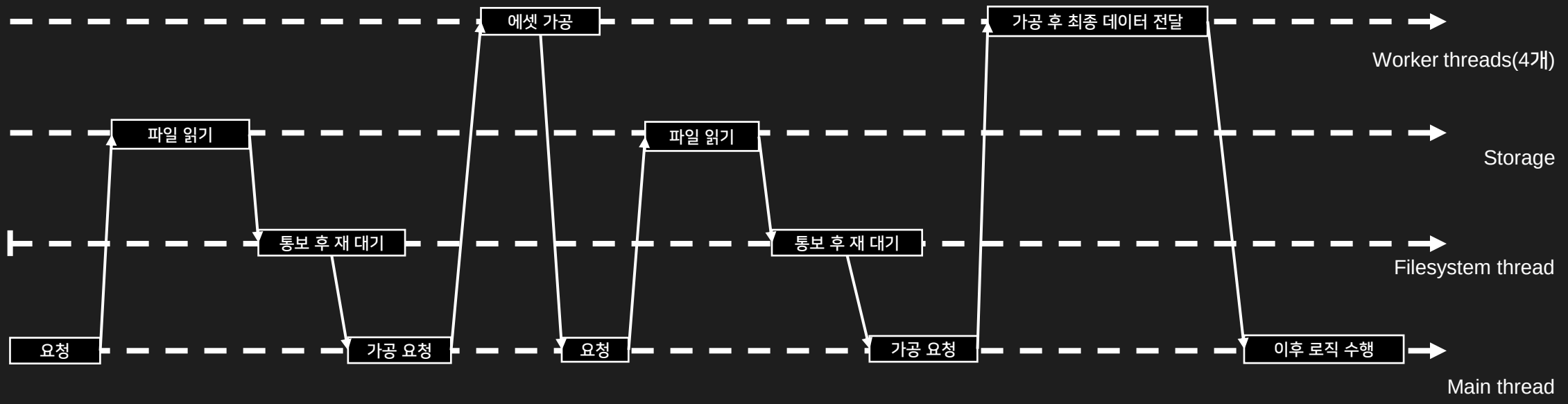
비동기 파일 로딩 과정

추가 파일에 대한 파일 읽기 진행 단계는 첫 요청과 동일합니다



비동기 파일 로딩 과정

모든 읽기 작업이 완료되면 최종적으로 가공된 에셋 데이터가 메인 스레드로 전달됩니다.



비동기 파일 로딩 과정

일련의 로딩 과정에서 크게 2가지 이슈를 처리해야 했습니다.

1. 특정 스레드로 특정 작업을 위임할 수 있는 구조가 필요.
2. 추가 파일 데이터에 대한 읽기 작업 요청으로 인한 최종 완료 통보 시점 지연.

특정 스레드로 특정 작업 위임하기

기본적인 구현 방법은 태스크 시스템 형태의 스레드 풀입니다.

여러 개의 스레드를 역할을 나눠 미리 생성하였습니다.

```
1 EngineTaskScheduler::EngineTaskScheduler( ) : m_taskScheduler( MAX_ENGINE_THREAD_GROUP,
  ThreadType::WorkerThreadCount )
2 { }
```

특정 스레드로 특정 작업 위임하기

생성하는 스레드의 역할은 열거형으로 다음과 같이 정의되어 있습니다.
주 스레드인 GameThread를 제외한 6개의 스레드가 생성됩니다.

```
namespace ThreadType
{
    enum
    {
        FileSystemThread,
        WorkerThread0,
        WorkerThread1,
        WorkerThread2,
        WorkerThread3,
        RenderThread,
        GameThread,
        WorkerThreadCount = GameThread,
    };
}
```

특정 스레드로 특정 작업 위임하기

그리고 태스크에는 workerAffinity 라는 변수를 통해서 어떤 스레드에서 작업을 수행할지 지정합니다.

```
1 class TaskBase
2 {
3   // ...
4 private:
5     std::size_t m_workerAffinity;
6 };
```

workerAffinity는 스레드의 열거형을 기반으로 하여 비트 시프트를 통해 계산 합니다.

```
1 template <std::size_t... N>
2 constexpr std::size_t WorkerAffinityMask( )
3 {
4     return ( ( 1 << N ) | ... );
5 }
```

특정 스레드로 특정 작업 위임하기

이렇게 계산된 workerAffinityMask 와 스레드의 타입을 검사하여 해당 태스크를 스레드에서 실행할지를 결정합니다.

```cpp
bool CheckWorkerAffinity( std::size_t workerId, std::size_t workerAffinity )
{
    assert( std::numeric_limits<std::size_t>::digits > workerId );
    return ( ( 1i64 << workerId ) & workerAffinity ) > 0;
}
```

```cpp
void WorkerThread( TaskScheduler* scheduler, Worker* worker )
{
// ...
    std::unique_lock taskLock( queue->m_taskLock );
    if ( queue->m_tasks.empty( ) == false )
    {
        task = queue->m_tasks.front( );

        if ( CheckWorkerAffinity( worker->m_threadType, task->WorkerAffinity( ) ) == false )
        {
            task = nullptr;
            continue;
        }

        queue->m_tasks.pop( );
        break;
    }
// ...
}
```

특정 스레드로 특정 작업 위임하기

실제 태스크 제출에는 다음과 같은 헬퍼 함수가 사용됩니다.

```
template <std::size_t... N, typename Lambda>
TaskHandle EnqueueThreadTask( Lambda lambda )
{
    ITaskScheduler* taskScheduler = GetInterface<ITaskScheduler>( );
    constexpr std::size_t affinityMask = WorkerAffinityMask<N...>( );
    TaskHandle taskGroup = taskScheduler->GetTaskGroup( );
    taskGroup.AddTask( Task<LambdaTask<Lambda>>::Create( affinityMask, lambda ) );
    bool success = taskScheduler->Run( taskGroup );
    assert( success );
    return taskGroup;
}
```

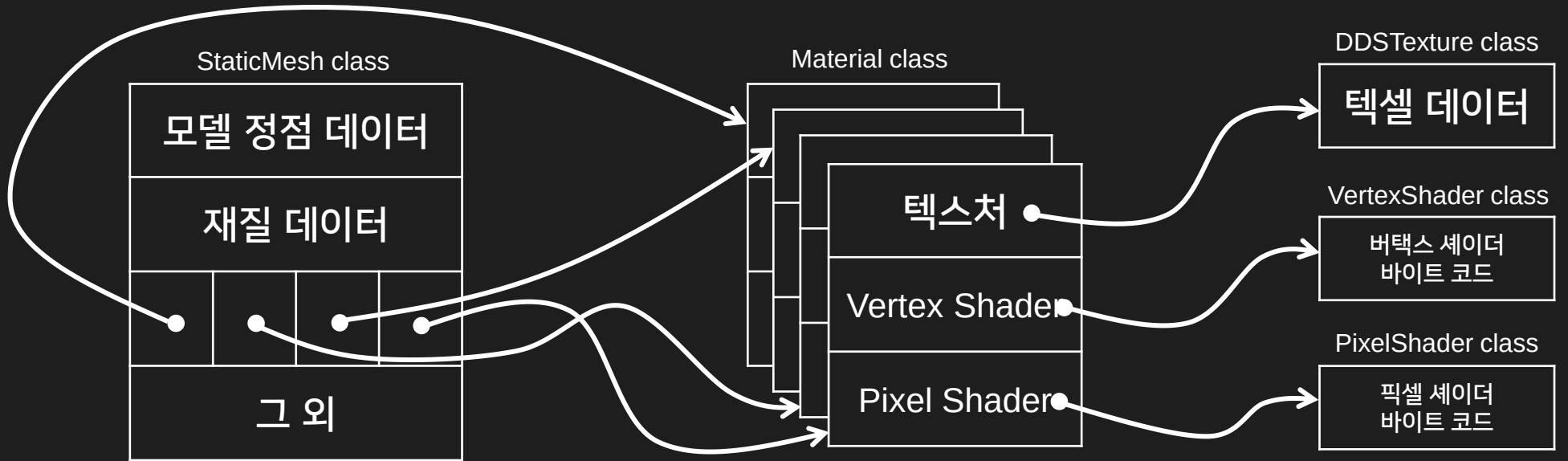
실제 호출 예시입니다. (주 스레드에서 실행되도록 작업을 위임)

```
EnqueueThreadTask<ThreadType::GameThread>( [this]( )
{
    UpdateState( );
} );
```

최종 데이터 가공 완료 통보하기

에셋의 종류에 따라서 추가적인 파일 읽기가 필요한 경우가 있습니다.

정적 메시 에셋의 클래스인 StaticMesh 클래스의 구조를 살펴보겠습니다.



최종 데이터 가공 완료 통보하기

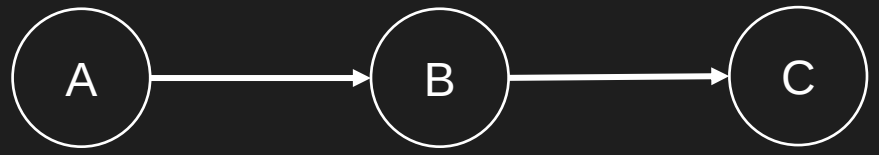
이런 구조로 인해서 모델 에셋을 온전하게 사용할 수 있는 시점은 모든 추가 파일의 데이터 읽기와 가공이 완료된 시점입니다.

추가 파일에 대한 처리의 완료를 기다리지 않고 로딩 완료 콜백을 호출한다면 필요한 데이터가 누락되는 등의 문제가 발생할 수 있습니다.

최종 완료 콜백은 이러한 처리가 모두 끝난 마지막에 호출되도록 적절하게 미뤄져야 합니다.

최종 데이터 가공 완료 통보하기

다음과 같은 단순한 선형 구조라면 완료 콜백을 지연시키기 위해 콜백체인을 이용할 수 있겠습니다.




```
1 void Read( auto callback )
2 {
3     // 읽기 완료 후 파일 가공 처리
4
5     // 종속된 파일을 읽어야 함
6     auto newCallback = [callback]()
7     {
8         // 파일 가공 처리 후 현재 파일에 대한 콜백 호출
9
10        // 완료 후 종속된 파일에 대한 콜백 호출
11        callback();
12    }
13
14    Read( newCallback );
15 }
```

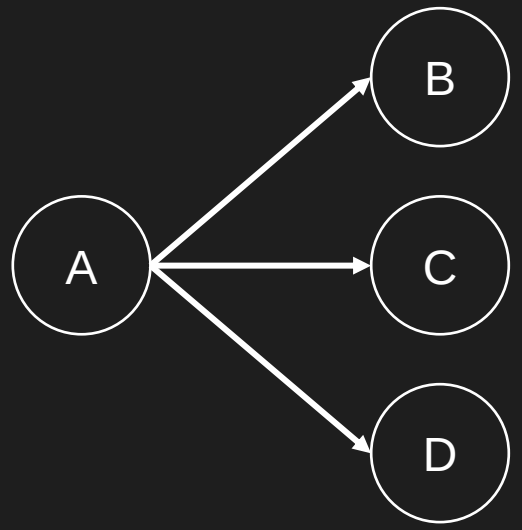


```
1 // 다음과 같이 콜백 체인이 구성 됨
2 auto bCallback = []()
3 {
4     // b 파일 가공 처리 후 b 파일에 대한 완료 통보
5
6     // 완료 후 a 파일의 콜백 호출
7     aCallback();
8 };
9
10 auto cCallback = []()
11 {
12     // c 파일 가공 처리 후 c 파일에 대한 완료 통보
13
14     // 완료 후 b 파일의 콜백 호출
15     bCallback();
16 }
```

최종 데이터 가공 완료 통보하기

하지만 이런 선형 구조는 추가 파일을 여러 개 로드해야 하는 경우에는 처리하기 어려워 집니다.

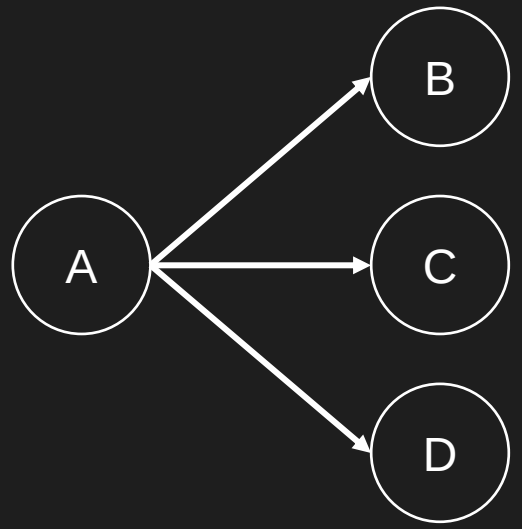
다음과 같은 구조에서 B 파일을 처리하고 난 다음 A 파일의 콜백을 바로 부를 수 없습니다. (C, D를 아직 안 읽었기 때문)




```
1 void Read( auto callback )
2 {
3     // 읽기 완료 후 파일 가공 처리
4
5     // 종속된 파일을 읽어야 함
6     auto newCallback = [callback]()
7     {
8         // 파일 가공 처리 후 현재 파일에 대한 콜백 호출
9
10        // 완료 후 종속된 파일에 대한 콜백 호출
11        callback();
12    }
13
14    Read( newCallback );
15 }
```

최종 데이터 가공 완료 통보하기

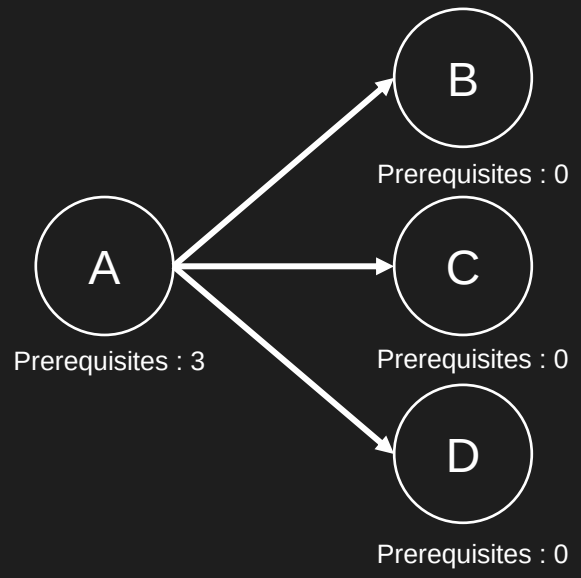
이를 해결 하기 위해 종속 관계를 저장해서 콜백 호출을 무시하도록 하였습니다.
주요 변수 2개를 살펴보겠습니다.




```cpp
// 몇몇 코드는 생략되어 있습니다.
class AssetLoaderHandle : public std::enable_shared_from_this<AssetLoaderHandle>
{
public:
    void AddPrerequisite( const AssetLoaderSharedHandle& prerequisite )
    {
        prerequisite->m_subSequentList.emplace_back( shared_from_this() );
        IncreasePrerequisite( );
    }

    bool HasPrerequisites( ) const
    {
        return m_prerequisites > 0;
    }

    void IncreasePrerequisite( )
    {
        ++m_prerequisites;
    }

    void DecreasePrerequisite( )
    {
        --m_prerequisites;
    }

private:
    std::vector<AssetLoaderSharedHandle> m_subSequentList;
    int m_prerequisites = 0;
};

```

최종 데이터 가공 완료 통보하기

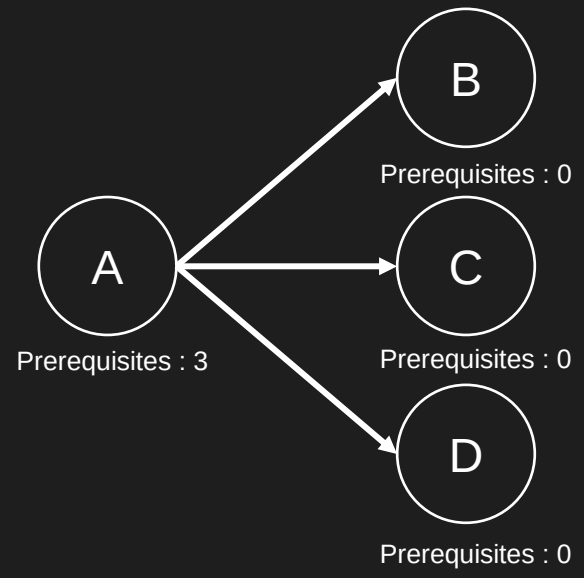
m_prerequisites 는 현재 에셋보다 먼저 처리되야 하는 에셋의 숫자입니다.




```cpp
// 몇몇 코드는 생략되어 있습니다.
class AssetLoaderHandle : public std::enable_shared_from_this<AssetLoaderHandle>
{
public:
    void AddPrerequisite( const AssetLoaderSharedHandle& prerequisite )
    {
        prerequisite->m_subSequentList.emplace_back( shared_from_this() );
        IncreasePrerequisite( );
    }

    bool HasPrerequisites( ) const
    {
        return m_prerequisites > 0;
    }

    void IncreasePrerequisite( )
    {
        ++m_prerequisites;
    }

    void DecreasePrerequisite( )
    {
        --m_prerequisites;
    }

private:
    std::vector<AssetLoaderSharedHandle> m_subSequentList;
    int m_prerequisites = 0;
};

```

최종 데이터 가공 완료 통보하기

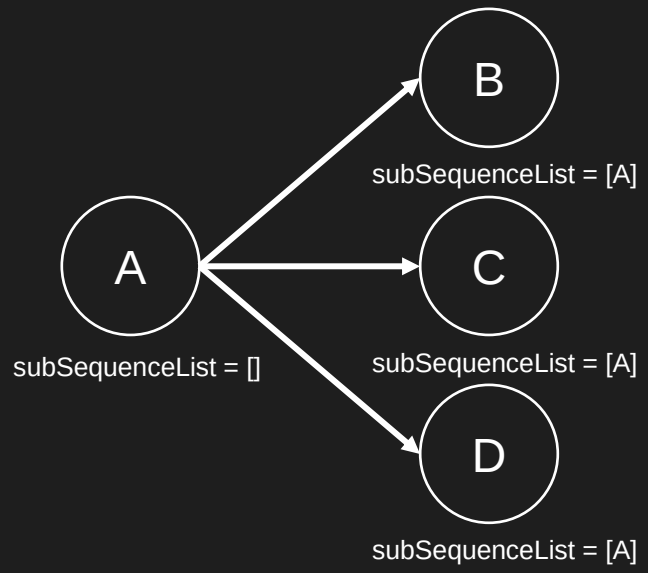
m_prerequisites 이 0보다 큰 경우 다음과 같이 콜백의 호출이 무시됩니다.




```cpp
void AssetLoaderHandle::ExecuteCompletionCallback( )
{
    assert( IsInGameThread( ) );

    if ( m_prerequisites > 0 )
    {
        return;
    }

    if ( m_loadComplete == false )
    {
        return;
    }

    if ( m_needPostProcess )
    {
        if ( auto pAsyncLoadable = std::static_pointer_cast<AsyncLoadableAsset>( m_loadedAsset ) )
        {
            pAsyncLoadable->PostLoad( );
        }
        m_needPostProcess = false;
    }

    if ( m_loadCompletionCallback.IsBound( ) )
    {
        m_loadCompletionCallback( m_loadedAsset );
    }

    for ( const auto& subSequent : m_subSequentList )
    {
        subSequent->DecreasePrerequisite( );
        subSequent->ExecuteCompletionCallback( );
    }

    m_subSequentList.clear( );
}
```

최종 데이터 가공 완료 통보하기

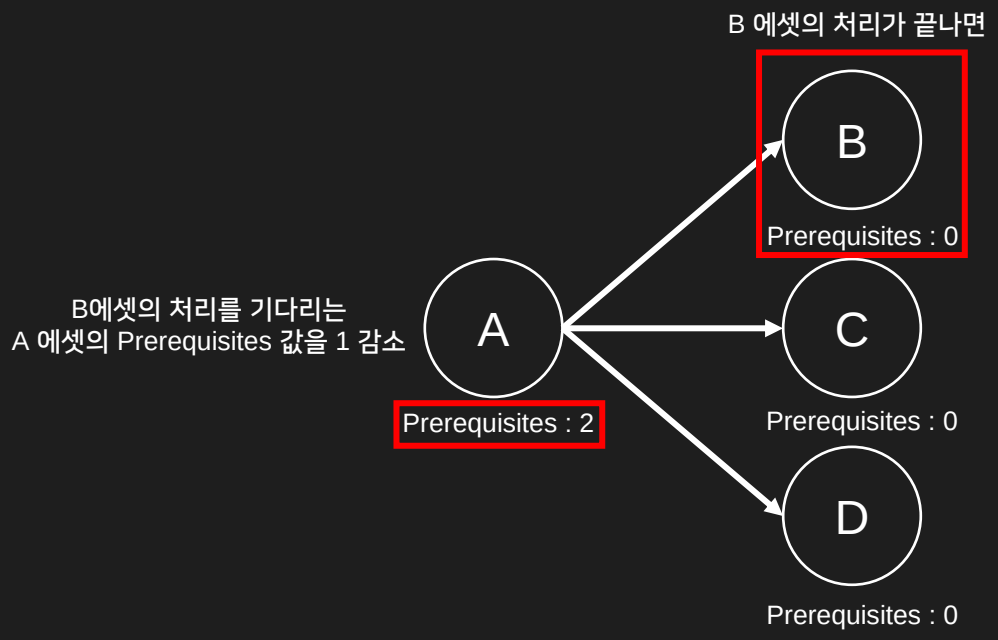
m_subSequentList 는 현재 에셋의 처리를 기다리고 있는 에셋 로더 핸들의 리스트입니다.




```cpp
// 몇몇 코드는 생략되어 있습니다.
class AssetLoaderHandle : public std::enable_shared_from_this<AssetLoaderHandle>
{
public:
    void AddPrerequisite( const AssetLoaderSharedHandle& prerequisite )
    {
        prerequisite->m_subSequentList.emplace_back( shared_from_this() );
        IncreasePrerequisite( );
    }

    bool HasPrerequisites( ) const
    {
        return m_prerequisites > 0;
    }

    void IncreasePrerequisite( )
    {
        ++m_prerequisites;
    }

    void DecreasePrerequisite( )
    {
        --m_prerequisites;
    }

private:
    std::vector<AssetLoaderSharedHandle> m_subSequentList;
    int m_prerequisites = 0;
};

```

최종 데이터 가공 완료 통보하기

현재 에셋의 처리가 끝나면 m_subSequentList 에 등록된 핸들에 대해 DecreasePrerequisite( ) 함수를 호출해서 m_prerequisites 를 감소 시킵니다.

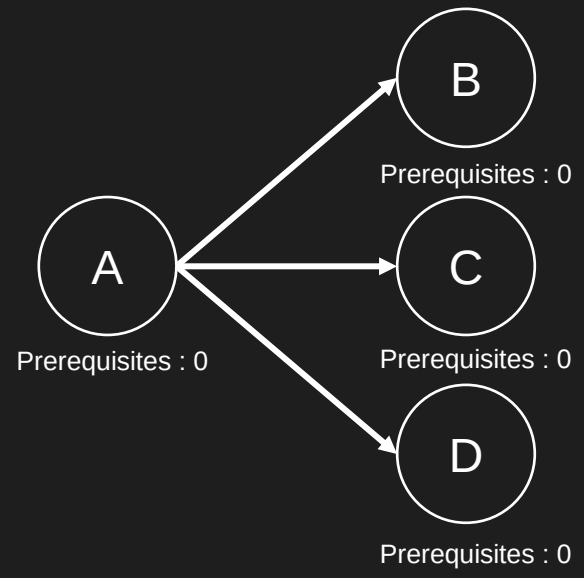



```cpp
void AssetLoaderHandle::ExecuteCompletionCallback( )
{
    assert( IsInGameThread( ) );

    if ( m_prerequisites > 0 )
    {
        return;
    }

    if ( m_loadComplete == false )
    {
        return;
    }

    if ( m_needPostProcess )
    {
        if ( auto pAsyncLoadable = std::static_pointer_cast<AsyncLoadableAsset>( m_loadedAsset ) )
        {
            pAsyncLoadable->PostLoad( );
        }
        m_needPostProcess = false;
    }

    if ( m_loadCompletionCallback.IsBound( ) )
    {
        m_loadCompletionCallback( m_loadedAsset );
    }

    for ( const auto& subSequent : m_subSequentList )
    {
        subSequent->DecreasePrerequisite( );
        subSequent->ExecuteCompletionCallback( );
    }

    m_subSequentList.clear( );
}
```

```cpp
void DecreasePrerequisite( )
{
    --m_prerequisites;
}
```

최종 데이터 가공 완료 통보하기

종속된 모든 에셋의 처리가 끝나게 되면 m_prerequisites의 값이 0이 되어 최종적으로 A 에셋의 콜백을 호출할 수 있게 됩니다.




```
1  void AssetLoaderHandle::ExecuteCompletionCallback( )
2  {
3      assert( IsInGameThread( ) );
4
5      if ( m_prerequisites > 0 )
6      {
7          return;
8      }
9
10     if ( m_loadComplete == false )
11     {
12         return;
13     }
14
15     if ( m_needPostProcess )
16     {
17         if ( auto pAsyncLoadable = std::static_pointer_cast<AsyncLoadableAsset>( m_loadedAsset ) )
18         {
19             pAsyncLoadable->PostLoad( );
20         }
21         m_needPostProcess = false;
22     }
23
24     if ( m_loadCompletionCallback.IsBound( ) )
25     {
26         m_loadCompletionCallback( m_loadedAsset );
27     }
28
29     for ( const auto& subSequent : m_subSequentList )
30     {
31         subSequent->DecreasePrerequisite( );
32         subSequent->ExecuteCompletionCallback( );
33     }
34
35     m_subSequentList.clear( );
36 }
37
```

최종 데이터 가공 완료 통보하기

이런 식으로 완료 콜백을 지연합니다. 이제 에셋 간 종속 관계를 파일 처리 과정에서 적절하게 연결하면 됩니다.

저는 현재 처리중인 에셋에 대한 에셋 로더 핸들을 다음과 같이 저장하고 파일 시스템에 비동기 읽기 요청 시 연결해주는 형태로 구현하였습니다.

```cpp
EnqueueThreadTask<WorkerThreads>(
[this, buffer, bufferSize, handle]( )
{
    SetHandleInProcess( handle );

    Archive ar( buffer, bufferSize );
    std::size_t assetID = 0;

    ar << assetID;

    std::shared_ptr<IAsyncLoadableAsset> newAsset( AssetFactory::GetInstance( ).CreateAsset( assetID ) );
    if ( newAsset != nullptr )
    {
        handle->SetLoadedAsset( newAsset );
        newAsset->Serialize( ar );

        if ( handle->HasPrerequisites( ) == false )
        {
            EnqueueThreadTask<ThreadType::GameThread>(
                [handle, newAsset]( )
            {
                assert( IsInGameThread( ) );
                handle->ExecuteCompletionCallback( );
            } );
        }
    }

    SetHandleInProcess( nullptr );
    delete[] buffer;
} );
```

읽기 완료 후 에셋 가공을 워커 스레드로 위임하는 코드

```cpp
thread_local static AssetLoaderSharedHandle AssetHandleInProcess;

void AssetLoader::SetHandleInProcess( const AssetLoaderSharedHandle& handle )
{
    AssetHandleInProcess = handle;
}
```

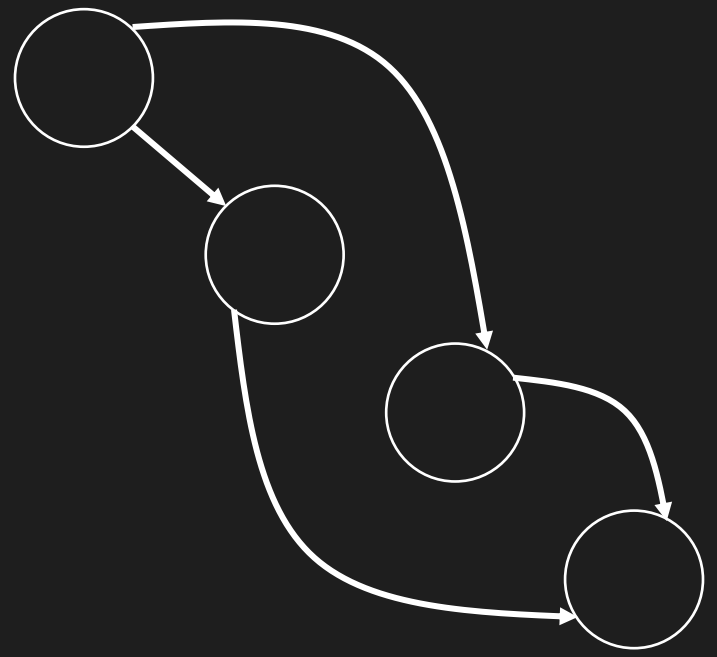
```cpp
void AssetLoader::AddPrerequisiteToDependantAsset( const AssetLoaderSharedHandle& handle )
{
    if ( ( AssetHandleInProcess != nullptr ) && handle->IsLoadingInProgress( ) )
    {
        AssetHandleInProcess->AddPrerequisite( handle );
    }
}
```

```cpp
void AddPrerequisite( const AssetLoaderSharedHandle& prerequisite )
{
    prerequisite->m_subSequentList.emplace_back( shared_from_this() );
    IncreasePrerequisite( );
}
```

최종 데이터 가공 완료 통보하기

지금까지의 과정은 DAG(Directed acyclic graph) 를 파일 읽기 과정에서 만드는 것입니다.

m_subSequentList 가 진입 간선(Incoming edge)의 개수 라고 생각하시면 될 것입니다.



마치며...

제가 구현해본 비동기 파일 로딩 과정은 이것이 전부입니다.

현재 로딩 중인 파일에 대한 읽기 요청이 또 들어 온 경우의 처리 등 자잘한 예외 처리 사항이 남아 있긴 하지만 핵심내용은 아니라 제외하였습니다.

더 자세한 코드를 보고 싶으시다면...

ppt의 코드는 설명을 위해 전체 코드의 일부분을 발췌하였습니다.
전체 코드는 아래 링크를 참조 바랍니다.

- https://github.com/xtozero/SSR/blob/multi-thread/Source/Engine/Public/AssetLoader/AssetLoader.h
- https://github.com/xtozero/SSR/blob/multi-thread/Source/Engine/Private/AssetLoader/AssetLoader.cpp
- https://github.com/xtozero/SSR/blob/multi-thread/Source/Core/Public/TaskScheduler.h
- https://github.com/xtozero/SSR/blob/multi-thread/Source/Core/Private/TaskScheduler.cpp