Unreal Fest Gold Coast 2024

How to Benefit from Multithreading in Your Unreal Engine Projects

Alex Stevens
Senior Software Engineer II
Gameloft

Agenda

Threading in Unreal
Brief outline of the threading systems.

Runnable Threads
For long running tasks.

Task System
A fully featured task manager with dependencies.

Threading Primitives
Atomics, Locks, Lock-free Lists, Synch Events, etc.

Thread Safety
Common gotchas, UObjects, BPs, etc.

Debugging Threads
For when things go awry.

Demos
Putting theory into practice.

Who Am I?

- Senior Software Engineer at Gameloft
- Previously: Evangelist at Epic Games
- Consulting and contracting for studios
- Worked on VR games for a long time
- Have a love for optimisation

Why Threading?

CPUs are wiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiide

Multi-core processors have become the norm and are becoming more and more parallelised in architecture.

Parallelisable Jobs

Some tasks are prime candidates to being parallelised and have a very clear road to an appropriate architecture.

Lazy Work

Sometimes some jobs just don’t need to be high priority nor be done on the game thread. So we fire and forget them.

Why this talk?

The number of teams writing their own systems is too damn high!

Threading in Unreal

- A number of threading systems in Unreal: some old, some new
- Built-in features that are heavily threaded like Mass and PCG
- What about threading our code base?

Systems:

- Runnable Threads
- Async, Promises, and Futures
- Thread Pool
- Task Graph
- Task System

The First Rule of Threading Club

Don’t optimise without profiling first.

Runnable Threads

- Intended to be long-lived
- Creating an FRunnableThread is expensive
- Render, RHI, and Audio threads are some examples
- Subclass FRunnable
- Create an FRunnableThread

```
class FRunnableWork : public FRunnable
{
public:
    virtual uint32 Run() override
    {
       // Do really long winded, but fun stuff here

       // When complete, return an exit status
       return 0;
    }
};

// Could also use TUniquePtr for these
FRunnableWork* Runnable = new FRunnableWork;
FRunnableThread* RunnableThread =
FRunnableThread::Create(Runnable,
TEXT("RunnableWork"));

// Optionally kill or run the thread forever
RunnableThread->Kill(true);

// Be sure to clean up though
delete RunnableThread;
delete Runnable;
```

Thread Pool

- Pre-creates a set number of threads that idle until provided work
- Programmers can quickly execute threaded tasks without waiting for a new thread to spool up
- The basis of threading systems in UE
- Jobs defined by implementing IQueuedWork
- Create your own pool via FQueuedThreadPool::Allocate or use global GThreadPool
- Also have FQueuedThreadPoolWrapper which allows max concurrency

```cpp
class FThreadWork : public IQueuedWork
{
public:
    virtual void DoThreadedWork() override
    {
        // Do the fun stuff here
    }
    virtual void Abandon() override
    {
        // Handle abandonment issues here
    }
};

// Queue up the work
FThreadWork* ThreadWork = new FThreadWork;
GThreadPool->AddQueuedWork(ThreadWork);

// Or potentially cancel it early (and clean up)
GThreadPool->RetractQueuedWork(ThreadWork);
delete ThreadWork;
```

Async, Promises, and Futures

- Can use Async to kick off short-lived lambda tasks
- ParallelFor to make short work of wide jobs
- Helper function to other engine threading systems
- Promises can be used to sign a contract that a result will be returned later
- Futures then can listen to that contract

```cpp
TFuture<bool> Future = Async(EAsyncExecution::ThreadPool, []
{
    // Do some work... return a result
    return true;
});

// Wait for and get result
check(Future.Get());

// ======================================================

TArray<FVector> Data;
ParallelFor(Data.Num(), [&Data](const int32 Index)
{
    const FVector& Location = Data[Index];
    // Do parallel work on items in Data
});

// ======================================================

TPromise<bool> Promise;
TFuture<bool> Future = Promise.GetFuture();
AsyncTask(ENamedThreads::ActualRenderingThread,
[Promise = MoveTemp(Promise)] mutable
{
    // Do work
    Promise.SetValue(true);
});

// Wait for and get result
check(Result.Get());
```

Threading Primitives

- TAtomic template
- FCriticalSection (long), FSpinLock (short), and TScopeLock
- Do synchronisations via ::GetSynchEventFromPool and ::ReturnSynchEventToPool

```cpp
TAtomic<int32> AtomicInteger;

// ===========================================

FCriticalSection Section;
UE::TScopeLock<FCriticalSection> Lock(&Section);

// ===========================================

UE::FSpinLock SpinLock;
UE::TScopeLock Lock = SpinLock;

// ===========================================

FEvent* Event =
FPlatformProcess::GetSynchEventFromPool();

// Wait for the event to be triggered
Event->Wait();

// In another thread
Event->Trigger();
FPlatformProcess::ReturnSynchEventToPool(Event);
```

Threading Primitives

- Lock-free Lists in Containers/LockFreeList.h
- LIFO, FIFO, etc. and pointer-based types
- Uses atomics with a linked list
- Extremely useful for feeding a job queue

```
typedef TFunction<void()> FTaskFunction;

TLockFreePointerListFIFO<FTaskFunction,
      PLATFORM_CACHE_LINE_SIZE> Queue;

// From a thread
Queue.Push(new FTaskFunction(Function));

// From another thread
while (const FTaskFunction* Task = Queue.Pop())
{
    (*Task)();
    delete Task;
}
```

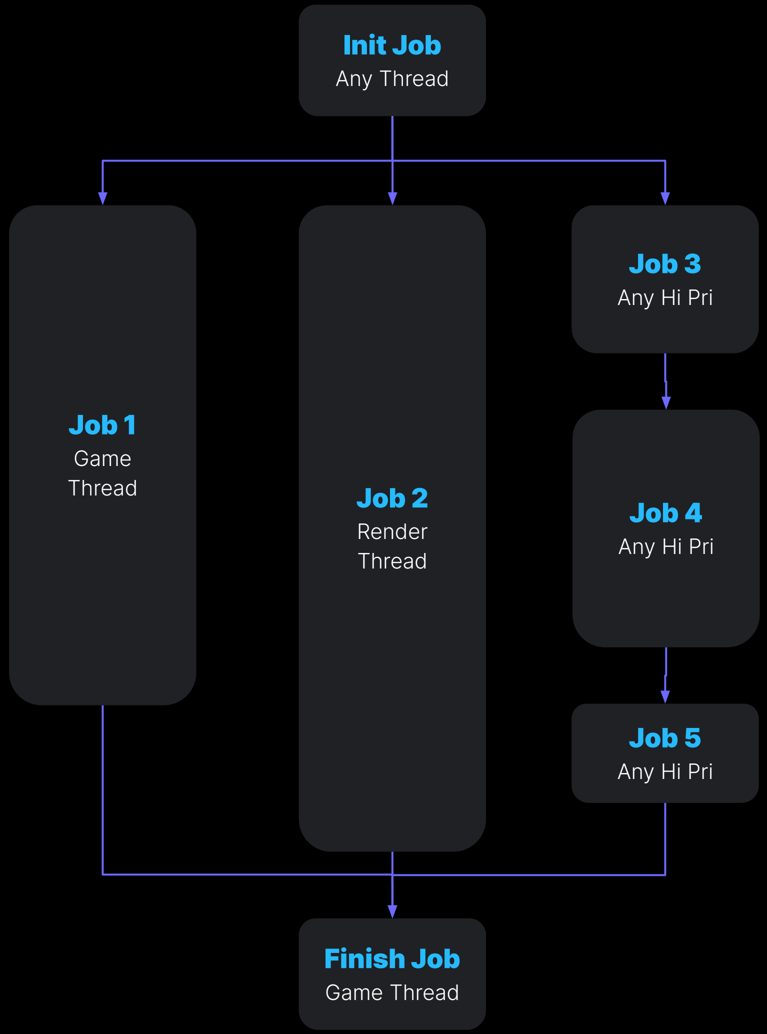
The Second Rule of Threading Club

Don’t write your own threading system!

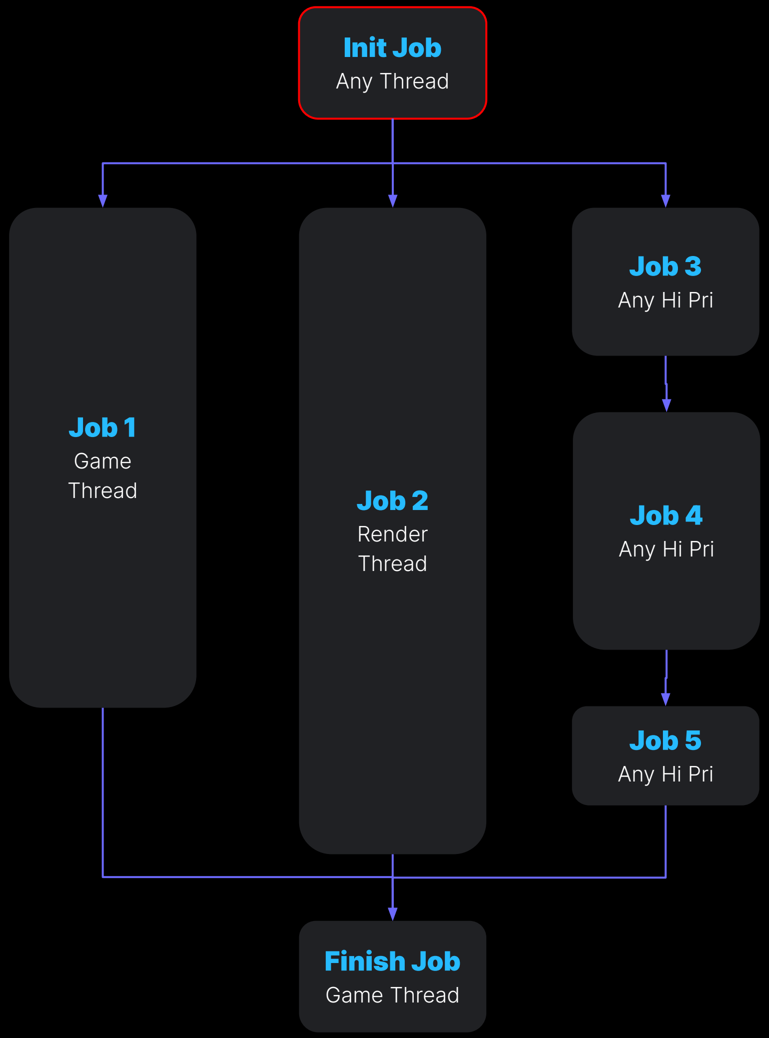
Task Graph

- Threading system that allows tasks to be queued up with prerequisites
- This is a Directed Acyclic Graph (DAG)
- Fantastic for going wide and long with threaded tasks

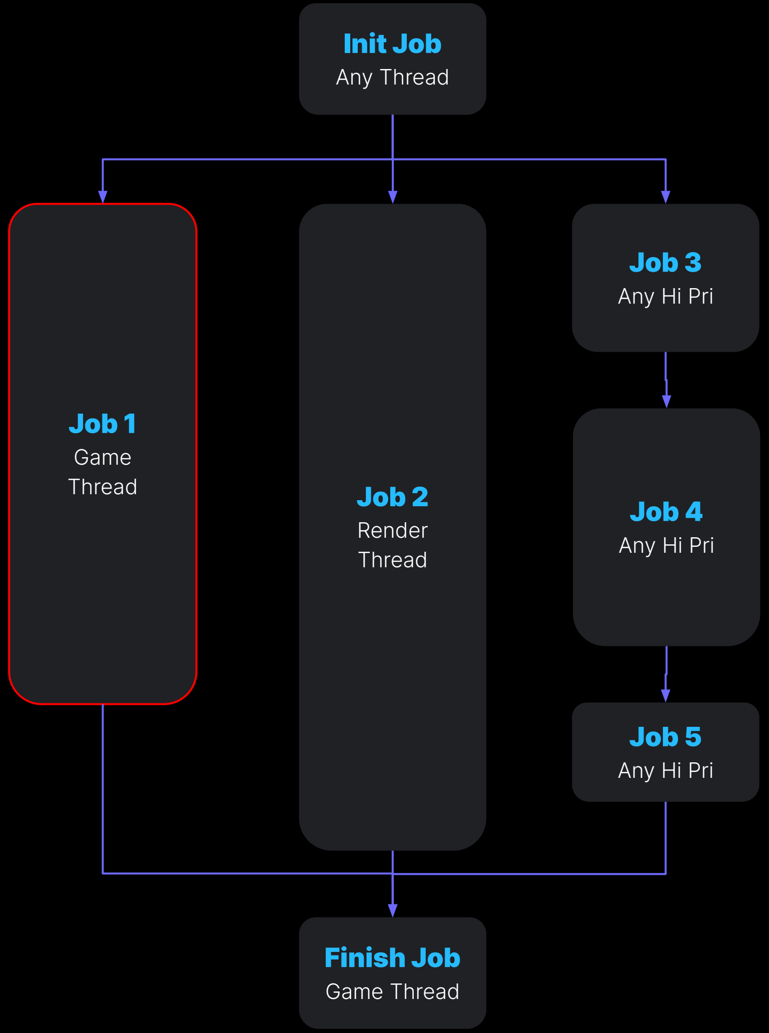
```
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
       // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```



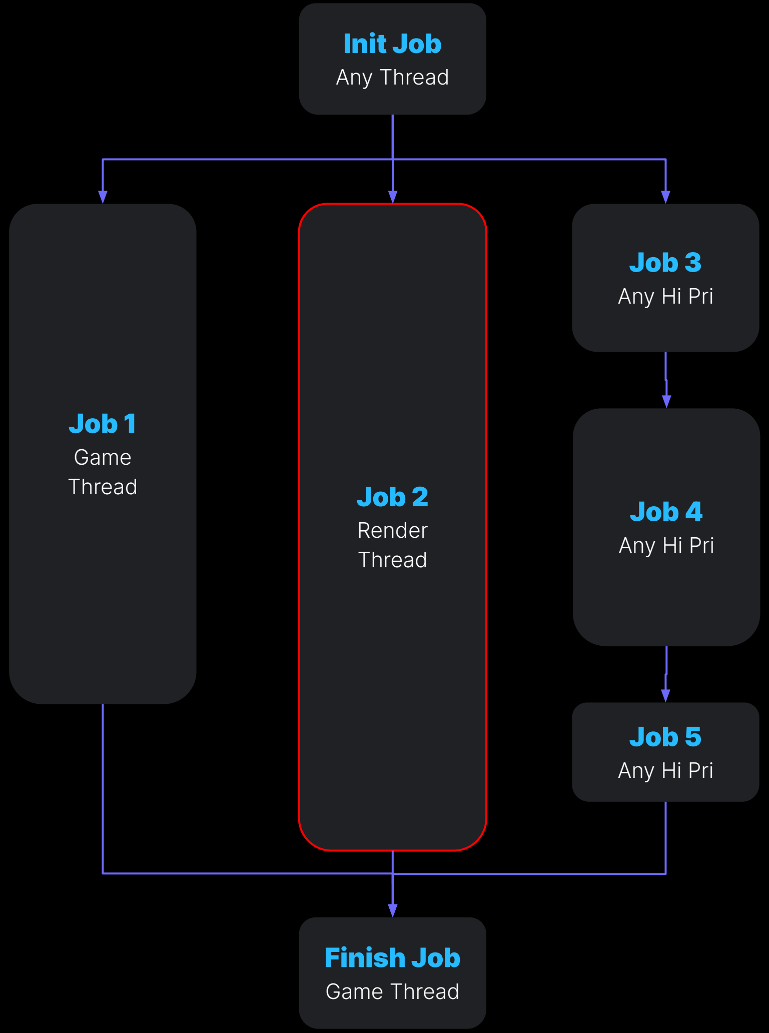
```cpp
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
        // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```



```
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
        // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```



```cpp
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
        // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```



Init Job
Any Thread

Job 1
Game Thread

Job 2
Render Thread

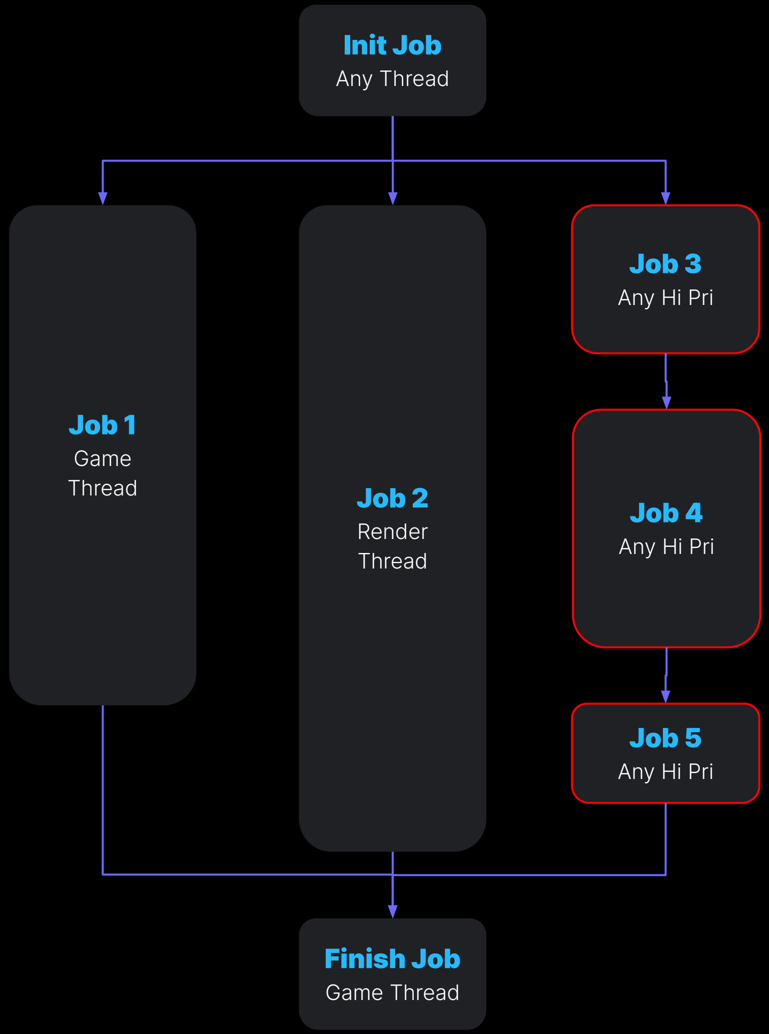
Job 3
Any Hi Pri

Job 4
Any Hi Pri

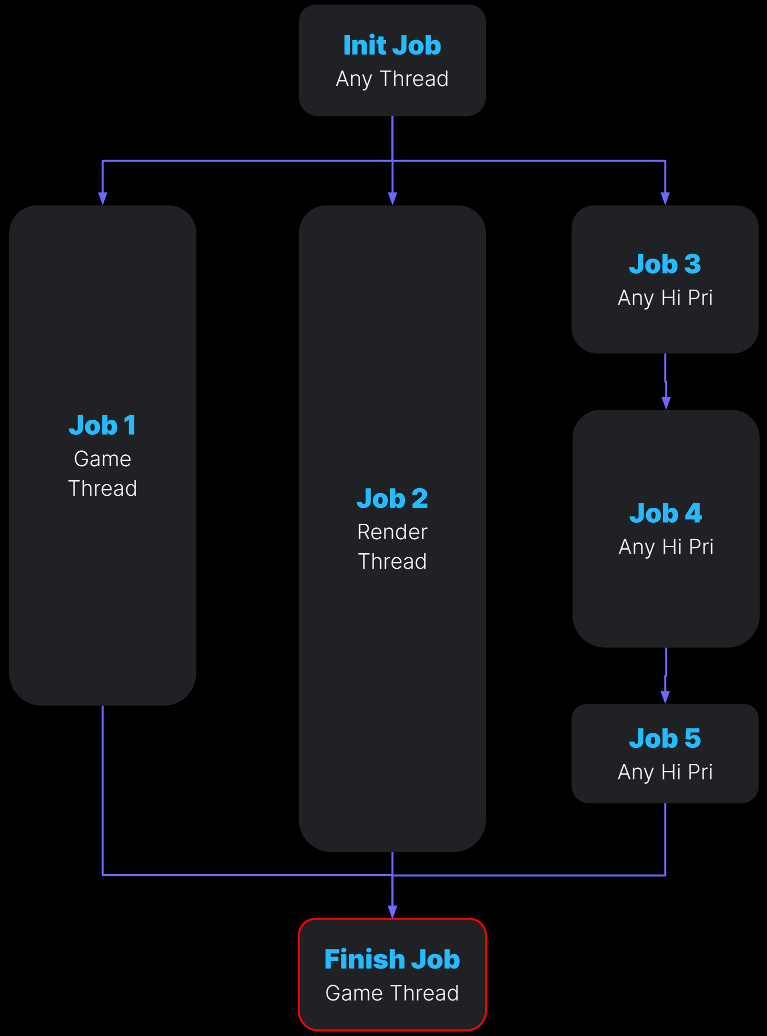
Job 5
Any Hi Pri

Finish Job
Game Thread

```
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
       // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```



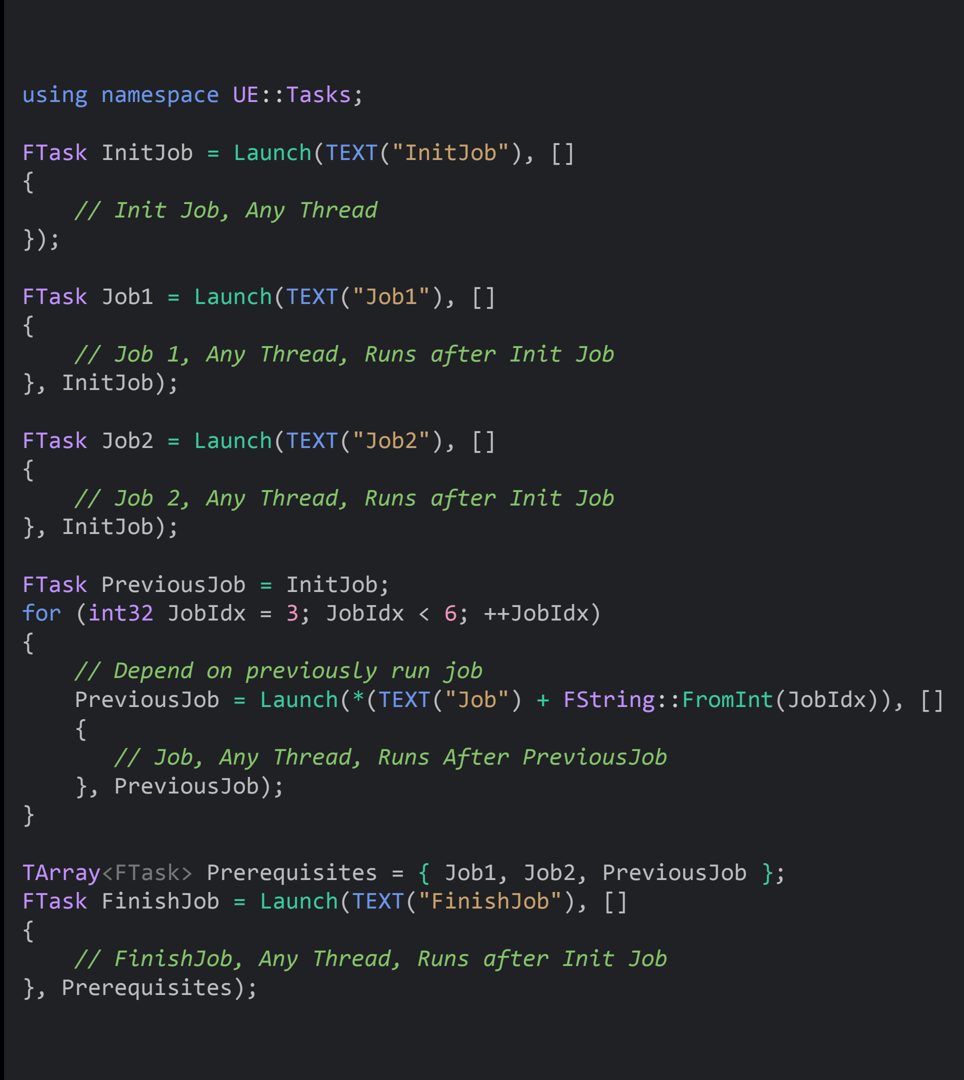
```
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
       // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```



```
FGraphEventRef InitJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // InitJob, Any Thread, Runs immediately
}, TStatId(), nullptr, ENamedThreads::AnyThread);

FGraphEventRef Job1 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 1, Game Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef Job2 = FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Job 2, Render Thread, Runs After InitJob
}, TStatId(), InitJob, ENamedThreads::GameThread);

FGraphEventRef PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = FFunctionGraphTask::CreateAndDispatchWhenReady([]
    {
        // Job, Any Hi Pri, Runs After PreviousJob
    }, TStatId(), PreviousJob, ENamedThreads::AnyHiPriThreadNormalTask);
}

FGraphEventArray Prerequisites = { Job1, Job2, PreviousJob };
FFunctionGraphTask::CreateAndDispatchWhenReady([]
{
    // Finish Job, GameThread, Runs After Everything else
}, TStatId(), &Prerequisites, ENamedThreads::GameThread);
```

Task System

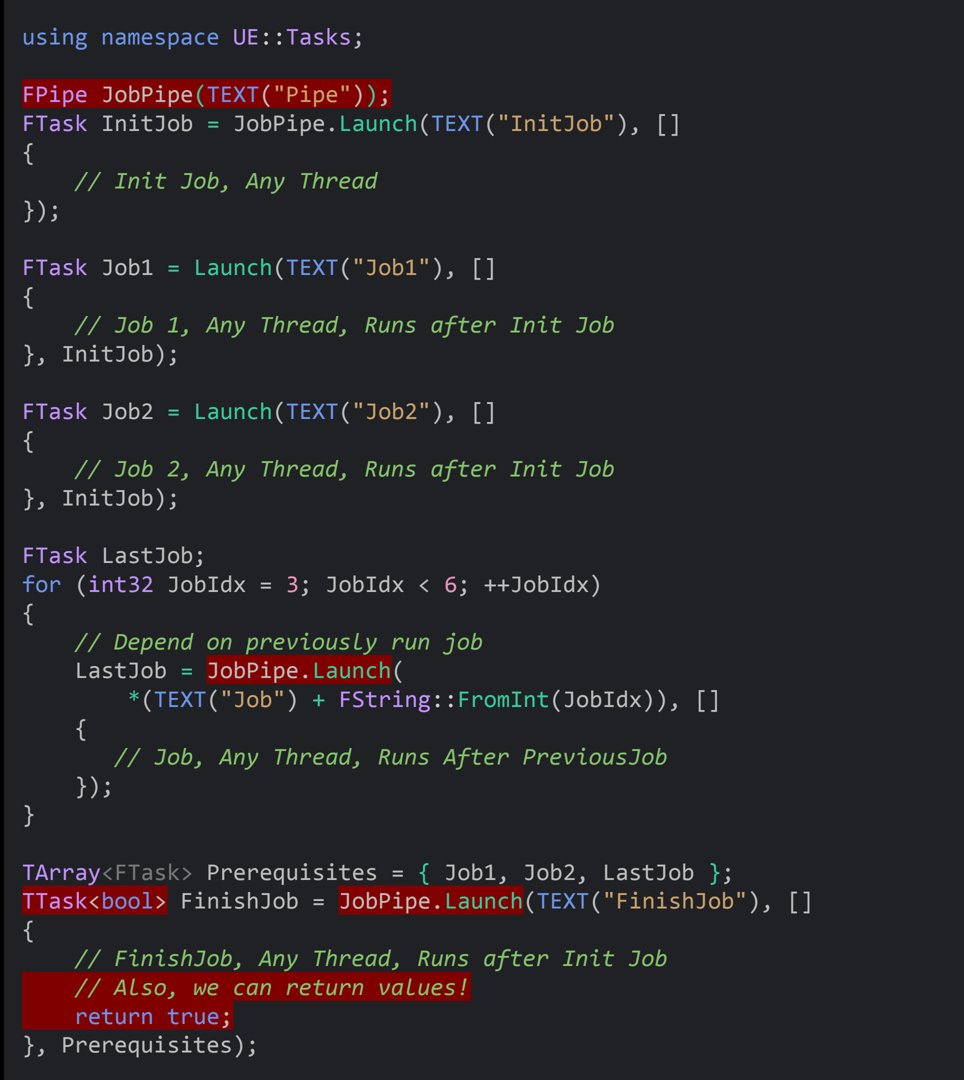
- An improvement on Task Graph
- Has nicer debugging features



```
using namespace UE::Tasks;

FTask InitJob = Launch(TEXT("InitJob"), []
{
    // Init Job, Any Thread
});

FTask Job1 = Launch(TEXT("Job1"), []
{
    // Job 1, Any Thread, Runs after Init Job
}, InitJob);

FTask Job2 = Launch(TEXT("Job2"), []
{
    // Job 2, Any Thread, Runs after Init Job
}, InitJob);

FTask PreviousJob = InitJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    PreviousJob = Launch(*(TEXT("Job") + FString::FromInt(JobIdx)), []
    {
       // Job, Any Thread, Runs After PreviousJob
    }, PreviousJob);
}

TArray<FTask> Prerequisites = { Job1, Job2, PreviousJob };
FTask FinishJob = Launch(TEXT("FinishJob"), []
{
    // FinishJob, Any Thread, Runs after Init Job
}, Prerequisites);
```

Task System

- Let’s write this the Task System way
- Chaining tasks using Pipes is clearer and safer



```
using namespace UE::Tasks;

FPipe JobPipe(TEXT("Pipe"));
FTask InitJob = JobPipe.Launch(TEXT("InitJob"), []
{
    // Init Job, Any Thread
});

FTask Job1 = Launch(TEXT("Job1"), []
{
    // Job 1, Any Thread, Runs after Init Job
}, InitJob);

FTask Job2 = Launch(TEXT("Job2"), []
{
    // Job 2, Any Thread, Runs after Init Job
}, InitJob);

FTask LastJob;
for (int32 JobIdx = 3; JobIdx < 6; ++JobIdx)
{
    // Depend on previously run job
    LastJob = JobPipe.Launch(
        *(TEXT("Job") + FString::FromInt(JobIdx)), []
    {
       // Job, Any Thread, Runs After PreviousJob
    });
}

TArray<FTask> Prerequisites = { Job1, Job2, LastJob };
TTask<bool> FinishJob = JobPipe.Launch(TEXT("FinishJob"), []
{
    // FinishJob, Any Thread, Runs after Init Job
    // Also, we can return values!
    return true;
}, Prerequisites);
```

Task System

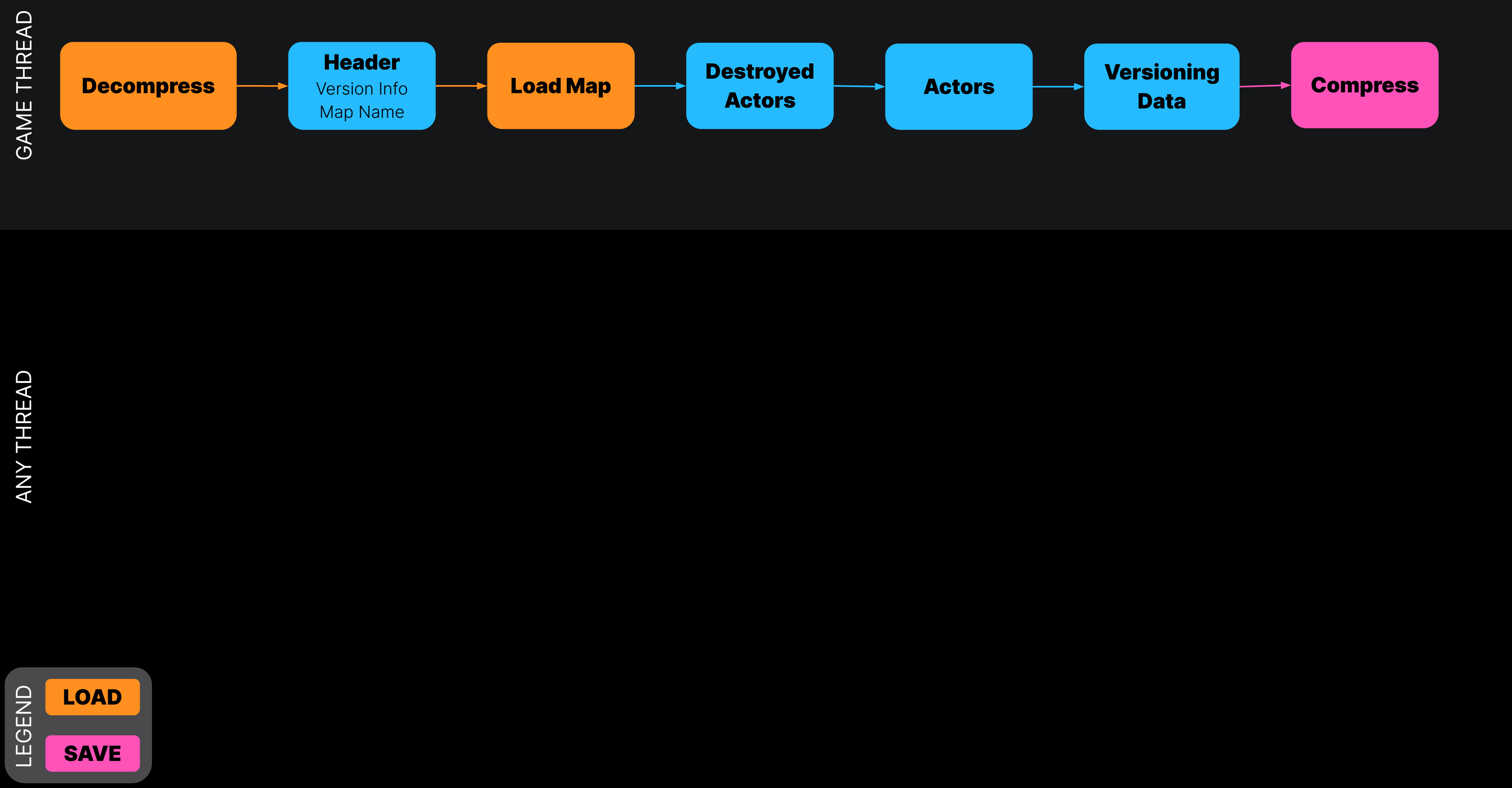
- Can conditionally nest tasks from within other tasks
- Has own synch events, that can be used as dependencies!
- However, currently doesn’t support directly executing on specified threads like the game thread...
- TASKGRAPH_NEW_FRONTEND=1

```
FTask Job1 = Launch(TEXT("Job1"), []
{
    FTaskEvent GameThreadEvent(UE_SOURCE_LOCATION);

    AsyncTask(ENamedThreads::GameThread, [GameThreadEvent] mutable
    {
       // Job 1, Game Thread, Runs after Init Job
       GameThreadEvent.Trigger();
    });

    // Now the Job 1 task will only complete when
    // this task event is triggered
    AddNested(GameThreadEvent);
}, InitJob);
```

Thread Debugging is Easier!

Run Insights with “task” channel.

Save Game System



GAME THREAD

Decompress

Header
Version Info
Map Name

Load Map

Destroyed Actors

Actors

Versioning Data

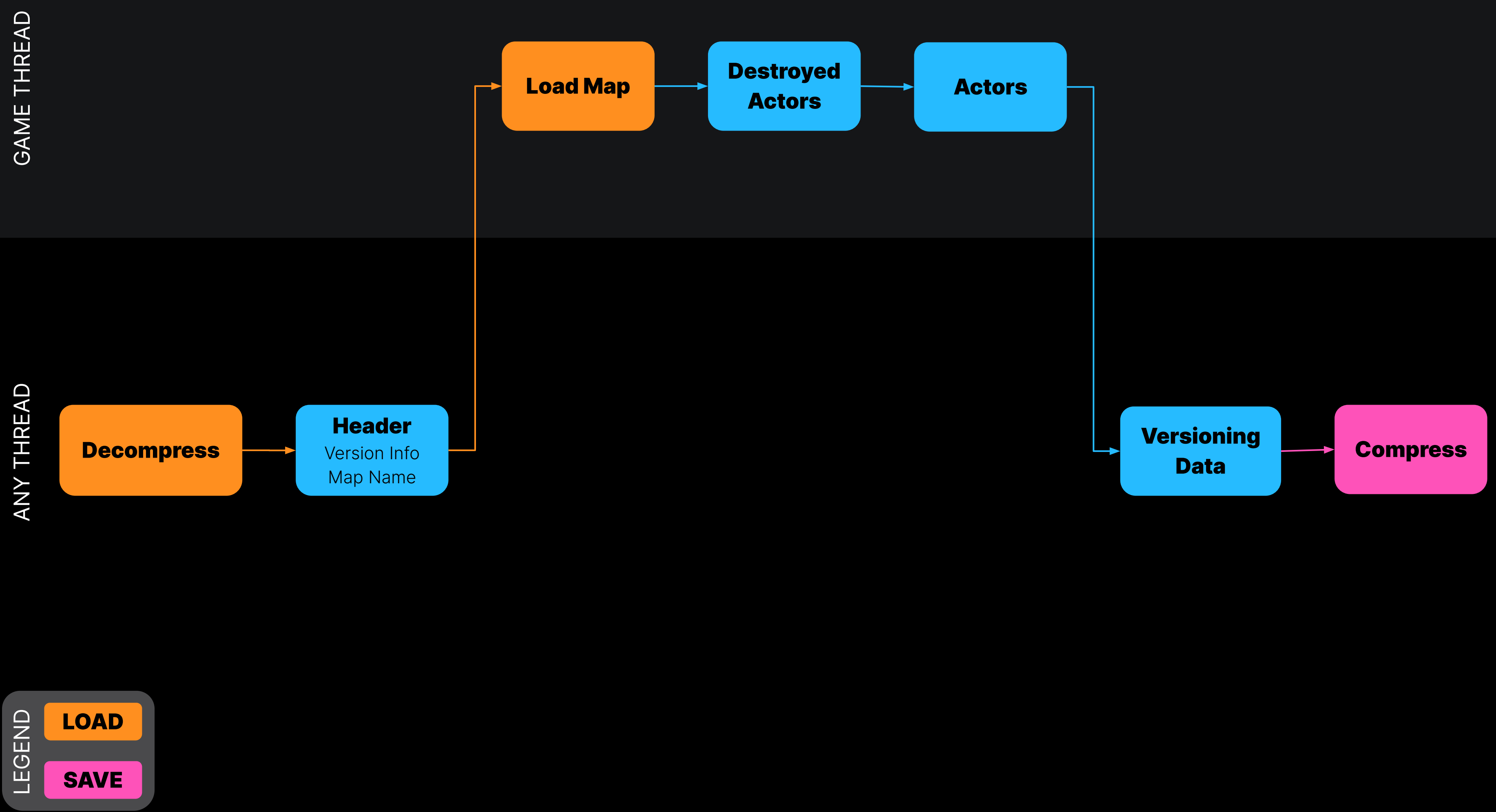
Compress

ANY THREAD

LEGEND

LOAD

SAVE



GAME THREAD

Load Map

Destroyed Actors

Actors

ANY THREAD

Decompress

Header
Version Info
Map Name

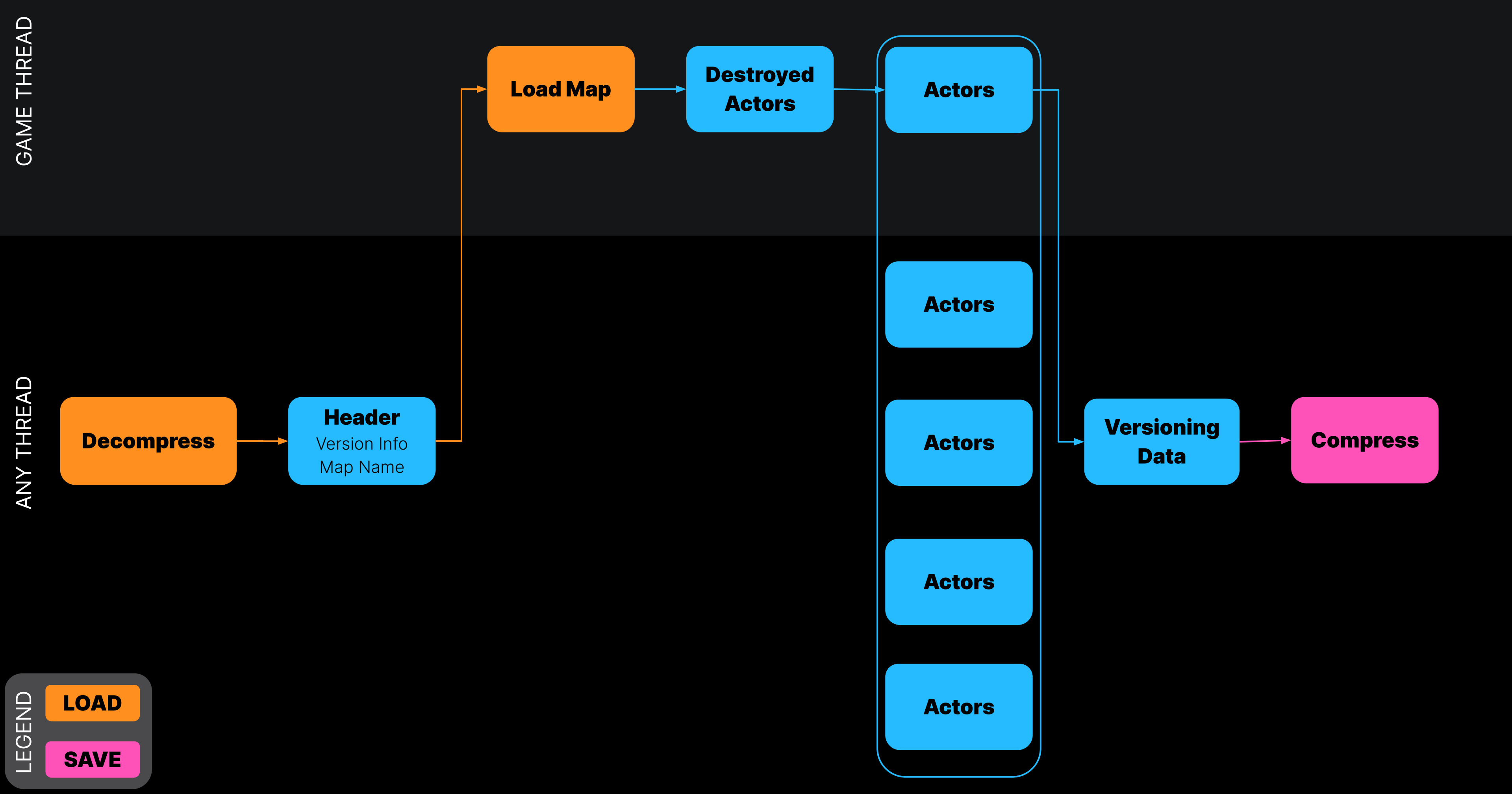
Versioning Data

Compress

LEGEND

LOAD

SAVE


GAME THREAD
ANY THREAD
Load Map
Destroyed Actors
Actors
Actors
Actors
Actors
Actors
Decompress
Header
Version Info
Map Name
Versioning Data
Compress
LEGEND
LOAD
SAVE

[Task System Demo]

Safety First

Blueprints are NOT Thread-Safe

Unless BP methods are explicitly marked as such, don’t execute them from a thread. Blueprint created methods can be marked thread-safe, but this is a contract by the programmer to say it “is”. There is no magic button to make something thread-safe.

Threading UObjects

Not thread-safe by default but can be made so. UObject creation is only ever able to be called from the Game Thread. If ParallelFor is run from the Game Thread, read-only and self-writes in multi-threaded tasks can be done safely on UObjects.

Thread Resources

It’s quite easy to run into memory leaks due to allocating thread resources. Instead, use threadsafe shared pointers passed between tasks to automatically clean up after a job has completed.

Summary

- Unreal Engine has many ways to do threading. Consider which one is best for your project
- Thread Safety for Blueprints and UObjects is possible, but caution must be taken
- The Task System is a highly flexible way to coordinate short-lived tasks. Whereas a task that lasts the lifetime of the engine should be executed in an FRunnableThread

Thank you!

Questions?

alex.stevens@gameloft.com

@MilkyEngineer

https://bit.ly/Threading_UFGC24