Tasks System

An Overview of the Tasks System.

Tasks System is a job manager that provides the capability to execute your gameplay code asynchronously. It supports building and running a directed acyclic graph of dependent tasks. It improves on the **TaskGraph**, the job manager used in **Unreal Engine (UE)**. Tasks System and TaskGraph both use the same backend (the scheduler and worker threads).

The main features are:

- Launching a task by providing a callable object that needs to be executed asynchronously.
- Waiting for task completion and/or retrieving the task execution result.
- Specifying task prerequisites — other tasks that must be completed before the task execution starts.
- Launching nested tasks from inside a task. A parent task is not completed until all its nested tasks are completed.
- Building task chains, also known as pipes.
- Using task events for synchronization and signaling between tasks.

All code samples assume using the namespace `UE::Tasks` for brevity.

Launching

To **Launch** a task, you will need to provide the task's debug name and a callable object "task body". For example:

```
Launch(
UE_SOURCE_LOCATION,
[]{ UE_LOG(LogTemp, Log, TEXT("Hello Tasks!")); }
);

```

Copy full snippet

The code above launches a task that will execute the given function asynchronously. The first parameter is the task's debug name (preferably unique). Its purpose is to help debug tasks and assist in finding the code that launched the task.

`UE_SOURCE_LOCATION` is a macro that generates a string in the format file name of the source file and the line where it is used. This example shows a "fire and forget" task, this means that you don't need to care what happens to the task after it has been launched because it is eventually executed.

Often you will need to wait for task completion or retrieve its execution result. This can be done by using the Task object returned by the Launch call:

```
FTask Task = Launch(UE_SOURCE_LOCATION, []{});

```

Copy full snippet

A task execution can return a result. `FTask` is an alias of `TTask<void>`, a specialization of generic `TTask<ResultType>`. `ResultType` should match the type of the result returned by the task body:

```
TTask<bool> Task = Launch(UE_SOURCE_LOCATION, []{ return true; });

```

Copy full snippet

Tasks are executed asynchronously and potentially concurrently with the launching thread, so their execution order is undefined. Though we can still affect the tasks execution order by specifying task priority. Task priorities are "high", "normal" (default), "background high", "background normal", and "background low". Tasks with higher priority are executed before tasks with lower priority.

```
Launch(UE_SOURCE_LOCATION, []{}, ETaskPriority::High);

```

Copy full snippet

A lambda function is typically used as a task body, though any callable object can be used too.

```
void Func() {}
Launch(UE_SOURCE_LOCATION, &Func);

struct FFunctor
{
void operator()() {}
};
Launch(UE_SOURCE_LOCATION, FFunctor{});

```

Copy full snippet

Technical Details

`FTask` is a handle of an actual task, which is similar to a smart pointer. It uses reference counting to manage its lifetime. Launching a task starts its lifetime and allocates the required resources. To release a held reference you can "reset" the task handle using the following:

```
FTask Task = Launch(UE_SOURCE_LOCATION, []{});
Task = {}; // release the reference

```

Copy full snippet

Releasing a task handle doesn't immediately lead to task destruction. The system holds its own reference used to execute the task. This reference is released after task completion.

Refer to Launch for additional information.

Waiting for Task Completion

You may often need to know if a task is completed, to wait for its completion, or to retrieve its execution result.

Task Command	Implementation Method
Check if a task is completed	Example: `bool bCompleted = Task.IsCompleted();` Copy full snippet
Wait for task completion	Example: `Task.Wait();` Copy full snippet
Wait for task completion with timeout	Example: `bool bTaskCompleted = Task.Wait(FTimespan::FromMillisecond(100));`

Task Command	Implementation Method
	  Copy full snippet

Wait until all tasks are completed	Example:
	```
	TArray<FTask> Tasks = …;
	Wait(Tasks);
	```
	Copy full snippet

Retrieve task execution result. The call is blocked until the task is completed and its result is ready.	Example:
	```
	TTask<int> Task = Launch
	(UE_SOURCE_LOCATION, []{ return 42; });
	int Result = Task.GetResult();
	```
	Copy full snippet

Waiting should be avoided if possible, as it limits scalability. Instead, we recommend you build task graphs by defining dependencies between tasks and designing task-based asynchronous APIs. See Wait, and `GetResult()` for more information.

Busy-waiting

A problem of waiting for task completion is that it blocks the current thread, therefore it isn't useful. An alternative approach is to use **Busy-waiting**. When using Busy-waiting, the thread will attempt to execute other tasks until the task that is being waited for is completed.

While Busy-waiting can be useful in a controlled environment, it has its own set of issues and should be used with caution. The main issue is that it isn't possible to control which tasks are picked by the scheduler to be executed during Busy-waiting.

This can lead to deadlocks(meaning a thread is busy waiting and has locked a non-reentrant mutex while a task picked up by the scheduler had tried to lock the same mutex), or result in worse performance where the scheduler picks up a long-running task while busy-waiting for a shorter one on the critical path.

Refer to `BusyWait()` for additional information.

Prerequisites

Tasks can have dependencies on other tasks. If task A can be executed only after task B is completed, then task B is called a **prerequisite** of task A and task A is called a **subsequent** of task B. This allows for building a directed acyclic graph of tasks.

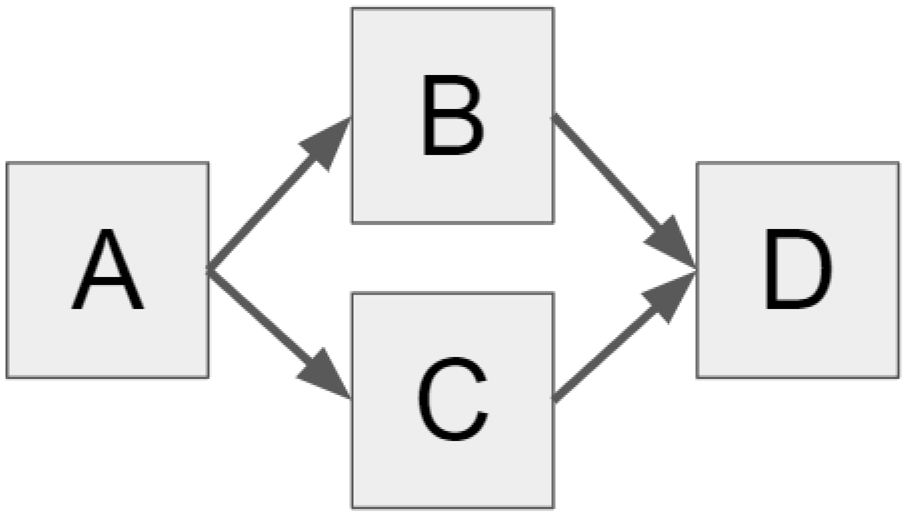
The main advantage of using task dependencies is that it doesn't block the worker threads. Additionally, dependencies allow you to force the task execution order, which is not normally guaranteed. The code below builds a simple Prerequisite to Subsequent dependency:

```
FTask Prerequisite = Launch(UE_SOURCE_LOCATION, []{});
FTask Subsequent = Launch(UE_SOURCE_LOCATION, []{}, Prerequisite);

```

Copy full snippet

In the code example below, the `Prerequisites()` is a helper function:



```
FTask A = Launch(UE_SOURCE_LOCATION, []{});
FTask B = Launch(UE_SOURCE_LOCATION, []{}, A);
FTask C = Launch(UE_SOURCE_LOCATION, []{}, A);
FTask D = Launch(UE_SOURCE_LOCATION, []{}, Prerequisites(B, C));

```

Copy full snippet

Refer to Launch for additional information.

Nested Tasks

Nested tasks are similar to prerequisites, but while prerequisites are execution dependencies, nested tasks are completion dependencies. Consider task A that launches task B during its execution, and task A can be completed only when its execution is finished and task B is completed. This is a common pattern when a system exposes a task-based async interface, but task B is part of the implementation therefore it's undesirable to leak this task.

The simplest implementation would be the following:

```
FTask TaskA = Launch(UE_SOURCE_LOCATION,
[]
{
```

```
FTask TaskB = Launch(UE_SOURCE_LOCATION, [] {});
TaskB.Wait();
}
);

```

Copy full snippet

This is a basic implementation to accomplish the task, but it is inefficient as the worker thread executing task A is blocked waiting for task B completion, therefore it is not used to execute other tasks.

The solution is to use nested tasks. In our example, task A is a parent task, and task B is a nested task because its execution should be nested inside task A's execution:

```
FTask TaskA = Launch(UE_SOURCE_LOCATION,
[]
{
FTask TaskB = Launch(UE_SOURCE_LOCATION, [] {});
AddNested(TaskB);
}
);
TaskA.Wait(); // returns only when both 'TaskA' and 'TaskB' are completed

```

Copy full snippet

AddNested adds the given task as nested to the task being executed by the current thread. It asserts if it isn't called from inside a task.

Refer to `AddNested()` for more information.

Pipes

A **Pipe** is a chain of tasks that are executed one after another (not concurrently). Consider a shared resource accessed from multiple threads. A classic approach to synchronize the access is to "lock" the resource by locking a mutex. This often brings a significant performance penalty as the thread gets blocked, especially if there is resource contention.

For complex resources, it is desirable to provide an async interface that allows initiating an async operation to work on the resource and an ability to check if the operation is completed(or is to subscribe to a completion notification).

Implementing an async interface often is not a trivial task. Pipes were designed to streamline this. The intention is to have a pipe per a shared resource. All accesses to the shared resource are performed inside tasks launched by the pipe. For example:

```
class FThreadSafeResource
{
public:
TTask<bool> Access()
{
return Pipe.Launch(TEXT("Access()"), [this] { return
ThreadUnsafeResource.Access(); });
}

FTask Mutate()
{
return Pipe.Launch(TEXT("Mutate()"), [this] { ThreadUnsafeResource.Mutate();
});
}
private:
FPipe Pipe{ TEXT("FThreadSafeResource pipe")};
FThreadUnsafeResource ThreadUnsafeResource;
};

FThreadSafeResource ThreadSafeResource;
//access the same instance concurrently from multiple threads
bool bRes = ThreadSafeResource.Access().GetResult();
FTask Task = ThreadSafeResource.Mutate();

```

Copy full snippet

`FThreadSafeResource` provides a public thread-safe async interface based on tasks. It encapsulates a thread-unsafe resource. The implementation is straightforward and consists of boilerplate code. Any access to the thread-unsafe resource happens inside piped tasks.

As those piped tasks are executed sequentially, there's no need for any additional synchronization. Pipes are lightweight objects, therefore they don't store a collection of their tasks. It's possible to have thousands of pipes without a significant drop in performance.

To make a task piped, it needs to be launched by a pipe:

```
FPipe Pipe{ UE_SOURCE_LOCATION };
FTask TaskA = Pipe.Launch(UE_SOURCE_LOCATION, []{});
FTask TaskB = Pipe.Launch(UE_SOURCE_LOCATION, []{});

```

Copy full snippet

TaskA and TaskB will not be executed concurrently, so they don't need to synchronize with each other to access a shared resource. While most of the time the order of execution is predictable, the order in which the tasks are launched is not guaranteed.

Piped tasks support the same features that other tasks do, for example, they can have dependencies and follow the order of behavior. First, dependencies are resolved then a task is piped. This means that a task with pending dependencies doesn't block a pipe execution and that dependencies can alter the piped tasks' execution order.

You can consider pipes to be **green threads**. Those green threads are executed by the worker threads, and can "jump threads". For instance, In the previous example, TaskA and TaskB can be executed by different threads.

- Pipes API is thread-safe.
- Pipe objects are non-copyable and non-movable.
- It is not possible to launch a task in multiple pipes.

Refer to FPipe for additional information.

Task Events

Task events are a special task type that doesn't have a task body and can't do any execution. A significant difference is that task events initially are not launched (signaled), and need to be explicitly triggered. Task events are useful as a synchronization and signaling primitive. They are similar to the one-time FEvent. They can be used as a prerequisite or subsequent of other tasks.

Below is a table that provides some examples of what can be done using Task Events.

Task Event Example	Implementation Method

Launch a task but hold its execution until explicitly released.	Example:
	`FTaskEvent Event{ UE_SOURCE_LOCATION };`
	`FTask Task = Launch(UE_SOURCE_LOCATION, []{}, Event);`
	`Event.Trigger();`
	Copy full snippet
	The event is used as a prerequisite for the task. Initially, the event is in a non-signaled state, therefore it hasn't been completed, which means that the task has a pending dependency and won't be scheduled and executed until it is resolved. Task events are switched to the signaled state by triggering them.

Use a task event as a joiner task.	Example:
	`FTask TaskA = Launch(UE_SOURCE_LOCATION, []{});`
	`FTask TaskB = Launch(UE_SOURCE_LOCATION, []{});`
	`FTaskEvent Joiner{ UE_SOURCE_LOCATION };`
	`Joiner.AddPrerequisites(Prer`

Task Event Example	Implementation Method
	`equisites(TaskA, TaskB));` `Joiner.Trigger();` `...` `Joiner.Wait();` Copy full snippet The joiner depends on TaskA and TaskB. Waiting for it means waiting for all its dependencies instead of waiting for them individually. `Prerequisites()` is a helper function.
Stop task execution in the middle and wait for an event to happen.	Example: `FTaskEvent Event{ UE_SOURCE_LOCATION };` `FTask Task = Launch(UE_SOURCE_LOCATION,` `[&Event]` `{` `...` `Event.Wait();` `...`

Task Event Example	Implementation Method

	`});`
	`...`
	`Event.Trigger();`
	
	Copy full snippet
	
	In general, waiting in the middle of a task is not the best idea for performance and scalability reasons. If you find yourself in such a situation, consider redesigning with prerequisites if possible.

Execute a task but don't flag it as completed automatically. Instead, "complete" it explicitly when it's convenient	Example:
	
	`FTaskEvent Event{ UE_SOURCE_LOCATION };`
	`FTask Task = Launch(UE_SOURCE_LOCATION,`
	`[&Event]`
	`{`
	`AddNested(Event);`

Task Event Example	Implementation Method
	`    });` `...` `Event.Trigger();` Copy full snippet

```
        });
    ...
    Event.Trigger();

```

Copy full snippet

See also: FTaskEvent

Debugging and Profiling

Every task, task event, or pipe has a user-provided debug name. This allows the capability to identify them during runtime in the debugger. Visual Studio native visualizers are provided for examining their internal state.

Unreal Insights adds the task trace channel that enables visualization of tasks and their lifetime events. Such as when a task was launched, scheduled, executed, and completed.

Refer to the Unreal Insights documentation for details.

Debugging and profiling are active areas of development and will be further improved in the future.