

Creating Multithreaded Skyrim Mods

This tutorial covers how modders can use Papyrus more effectively by leveraging its inherent multithreading capability. This guide includes plenty of examples and explanations to help you understand the design pattern. Using multithreading can greatly increase the performance of mods that have many external function calls back-to-back (*function A calls external function B, then calls external function C, then calls...*) or deeply nested (*function A calls external function B calls external function C calls...*) and help execute time-critical tasks, at the cost of a short burst of resource utilization.

[Papyrus is a threaded scripting language](#). However, it can be a challenge to harness this attribute of the language.



The intended audience for this guide is intermediate to expert Papyrus developers. This design pattern **requires SKSE** for its use of [Mod Events](#).

The examples provided are intended to be used as a reference to adapt to your own needs; as each mod's needs are different, and because of the way Papyrus (and Skyrim) is designed, writing a generic framework that provides a solution for everyone is not possible; change it to fit your unique requirements.

Please take your time going through this guide; there is a lot of information, but once you've grasped the idea, you'll be up and running in no time. There are a lot of code dependencies, so some of what you'll be doing may not make sense until the end.

Contents

Should I Multithread?

The first question to answer is whether or not a multithreading solution is a good fit for your mod. There's no sense in refactoring hundreds of lines of code if you're not going to stand to benefit from it.

Does your mod / script:

- Have many external function calls to complete a single task?
- Have many objects that must be placed quickly using things like `MoveTo()`?
- Extensively or repeatedly use [latent functions](#)?
- Have time-critical tasks that rely on the results of other (potentially slow) functions?
- Have need of doing the same thing to a large group of objects?


If you answered "yes" to any of these bullets, a multithreaded design pattern may increase the performance of your mod. This pattern provides two distinct advantages:


1. Multithreading takes tasks that would otherwise be run in sequence and allows them to run simultaneously, which can reduce the time it takes to complete all tasks.

2. Due to the way the Papyrus scheduler must sync external calls to frames, many external calls can add a great deal of overhead (see [this page](#) on notes regarding how external calls suspend and resume threads); this pattern can greatly reduce the number of external function calls your script must use at any one time.

Only profiling your scripts by using one of the various profiling functions can tell you whether or not these patterns will improve your mod's behavior. I have personally seen performance over 10 times faster (an action that once took ~8.5 seconds to now takes ~0.5 seconds) in my own mods using this method.

Note that by spinning up many threads simultaneously, you are invariably placing increased load on the Papyrus VM for as long as it takes your threads to complete. Ideally, this should be a much shorter frame of time than if the task were done in a single thread. You must decide whether or not the narrow "spike" of resource consumption using threads is better than the more spread-out "swell" of a single thread calling many functions back-to-back. Again, **profile** before and after!

 This design pattern is **not** intended to replace the way you do things in every script. It is meant to tackle increasing the performance of *specific, slow, and usually repetitive* tasks.

 Keep in mind that asynchronous operations means that you don't know how fast, or in what order, your threads will run or finish.

If the tasks you need to perform are order-dependent, the methods detailed below are probably not a good fit for your needs.

A Private Army

In this example, we are developing a Conjuraction mod. We need to spawn 20 guards very quickly when the player casts a spell; ideally, they should all appear at close to the same time. We also need to keep track of the guards we create, so we can destroy them after the spell ends. This guide will not cover creating a spell, instead we will skip to a point after we've created our Spell and our MagicEffect that we want to add a script to.

We come up with the following script to drop onto our MagicEffect in the Creation Kit when our spell is cast:

```
scriptname SummonArmy extends ActiveMagicEffect
```

```
ActorBase property Guard auto
ObjectReference property GuardMarker auto
Actor Guard1
Actor Guard2
...
Actor Guard20
```

```
Event OnEffectStart(Actor akTarget, Actor akCaster)
    if akCaster == Game.GetPlayer()
        ;Place actors according to the player's position, taking into account walls,
        obstacles, etc
```

```

    MoveGuardMarkerNearPlayer(1) ;Moves the GuardMarker where the guard is supposed
to go; maybe some GetPositions, etc
    Guard1 = GuardMarker.PlaceAtMe(Guard)
    MoveGuardMarkerNearPlayer(2)
    Guard2 = GuardMarker.PlaceAtMe(Guard)
    ;...and so on
    MoveGuardMarkerNearPlayer(20)
    Guard20 = GuardMarker.PlaceAtMe(Guard)
endif
endEvent

```

```

Event OnEffectFinish(Actor akTarget, Actor akCaster)
if akCaster == Game.GetPlayer()
    Guard1.Disable()
    Guard1.Delete()
    ;...and so on
    Guard20.Disable()
    Guard20.Delete()
endif
endEvent

```

We test this in-game, and we see each guard appear one-by-one. Your users complain that the spell is "slow" and "clunky". You'd like things to appear much faster so that the spell feels responsive.

We have (for illustration purposes) some preprocessing that needs to happen (`MoveGuardMarkerNearPlayer(int Index)`) before we know where to put the guard, which turns out to be slow (several `MoveTo()`, etc). We have decided that multithreading this task would be much faster than placing each Actor one-by-one.

Two Approaches: Futures and Callbacks

There are two basic threaded patterns that you can decide to implement. They each have pros and cons. You will need to decide which approach is best for your application.

Futures

As Papyrus developers, we are accustomed to calling functions, having those functions return values, and storing those returned values. You're probably used to seeing code like this:

```

ObjectReference my_sword = PlayerRef.PlaceAtMe(Sword)

```

In the Futures pattern, we avoid calling functions like this directly. Instead, we call a function on a special script we will write, the **Thread Manager**, that will delegate our work to a **thread**.

Instead of receiving a return value, we will receive something called a **Future**. A `Future` is not the return value; instead, it *represents* the return value *at some point in the future*. It can be thought of as a placeholder for the "real" value.



- **Thread** - An individual script instance that does work. Returns results to a Future or raises a callback event.
 - **Thread Manager** - A script that controls which thread handles a task. Returns a Future to the user of the script, if using that pattern.
 - **Future** - An object that will contain the thread's return value at some point in the future.
-

So our code using a Future pattern might look something like this:

```
ObjectReference my_sword_future = ThreadManager.PlaceAtMeAsync(Sword)
ThreadManager.wait_all()
```

`PlaceAtMeAsync` is a function we've written that gets assigned to a thread. A thread that has been given data to work on is referred to as being **queued**. `wait_all()` tells the Thread Manager that it should start running any queued threads and that we will wait until they're finished.

Later, when we decide we want the result of our thread, we just ask for it:

```
ObjectReference my_sword = (my_sword_future as FutureScript).get_result()
```

Why would we want to do this? In the above example, it might not make much sense. But what if our code looked more like,

```
ObjectReference my_sword = PlayerRef.PlaceAtMe(Sword)
my_sword.MoveTo(SwordPositionMarker)
SwordPositionMarker.MoveTo(OriginLocation)
my_sword.SetAngle(my_sword.GetAngleX(), my_sword.GetAngleY(),
my_sword.GetAngleZ() + 120.0)
;...and so on
```

If we had to do this collection of operations on not just one sword, but say, two dozen, things start to take a while to process; it might be a few seconds before anything ever even happens to sword #12, 18, or 22. And everything happens one by one. With the Future pattern, calling `PlaceAtMeAsync()` would return almost immediately, leaving your script free to do other things while all of your swords are placed and moved. And with threads, all of these swords would be placed and moved nearly simultaneously. When you're ready to get each sword's `ObjectReference`, you call `get_result()` on your Future.

Futures Pros

- **Pull-based:** Using Futures is a *pull* pattern, where you must explicitly ask (pull) for the results of a thread you have started by calling `get_result()`.
- **Control who can access results:** The result can only be retrieved by someone who knows the Future of your thread. You have control over who can retrieve your results.
- **Control when results are retrieved:** The result is only retrieved when you ask for it. This is important when you need to retrieve results in a particular order.
- **Easier to trace execution order:** A thread can be started and results retrieved all within the same function; your code does not have to "jump around" as much as it does in the Callback pattern.
- **Abstracts away "locks":** With Futures, we don't have to worry about two threads accidentally manipulating the same variable in the wrong order because one finished faster or slower than the other. We just request our results from our futures in the order that we want them.
- **Requires less state management:** Managing the state of your script is almost as simple as when you wrote scripts in a single thread calling functions directly.

Futures Cons

- **Harder to understand:** Implementing a Future-based approach requires learning several new concepts.
- **Harder to implement:** There are more pieces involved in setting up a Future-based approach.
- **More overhead (slower):** A Future-based approach can be slower than using a Callback approach, due to the fact that Futures are ObjectReferences and must be created and destroyed when a thread runs and data is read.
- **Results availability and delays:** If you call `get_result()` on a Future, and the result is not yet ready, the script will wait until it is, and then return the result. You are at the mercy of the thread to return a value to the future until you can continue. For some applications, this may be considered a pro.
- **Harder to make results public:** If you have many intended consumers of your thread's results (many scripts, or even scripts on other people's mods), using Futures may be burdensome for getting your results to everyone who needs them.
- **May require polling:** If you can't afford to block execution on `get_result()`, you may have to poll the Future's `done()` function to check whether the thread has finished.

Callbacks

Callbacks are similar to Futures in that we start **threads** using a **Thread Manager** to do work for us, instead of calling functions directly and sequentially. The difference is that when we start the thread, there is no return value; instead, the thread will *call back* to tell us when it is finished, and what the result was. It does this by raising a Mod Event.



- **Callback** - A Mod Event that is raised when a thread completes. Passes the thread results in the event parameters.

Using the above example from Futures, our code might look like:

```
Event OnInit()  
    RegisterForModEvent("MyMod_PlaceSwordAsyncCallback")  
endEvent  
  
function SomeFunction()  
    ThreadManager.PlaceSwordAsync(Sword)  
    ThreadManager.wait_all()  
endFunction  
  
;...then somewhere else, in your script  
  
Event PlaceSwordAsyncCallback(ObjectReference akPlacedObject)  
    ;Anyone that registers for the mod event can get this, too!  
    my_sword = akPlacedObject  
endEvent
```

Callback Pros

- **Push-based:** Using callbacks is a *push* pattern, where results are returned to you as soon as they're available instead of having to request them.
- **Anyone can access results:** The results of a thread are available to anyone who registered for the event that returns them.
- **Results received without delays:** Unlike Futures, you do not have to block your script pending results being available. Just register for the appropriate event and react to it.
- **No polling:** You no longer have to potentially poll for whether or not your results are ready.
- **Easier to understand:** The concepts in a Callback pattern are nothing new to anyone who knows how to use Mod Events.
- **Easier to implement:** There are comparatively fewer things to deal with when using a Callback pattern.
- **Less overhead (faster):** Using a callback pattern can be a bit faster than a Future-based approach.

Callback Cons

- **...Anyone can access results:** You have no control over who is able to consume your results.
- **No control when results are retrieved:** You have no control over when a result will be retrieved, or in what order. You must be able to react to the result events that are raised, and you must assume that threads can finish in any order.
- **More difficult to trace execution order:** A callback pattern can make the script flow more difficult to follow and debug, since the function where a thread is started and the event that it returns results to will be in two (or more) different places.
- **Locks required:** Locks are required if you have two threads that may write to the same variable.

- **Requires more state management:** You can receive result callbacks at any time, which may make it necessary for you to re-evaluate the script's current state each time you receive one, depending on your application.

More details about each approach are available in the next tutorials, with example code and definitions. Press on!