Creating Multithreaded Skyrim Mods Part 3 - Callbacks

creationkit.com/index.php

We will be implementing a multithreaded solution to our example problem (a Conjuration mod that spawns many actors) using the **Callback pattern**.

Download Tutorial Example Plugin - A fully functional, installable mod. Includes all tutorial files and source code.

Contents

Pattern Overview

To recap the Pros and Cons of this approach:

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:** Their 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.

Here is a diagram of how the Callback pattern works.

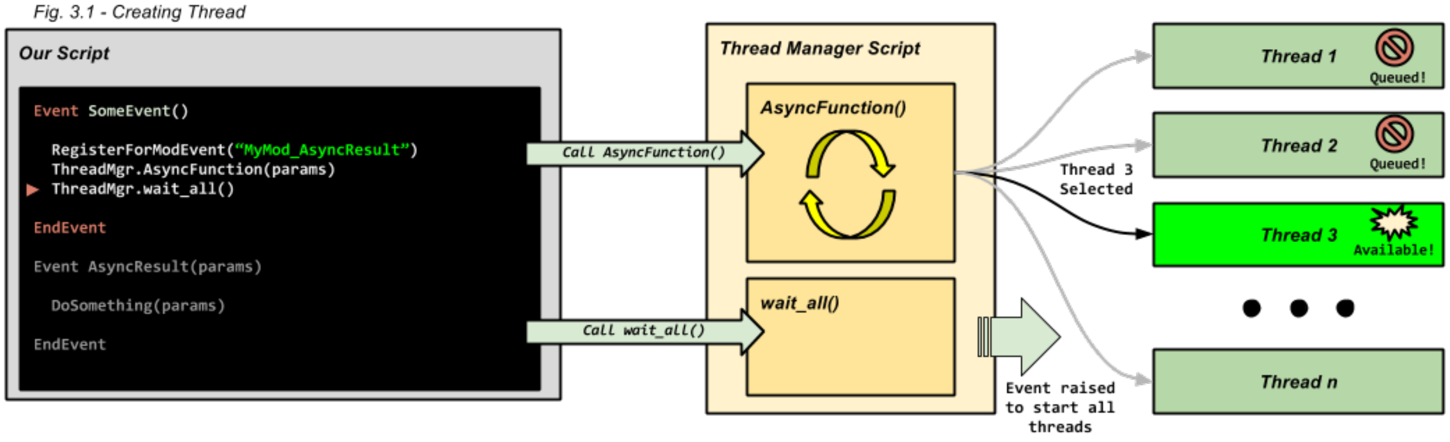


Fig. 3.1 - Creating Thread

```
Event SomeEvent()

  RegisterForModEvent("MyMod_AsyncResult")
  ThreadMgr.AsyncFunction(params)
  ThreadMgr.wait_all()

EndEvent

Event AsyncResult(params)

  DoSomething(params)

EndEvent
```

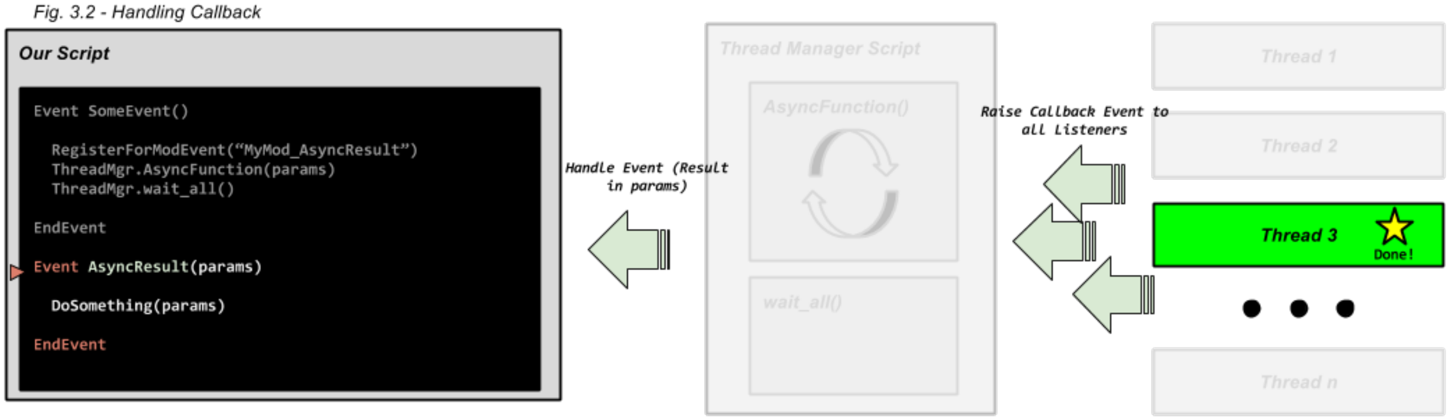


Fig. 3.2 - Handling Callback

```
Event SomeEvent()

  RegisterForModEvent("MyMod_AsyncResult")
  ThreadMgr.AsyncFunction(params)
  ThreadMgr.wait_all()

EndEvent

Event AsyncResult(params)

  DoSomething(params)

EndEvent
```

Above, you can see that the sequence is:

1. Register for our Callback Event and call a function on our **Thread Manager**.
2. The Thread Manager delegates the work to an available **thread**.
3. After the thread finishes, we handle the callback in an event.

That is the Callback pattern. Just like the Futures pattern, we will now piece it all together.

Creation Kit

Create Quest: Begin by opening the Creation Kit and creating a new Quest. We'll call our quest **GuardPlacementQuest**. Click OK to save and close the quest, then open it again (to prevent the CK from crashing). Make sure that "Start Game Enabled", "Run Once", "Warn on Alias Failure" and "Allow repeated stages" are unchecked. Click OK to close it again.

- **Fig. 3.3**:
  Create Quest

Threads

The thread is what will perform the work we want to perform in parallel. Just like the `PlaceAtMe()` needed to spawn our guards, we expect the result of our Thread to be an ObjectReference.

First, let's define a base Thread "class", called **GuardPlacementThread**.

```
scriptname GuardPlacementThread extends Quest hidden

;Thread variables
bool thread_queued = false

;Variables you need to get things done go here
ActorBase theGuard
Static theMarker

;Thread queuing and set-up
ObjectReference function get_async(ActorBase akGuard, Static akXMarker)

    ;Let the Thread Manager know that this thread is busy
    thread_queued = true

    ;Store our passed-in parameters to member variables
 theGuard = akGuard
 theMarker = akXMarker
endFunction

;Allows the Thread Manager to determine if this thread is available
bool function queued()
 return thread_queued
endFunction

;For Thread Manager troubleshooting.
bool function force_unlock()
    clear_thread_vars()
    thread_queued = false
    return true
endFunction

;The actual set of code we want to multithread.
Event OnGuardPlacement()
 if thread_queued
  ;OK, let's get some work done!
  ObjectReference tempMarker = Game.GetPlayer().PlaceAtMe(theMarker)
  MoveGuardMarkerNearPlayer(tempMarker)
  ObjectReference result = tempMarker.PlaceAtMe(theGuard)
  tempMarker.Disable()
  tempMarker.Delete()

        ;OK, we're done - raise event to return results
  RaiseEvent_GuardPlacementCallback(result)

        ;Set all variables back to default
  clear_thread_vars()

        ;Make the thread available to the Thread Manager again
  thread_queued = false
 endif
endEvent

;Called from Event OnGuardPlacement
function MoveGuardMarkerNearPlayer(ObjectReference akMarker)
 ;Some difficult calculations, etc
EndFunction

function clear_thread_vars()
 ;Reset all thread variables to default state
 theGuard = None
 theMarker = None
endFunction

;Create the callback
function RaiseEvent_GuardPlacementCallback(ObjectReference akGuard)
    int handle = ModEvent.Create("MyMod_GuardPlacementCallback")
    if handle
     ModEvent.PushForm(handle, akGuard as Form)
        ModEvent.Send(handle)
    else
        ;pass
    endif
endFunction
```

As you can see, our thread does a few important things:

- It has a `get_async()` function, which takes in all of the parameters necessary to do the work we need to perform.
- `Event OnGuardPlacement()` will fire if the Thread Manager raises the event.
- The thread "calls back" to any scripts that have registered for our callback event.
- It clears all of the member variables using `clear_thread_vars()`.

- We set `thread_queued` back to `False`, which tells the Thread Manager that this thread is available to be used again.

Reacting to an Event allows the `Event OnGuardPlacement()` to begin working in parallel to other threads. If we called the functions that do work directly from `get_async()`, the calling script would block until the work was complete, which would defeat the purpose.

Be diligent about error handling and what could go wrong while your thread is running. If your thread aborts before it can set `thread_queued` back to `False`, your thread will become locked and unusable until it times out on the next `get_result()`. If the thread is hung waiting for an external function call that will never return (such as a `PlaceAtMe()` on an ObjectReference that cannot complete its `OnInit()` block), it may become permanently locked.

Now that we've set up our base Thread script, we will create 10 child scripts that will extend this one. They will each contain only one line, the scriptname definition.

```
;GuardPlacementThread01.psc
scriptname GuardPlacementThread01 extends GuardPlacementThread

;GuardPlacementThread02.psc
scriptname GuardPlacementThread02 extends GuardPlacementThread

...

;GuardPlacementThread09.psc
scriptname GuardPlacementThread09 extends GuardPlacementThread

;GuardPlacementThread10.psc
scriptname GuardPlacementThread10 extends GuardPlacementThread
```

Once all of your Thread child scripts are saved and compiled, attach the 10 child scripts to your Quest.

"***But wait,***" you ask. "***We need to place 20 guards, but we only have 10 threads. Won't something break?*** " The Thread Manager, which we'll talk about next, can handle having more work than there are threads!

Thread Manager

We will next define the Thread Manager script. This script handles delegating our work to an available thread. If a thread is not available, it waits until one is.

Declare any properties that your threads will need in this script; the threads themselves will not have properties defined (since this would be tedious to hook up in the Creation Kit for each thread).

```
scriptname GuardPlacementThreadManager extends Quest

Quest property GuardPlacementQuest auto
{The name of the thread management quest.}

Static property XMarker auto
{Tedious to define properties in the threads and hook up in CK over and over, so
define things we need here. MoveGuardMarkerNearPlayer() needs XMarkers.}

GuardPlacementThread01 thread01
GuardPlacementThread02 thread02
;...and so on
GuardPlacementThread09 thread09
GuardPlacementThread10 thread10

Event OnInit()
    ;Register for the event that will start all threads
    ;NOTE - This needs to be re-registered once per load! Use an alias and
OnPlayerLoadGame() in a real implementation.
    RegisterForModEvent("MyMod_OnGuardPlacement", "OnGuardPlacement")

    ;Let's cast our threads to local variables so things are less cluttered in our
code
    thread01 = GuardPlacementQuest as GuardPlacementThread01
    thread02 = GuardPlacementQuest as GuardPlacementThread02
    ;...and so on
    thread09 = GuardPlacementQuest as GuardPlacementThread09
    thread10 = GuardPlacementQuest as GuardPlacementThread10
EndEvent

;The 'public-facing' function that our MagicEffect script will interact with.
function PlaceConjuredGuardAsync(ActorBase akGuard)
    if !thread01.queued()
        thread01.get_async(akGuard, XMarker)
    elseif !thread02.queued()
 thread02.get_async(akGuard, XMarker)
    ;...and so on
    elseif !thread09.queued()
        thread09.get_async(akGuard, XMarker)
    elseif !thread10.queued()
        thread10.get_async(akGuard, XMarker)
    else
 ;All threads are queued; start all threads, wait, and try again.
        wait_all()
        PlaceConjuredGuardAsync(akGuard)
 endif
endFunction
```

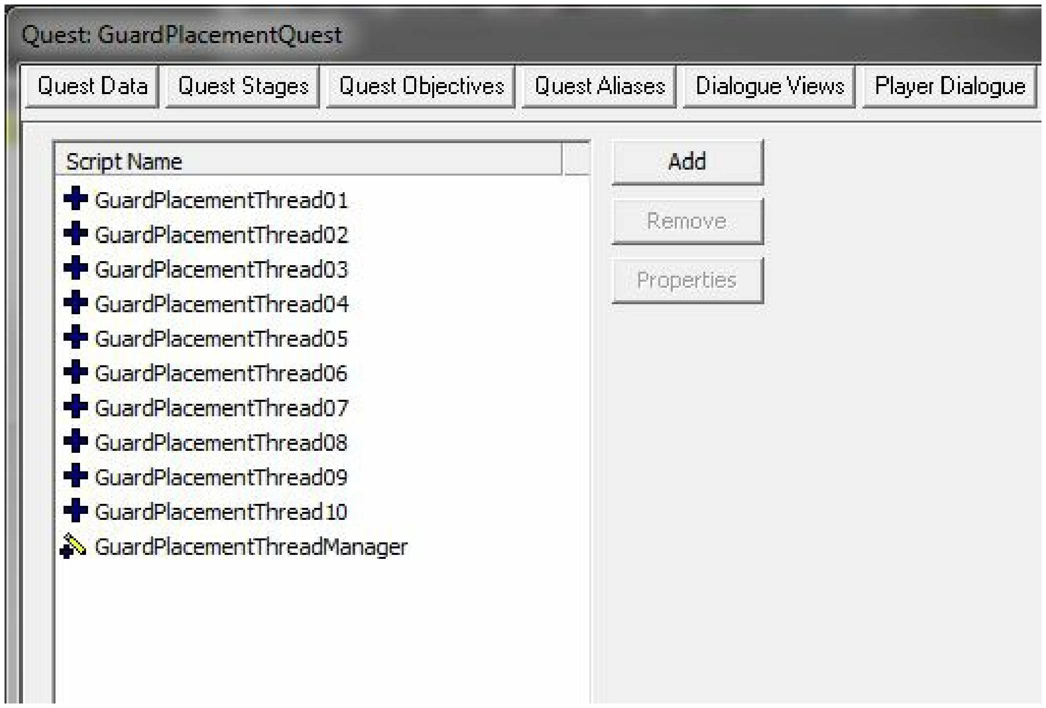
```
function wait_all()
    RaiseEvent_OnGuardPlacement()
    begin_waiting()
endFunction

function begin_waiting()
    bool waiting = true
    int i = 0
    while waiting
        if thread01.queued() || thread02.queued() || thread03.queued() ||
thread04.queued() || thread05.queued() || \
           thread06.queued() || thread07.queued() || thread08.queued() ||
thread09.queued() || thread10.queued()
            i += 1
            Utility.wait(0.1)
            if i >= 100
                debug.trace("Error: A catastrophic error has occurred. All threads
have become unresponsive. Please debug this issue or notify the author.")
                i = 0
                ;Fail by returning None. The mod needs to be fixed.
                return
            endif
        else
            waiting = false
        endif
    endWhile
endFunction

;Create the ModEvent that will start this thread
function RaiseEvent_OnGuardPlacement()
    int handle = ModEvent.Create("MyMod_OnGuardPlacement")
    if handle
        ModEvent.Send(handle)
    else
        ;pass
    endif
endFunction
```

The PlaceConjuredGuardAsync() function handles making sure that our work gets delegated to an available thread.

Threads begin working when either:

- All threads are queued.
- When wait_all() is called from the calling script.

Compile and attach this script to your GuardPlacementQuest. then, double-click the Thread Manager script and **fill the properties**. Once you've done that, your quest's script section should look something like this:



GuardPlacementThread01
GuardPlacementThread02
GuardPlacementThread03
GuardPlacementThread04
GuardPlacementThread05
GuardPlacementThread06
GuardPlacementThread07
GuardPlacementThread08
GuardPlacementThread09
GuardPlacementThread10
GuardPlacementThreadManager

Tying it All Together

Now that we've created our Threads and our Thread Manager, we can start to put them to work. Since we aren't calling the functions we want to execute directly, we need to change how we do things slightly.

The previous execution flow was:

1. Call each function, one by one, and store the results. (`PlaceAtMe()`, etc)

The flow using threads now is:

1. Call an Async function on our Thread Manager.
2. Handle return events as they are raised and store our results.

In our original ActiveMagicEffect script, we did all of our MoveGuardMarkerNearPlayer() and PlaceAtMe() calls in a row, getting a series of Actor references for our guards in return. We're going to modify that slightly to use our shiny new threaded placement system.

```
scriptname SummonArmy extends ActiveMagicEffect

Quest property GuardPlacementQuest auto
{We need a reference to our quest with the threads and Thread Manager defined.}
ActorBase property Guard auto

ObjectReference Guard1
ObjectReference Guard2
```

```
;...and so on
ObjectReference Guard9
ObjectReference Guard10

Event OnEffectStart(Actor akTarget, Actor akCaster)
 if akCaster == Game.GetPlayer()
  ;Cast the Quest as our Thread Manager and store it
  GuardPlacementThreadManager threadmgr = GuardPlacementQuest as
GuardPlacementThreadManager

  ;Register for the callback event
  RegisterForModEvent("MyMod_GuardPlacementCallback", "GuardPlacementCallback")

  ;Call PlaceConjuredGuardAsync for each Guard and store the returned Future
  threadmgr.PlaceConjuredGuardAsync(Guard)
  threadmgr.PlaceConjuredGuardAsync(Guard)
  ;...and so on
  threadmgr.PlaceConjuredGuardAsync(Guard)
  threadmgr.PlaceConjuredGuardAsync(Guard)
  threadmgr.wait_all()
 endif
endEvent

Event OnEffectFinish(Actor akTarget, Actor akCaster)
 if akCaster == Game.GetPlayer()
  DisableAndDelete(Guard1)
  DisableAndDelete(Guard2)
                ;...and so on
  DisableAndDelete(Guard9)
  DisableAndDelete(Guard10)
 endif
endEvent

bool locked = false
Event GuardPlacementCallback(Form akGuard)
 ;A spin lock is required here to prevent us from writing two guards to the same
variable
 while locked
  Utility.wait(0.1)
 endWhile
 locked = true

 ObjectReference myGuard = akGuard as ObjectReference

 if !Guard1
  Guard1 = myGuard
 elseif !Guard2
  Guard2 = myGuard
 ;...and so on
 elseif !Guard9
```

```
  Guard9 = myGuard
 elseif !Guard10
  Guard10 = myGuard
 endif

 locked = false
endEvent

function DisableAndDelete(ObjectReference akReference)
 akReference.Disable()
 akReference.Delete()
endFunction
```

Here, instead of doing the work in our script, registered for a callback Mod Event and delegated the work to the Thread Manager. We then called the Thread Manager's `wait_all()` function to make sure every thread has completed before continuing. Our return values are handed to us when the `GuardPlacementCallback()` event is raised.

You'll notice that our callback event employs a spin lock. This is very important, since it is possible for two callback events to accidentally write to the same variable using this pattern.

Notes on Callbacks

- You can create as many threads as you want, but I wouldn't recommend more than 10 or so. It depends on your needs, the strain each thread places on the Papyrus VM, and how quickly you need your results.
- If you need to perform a set of actions that are not all the same, the Thread Manager might not be best for you. You may want to create different thread base scripts purpose-built for your various tasks and then call their get_async() functions directly, blocking on `queued()` until they're available. You can still run many different tasks concurrently this way, even if they're not the same.

Playing the Example Plugin

Download Tutorial Example Plugin - A fully functional, installable mod. Includes all tutorial files and source code.

The example plugin can be installed using a mod manager, or by dragging all of the zipped files into the Skyrim\Data directory of your installation.

There are some differences between the examples provided here and the example plugin's code. Particularly, the example plugin is implemented with 20 threads instead of 10, and shows how you might scale the number of threads you use up or down.

When you start the game, you will be given 3 spells: *Summon Army (Single Threaded)* , *Summon Army (10 Threads)* , and *Summon Army (20 Threads)* . These will let you play around with the example scenario provided and see the time difference in completing the script between 1, 10, and 20 threads. Casting the spell will summon 20 Stormcloak

Soldiers, which will disappear after 15 seconds. Wait until all guards have disappeared before casting the spell (or another spell) again.

In my personal experience, I saw greatly diminishing returns after 10 threads in this example.

- **1 Thread:** Avg. 3.4 seconds to complete
- **10 Threads:** Avg. 0.8 seconds to complete
- **20 Threads:** Avg. 0.5 seconds to complete

Profiling your script is critical to determine if your unique application would benefit the most from more or less threads (or threading at all).

Your experience and times may differ based on your current load order and system performance. Give it a try and see what results you obtain.

This example plugin is provided to help understand the principles outlined in this tutorial, not for real gameplay. The mod's spells will cease to function after saving and loading the game.