Creating Multithreaded Skyrim Mods Part 2 - Futures

creationkit.com/index.php

This tutorial picks up where our introduction left off. We will be implementing a multithreaded version of our example mod (a Conjuration mod that spawns many actors) using the **Futures 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:

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 than in a Callback 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.

Here is a diagram of how the Futures pattern works.

Fig. 2.1 - Selecting Thread, storing Future

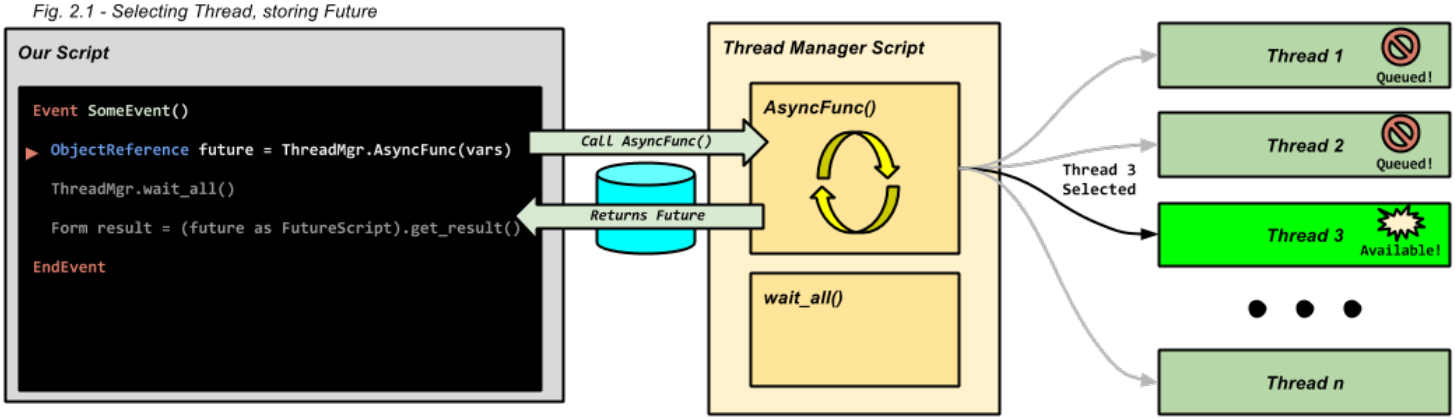


Fig. 2.2 - Starting Threads

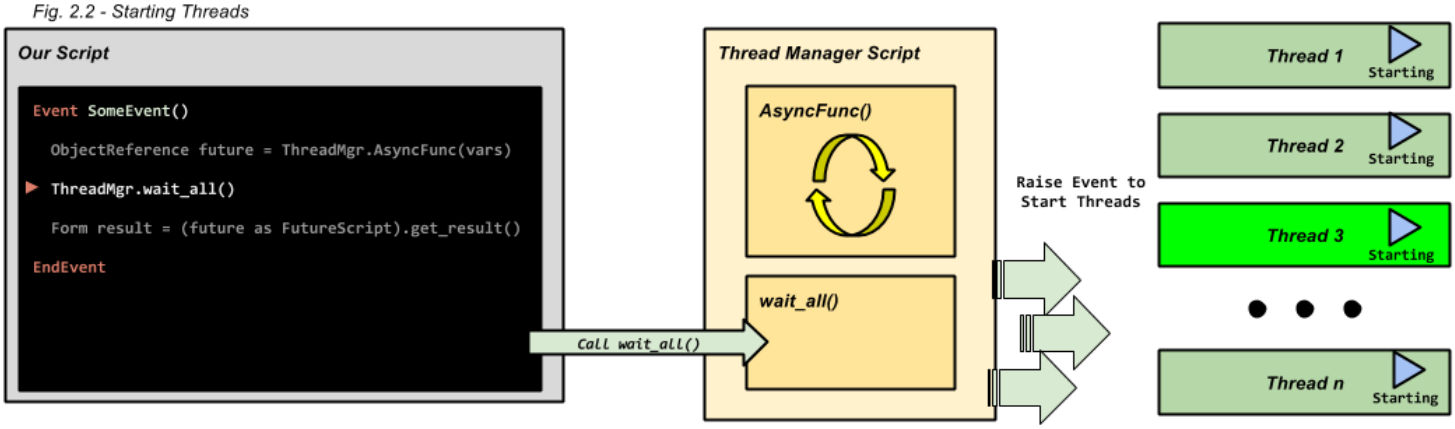
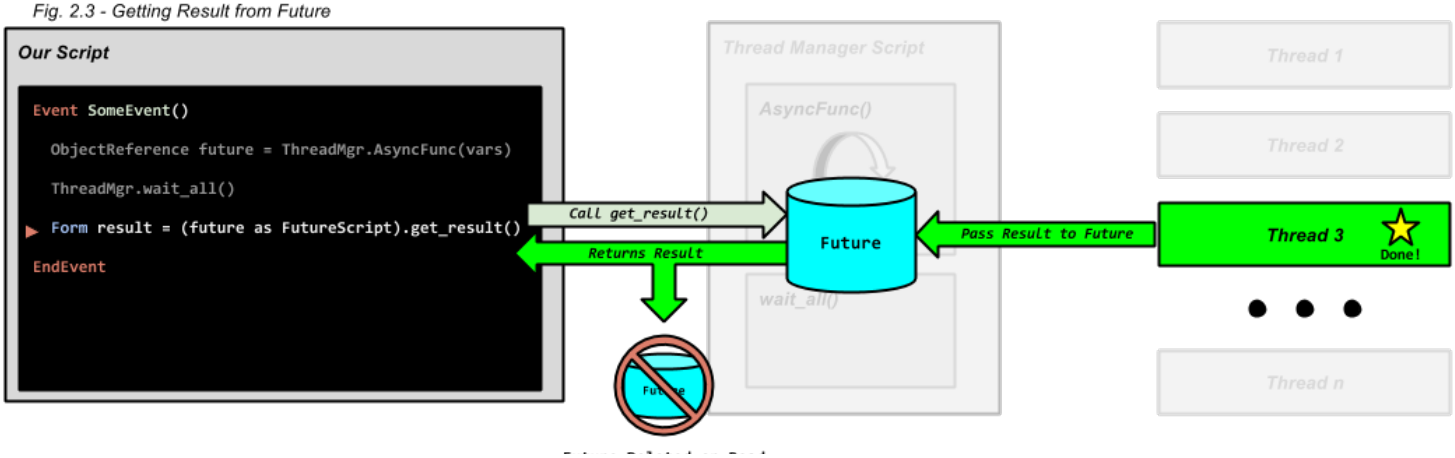


Fig. 2.3 - Getting Result from Future



Above, you can see that the sequence is:

1. Call a function on our **Thread Manager**.
2. The Thread Manager delegates the work to an available **thread**.
3. The Thread Manager returns a **Future** to the caller, who stores it as an ObjectReference.
4. The calling script calls `wait_all()`, which starts all queued threads and waits for them to finish.
5. The calling script calls `get_result()` on the Future. The Future returns the result to the caller.
6. After the result has been read, the Future is deleted.

That is the Futures pattern in a nutshell. Now, we will implement the various parts of this pattern, and put it all together at the end.

Creation Kit

1. **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.
2. **Create Future (Activator):** Next, we want to create an object we will need later, called a `Future`. We'll get into what these do later. Open the Activator tree in the Creation Kit Object Window, and find '**xMarkerActivator**'. Right click and Duplicate this object. Double-click the duplicate and rename it's Editor ID to identify it later; we'll call ours **GuardPlacementFutureActivator**.
3. **Create Anchor (Object Reference):** We now want to create a "Future Anchor". This is an XMarker object reference that we will be placing in a far-off, unused cell. You can create your own blank cell, but **AAADeleteWhenDoneTestJeremy** is also a good candidate. Wherever you decide to place it, drag an XMarker Static from the Object Window of the Creation Kit into the Render Window and name the reference. We'll name ours **GuardPlacementFutureAnchor**. We'll use this to `PlaceAtMe()` Futures on this object later on.

- **Fig. 2.4**:
  Create Quest

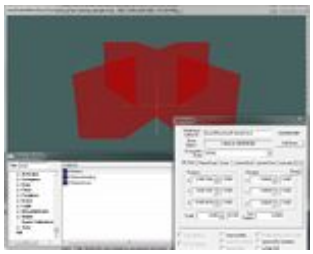
- **Fig. 2.5**:
  Create Future Activator

- **Fig. 2.6**:
  Create Anchor

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 control variables
ObjectReference future
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(Activator akFuture, ObjectReference akFutureAnchor,
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

 ;Create the Future that will contain our result
 future = akFutureAnchor.PlaceAtMe(akFuture)
 return future
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 has_future(ObjectReference akFuture)
    if akFuture == future
        return true
    else
        return false
    endif
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)  ;We could have
passed PlayerRef in as a get_async() parameter, too
  MoveGuardMarkerNearPlayer(tempMarker)
  ObjectReference result = tempMarker.PlaceAtMe(theGuard)
                tempMarker.Disable()
                tempMarker.Delete()

                ;OK, we're done - let's pass the result back to the future
                ;UNCOMMENT THIS after compiling GuardPlacementFuture
  ;(future as GuardPlacementFuture).result = result

                ;Set all variables back to default
  clear_thread_vars()

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

;Another function that does things we want to multithread.
function MoveGuardMarkerNearPlayer(ObjectReference akMarker)
 ;Expensive SetPosition, GetPosition, FindNearestRef, etc calls here (illustration
only)
endFunction

function clear_thread_vars()
 ;Reset all thread variables to default state
 theGuard = None
 theMarker = None
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.
- `get_async()` creates a `Future` which will eventually make its way back to the script that called our Thread Manager function.
- `Event OnGuardPlacement()` will fire if the Thread Manager raises the event.
- The thread returns its results back to the Future it created.
- 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).

In the end, the function that we call in our Thread Manager will return a `Future`, which we can use to get our return value later.

```
scriptname GuardPlacementThreadManager extends Quest

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

Activator property GuardPlacementFutureActivator auto
{Our Future object.}

ObjectReference property GuardPlacementFutureAnchor auto
{Our Future Anchor object reference.}

Static property XMarker auto
{Something a thread needs; our threads don't declare their own properties.}

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.
ObjectReference function PlaceConjuredGuardAsync(ActorBase akGuard)
 ObjectReference future
 while !future
  if !thread01.queued()
   future = thread01.get_async(GuardPlacementFutureActivator,
GuardPlacementFutureAnchor, akGuard, XMarker)
  elseif !thread02.queued()
   future = thread02.get_async(GuardPlacementFutureActivator,
```

```
GuardPlacementFutureAnchor, akGuard, XMarker)
  ...
  elseif !thread09.queued()
   future = thread09.get_async(GuardPlacementFutureActivator,
GuardPlacementFutureAnchor, akGuard, XMarker)
  elseif !thread10.queued()
   future = thread10.get_async(GuardPlacementFutureActivator,
GuardPlacementFutureAnchor, akGuard, XMarker)
  else
   ;All threads are queued; start all threads, wait, and try again.
                        wait_all()
  endif
 endWhile

 return future
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

;A helper function that can avert permanent thread failure if something goes wrong
function TryToUnlockThread(ObjectReference akFuture)
    bool success = false
    if thread01.has_future(akFuture)
        success = thread01.force_unlock()
```

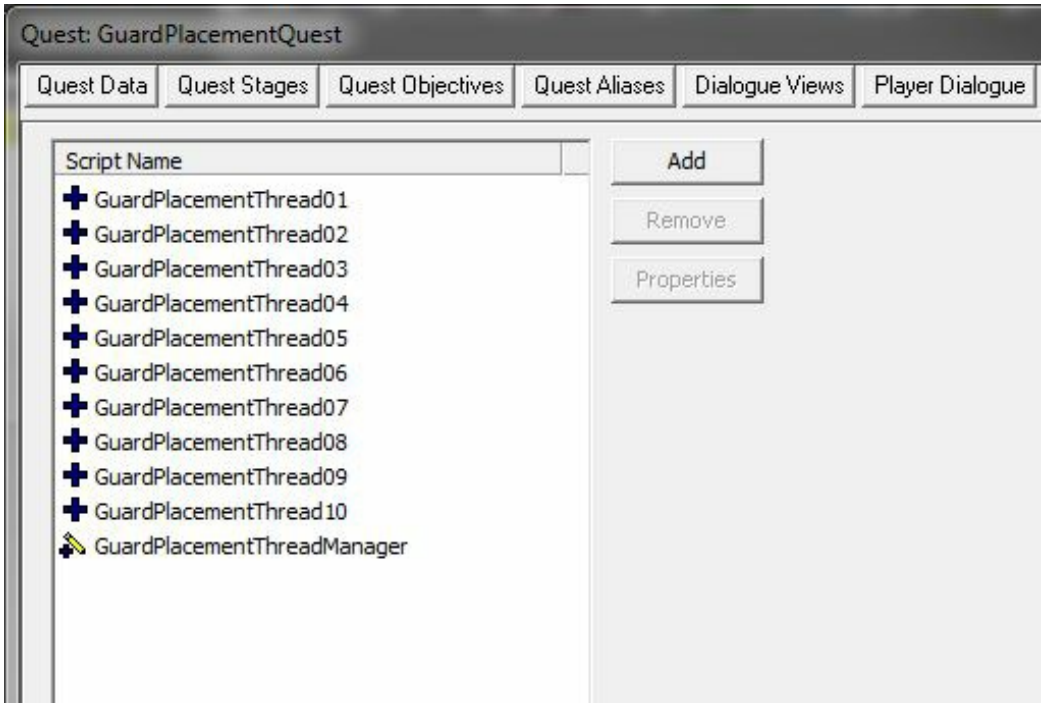
```
    elseif thread02.has_future(akFuture)
        success = thread02.force_unlock()
    ;...and so on
    elseif thread09.has_future(akFuture)
        success = thread09.force_unlock()
    elseif thread10.has_future(akFuture)
        success = thread10.force_unlock()
    endif

    if !success
        debug.trace("Error: A thread has encountered an error and has become
unresponsive.")
    else
        debug.trace("Warning: An unresponsive thread was successfully unlocked.")
    endif
endFunction

;Create the ModEvent that will start all threads
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. The function then returns a `Future` once an available thread is found.

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:



Back to the Future

Futures are a concept from parallel processing. It can be thought of as a placeholder in lieu of your result until your result has arrived. Like the Google App Engine version that this was inspired by, when the Future is created, it will probably not have any results yet. Your script can store a `Future` and later call the `Future` object's `get_result()` function, which should return your results immediately.

- A few notes about Futures:
  - Futures **contain the result** of a thread that has finished.
  - Futures are lightweight Activator ObjectReferences placed in an unloaded cell.
  - A `Future` is **temporary**, and exists until the result is read, after which, the `Future` is destroyed. Make sure to save your results to your own variable if you will need them later, since the `Future` will no longer exist after calling `get_result()`. This keeps the number of ObjectReferences created under control, and helps prevent save game size bloat.
  - `get_result()` is technically *blocking*, meaning it waits until results are received from the thread, and then returns. However, since `wait_all()` waits for all threads to complete, there should be no reason you should have to wait on your results when calling this function, unless something went wrong.
  - The result of a `Future` is the same as the result of any other function in Papyrus, and can return None, false, etc if an error is encountered. Code the result of a `Future` like you would the result of anything else and anticipate errors accordingly.
  - Futures will attempt to unlock threads that have become unresponsive.

Let's create our Future:

```
scriptname GuardPlacementFuture extends ObjectReference

Quest property GuardPlacementQuest auto

ObjectReference r
ObjectReference property result hidden
 function set(ObjectReference akResult)
  done = true
  r = akResult
 endFunction
endProperty

bool done = false
bool function done()
 return done
endFunction

ObjectReference function get_result()
 ;Terminate the request after 10 seconds, or as soon as we have a result
 int i = 0
 while !done && i < 100
  i += 1
  utility.wait(0.1)
 endWhile
 RegisterForSingleUpdate(0.1)

        if i >= 100
                ;Our thread probably encountered an error and is locked up; we need
to unlock it.
                (GuardPlacementQuest as
GuardPlacementThreadManager).TryToUnlockThread(self as ObjectReference)
        endif
 return r
endFunction

Event OnUpdate()
 self.Disable()
 self.Delete()
endEvent
```

This script should be **compiled and attached to the Future Activator** object we created earlier. After you've attached it, make sure to **fill the properties.**

Note the Type of the result; this could be changed to any data type you need to return.

As a best practice, only interface with the `Future` using its member functions, `done()` and `get_result()`.

Quick Detour: Revisiting the Thread Script

There was a line from our thread script that was commented out, because our Future script didn't exist yet:

```
;(future as GuardPlacementFuture).result = result
```

Go back and uncomment this line and recompile the parent thread script. You don't need to recompile all of the children.

This is important! If you don't uncomment this line, your thread will never return results to the Future!

Tying it All Together

Now that we've created our Threads, our Thread Manager, and our Future script, 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, and store the `Future` it returns.
2. Later, call the `get_results()` function of the `Future` to retrieve the 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
...
ObjectReference Guard20

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

  ;Call PlaceConjuredGuardAsync for each Guard and store the returned Future
  ObjectReference Guard1Future = threadmgr.PlaceConjuredGuardAsync(Guard)
  ObjectReference Guard2Future = threadmgr.PlaceConjuredGuardAsync(Guard)
  ObjectReference Guard3Future = threadmgr.PlaceConjuredGuardAsync(Guard)
  ;...and so on
  ObjectReference Guard19Future = threadmgr.PlaceConjuredGuardAsync(Guard)
  ObjectReference Guard20Future = threadmgr.PlaceConjuredGuardAsync(Guard)

               ;Begin working and wait for all of our threads to complete.
               threadmgr.wait_all()

  ;Collect the results
  Guard1 = (Guard1Future as GuardPlacementFuture).get_result()
  Guard2 = (Guard2Future as GuardPlacementFuture).get_result()
  Guard3 = (Guard3Future as GuardPlacementFuture).get_result()
  ;...and so on
  Guard19 = (Guard19Future as GuardPlacementFuture).get_result()
  Guard20 = (Guard20Future as GuardPlacementFuture).get_result()
 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
```

Here, instead of doing the work in our script, we delegated the work to the Thread Manager, and stored the Futures that it returned to us. Then, we gathered the results using our Futures' `get_result()` function. We don't have to worry about our threads or the state of the Futures; those are freed up and cleared for us by the system.

Even though all of the threads are working in parallel and might not finish at the same time, the `get_result()` function will wait until a result is available before returning. We can be sure that we will get the results even if they are processed out of order. For instance, if thread 2 completed before thread 1, calling the thread 1 Future's `get_result()` function will pause the script until a result is available. Then the thread 2 Future's result is gathered, and so on.

Notes on Futures

- Make sure to always call wait_all() after calling your asynchronous functions, or your threads **will not start**.
- We call `RegisterForModEvent()` on our Thread Manager's `OnInit()` block. Remember that this will need to be re-registered after **every game load**. You will need to define a Player Alias with an attached script that has an `OnPlayerLoadGame()` event defined that re-registers for this mod event. Any script attached to the quest with the threads can register for the event, and all threads will begin receiving those events.
- Be a good Papyrus and Skyrim citizen and read the results from your Futures as soon as you are able so that they can be disposed of. If Futures begin to pile up without being read and destroyed, save game bloat could occur.
- If you are running operations in an always-on background script that you want to multithread, and you will always have the same number of results back, it may make more sense for you to implement a static set of Future references that are never destroyed that you continue to reuse. This would prevent the churn of Futures being created and destroyed and may lend itself to faster performance. Keep in mind that this would probably result in some data loss if your Futures are not read from regularly as the new results overwrite the old ones.
- 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. 1.4 seconds to complete
- **20 Threads:** Avg. 1.1 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.