Patterns for using the InitOnce functions

devblogs.microsoft.com/oldnewthing/20110408-01

April 8, 2011

Raymond Chen

Since writing lock-free code is is such a headache-inducer, you're probably best off making some other people suffer the headaches for you. And those other people are the kernel folks, who have developed quite a few lock-free building blocks so you don't have to. For example, there's a collection of functions for manipulating interlocked lists. But today we're going to look at the one-time initialization functions.

The simplest version of the one-time initialization functions isn't actually lock-free, but it does implement the double-checked-lock pattern for you so you don't have to worry about the details. The usage pattern for the `InitOnceExecuteOnce` function is pretty simple. Here it is in its simplest form:

```
int SomeGlobalInteger;
BOOL CALLBACK ThisRunsAtMostOnce(
    PINIT_ONCE initOnce,
    PVOID Parameter,
    PVOID *Context)
{
    calculate_an_integer(&SomeGlobalInteger);
    return TRUE;
}
void InitializeThatGlobalInteger()
{
    static INIT_ONCE initOnce = INIT_ONCE_STATIC_INIT;
    InitOnceExecuteOnce(&initOnce,
                        ThisRunsAtMostOnce,
                        nullptr, nullptr);
}
```

In the simplest form, you give `InitOnceExecuteOnce` an `INIT_ONCE` structure (where it records its state), and a callback. If this is the first time that `InitOnceExecuteOnce` is called for a particular `INIT_ONCE` structure, it calls the callback. The callback can do whatever it likes, but presumably it's doing some one-time initialization. If another thread calls `InitOnceExecuteOnce` on the same `INIT_ONCE` structure, that other thread will wait until the first thread is finished its one-time execution.

We can make this a tiny bit fancier by supposing that the calculation of the integer can fail.

```
BOOL CALLBACK ThisSucceedsAtMostOnce(
    PINIT_ONCE initOnce,
    PVOID Parameter,
    PVOID *Context)
{
    return SUCCEEDED(calculate_an_integer(&SomeGlobalInteger));
}
BOOL TryToInitializeThatGlobalInteger()
{
    static INIT_ONCE initOnce = INIT_ONCE_STATIC_INIT;
    return InitOnceExecuteOnce(&initOnce,
                               ThisSucceedsAtMostOnce,
                               nullptr, nullptr);
}
```

If your initialization function returns `FALSE`, then the initialization is considered to have failed, and the next time somebody calls `InitOnceExecuteOnce`, it will try to initialize again.

A slightly fancier use of the `InitOnceExecuteOnce` function takes advantage of the `Context` parameter. The kernel folks noticed that an `INIT_ONCE` structure in the "initialized" state has a lot of unused bits, and they've offered to let you use them. This is convenient if the thing you're initializing is a pointer to a C++ object, because it means that there's only one thing you need to worry about instead of two.

```
BOOL CALLBACK AllocateAndInitializeTheThing(
    PINIT_ONCE initOnce,
    PVOID Parameter,
    PVOID *Context)
{
    *Context = new(nothrow) Thing();
    return *Context != nullptr;
}
Thing *GetSingletonThing(int arg1, int arg2)
{
    static INIT_ONCE initOnce = INIT_ONCE_STATIC_INIT;
    void *Result;
    if (InitOnceExecuteOnce(&initOnce,
                        AllocateAndInitializeTheThing,
                        nullptr, &Result))
    {
        return static_cast<Thing*>(Result);
    }
    return nullptr;
}
```

The final parameter to `InitOnceExecuteOnce` function receives the magic almost-pointer-sized data that the function will remember for you. Your callback function passes this magic value back through the `Context` parameter, and then `InitOnceExecuteOnce` gives it back to you as the `Result`.

As before, if two threads call `InitOnceExecuteOnce` simultaneously on an uninitialized `INIT_ONCE` structure, one of them will call the initialization function and the other will wait.

Up until now, we've been looking at the synchronous initialization patterns. They aren't lock-free: If you call `InitOnceExecuteOnce` and initialization of the the `INIT_ONCE` structure is already in progress, your call will wait until that initialization attempt completes (either successfully or unsuccessfully).

More interesting is the asynchronous pattern. Here it is, as applied to our `Singleton-Manager` exercise:

```
 SingletonManager(const SINGLETONINFO *rgsi, UINT csi)
               : m_rgsi(rgsi), m_csi(csi),
                 m_rgio(new INITONCE[csi]) {
   for (UINT iio = 0; iio < csi; iio++) {
    InitOnceInitialize(&m_rgio[iio]);
   }
 }
 ...
 // Array that describes objects we've created
 // runs parallel to m_rgsi
 INIT_ONCE *m_rgio;
};
ITEMCONTROLLER *SingletonManager::Lookup(DWORD dwId)
{
 ... same as before until we reach the "singleton constructor pattern"
 void *pv = NULL;
 BOOL fPending;
 if (!InitOnceBeginInitialize(&m_rgio[i], INIT_ONCE_ASYNC,
                              &fPending, &pv)) return NULL;
 if (fPending) {
  ITEMCONTROLLER *pic = m_rgsi[i].pfnCreateController();
  if (pic && InitOnceComplete(&m_rgio[i],
                              INIT_ONCE_ASYNC, pic)) {
   pv = pic;
  } else {
   // lost the race - discard ours and retrieve the winner
   delete pic;
   InitOnceBeginInitialize(&m_rgio[i], INIT_ONCE_CHECK_ONLY,
                           &fPending, &pv);
  }
 }
 return static_cast<ITEMCONTROLLER *>(pv);
}
```

The pattern for asynchronous initialization is as follows:

- Call `InitOnceBeginInitialize` in async mode.
- If it returns `fPending == FALSE`, then initialization has already been performed and you can go ahead and use the result passed back in the final parameter.
- Otherwise, initialization is pending. Do your initialization, but remember that since this is a lock-free algorithm, there can be many threads trying to initialize simultaneously, so you have to be careful how you manipulate global state. This pattern works best if initialization takes the form of creating a new object (because that means multiple threads performing initialization are each creating independent objects).
- If you successfully created the object, call `InitOnceComplete` with the result of your initialization.
- If `InitOnceComplete` succeeds, then you won the initialization race, and you're done.
- If `InitOnceComplete` fails, then you lost the initialization race and should clean up your failed initialization. In that case, you should call `InitOnceBeginInitialize` one last time to get the answer from the winner.

it's conceptually simple; it just takes a while to explain. but at least now it's in recipe form.

Exercise: Instead of calling `InitOnceComplete` with `INIT_ONCE_INIT_FAILED`, what happens if the function simply returns without ever completing the init-once?

Exercise: What happens if two threads try to perform asynchronous initialization and the first one to complete fails?

Exercise: Combine the results of the first two exercises and draw a conclusion.

Update: I got it wrong in the case of a failed asynchronous initialization. You're just supposed to abandon the initialization rather than report failure. The code and discussion have been updated.

Raymond Chen

Follow