Lock-free algorithms: The singleton constructor

devblogs.microsoft.com/oldnewthing/20110406-00

April 6, 2011

Raymond Chen

The first half may be familiar to many (most?) readers, but there's an interesting exercise at the bottom.

A very useful pattern for the Interlocked* functions is lock-free lazy initialization via `InterlockedCompareExchangePointerRelease`. Yes, that's a really long function name, but it turns out every part of it important.

```
Widget *g_pwidCached;
Widget *GetSingletonWidget()
{
 Widget *pwid = g_pwidCached;
 if (!pwid) {
  pwid = new(nothrow) Widget();
  if (pwid) {
   Widget *pwidOld = reinterpret_cast<Widget*>
       (InterlockedCompareExchangePointerRelease(
          &reinterpret_cast<PVOID&>(g_pwidCached),
          pwid, NULL));
   if (pwidOld) {
    delete pwid; // lost the race - destroy the redundant copy
    pwid = pwidOld; // use the old one
   }
  }
 }
 return pwid;
}
```

This is a double-check lock, but without the locking. Instead of taking lock when doing the initial construction, we just let it be a free-for-all over who gets to create the object. If five threads all reach this code at the same time, sure, let's create five objects. After everybody creates what they think is the winning object, they called `InterlockedCompareExchange-PointerRelease` to attempt to update the global pointer.

The parts of the name of the `InterlockedCompareExchangePointerRelease` function work like this:

- `Interlocked` : The operation is atomic. This is important to avoid two threads successfully updating the value of `g_pwidCached` .
- `Compare` : The value in `g_pwidCached` is compared against `NULL` .
- `Exchange` : If the values are equal, then `g_pwidCached` is set to `pwid` . This, combined with the comparison, ensures that only one thread gets to set the value of `g_pwidCached` .
- `Pointer` : The operations are on pointer-sized data.
- `Release` : The operation takes place with release semantics. This is important to ensure that the `pwid` we created is fully-constructed before we publish its pointer to other processors.

This technique is suitable when it's okay to let multiple threads try to create the singleton (and have all the losers destroy their copy). If creating the singleton is expensive or has unwanted side-effects, then you don't want to use the free-for-all algorithm.

Bonus reading:

- One-Time Initialization helper functions save you from having to write all this code yourself. They deal with all the synchronization and memory barrier issues, and support both the one-person-gets-to-initialize and the free-for-all-initialization models.
- A lazy initialization primitive for .NET provides a C# version of the same.

Okay, now here's the interesting exercise. This is an actual problem I helped out with, although details have been changed for expository purposes.

> We have a data structure which manages a bunch of singleton objects, let's say that they are instances of a structure called `ITEMCONTROLLER` and they are keyed by a 32-bit ID. We're looking for design suggestions on making it thread-safe. The existing code goes like this (pseudocode):

```
struct ITEMCONTROLLER;
struct SINGLETONINFO {
 DWORD dwId;
 ITEMCONTROLLER *(*pfnCreateController)();
};
class SingletonManager {
public:
 // rgsi is an array that describes how to create the objects.
 // It's a static array, with csi in the range 20 to 50.
 SingletonManager(const SINGLETONINFO *rgsi, UINT csi)
               : m_rgsi(rgsi), m_csi(csi),
                 m_rgcs(NULL), m_ccs(0), m_ccsAlloc(0) { }
 ~SingletonManager() { ... }
 ITEMCONTROLLER *Lookup(DWORD dwId);
private:
 struct CREATEDSINGLETON {
  DWORD dwId;
  ITEMCONTROLLER *pic;
 };
private:
 const SINGLETONINFO *m_rgsi;
 int m_csi;
 // Array that describes objects we've created
 CREATEDSINGLETON *m_rgcs;
 int m_ccs;
};
ITEMCONTROLLER *SingletonManager::Lookup(DWORD dwId)
{
 int i;
 // See if we already created one
 for (i = 0; i < m_ccs; i++) {
  if (m_rgcs[i].dwId == dwId)
   return m_rgcs[i].pic;
 }
 // Not yet created - time to create one
 ITEMCONTROLLER *pic;
 for (i = 0; i < m_rgsi; i++) {
  if (m_rgsi[i].dwId == dwId) {
   pic = m_rgsi[i].pfnCreateController();
   break;
  }
 }
 if (pic == NULL) return;
 ... if m_rgcs == NULL then allocate it and update m_ccsAlloc
 ... else realloc it bigger and update m_ccsAlloc
 // append to our array so we can find it next time
 m_rgcs[m_ccs].dwId = dwId;
 m_rgcs[m_ccs].pic  = pic;
 m_ccs++;
 return pic;
}
```

> In words, the `SingletonManager` takes an array of `SINGLETONINFO` structures, each of which contains an ID and a function to call to create the object with that ID. To look up an entry, we first check if we already created one; if so, then we just return the existing one. Otherwise, we create the object (using `pfnCreateController`) and add it to our array of created objects.
>
> Our initial inclination is to put a critical section around the entire `Lookup` function, but maybe there's something more clever we can do here. Maybe a slim reader-writer lock?

Bonus chatter: Although it's the case on Windows that the plain versions of the interlocked functions impose both acquire and release semantics, other platforms may not follow Windows' lead. In particular, on the XBOX360 platform, the plain versions of the interlocked functions impose neither acquire nor release semantics. I don't know what the rules are for Windows CE.

Erratum: I once knew but subsequently forgot that the singleton pattern described in this article (with the `InterlockedCompareExchangePointer`) is not safe on some CPU architectures. An additional `MemoryBarrier()` needs to be inserted after the fetch of the single pointer to ensure that indirections through it will retrieve the new values and not any cached old values:

```
Widget *GetSingletonWidget()
{
 Widget *pwid = g_pwidCached;
 if (!pwid) {
  ...
 } else {
  // Ensure that dereferences of pwid access new values and not old
  // cached values.
  MemoryBarrier();
 }
 return pwid;
}
```

The discussion of lock-free algorithms continues (with probably more errors!) next time.

Raymond Chen

Follow