Lock-free algorithms: The singleton constructor (answer to exercises)

devblogs.microsoft.com/oldnewthing/20110408-00

April 8, 2011

Raymond Chen

A few days ago, I asked you to make an existing class multithread-safe. The class caches objects called `SINGLETONINFO` which are indexed by a 32-bit ID. The cache is implemented as an array that dynamically resizes as more items are added to it. A naïve multithreaded version might use a slim reader-writer lock with shared access on reads, exclusive access on writes, and mixed access on the treacherous "create if it doesn't already exist" path.

Let's see. First of all, the function doesn't allocate the memory for the cache until somebody actually tries to look something up. But duh, that's the whole point of the class: To look up things! The only time this lazy-initialization actually provides a benefit is if somebody creates a `SingletonManager`, *calls no methods on it*, and then destroys it.

This doesn't happen in practice, and even if it did, it's certainly not a scenario we're going to optimize for. Get rid of the lazy-initialization of the cache; it makes multithreading unnecessarily complicated.

Second, since the only way an `ITEMCONTROLLER` can get into the cache is via the `SINGLETONINFO`, if a `SingletonManager` is told, "Here are 30 item IDs and their corresponding controller creation functions," then the cache can never hold more than 30 items. If you only know how to create 30 items, and you never create more than one copy of each item, then you're never going to create more than 30 items.

Therefore, instead of managing a dynamically-growing array, we can allocate a fixed-size array at construction of length equal to the number of `SINGLETONINFO` elements. This avoids having to lock around the code that reallocates the array. Since the array length is in the range 30–50, we don't have the problem of allocating megabytes of memory to track just a few objects. In the worst case, we allocate a 50-element cache.

Next, we can store each `ITEMCONTROLLER` in the same position in the cache array as it exists in the `SINGLETONINFO` array.

With these simplifications, we see that we don't need to do any locking or complicated duplicate-detection. After locating the ID in the `SINGLETONINFO` array, look at the corresponding entry in the cache array and perform a singleton initialization there.

```
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_rgpic(new ITEMCONTROLLER*[csi]) { }
 ~SingletonManager() { ... }
 ITEMCONTROLLER *Lookup(DWORD dwId);
private:
 const SINGLETONINFO *m_rgsi;
 int m_csi;
 // Array that describes objects we've created
 // runs parallel to m_rgsi
 ITEMCONTROLLER *m_pic;
};
ITEMCONTROLLER *SingletonManager::Lookup(DWORD dwId)
{
 int i;
 // Convert ID to index
 for (i = 0; i = m_csi) return NULL; // not something we know about
 // Singleton constructor pattern
 if (!m_rgpic[i]) {
  ITEMCONTROLLER *pic = m_rgsi[i].pfnCreateController();
  if (!pic) return NULL;
  if (InterlockedCompareExchangePointerRelease(
          &reinterpret_cast<PVOID&>(m_rgpic[i]),
          pic, NULL) != 0) {
   delete pic; // lost the race - destroy the redundant copy
  }
 }
 MemoryBarrier();
 return m_rgpic[i];
}
```

Comments on proposed solutions: Gabe pointed out that the reallocation was a sticking point which made a lock-free implementation difficult if not impossible. Credit to him for recognizing the problem.

Thorsten proposed using a linked list instead of an array to avoid the reallocation problem.

Ray Trent reminded us of the C++ function-local static technique, which works if it's what you need, but it has its own problems, such as lack of thread-safety (up until perhaps two weeks ago), and the fact that it doesn't generalize to a solution to the exercise. The not-thread-safe-ness of C++ static initialization was called out as a *feature* in early versions of the C++ language specification (to permit recursive initialization). This was revised in the ISO version of C++, which declared that if control enters a function which is in the middle of initializing its statics, the behavior is *undefined*. I don't know what C++0x has to say about the subject, but seeing as the standard was approved only two weeks ago and hasn't even been formally published yet, it seems premature to expect all compilers to conform to any new multi-threading semantics.

Note that the function-local static technique works only if you want a process-wide singleton. If you need a singleton with a tighter scope (say, "one object per thread" or "one object per transaction"), then the function-local static technique will not work. Which after all was the point of the SingletonManager class: To manage singletons relative to its own scope, not globally. If you had wanted global singletons, then you wouldn't need a singleton manager; you would just have each object manage its own singleton.

To elaborate: Suppose you have an object with a bunch of components. Most clients don't use all the components, so you want to lazy-create those components. Say, each `Transaction` can have an error log file, but you don't want to create the error log file until an error occurs. On the other hand, you want all the errors for a single transaction to go into the same log file.

```
class LogFile : public ITEMCONTROLLER
{
public:
  static ITEMCONTROLLER *Create() { return new LogFile(); }
};
const SINGLETONINFO c_rgsiTransactions[] = {
  { LOGFILE_ID, LogFile::Create };
};
class Transaction
{
public:
  Transaction()
    : m_singletons(c_rgsiTransactions,
                   ARRAYSIZE(c_rgsiTransactions))
  { }
  void LogError(blah blah)
  {
    LogFile *plog = static_cast<LogFile*>
                        (m_singletons.Lookup(LOGFILE_ID));
    if (plog) plog->Log(blah blah);
  }
private:
  SingletonManager m_singletons;
};
```

The singleton manager makes sure that each transaction has at most one log file. But we can't use the function-local static technique in `LogFile::Create`, because we want multiple log files in general, just a singleton log file per transaction. If we had used the function-local static technique in `LogFile::Create`, then all errors would have been logged into a giant log file instead of a separate log file per transaction.

Scott tried to update the singleton atomically but forgot about the thread safety of the reallocation, and the solution had its own holes too. Alex Grigoriev's solution is the classic back-off-and-retry algorithm modulo forgetting to protect against reallocation.

nksingh was the first to observe that the reallocation could be removed, and effectively came up with the solution presented here. (But missed the further optimization that the `dwId` member was redundant.) He also recommended using the InitOnce functions, which is something I too recommend. We'll look at the InitOnce functions in a separate article since this one is getting kind of long.

Raymond Chen

Follow