A cache with a bad policy is another name for a memory leak

devblogs.microsoft.com/oldnewthing/20060502-07

May 2, 2006

Raymond Chen

A common performance trick is to reduce time spent in the heap manager by caching the last item freed (or maybe the last few) so that a subsequent allocation can just re-use the item rather than having to go make a new one. But you need to be careful how you do this or you can end up making things worse rather than better. Here's an example motivated by an actual problem the Windows performance team researched.

Consider a cache of variable-sized buffers. I will use only a one-entry cache for simplicity. In real life, the cache would be more complicated: People tend to have a deeper cache of four to ten entries, and you would have to ensure that only one thread used the cache at a time; typically this is done by associating the cache with something that has thread affinity. Furthermore, you probably would keep the size of the cached buffer in a member variable instead of calling `LocalSize` all the time. I've left out all these complications to keep the presentation simple.

```
class BufferCache {
public:
 BufferCache() : m_pCache(NULL) { }
 ~BufferCache() { LocalFree(m_pCache); }
 void *GetBuffer(SIZE_T cb);
 void ReturnBuffer(void *p);
private:
 void *m_pCache;
};
```

If a request for a memory buffer arrives and it can be satisfied from the cache, then the cached buffer is returned. Otherwise, a brand new buffer is allocated.

```
void *BufferCache::GetBuffer(SIZE_T cb)
{
 // Satisfy from cache if possible
 if (m_pCache && LocalSize(m_pCache) >= cb) {
  void *p = m_pCache;
  m_pCache = NULL;
  return p;
 }
 return LocalAlloc(LMEM_FIXED, cb);
}
```

When a buffer is returned to the cache, we compare it against the item already in the cache and keep the bigger one, since that is more likely to satisfy a `GetBuffer` in the future. (In the general case of a multiple-entry cache, we would free the smallest entry.)

```
// Flawed design - see discussion
void BufferCache::ReturnBuffer(void *p)
{
 SIZE_T cb = LocalSize(p);
 if (!m_pCache || cb > LocalSize(m_pCache)) {
  // Returned buffer is bigger than the cache:
  // Keep the returned buffer
  LocalFree(m_pCache);
  m_pCache = p;
 } else {
  // Returned buffer is smaller than the cache:
  // Keep the cache
  LocalFree(p);
 }
}
```

Why is this a flawed design? I'll let you think about this for a while.

No really, I want you to think about it.

Are you thinking? Take your time; I'll be here when you're done.

Okay, since I know you haven't actually thought about it but are just sitting there waiting for me to tell you, I'll give you a bit of a nudge.

The distribution of buffer sizes is rarely uniform. The most common distribution is that small buffers are popular, with larger and larger buffers being required less and less often. Let's write a sample program that allocates and frees memory according to this pattern. To make the bad behavior easier to spot in a short run, I'm going to use a somewhat flat distribution and say that half of the buffers are small, with larger buffers becoming less popular according to exponential decay. In practice, the decay curve is usually much, much steeper.

```
#include <vector>
#include <iostream>
// Since this is just a quick test, we're going to be sloppy
using namespace std; //  sloppy
int __cdecl main(int argc, char **argv)
{
 BufferCache b;
 // seeding the random number generator is not important here
 vector<void *> v; // keeps track of allocated memory
 for (;;) {
  // randomly allocate and free
  if (v.size() == 0 || (rand() & 1)) { // allocate
   SIZE_T cb = 100;
   while (cb < 1024 * 1024 && (rand() & 1)) {
    cb *= 2; // exponential decay distribution up to 1MB
   }
   void* p = b.GetBuffer(cb);
   if (p) {
    cout << " A" << LocalSize(p) << "/" << cb;
    v.push_back(p);
   }
  } else { // free
   int victim = rand() % v.size(); // choose one at random
   cout << " F" << LocalSize(v[victim]);
   b.ReturnBuffer(v[victim]); // free it
   v[victim] = v.back();
   v.pop_back();
  }
 }
}
```

This short program randomly allocates and frees memory from the buffer cache, printing (rather cryptically) the size of the blocks allocated and freed. When memory is allocated, it prints “A1/2” where “1” is the size of the block actually allocated and “2” is the size requested. When freeing memory, it prints “F3” where “3” is the size of the block allocated. Run this program, let it do its thing for maybe ten, fifteen seconds, then pause the output and study it. I’ll wait. If you’re too lazy to actually compile and run the program, I’ve included some sample output for you to study:

```
F102400 A102400/400 F800 F200 A800/100 A200/200 A400/400
A400/400 A200/200 F1600 A1600/100 F100 F800 F25600 A25600/200
F12800 A12800/200 F200 F400 A400/100 F200 A200/100 A200/200
A100/100 F200 F3200 A3200/400 A200/200 F51200 F800 F25600
F1600 F1600 A51200/100 F100 A100/100 F3200 F200 F409600 F100
A409600/400 A100/100 F200 F3200 A3200/800 A400/400 F800 F3200
F200 F12800 A12800/200 A100/100 F200 F25600 F400 F6400
A25600/100 F100 F200 F400 F200 F800 F400 A800/800 A100/100
```

Still waiting.

Okay, maybe you don't see it. Let's make the effect even more obvious by printing some statistics periodically. Of course, to generate the statistics, we need to keep track of them, so we'll have to remember how big the requested buffer was (which we'll do in the buffer itself):

```
int __cdecl main(int argc, char **argv)
{
 BufferCache b;
 // seeding the random number generator is not important here
 vector<void *> v; // keeps track of allocated memory
 SIZE_T cbAlloc = 0, cbNeeded = 0;
 for (int count = 0; ; count++) {
  // randomly allocate and free
  if (v.size() == 0 || (rand() & 1)) { // allocate
   SIZE_T cb = 100;
   while (cb < 1024 * 1024 && !(rand() % 4)) {
    cb *= 2; // exponential decay distribution up to 1MB
   }
   void* p = b.GetBuffer(cb);
   if (p) {
    *(SIZE_T*)p = cb;
    cbAlloc += LocalSize(p);
    cbNeeded += cb;
    v.push_back(p);
   }
  } else { // free
   int victim = rand() % v.size(); // choose one at random
   cbAlloc -= LocalSize(v[victim]);
   cbNeeded -= *(SIZE_T*)v[victim];
   b.ReturnBuffer(v[victim]); // free it
   v[victim] = v.back();
   v.pop_back();
  }
  if (count % 100 == 0) {
   cout << count << ": " << v.size() << " buffers, "
        << cbNeeded << "/" << cbAlloc << "="
        << cbNeeded * 100.0 / cbAlloc << "% used" << endl;
  }
 }
}
```

This new version keeps track of how many bytes were allocated as opposed to how many were actually needed, and prints a summary of those statistics every hundred allocations. Since I know you aren't actually going to run it yourself, I've run it for you. Here is some sample output:

```
0: 1 buffers, 400/400=100% used
100: 7 buffers, 4300/106600=4.03377% used
200: 5 buffers, 1800/103800=1.7341% used
300: 19 buffers, 9800/115800=8.46287% used
400: 13 buffers, 5100/114000=4.47368% used
500: 7 buffers, 2500/28100=8.8968% used
...
37200: 65 buffers, 129000/2097100=6.15135% used
37300: 55 buffers, 18100/2031400=0.891011% used
37400: 35 buffers, 10400/2015800=0.515924% used
37500: 43 buffers, 10700/1869100=0.572468% used
37600: 49 buffers, 17200/1874000=0.917823% used
37700: 75 buffers, 26000/1889900=1.37573% used
37800: 89 buffers, 30300/1903100=1.59214% used
37900: 91 buffers, 29600/1911900=1.5482% used
```

By this point, the problem should be obvious: We're wasting insane quantities of memory. For example, after step 37900, we've allocated 1.8MB of memory when we needed only 30KB, for a waste of over 98%.

How did we go horribly wrong?

Recall that most of the time, the buffer being allocated is a small buffer, and most of the time, a small buffer is freed. But it's the rare case of a large buffer that messes up everything. The first time a large buffer is requested, it can't come from the cache, since the cache has only small buffers, so it must be allocated. And when it is returned, it is kept, since the cache keeps the largest buffer.

The next allocation comes in, and it's probably one of the common-case small buffers, and it is given the cached buffer—which is big. You're wasting a big buffer on something that needs only 100 bytes. Some time later, another rare big buffer request comes in, and since that other big buffer got wasted on a small allocation, you have to allocate a new big buffer. You allocated two big buffers even though you need only one. Since big buffers are rare, it is unlikely that a big buffer will be given to a caller that actually **needs** a big buffer; it is much more likely to be given to a caller that needs a small buffer.

> Bad effect 1: Big buffers get wasted on small callers.

Notice that once a big buffer enters the system, it is hard to get rid of, since a returned big buffer will be compared against what is likely to be a small buffer, and the small buffer will lose.

> Bad effect 2: Big buffers rarely go away.

The only way a big buffer can get freed is if the buffer in the cache is itself already a big buffer. If instead of a one-entry cache like we have here, you keep, say, ten buffers in your buffer cache, then in order to free a big buffer, you have to have eleven consecutive

`ReturnBuffer` calls, all of which pass a big buffer.

> Bad effect 3: The more efficient you try to make your cache, the more wasteful it gets!

What's more, when that eleventh call to `ReturnBuffer` is made with a big buffer, it is only the smallest of the big buffers that gets freed. The biggest buffers stay.

> Bad effect 4: When a big buffer does go away, it's only because you are keeping an even bigger buffer!

> Corollary: The biggest buffer never gets freed.

What started out as an "obvious" decision in choosing which buffer to keep has turned into a performance disaster. By favoring big buffers, you allowed them to "poison" the cache, and the longer you let the system run, the more allocations end up being big "poisoned" buffers. It doesn't matter how rare those big blocks are; you will eventually end up in this state. It's just a matter of time.

When the performance team tries to explain this problem to people, many of them get the mistaken impression that the problem is merely that there is wasted space in the cache. But look at our example: Our cache has only one entry and we are still wasting over 90% of the memory. That's because the waste is not in the memory being held by the cache, but rather is in the memory that the cache *hands out*. (It's sort of like that scene in *It's a Wonderful Life* where George Bailey is explaining where all the money is. It's not in the bank; it's in all the places that got money *from* the bank.)

My recommendations:

- Instrument your cache and understand what your program's memory allocation patterns are.
- Use that information to pick a size cutoff point beyond which you simply will not use the cache at all. This ensures that big buffers never get into the cache in the first place. Choosing this cutoff point is usually extremely easy once you look at then allocation histogram.
- Although you've taken the big buffers out of the picture, you will still have the problem that the small buffers will gradually grow up to your cutoff size. (*I.e.*, you still have the same problem, just in miniature.) Therefore, if the cache is full, you should just free the most recently returned buffer regardless of its size.
- Do not use the cached buffer if the waste is too great. You might decide to use multiple "buckets" of cached entries, say one for buffers below 100 bytes, another for buffers between 100 and 200 bytes, and so on. That way, the waste per allocation is never more than 100 bytes.
- Finally, reinstrument your cache to ensure that you're not suffering from yet some other pathological behavior that I haven't taken into account.

Here's a new `ReturnBuffer` implementation that takes some of the above advice into account. Instrumentation shows that three quarters of the allocations are in the 100–200 byte range, so let's cap our cache at 200 bytes.

```
void BufferCache::ReturnBuffer(void *p)
{
 if (m_pCache == NULL && LocalSize(p) <= 200) {
  m_pCache = p;
 } else {
  LocalFree(p);
 }
}
```

With this one seemingly-minor change, our efficiency stays above 90% and occasionally even gets close to 100%:

```
0: 1 buffers, 400/400=100% used
100: 7 buffers, 4300/4400=97.7273% used
200: 5 buffers, 1800/1800=100% used
300: 19 buffers, 9800/9800=100% used
400: 13 buffers, 5100/5100=100% used
500: 7 buffers, 2500/2600=96.1538% used
...
37200: 65 buffers, 129000/130100=99.1545% used
37300: 55 buffers, 18100/18700=96.7914% used
37400: 35 buffers, 10400/11000=94.5455% used
37500: 43 buffers, 10700/11000=97.2727% used
37600: 49 buffers, 17200/18000=95.5556% used
37700: 75 buffers, 26000/26800=97.0149% used
37800: 89 buffers, 30300/31900=94.9843% used
37900: 91 buffers, 29600/30600=96.732% used
```

Don't forget to check out performance guru Rico Mariani's reminder that Caching implies Policy. As he explained to me, "Cache policy is everything so you must be dead certain that your policy is working as you intended. A cache with a bad policy is another name for a memory leak."

Raymond Chen

Follow