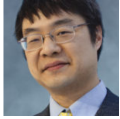
Loading the dictionary, part 6: Taking advantage of our memory allocation pattern

devblogs.microsoft.com/oldnewthing/20050519-00

May 19, 2005



Raymond Chen

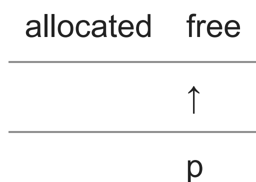
After our latest round of optimization, the 100ms barrier teased us, just milliseconds away. Profiling the resulting program reveals that 60% of the CPU is spent in `operator new`. Is there anything we can do about that?

Indeed, we can. Notice that the memory allocation pattern for the strings in our dictionary is quite special: Once a string is allocated into the dictionary, it is never modified or freed while the dictionary is in use. When the dictionary is freed, all the strings are deleted at once. This means that we can design an allocator tailored to this usage pattern.

I don't know whether there is a standard name for this thing, so I'm just going to call it a `StringPool`. A string pool has the following characteristics:

- Once you allocate a string, you can't modify or free it as long as the pool remains in existence.
- If you destroy the string pool, all the strings in it are destroyed.

We implement it by using the same type of fast allocator that the CLR uses: A single pointer. [25 May 2005: The blog server software corrupts the diagram, sorry.]



To allocate memory, we just increment `p` by the number of bytes we need. If we run out of memory, we just allocate a new block, point `p` to its start, and carve the memory out of the new block. Destroying the pool consists of freeing all the blocks.

Note also that this memory arrangement has very good locality. Instead of scattering the strings all over the heap, they are collected into one location. Furthermore, they are stored in memory **in exactly the order we're going to access them**, which means no wasted page faults or cache lines. (Well, you don't know that's the order we're going to access them, but it's true. This is one of those "performance-guided designs" I mentioned a little while ago.)

```
class StringPool
{
public:
 StringPool();
 ~StringPool();
 LPWSTR AllocString(const WCHAR* pszBegin, const WCHAR* pszEnd);
private:
 union HEADER {
  struct {
   HEADER* m_phdrPrev;
   SIZE_T  m_cb;
  };
  WCHAR alignment;
 };
 enum { MIN_CBCHUNK = 32000,
        MAX_CHARALLOC = 1024*1024 };
private:
 WCHAR*  m_pchNext;   // first available byte
 WCHAR*  m_pchLimit;  // one past last available byte
 HEADER* m_phdrCur;   // current block
 DWORD   m_dwGranularity;
}; // colorization fixed 25 May
```

Each block of memory we allocate begins with a `StringPool::HEADER` structure, which we use to maintain a linked list of blocks as well as providing enough information for us to free the block when we're done.

Exercise: Why is `HEADER` a union containing a structure rather than just being a structure? What is the significance of the `alignment` member?

```
inline RoundUp(DWORD cb, DWORD units)
{
    return ((cb + units - 1) / units) * units;
}
StringPool::StringPool()
 : m_pchNext(NULL), m_pchLimit(NULL), m_phdrCur(NULL)
{
 SYSTEM_INFO si;
 GetSystemInfo(&si);
 m_dwGranularity = RoundUp(sizeof(HEADER) + MIN_CBCHUNK,
                           si.dwAllocationGranularity);
}
```

At construction, we compute the size of our chunks. We base it on the system allocation granularity, choosing the next multiple of the system allocation granularity that is at least `sizeof(HEADER) + MIN_CBCHUNK` in size. Since a chunk is supposed to be a comfortably large block of memory, we need to enforce a minimum chunk size to avoid having an enormous number of tiny chunks if we happen to be running on a machine with a very fine allocation granularity.

```
LPWSTR StringPool::AllocString(const WCHAR* pszBegin, const WCHAR* pszEnd)
{
 size_t cch = pszEnd - pszBegin + 1;
 LPWSTR psz = m_pchNext;
 if (m_pchNext + cch <= m_pchLimit) {
  m_pchNext += cch;
  lstrcpynW(psz, pszBegin, cch);
  return psz;
 }
 if (cch > MAX_CHARALLOC) goto OOM;
 DWORD cbAlloc = RoundUp(cch * sizeof(WCHAR) + sizeof(HEADER),
                                                      m_dwGranularity);
 BYTE* pbNext = reinterpret_cast<BYTE*>(
                  VirtualAlloc(NULL, cbAlloc, MEM_COMMIT, PAGE_READWRITE));
 if (!pbNext) {
OOM:
  static std::bad_alloc OOM;
  throw(OOM);
 }
 m_pchLimit = reinterpret_cast<WCHAR*>(pbNext + cbAlloc);
 HEADER* phdrCur = reinterpret_cast<HEADER*>(pbNext);
 phdrCur->m_phdrPrev = m_phdrCur;
 phdrCur->m_cb = cbAlloc;
 m_phdrCur = phdrCur;
 m_pchNext = reinterpret_cast<WCHAR*>(phdrCur + 1);
 return AllocString(pszBegin, pszEnd);
}
```

To allocate a string, we first try to carve it out of the remainder of the current chunk. This nearly always succeeds.

If the string doesn't fit in the chunk, we allocate a new chunk based on our allocation granularity. To avoid integer overflow in the computation of the desired chunk size, we check against a fixed "maximum allocation" and go stright to the out-of-memory handler if it's too big.

Once we have a new chunk, we link it into our list of `HEADER` s and abandon the old chunk. (Yes, this wastes some memory, but in our usage pattern, it's not much, and trying to squeeze out those last few bytes isn't worth the added complexity.) Once that's done, we try to allocate

again; this second time will certainly succeed since we made sure the new chunk was big enough. (And any decent compiler will detect this as a tail recursion and turn it into a "goto".)

There is subtlety here. Notice that we do not update `m_pchNext` until after we're sure we either satisfied the allocation or allocated a new chunk. This ensures that our member variables are stable at the points where exceptions can be thrown. Writing exception-safe code is hard, and seeing the difference between code that is and isn't exception safe is often quite difficult.

```
StringPool::~StringPool()
{
 HEADER* phdr = m_phdrCur;
 while (phdr) {
  HEADER hdr = *phdr;
  VirtualFree(hdr.m_phdrPrev, hdr.m_cb, MEM_RELEASE);
  phdr = hdr.m_phdrPrev;
 }
}
```

To destroy the string pool, we walk our list of chunks and free each one. Note the importance of copying the `HEADER` out of the chunk before we free it!

Using this string pool requires only small changes to the rest of our program.

```
struct DictionaryEntry
{
 bool Parse(const WCHAR *begin, const WCHAR *end, StringPool& pool);
 // void Destruct() {
 //  delete[] m_pszTrad;
 //  delete[] m_pszSimp;
 //  delete[] m_pszPinyin;
 //  delete[] m_pszEnglish;
 // }
 LPWSTR m_pszTrad;
 LPWSTR m_pszSimp;
 LPWSTR m_pszPinyin;
 LPWSTR m_pszEnglish;
};
class Dictionary
{
public:
 Dictionary();
 // ~Dictionary();
 int Length() { return v.size(); }
private:
 vector v;
 StringPool m_pool;
};
// Dictionary::~Dictionary()
// {
//  for (vector<DictionaryEntry>::iterator i = v.begin();
//         i != v.end(); i++) {
//    i->Destruct();
//   }
// }
```

We no longer need to free the strings in the `DictionaryEntry` manually, so the `Destruct` method and the `Dictionary` destructor can go.

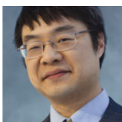
```
bool DictionaryEntry::Parse(
   const WCHAR *begin, const WCHAR *end,
   StringPool& pool)
{
 const WCHAR* pch = std::find(begin, end, L' ');
 if (pch >= end) return false;
 m_pszTrad = pool.AllocString(begin, pch);
 begin = std::find(pch, end, L'[') + 1;
 if (begin >= end) return false;
 pch = std::find(begin, end, L']');
 if (pch >= end) return false;
 m_pszPinyin = pool.AllocString(begin, pch);
 begin = std::find(pch, end, L'/') + 1;
 if (begin >= end) return false;
 for (pch = end; *--pch != L'/'; ) { }
 if (begin >= pch) return false;
 m_pszEnglish = pool.AllocString(begin, pch);
 return true;
}
Dictionary::Dictionary()
{
 ...
    if (de.Parse(buf, buf + cchResult, m_pool)) {
 ...
}
```

And finally, we pass our string pool to `DictionaryEntry::Parse` so it knows where to get memory for its strings from.

With these changes, the dictionary loads in 70ms (or 80ms if you include the time it takes to destroy the dictionary). This is 70% faster than the previous version, and over three times as fast if you include the destruction time.

And now that we've reached our 100ms goal, it's a good time to stop. We've gotten the running time of dictionary loading down from an uncomfortable 2080ms to a peppier 70ms, a nearly 30-fold improvement, by repeatedly profiling and focusing on where the most time is being spent. (I have some more simple tricks that shave a few more milliseconds off the startup time. Perhaps I'll bring them into play if other changes to startup push us over the 100ms boundary. As things stand, the largest CPU consumers are `MultiByteToWideChar` and `lstrcpynW`, so that's where I would focus next.)

That's the end of the first stage. The next stage will be displaying the dictionary in an owner-data listview, but you'll have to wait until next month.



Raymond Chen

Follow