Loading the dictionary, part 2: Character conversion

devblogs.microsoft.com/oldnewthing/20050511-46

May 11, 2005

Raymond Chen

When you want to optimize a program, you first need to know where the time is being spent. There's no point optimizing a function that isn't actually responsible for your poor performance. For example, if a particular function is responsible for 2% of your CPU time, then even if you optimized it down to infinite speed, your program would speed up at best by only little over 2%. In the comments to yesterday's entry, several people put forth suggestions as to how the program could be optimized, in the process quite amply demonstrating this principle. None of the people who made suggestions actually investigated the program to see where the glaring bottleneck was.

(Rico Mariani points out that you also need to take performance in account when doing high level designs, choosing algorithms and data structures that are suitable for the level of performance you need. If profiling reveals that a fundamental design decision is responsible for a performance bottleneck, you're in big trouble. You will see this sort of performance-guided design as the program develops. And you should check out Performance Quiz #6 which starts with the very program we're developing here.)

Upon profiling our dictionary-loader, I discovered that 80% of the CPU time was spent in `getline`. Clearly this is where the focus needs to be. Everything else is just noise.

Digging a little deeper, it turns out that 29% of the CPU time was spent by `getline` doing character set conversion in `codecvt::do_in`. Some debugging revealed that `codecvt::do_in` was being called millions of times, each time converting just one or two characters. In fact, for each character in the file, `codecvt::do_in` was called once and sometimes twice!

Let's get rid of the piecemeal character set conversion and instead convert entire lines at a time.

```
Dictionary::Dictionary()
{
 std::ifstream src;
 typedef std::codecvt<wchar_t, char, mbstate_t> widecvt;
 std::locale l(".950");
 const widecvt& cvt = _USE(l, widecvt); // use_facet<widecvt>(l);
 src.open("cedict.b5");
 string s;
 while (getline(src, s)) {
  if (s.length() > 0 && s[0] != L'#') {
   wchar_t* buf = new wchar_t[s.length()];
   mbstate_t state = 0;
   char* nextsrc;
   wchar_t* nextto;
   if (cvt.in(state, s.data(), s.data() + s.length(), nextsrc,
                  buf, buf + s.length(), nextto) == widecvt::ok) {
    wstring line(buf, nextto - buf);
    DictionaryEntry de;
    if (de.Parse(line)) {
     v.push_back(de);
    }
   }
   delete[] buf;
  }
 }
}
```

Instead of using a `wifstream`, we just use a non-Unicode `ifstream` and convert each line to Unicode manually. Doing it a line at a time rather than a character at a time, we hope, will be more efficient.

We ask code page 950 for a converter, which we call `cvt`. Notice that the Microsoft C++ compiler requires you to use the strange `_USE` macro instead of the more traditional `use_facet`.

For each line that isn't a comment, we convert it to Unicode. Our lives are complicated by the fact that `codecvt::in` requires pointers to elements rather than iterators, which means that we can't use a `wstring` or a `vector`; we need a plain boring `wchar_t[]` array. (Notice that we can cheat on the "from" buffer and use the `string::data()` function to get at a read-only array representation of the string.) If the conversion succeeds, we convert the array into a proper string and continue as before.

With this tweak, the program now loads the dictionary in 1120ms (or 1180ms if you include the time it takes to destroy the dictionary). That's nearly twice as fast as the previous version.

You might think that we could avoid redundant allocations by caching the temporary conversion buffer between lines. I tried that, and surprisingly, it actually slowed the program down by 10ms. Such is the counter-intuitive world of optimization. That's why it's important

to identify your bottlenecks via measurement instead of just guessing at them.

Raymond Chen

Follow