What is DLL import binding?

devblogs.microsoft.com/oldnewthing/20100318-00

March 18, 2010

Raymond Chen

Last time, we saw how *hinting* is used to speed up the resolving of imported functions. Today, we'll look at *binding*.

Recall that the module loader resolves imports by locating the function in the export table of the linked-to DLL and recording the results in the loaded module's table of imported function addresses so that code from the module can jump indirectly through the table and reach the target function.

One of the consequences of this basic idea is that the table of imported function addresses is written to at module load time. Writeable data in a module is stored in the form of copy-on-write pages. Copy-on-write pages are a form of computer optimism: "I'm going to assume that nobody writes to this page, so that I can share it among all copies of the DLL loaded into different processes" (assuming other conditions are met, not important to this discussion; don't make me bring back the nitpicker's corner). "In this way, I can conserve memory, leaving more memory available for other things." But once you write to the page, that assumption is proven false, and the memory manager needs to make a private copy of the page for your process. If two processes load your DLL, they each get their own copy of the memory once they write to it, and the opportunity to share the memory between the two DLLs is lost.

What is particularly sad is when the copy-on-write page is forced to be copied because two processes wrote to the pages, even if the processes *wrote the same value*. Since the two pages are now once again identical, they could in principle be shared again. (The memory manager doesn't do `memcmp`s of every potentially-shared page each time you write to it, on the off chance that you happened to make two pages coincidentally identical. Once a copy-on-write page is written to, the memory manager makes the copy and says, "Oh well, it was good while it lasted.")

One of the cases where two processes both write to the page and write the same value is when they are resolving imports to the same DLL. In that case, the call to `GetProcAddress` will return the same value in both processes (assuming the target DLL is loaded at the same base address in both processes), and you are in the sad case where two processes dirty the page by writing the same value.

To make this sad case happy again, the module loader has an optimization to avoid writing to pages it doesn't have to: We pre-initialize the values in the table of imported function addresses to a prediction as to what the actual address of the function will be. Then we can have the module loader compare the return value of `GetProcAddress` against the predicted value, and if they are the same, it skips the write. In context diff format:

```
// error checking deleted since it's not relevant to the discussion
for (Index = 0; Index < NumberOfImportedFunctions; Index++) {
  FunctionPointer = GetProcAddress(hinst, ImportEntry[Index]);
- TableEntry[Index] = FunctionPointer;
+ if (TableEntry[Index] != FunctionPointer)
+   TableEntry[Index] = FunctionPointer;
}
```

But wait, we can optimize this even more. How about avoiding the entire loop? This saves us the trouble of having to call `GetProcAddress` in the first place.

There is an extra field in the import descriptor table entry called `TimeDateStamp` which records the timestamp of the DLL from which the precomputed function pointer values were obtained. Every DLL has a timestamp, recorded in the module header information. (The format of this timestamp is in seconds since January 1, 1970, commonly known as unix time format.) Before the module loader resolves imported functions, it compares the timestamp in the import descriptor table entry against the timestamp in the actual DLL that got loaded. If they match (and if the actual DLL was loaded at its preferred base address), then the module loader skips the loop entirely: All the precomputed values are correct.

That's the classical model for binding. There have been some changes since the original implementation, but they don't change the underlying principle: Precompute the answers and associate them with a key which lets you determine whether the information against which the values were precomputed matches the information that you actually have.

Binding therefore is a performance optimization to address both wall-clock running time (by reducing the amount of computation performed at module load time) and memory consumption (by reducing the number of copy-on-write pages actually written to).

Exercise: Why is the timestamp stored in the module header? Why not just use the actual file last-modified time?

Exercise: When you rebase a DLL, does it update the timestamp?

Raymond Chen

Follow