The PowerPC 600 series, part 9: The table of contents

devblogs.microsoft.com/oldnewthing/20180816-00

August 16, 2018

Raymond Chen

We saw that the PowerPC 600 series gives you absolute addressing to the top and bottom 32KB of address space. But that doesn't buy you much on Windows NT programs, because all of those addresses are not usable by 32-bit programs. By convention, the *r2* register contains a value called the *table of contents*, which is a pointer to a list of interesting constants the function needs. You can put addresses of global variables here, or you can put other useful constants.

In principle, each function gets its own table of contents, but in practice, the Microsoft compiler generates a single table of contents for the entire module, similar to what the Itanium does. In theory, you could even put your variables directly in the table of contents (which is what the Itanium does), but the Microsoft compiler doesn't. It puts the table of contents in read-only memory. In Itanium-speak you might say that every global variable is considered large. I'm guessing this is to improve page sharing between processes since the table of contents would otherwise be a mix of read-write data and read-only data, but it does mean that accessing any global variable requires *two* memory accesses:

```
    lwz     r3, n(r2)       ; load pointer to variable from toc
    lwz     r3, (r3)        ; load the variable's value
```

The displacement field of the load instruction has a reach of ±32KB, which means that your table of contents has a comfortable maximum size of 64KB. (You would naturally set your table of contents pointer to be 32KB past the start of the table of contents, so that you could take advantage of negative offsets.) But what if you have more than 16384 global objects? No problem, because you don't need a separate pointer in the table of contents for each global object. You can group your global objects into chunks of 64KB and use a single pointer to access the entire chunk. If you have 16384 pointers, each of which can access 64KB of memory, the total amount of memory addressible from the table of contents is one gigabyte, which is hopefully enough to cover all your global objects.

(Also, if you have a monstrous 1-gigabyte global array, you can dedicate a single table of contents entry to that global array. You don't need a separate entry for each 64KB chunk.)

Note that you can have global things other than variables. For example, you'll probably have jump tables for switch statements and vtables for virtual functions.

Since each function requires its table of contents to be set properly, a function pointer on PowerPC is not a pointer to the first instruction. Instead, it's a pointer to a structure consisting of two pointers: The first pointer points to the first instruction of the function, and the second pointer is the table of contents for the function.[1]

The sequence for calling through a function pointer goes like this:

```
    ; call the function pointed to by r11
    ; assumes that our function's toc is saved on the stack at n(r1)
    lwz     r12, (r11)   ; get the code pointer
    lwz     r2, 4(r11)   ; set r2 to the toc for the function being called
    mtctr   r12          ; put code pointer in ctr
    bctrl                ; branch to ctr and link
    lwz     r2, n(r1)    ; restore our toc
```

We load the code pointer and put it into *ctr*. We also load the table of contents for the target function into *r2* so it can access its global variables. We then call the function by calling through *ctr*, and when the function returns, we restore our function's *r2* from wherever we had saved it (typically the stack).

If you're calling a function within the same module, you don't need to update *r2* because all the functions in a module use the same table of contents.

But what if you don't know whether the function is in the same module? For example, it might be an import stub for a naïvely-imported function. Now, in the modern days of link-time code generation, you can tell whether the destination is in the module or not, but in the old days of classical compiling and linking, the only time the compiler would be certain that the target function is in the same module is when the target function is defined in the same translation unit. Otherwise, the compiler isn't quite sure. It could do like the Itanium does and always include a reload of *r2* after the call returns, just in case. But that costs a memory access, so the PowerPC does things a little differently. To dig into what happens, we need to learn about the rest of the PowerPC calling convention, which we'll start looking at next time.

[1] Other ABIs add a third pointer to the structure, called the "environment". Windows NT makes do with just two pointers.

Raymond Chen

Follow