Why load fs:[0x18] into a register and then dereference that, instead of just going for fs:[n] directly?

devblogs.microsoft.com/oldnewthing/20220919-00

September 19, 2022

Raymond Chen

For the purpose of this discussion, I'm going to assume you are familiar with the x86 32-bit and 64-bit instruction set architectures.

In Windows on x86, a pointer to per-thread information is kept in the `fs` register (for x86-32) or the `gs` register (for x86-64). If you disassemble through the kernel, you'll see that accesses to the per-thread information usually goes through two steps:

```
    mov     eax, dword ptr fs:[0x00000018]
    mov     eax, dword ptr [eax+n]
```

Why do it this way when you could combine it into one?

```
    mov     eax, dword ptr fs:[n]
```

Access to the per-thread data is abstracted into the helper function `NtCurrentTeb`, and the definition of that function changes based on the processor architecture.

```
// x86-32
return (struct _TEB *) (ULONG_PTR) __readfsdword (0x18);

// x86-64
return (struct _TEB *)__readgsqword(FIELD_OFFSET(NT_TIB, Self));
```

One option for optimizing access to per-thread data is to create a custom accessor for each thing you need.

```
inline DWORD GetLastErrorFromTEB()
{
#if defined(_M_IX86)
    return __readfsdword(FIELD_OFFSET(TEB, LastErrorCode));
#elif defined(_M_AMD64)
    return __readgsdword(FIELD_OFFSET(TEB, LastErrorCode));
#else
    ... other architectures here ...
#endif
}
```

This gets rather unwieldy as the number of fields in the `TEB` increases, since each one needs its own custom accessor. So maybe you abstract it into a macro.

```
#if defined(_M_IX86)
#define GetDwordFieldFromTEB(Field) \
    __readfsdword(FIELD_OFFSET(TEB, Field))
#define GetPointerFieldFromTEB(Field) \
    __readfsdword(FIELD_OFFSET(TEB, Field))
#elif defined(_M_AMD64)
#define GetDwordFieldFromTEB(Field) \
    __readgsdword(FIELD_OFFSET(TEB, Field))
#define GetPointerFieldFromTEB(Field) \
    __readgsqword(FIELD_OFFSET(TEB, Field))
#else
    ... other architectures here ...
#endif

DWORD GetLastError()
{
    return GetDwordFieldFromTEB(LastErrorCode);
}

PEB GetProcessEnvironmentBlock()
{
    return (PEB)GetPointerFieldFromTEB(Peb);
}
```

Another option is to teach the compiler's peephole optimizer that, if compiling in Windows ABI mode, it can fold the instruction sequence, provided the first `fs` access is to exactly the value `0x18`. This would be a very special compiler optimization that would kick in only for a limited audience. While it's true that the compiler team has been known to produce custom versions of the compiler for one-off situations, the savings for this particular micro-optimization would be, if you're lucky, a few thousands of instructions. That's a lot of work to save a few dozen kilobytes.

Furthermore, switching over could end up being worse: The offset field of an absolute selector-relative load is a 32-bit field, whereas the offset when applied to a general-purpose register can be made as small as eight bits. There is typically a penalty for accessing memory through a segment register whose base is not zero.[1] Furthermore, you have to pay for the prefix byte, and are probably taking the processor down an execution path that is not heavily optimized. If you are going to access per-thread data more than once in a function, you may very well have been better off just caching the result in a general-purpose register so you could use the smaller (and probably more efficient) flat addressing mode instead.

Even if all the calculations show that accessing the per-thread data from scratch is better than caching it in a general-purpose register (say, because the x86-32 is so register-starved that freeing up even one register is a big deal), you have to redo the cost/benefit calculations

for the other architectures.

```
// arm
return (struct _TEB *)(ULONG_PTR)_MoveFromCoprocessor(CP15_TPIDRURW);

// arm64
return (struct _TEB *)__getReg(18); // register x18

// alpha
return (struct _TEB *)_rdteb(); // PAL instruction

// ia64
return (struct _TEB *)_rdtebex(); // register r13

// MIPS
return (struct _TEB *)((PCR *)0x7ffff000)->Teb;

// PowerPC
return (struct _TEB *)__gregister_get(13); // register r13
```

These architectures break down into two categories: Those for which the offset from the TEB register can be folded into the instruction, and those for which it cannot.

Architecture	Can fold?	Notes
x86-32	Yes	`mov eax, fs:[n]`
x86-64	Yes	`mov rax, gs:[n]`
arm	No	Cannot use offset load from coprocessor register
arm64	Yes	`ldr x0, [x18, #n]`
alpha	No	Address returned by dedicated instruction
ia64	Yes	Can calculate effective address directly from r13
MIPS	No	No absolute addressing mode
PowerPC	Yes	`lwz r0, n(r13)`

For the architectures that don't support folding, there's no benefit to the optimization. And fetching the address from scratch is a pessimization on Alpha, since the call to get the address of per-thread data is a system call.

In order to benefit from this optimization on processors where it's helpful, without hurting the ones where it's harmful, you may end up having to write two versions of every function: One which gets the per-thread pointer from scratch every time, and one which caches it. The

compiler cannot do this transformation for you because it doesn't know whether the per-thread data can safely be cached: If there is a fiber switch, then the per-thread data cannot be cached since the fiber might be resuming on a different thread!

Given that this optimization may not actually be beneficial, and even if it is, the benefit is slight, and even that slight benefit comes at a cost to other architectures, you're probably better off not bothering with this micro-optimization and just writing portable code.

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

- Chapter 2.1 *Intel Microarchitecture Code Name Sandy Bridge*, Section 2.1.5.2 *L1 DCache*: "If segment base is not zero, load latency increases."
- Chapter 13 *Intel Atom Microarchitecture and Software Optimization*, Section 13.3.3.3 *Segment Base*: "Non-zero segment base will cause load and store operations to experience a delay."

Fog, Agner. The microarchitecture of Intel, AMD, and VIA CPUs, August 17, 2021.

> Chapter 19 *AMD K8 and K10 pipeline*: "The time it takes to calculate an address and read from that address in the level-1 cache is 3 clock cycles if the segment base is zero and 4 clock cycles if the segment base is nonzero, according to my measurements."

Raymond Chen