The history of calling conventions, part 1

devblogs.microsoft.com/oldnewthing/20040102-00

January 2, 2004

Raymond Chen

The great thing about calling conventions on the x86 platform is that there are so many to choose from! In the 16-bit world, part of the calling convention was fixed by the instruction set: The BP register defaults to the SS selector, whereas the other registers default to the DS selector. So the BP register was necessarily the register used for accessing stack-based parameters. The registers for return values were also chosen automatically by the instruction set. The AX register acted as the accumulator and therefore was the obvious choice for passing the return value. The 8086 instruction set also has special instructions which treat the DX:AX pair as a single 32-bit value, so that was the obvious choice to be the register pair used to return 32-bit values. That left SI, DI, BX and CX. (Terminology note: Registers that do not need to be preserved across a function call are often called "scratch".) When deciding which registers should be preserved by a calling convention, you need to balance the needs of the caller against the needs of the callee. The caller would prefer that all registers be preserved, since that removes the need for the caller to worry about saving/restoring the value across a call. The callee would prefer that no registers be preserved, since that removes the need to save the value on entry and restore it on exit. If you require too few registers to be preserved, then callers become filled with register save/restore code. But if you require too many registers to be preserved, then callees become obligated to save and restore registers that the caller might not have really cared about. This is particularly important for leaf functions (functions that do not call any other functions). The non-uniformity of the x86 instruction set was also a contributing factor. The CX register could not be used to access memory, so you wanted to have some register other than CX be scratch, so that a leaf function can at least access memory without having to preserve any registers. So BX was chosen to be scratch, leaving SI and DI as preserved.

So here's the rundown of 16-bit calling conventions:

All
All calling conventions in the 16-bit world preserve registers BP, SI, DI (others scratch) and put the return value in DX:AX or AX, as appropriate for size.

C (__cdecl)

Functions with a variable number of parameters constrain the C calling convention considerably. It pretty much requires that the stack be caller-cleaned and that the parameters be pushed right to left, so that the first parameter is at a fixed position relative to the top of the stack. The classic (pre-prototype) C language allowed you to call functions without telling the compiler what parameters the function requested, and it was common practice to pass the wrong number of parameters to a function if you "knew" that the called function wouldn't mind. (See "open" for a classic example of this. The third parameter is optional if the second parameter does not specify that a file should be created.)
In summary: Caller cleans the stack, parameters pushed right to left.

Function name decoration consists of a leading underscore. My guess is that the leading underscore prevented a function name from accidentally colliding with an assembler reserved word. (Imagine, for example, if you had a function called "call".)

Pascal (__pascal)
Pascal does not support functions with a variable number of parameters, so it can use the callee-clean convention. Parameters are pushed from left to right, because, well, it seemed the natural thing to do. Function name decoration consists of conversion to uppercase. This is necessary because Pascal is not a case-sensitive language.
Nearly all Win16 functions are exported as Pascal calling convention. The callee-clean convention saves three bytes at each call point, with a fixed overhead of two bytes per function. So if a function is called ten times, you save 3*10 = 30 bytes for the call points, and pay 2 bytes in the function itself, for a net savings of 28 bytes. It was also fractionally faster. On Win16, saving a few hundred bytes and a few cycles was a big deal.

Fortran (__fortran)
The Fortran calling convention is the same as the Pascal calling convention. It got a separate name probably because Fortran has strange pass-by-reference behavior.

Fastcall (__fastcall)
The Fastcall calling convention passes the first parameter in the DX register and the second in the CX register (I think). Whether this was actually faster depended on your call usage. It was generally faster since parameters passed in registers do not need to be spilled to the stack, then reloaded by the callee. On the other hand, if significant computation occurs between the computation of the first and second parameters, the caller has to spill it anyway. To add insult to injury, the called function often spilled the register into memory because it needed to spare the register for something else, which in the "significant computation between the first two parameters" case means that you get a double-spill. Ouch! Consequently, __fastcall was typically faster only for short leaf functions, and even then it might not be.

Okay, those are the 16-bit calling conventions I remember. Part 2 will discuss 32-bit calling conventions, if I ever get around to writing it.

Raymond Chen

Follow