Why does the usage of the initial registers of a Win32 process depend on whether it is a 32-bit or 64-bit process?

devblogs.microsoft.com/oldnewthing/20230321-00

March 21, 2023

Raymond Chen

Someone noticed that when you create a process suspended and snoop at its registers, the results vary depending on wither it is a 32-bit or 64-bit process.

For a 32-bit process, the initial register state puts something in `eax` and something else in `ebx`.

For a 64-bit process, the initial register state puts something in `rcx` and something else in `rdx`.

Why do 32-bit and 64-bit processes use different registers to pass the initial state? Either make the 32-bit initial state use `ecx` and `edx`, or make the 64-bit initial state use `rax` and `rbx`. This appears to be an intentional divergence. What's the reason for it?

First of all, note that all of what I'm writing here is internal implementation detail that can change at any time. I'm discussing it to satisfy your curiosity, not to provide information that you can rely on.

Okay, so back to the question. Why do 32-bit and 64-bit processes disagree?

Well, really, the question is "What makes you think they should agree?"

I sort of hid an assumption in the question. Did you spot it?

The customer is asking not about 32-bit and 64-bit Windows, but about the x86-32 and x86-64 processor architectures. The question is based on a limited understanding of the world of CPUs. "We got both kinds. We got x86-32 *and* x86-64!"

Windows has supported many 32-bit processor architectures, and I've covered many of them in the past: x86-32, Alpha AXP, MIPS III, PowerPC, SuperH-3, and ARM. It also has supported a number of 64-bit processor architectures, including Alpha AXP (using all 64 bits this time), Itanium, x86-64, and AArch64.

All of these architectures are different, and there's no *a priori* expectation that any two of them match up in register usage in any particular way.

Here's a comparison of calling conventions, with a lot of details omitted.

32-bit architectures	**x86-32**	**Alpha AXP**	**MIPS III**	**PowerPC**	**SuperH-3**	**ARM**
iarg1	[*esp*+4]	*a0*	*a0*	*r3*	*r4*	*a1*
iarg2	[*esp*+8]	*a1*	*a1*	*r4*	*r5*	*a2*
iarg3	[*esp*+12]	*a2*	*a2*	*r5*	*r6*	*a3*
iarg4	[*esp*+16]	*a3*	*a3*	*r6*	*r7*	*a4*
iarg5	[*esp*+20]	*a4*	*a4*	*r7*	@(16, *r15*)	[*sp*, #0]
iarg6	[*esp*+24]	*a5*	20(*sp*)	*r8*	@(20, *r15*)	[*sp*, #4]
iarg7	[*esp*+28]	0(*sp*)	24(*sp*)	*r9*	@(24, *r15*)	[*sp*, #8]
iarg8	[*esp*+32]	8(*sp*)	28(*sp*)	*r10*	@(28, *r15*)	[*sp*, #12]
iarg9	[*esp*+36]	16(*sp*)	32(*sp*)	32(*r1*)	@(32, *r15*)	[*sp*, #16]
fpargs	on stack	*f16…f21*	*f12…f15*	*f1…f13*	*fr4…fr7*	*d0…d7*
iret	*eax*, *edx*	*v0*	*v0*, *v1*	*r3*	*r0*	*a1*, *a2*
fpret	*st(0)*, *st(1)*	*f0*, *f1*	*f0*/*f1*, *f2*/*f3*	*f1*	*fr0*	*d0*, *d1*
home space?	no	no	yes	yes	yes	no
i/fp separate alloc	no	yes	no	yes	sort-of	yes
reuse partial fp regs	no	no	no	no	yes	yes
stack alignment	4	16	8	8	4	8
red zone	0	0	0	232	0	8

And for 64-bit architectures:

64-bit architectures	**Alpha AXP**	**Itanium**	**x86-64**	**AArch64**
iarg1	*a0*	*r32*	*rcx*	*x0*
iarg2	*a1*	*r33*	*rdx*	*x1*
iarg3	*a2*	*r34*	*r8*	*x2*
iarg4	*a4*	*r35*	*r9*	*x3*
iarg5	*a4*	*r36*	[*rsp*+32]	*x4*
iarg6	*a5*	*r37*	[*rsp*+40]	*x5*
iarg7	0(*sp*)	*r38*	[*rsp*+48]	*x6*
iarg8	8(*sp*)	*r39*	[*rsp*+56]	*x7*
iarg9	16(*sp*)	[*sp*]	[*rsp*+64]	[*sp*, #0]
fpargs	*f16*…*f21*	*f32*…*f39*	*xmm0*…*xmm3*	*v0*…*v7*
iret	*v0*	*ret0*…*ret3*	*rax*	*x0*
fpret	*f0*, *f1*	*f8*	*xmm0*	*d0*, *d1*
home space?	no	no	yes	no
i/fp separate alloc	yes	yes	no	yes
reuse partial fp regs	no	no	no	no
stack alignment	16	16	16	16
red zone	0	−16	0	16

Certainly you don't expect all of these process to agree on what registers to use. They don't even all have the same registers to begin with!

Okay, so maybe the question is "Yes, I know that Windows supports more than just x86-32 and x86-64, but the two architectures are clearly descended from each other, so why are things so different between them?"

Well, why should they be the same? After all, x86-32 descended from 8086, but it's not like we still using the 8086 calling convention in x86-64 code. With a newer processor, we can take advantage of newer features, and that means we can re-optimize the calling convention to take advantage of them: Use the SSE registers for floating point instead of the legacy *st(n)* registers. Use compile-time exception handling tables instead of run-time stack threading. Increase the stack alignment requirements to be SSE-friendly. Pass parameters in registers rather than on the stack, now that we are no longer under severe register pressure.

We also see changes when moving from 32-bit ARM to 64-bit AArch64: The number of register-based parameters increases from four to eight, the partial floating point register backfill was dropped, the stack alignment became stricter, and the red zone expanded.

I mean, clearly you have to change *something* because the 64-bit registers are bigger than the 32-bit registers. At a minimum, you'll have to expand the register sizes. And once you decide to expand the register sizes, you've committed to the cost of change, so you may as well get your money's worth.

One thing you might notice is that the 32-bit and 64-bit Alpha AXP calling conventions are identical. What happened here? Was the 32-bit calling convention so perfect that nothing had to be improved for the 64-bit convention? What about the whole "expanding registers from 32-bit to 64-bit requires a change at least for the new register sizes"?

Recall that the Alpha AXP was always a 64-bit processor. There was no 32-bit Alpha AXP processor. The "32-bit" Alpha AXP calling convention was developed with a 64-bit processor in hand, just with the understanding that pointers are only 32 bits in size, for now. (Though you could ask for memory in the parts of the address space that require the use of 64-bit pointers. It would then be on you to figure out how to cajole the compiler into using 64-bit pointers.)

When the 32-bit ABI for Alpha AXP was invented, the processor already had 64-bit registers and full support for 64-bit operations. It's just that 32-bit Windows voluntarily restricted itself to 32 bits of address space. When designing the calling convention, the ABI designers made parameters 64-bit values, even if only the lower 32 bits were significant in practice. Everything was carefully designed so that the 32-bit calling convention could be repurposed as a 64-bit calling convention without any changes.[1] (This came in handy when the Alpha AXP was used as a proof-of-concept hardware platform for 64-bit Windows, since it avoided having to change large portions of the compiler.) In other words, the 32-bit ABI for Alpha AXP was invented with the power of clairvoyance: They knew what the 64-bit process was going to look like, and they could design the 32-bit ABI to be identical to the 64-bit one.

One thing you may notice is that all of the calling conventions pass parameters in registers except for one: x86-32. Once again, the x86 is the weirdo. The internal kernel infrastructure for creating a process lets you specify an initial register state and an initial instruction pointer, but it doesn't let you describe the contents of the stack. This means that all parameters must be passed in registers. This is straightforward for the register-based calling conventions, since they can just put the parameters directly in the initial register state. But for x86-32, that doesn't work.

What happens on x86-32 is that the kernel puts the parameters in some dummy registers, and then sets the initial instruction pointer not to the start of the process but rather to a helper function written in assembly language that takes those values from registers, pushes them onto the stack (to conform with the x86-32 calling convention), and then calls the *real* process start function.

The registers to use for this "custom calling convention" are completely arbitrary, and the kernel folks chose *eax* and *ebx* out of alphabetical convenience.[2] This choice was made several decades before the x86-64 convention was invented, so there was nothing to be compatible with.

So that's the reason why the x86-32 and x86-64 architectures disagree on how to pass the initial parameters to the process. There was no reason why they had to agree in the first place. The 32-bit version picked two registers arbitrarily, and those didn't happen to correspond in an attractive way with the x86-64 calling convention that would come later.[3]

[1] In theory, this would even let 32-bit and 64-bit Alpha AXP code coexist within a process, since they could just call into each other without having to do any calling convention thunking. The 64-bit code would have to be careful to pass only pointers to memory addressible with 32-bit pointers.

[2] What the kernel folks could have done was declare the process start function as using the `__fastcall` calling convention, which takes the first two parameters in `ecx` and `edx`. That would have avoided having to write the little helper function.

[3] I guess you could turn the question around and ask "Why doesn't the x86-64 calling convention use `rax` and `rbx` for the first two register parameters, so it would align in an attractive manner with the custom calling convention used by this one specific corner of the kernel." That was a one-off dark corner of the kernel that uses a custom calling convention that only a handful of people even know about, so there's no reason that the people who designed the x86-64 calling convention even knew about it, much less possessed any desire to align with it in an attractive manner. And even if they knew about it, there's really no need to accommodate it when designing a calling convention for general-purpose computing. There are probably quite a few one-off custom calling conventions scattered around the system. The one used by the kernel for starting new processes isn't particularly prominent. (Indeed, it's so deeply buried that it's probably one of the most *obscure* ones.)