Data breakpoints are based on the linear address, not the physical address

devblogs.microsoft.com/oldnewthing/20080509-00

May 9, 2008

Raymond Chen

When you ask the debugger to set a read or write breakpoint, the breakpoint fires only if the address is read from or written to by the address you specify. If the memory is mapped to another address and modified at that other address, then your breakpoint won't see it. For example, if you have multiple views on the same data, then modifications to that data via alternate addresses will not trigger the breakpoint. The hardware breakpoint status is part of the processor context, which is maintained on a per-thread basis. Each thread maintains its own virtualized hardware breakpoint status. You don't notice this in practice because debuggers are kind enough to replicate the breakpoint state across all threads in a process so that the breakpoint fires regardless of which thread triggers it. But that replication typically doesn't extend beyond the process you're debugging; the debugger doesn't bother replicating your breakpoints into other processes! This means that if you set a write breakpoint on a block of shared memory, and the write occurs in some other process, your breakpoint won't fire since it's not your process that wrote to it. When you call into kernel mode, there is another context switch, this time between user mode and kernel mode, and the kernel mode context of course doesn't have your data breakpoint. Which is a good thing, because if that data breakpoint fired in kernel mode, how is your user-mode debugger expected to be able to make any sense of it? The breakpoint fired when executing code that user mode doesn't have permission to access, and it may have fired while the kernel mode code owned an important critical section or spinlock, a critical section the debugger itself may very well need. Imagine if the memory were accessed by the keyboard driver. Oops, now your keyboard processing has been suspended. Even worse, what if the memory were accessed by a a hardware interrupt handler? Hardware interrupt handlers can't even access paged memory, much less allow user-mode code to run. This "program being debugged takes a lock that the debugger itself needs" issue isn't usually a problem when a user-mode debugger debugs a user-mode process, because the locks held by a user-mode process typically affect only that process. If a process takes a critical section, sure that may deadlock the process, but the debugger is not part of the process, so it doesn't care. Of course, the "debugger is its own world" principle falls apart if the debugger is foolish enough to require a lock that the program being debugged also uses. Debugger authors therefore must be careful to avoid these sorts of cross-process dependencies. (News flash: Writing a debugger is hard.) You can still run into trouble if the program being debugged has done something with global consequences like create a fullscreen topmost window (thereby covering the debugger) or installed a global keyboard hook (thereby interfering with typing). If you've tried debugging a system service, you may have run into this sort of cross-process deadlock. For example, if you debug the service that is responsible for the networking client, and the debugger tries to access the network (for example, to load symbols), you've created a deadlock since the debugger needs to access the network, which it can't do because the networking service is stopped in the debugger. Hardware debugging breakpoints are a very convenient tool for chasing down bugs, but you have to understand their limitations.

Additional reading: Data breakpoint oddities.

Raymond Chen

Follow