Nynaeve

Adventures in Windows debugging and reverse engineering.

« Don't mind the conflicting advice…
Debugger flow control: More on breakpoints (part 2) »

Debugger flow control: Hardware breakpoints vs software breakpoints

In debugging parlance, there are two kinds of breakpoints that you may run across – "hardware" breakpoints, and "software breakpoints". While the two overlap to a certain degree, it is important to know the differences between the two, and when it is better to use a "hardware" or "software" breakpoint.

For the purposes of this discussion, I'll stick to using WinDbg on an x86 target. The same general concepts apply to other architectures (especially x64, which works near identically), and the commands to set breakpoints are the same, but details such as where and how many hardware or software breakpoints you may set slightly vary from platform to platform.

In most debugging scenarios, you have probably just used software breakpoints exclusively. Software breakpoints are issued by the *bp* or *bu* commands (breakpoint and deferred breakpoint, respectively). These breakpoints are fairly simple and straightforward; they cause the processor to halt in the debugger whenever a thread attempts to execute a piece of code that you set a breakpoint on. Typically, you may set any number of software breakpoints that you want at the same time. Software breakpoints may only be targetted at code; there is no support for setting a "memory breakpoint" via a software breakpoint. Many features such as stepping over a call or going to the return address of a function also implicitly use a temporary software breakpoint that is removed once execution hits it the first time.

Hardware breakpoints, on the other hand, are much more powerful and flexible than software breakpoints. Unlike software breakpoints, you may use hardware breakpoints to set "memory breakpoints", or a breakpoint that is fired when any instruction attempts to read, write, or execute (depending on how you configure the breakpoint) a specific address. (There is also support for setting breakpoints on I/O port access, but I'll not cover that feature here, as it is typically of very limited applicibility for every-day debugging tasks.) Hardware breakpoints have some limitations, however; the main limit being that the number of hardware breakpoints that you may have active is extremely limited (on x86, you may only have four hardware breakpoints active at the same time).

Now that we have a basic overview of what the two breakpoint types are, let's dig a bit deeper and see how they work under the hood, and when you might use them.

The way software breakpoints work is fairly simple. Speaking about x86 specifically, to set a software breakpoint, the debugger simply writes an *int 3* instruction (opcode 0xCC) over the first byte of the target instruction. This causes an interrupt 3 to be fired whenever execution is transferred to the address you set a breakpoint on. When this happens, the debugger "breaks in" and swaps the 0xCC opcode byte with the original first byte of the instruction when you set the breakpoint, so that you can continue execution without hitting the same breakpoint immediately. There is actually a bit more magic involved that allows you to continue execution from a breakpoint and not hit it immediately, but keep the breakpoint active for future use; I'll discuss this in a future posting.

Now, you might be tempted to say that this isn't really how software breakpoints work, if you have ever tried to disassemble or dump the raw opcode bytes of anything that you have set a breakpoint on, because if you do that,

you'll not see an *int 3* anywhere where you set a breakpoint. This is actually because the debugger tells a lie to you about the contents of memory where software breakpoints are involved; any access to that memory (through the debugger) behaves as if the original opcode byte that the debugger saved away was still there.

Now that we know how software breakpoints work at a high level, it's time to talk about the other side of the story, hardware breakpoints.

Hardware breakpoints are, as you might imagine given the name, set with special hardware support. In particular, for x86, this involves a special set of perhaps little-known registers know as the "Dr" registers (for debug register). These registers allow you to set up to four (for x86, this is highly platform specific) addresses that, when either read, read/written, or executed, will cause the processor to throw a special exception that causes execution to stop and control to be transferred to the debugger.

Given that on x86, you can only have four hardware breakpoints active at once, why would anyone possibly want to use them?

Well, the main strength of hardware breakpoints is that you can use them to halt on non-execution accesses to memory locations. This is actually an extremely useful capability; for example, if you were debugging a memory corruption problem where an initial instance of corruption eventually causes a crash, your initial reaction would probably be something on the lines of "gee, if I know who caused the corruption in the first place, this would be much, much easier to debug" – and this is exactly what hardware breakpoints let you do. In essence, you can use a hardware breakpoint to tell the processor to stop when a specific variable (address) is read or read/written to. You can also use hardware breakpoints to break in on code execution as well, although in the typical case, it is more common to use software breakpoints for that purpose due to the relaxed restrictions on how many breakpoints you may have active at once.

That's the high level overview of the two main types of breakpoints you'll encounter in a debugger. In some upcoming postings, I'll go into some specifics as to how certain edge cases (such as stepping over a call) are implemented, and describe other situations where you'll find it very useful to use one kind of breakpoint instead of another. I am also planning on discussing how some of the other debugger flow control features are really implemented (such as tracing / single step), and what the consequences of using each flow control method are on the debuggee.

This entry was posted on Tuesday, November 7th, 2006 at 11:32 am and is filed under Debugging, Windows. You can follow any responses to this entry through the RSS 2.0 feed. Both comments and pings are currently closed.

7 Responses to "Debugger flow control: Hardware breakpoints vs software breakpoints"

1. *Andrew* says:
December 15, 2006 at 11:58 am

   I just want to encourage you to continue posting articles like these – I found it very informative.

2. *Skywing* says:
December 15, 2006 at 12:06 pm

   Thanks! I am planning on continuing this series in the near future, as time permits.

3. *Marc* says:
September 12, 2007 at 4:48 pm

   Very helpful. Thanks!

4. *P. Sridharan* says:
August 7, 2008 at 8:07 am

Very useful.

5. *IvanChen* says:
October 20, 2008 at 8:45 pm

Thanks, :)
yesterday I got an error message came from the cdb, when I set up a data breakpoint on some addresee , at the initial breakpoint. And the error message lead me here

Thanks

6. *stephen* says:
November 5, 2008 at 5:56 pm

Here is a challenge for you…if you are writing a userspace debugger in Windows, how do you actually *set* a hardware breakpoint without modifying DR0 registers manually when you catch each CreateThread debug event????
if you can answer this I will be your best friend.

7. *Raju* says:
July 29, 2010 at 1:54 am

Informative and very well explained.
Thanks
