How to view the stack of threads that were terminated as part of process teardown from the kernel debugger

devblogs.microsoft.com/oldnewthing/20120517-00

May 17, 2012

Raymond Chen

As we saw some time ago, process shutdown is a multi-phase affair. After you call `ExitProcess`, all the threads are forcibly terminated. After that's done, each DLL is sent a `DLL_PROCESS_DETACH` notification. You may be debugging a problem with `DLL_PROCESS_DETACH` handling that suggests that some of those threads were not cleaned up properly. For example, you might assert that a reference count is zero, and you find during process shutdown that this assertion sometimes fires. Maybe you terminated a thread before it got a chance to release its reference? How can you test this theory if the thread is already gone?

It so happens that when all the threads are terminated during the early phase of process shutdown, the kernel is a bit lazy and doesn't free their stacks. It figures, hey, the entire process is going away soon, so the stack memory is going to be cleaned up as part of process termination. (It's sort of the kernel equivalent of not bothering to sweep the floor of a building that's about to be demolished.) You can use this to your advantage by *grovelling the stacks that were left behind*.

Hey, this is why you get called in to debug the hard stuff, right?

> Before continuing, I need to emphasize that this information is **for debugging purposes only**. The structures and offsets are all implementation details which can change from release to release.

The first step is to identify where all the stacks are. The direct approach is difficult because the stacks can be all different sizes, so it's not easy to pick them out of a line-up. But one thing does come in a consistent size: The TEB.

From the kernel debugger, use the `!process` command to dump the process you are interested in, and from the header information, extract the `VadRoot`.

```
1: kd> !process -1
PROCESS 8731bd40  SessionId: 1  Cid: 0748    Peb: 7ffda000  ParentCid: 0620
    DirBase: 4247b000  ObjectTable: 96f66de0  HandleCount: 104.
    Image: oopsie.exe
    VadRoot 893de570 Vads 124 Clone 0 Private 518. Modified 643. Locked 0.
    DeviceMap 995628c0
```

Dump this VAD root with the `!vad` command, and pay attention only to the entries which say `1 Private READWRITE`.

```
1: kd> !vad 893de570
VAD     level      start      end    commit
... ignore everything except "1 Private READWRITE" ...
8730a5f0 ( 6)         50       50          1 Private      READWRITE
9ab0cb40 ( 5)         60       7f          1 Private      READWRITE
893978b0 ( 6)         80       9f          1 Private      READWRITE
87302d30 ( 5)        110      110          1 Private      READWRITE
889693f8 ( 6)        120      121          1 Private      READWRITE
872f3fb8 ( 6)        170      170          1 Private      READWRITE
87089a80 ( 6)        1a0      1a0          1 Private      READWRITE
8cbf1cb0 ( 5)        1c0      1df          1 Private      READWRITE
88c079d0 ( 6)        1e0      1e0          1 Private      READWRITE
9abc33e0 ( 6)        410      48f          1 Private      READWRITE
873173b0 ( 7)        970      970          1 Private      READWRITE
8ca1c158 ( 7)      7ffd5    7ffd5          1 Private      READWRITE
88c02a78 ( 6)      7ffd6    7ffd6          1 Private      READWRITE
872f9298 ( 5)      7ffd7    7ffd7          1 Private      READWRITE
8750d210 ( 7)      7ffd8    7ffd8          1 Private      READWRITE
87075ce8 ( 6)      7ffda    7ffda          1 Private      READWRITE
87215da0 ( 4)      7ffdc    7ffdc          1 Private      READWRITE
872f2200 ( 6)      7ffdd    7ffdd          1 Private      READWRITE
8730a670 ( 5)      7ffdf    7ffdf          1 Private      READWRITE
```

(If you are debugging from user mode, then you can use `!vadump` but the output format is different.)

Each of these is a candidate TEB. In practice, TEBs tend to be allocated at the high end of memory, so the ones with a low `start` value are probably red herrings. Therefore, you should investigate these candidates in reverse order.

For each candidate, take the `start` address and append three zeroes. (Each page on x86 is 4KB, which conveniently maps to 1000 in hex.) Dump the first seven pointers of the TEB with the `dp xxxxx000 L7` command.

```
1: kd> dp 7ffdf000 L7
7ffdf000  0016fbb0 00170000 0016b000 00000000
7ffdf010  00001e00 00000000 7ffdf000 ← hit
```

If the TEB is valid, then the seventh pointer points back to the start of the TEB. In a valid TEB, the second and third values are the stack limits; in this case, the candidate stack lives between `0016b000` and `00170000`. (As a double-check, you can verify that the upper limit of the stack, `00170000` in this case, matches up with the end of a VAD allocation in the `!vad` output above.)

Now that you know where the stack is, you can `dps` it and look for EBP frames. (I usually start about two to four pages below the upper limit of the stack.) Test out each candidate EBP frame with the `k=` command until you find one that seems to be solid. Record this candidate stack trace in a text file for further study.

Repeat for each candidate TEB, and you will eventually reconstruct what each thread in the process was doing at the moment it was terminated. If you're really lucky, you might even see the code that incremented the reference count but was terminated before it could release it.

The above discussion also applies to debugging 64-bit processes. However, instead of looking for `1 Private READWRITE` pages, you want to look for `2 Private READWRITE` pages. As an additional wrinkle, if you are debugging ia64, then converting a page frame to a linear address is sadly not as simple as appending three zeroes. Pages on ia64 are 8KB, not 4KB, so you need to shift the value left by 25 bits: Add three zeroes and then multiply by two.

And finally, if you are debugging a 32-bit process on x64, then you want to look for `3 Private READWRITE` pages, but add 2 before appending the three zeroes. That's because the TEB for a 32-bit process on x64 is really two TEBs glued together: A 64-bit TEB followed by a 32-bit TEB.

Note: I did not come up with this debugging technique on my own. I learned it from an even greater debugging genius.

Next time, we'll look at debugging this issue from a user-mode debugger.

Trivia: The informal term for these terminated-but-not-yet-completely-destroyed threads is *ghost threads*. The term was coined by the Exchange support team, because they often have to study server failures that require them to do this type of investigation, and they needed a cute name for it.

Raymond Chen

Follow