The case of the stack overflow exception when the stack is nowhere near overflowing

devblogs.microsoft.com/oldnewthing/20220204-00

February 4, 2022

Raymond Chen

A customer was confused by crash dumps that showed their program crashing with a stack overflow exception even though inspection of the stack showed that the program hadn't used anywhere near 1MB of stack. One megabyte is the default stack size, so why did the program encounter a stack overflow when there was still plenty of room?

One thing to check is whether the stack really is one megabyte in size. You can do this by inspecting the stack limits in the TEB.[1] The default stack size is determined by the value of `SizeOfStackReserve` in the PE header, and if you don't customize it, the linker defaults to one megabyte. You can also pass a custom stack size to the `CreateThread` to create a thread with a specific stack size. You can read more details on docs.microsoft.com.

In this case, the program did not create any threads with custom stack sizes, and the default stack reserve was still at the default of one megabyte. So it's not the case that the program was running on an artificially small stack.

Another possibility is that the system ran out of memory. Even though a megabyte of memory is reserved for the stack, the memory is allocated only on demand, as we learned last time when we took a closer look at the stack guard page. If the system cannot allocate memory to replace the guard page, then you get a stack overflow exception.

The customer used the `.dumpdebug` command in the debugger to look at the memory usage and other statistics that were captured when the dump was created. Their executable ran as part of a job object, so the numbers to look at are the *JobPrivateCommitUsage*, *JobPeak-PrivateCommitUsage*, and the *JobPrivateCommitLmit*. These numbers showed that the commit usage and peak commit usage were nowhere near the commit limit, so the issue is not that the process ran out of memory.

Maybe the problem is that a rogue pointer falsely triggered the guard page, causing it to turn into a regular committed page and leaving the thread with no guard page. We can investigate this possibility with the `!address` debugger command:

```
        BaseAddress      EndAddress+1        RegionSize     Type       State
Protect             Usage
-------------------------------------------------------------------------------
-------------------------------------
+        0`7ffe7000       7f`fa4d0000       7f`7a4e9000             MEM_FREE
PAGE_NOACCESS                      Free
+       7f`fa4d0000       7f`fa53c000        0`0006c000 MEM_PRIVATE MEM_RESERVE
Stack      [~0; 9024.9308]
        7f`fa53c000       7f`fa53f000        0`00003000 MEM_PRIVATE MEM_COMMIT
PAGE_READWRITE|PAGE_GUARD          Stack      [~0; 9024.9308]
        7f`fa53f000       7f`fa550000        0`00011000 MEM_PRIVATE MEM_COMMIT
PAGE_READWRITE                     Stack      [~0; 9024.9308]
+       7f`fa550000       7f`fa5bc000        0`0006c000 MEM_PRIVATE MEM_RESERVE
Stack      [~1; 9024.8070]
        7f`fa5bc000       7f`fa5bf000        0`00003000 MEM_PRIVATE MEM_COMMIT
PAGE_READWRITE|PAGE_GUARD          Stack      [~1; 9024.8070]
        7f`fa5bf000       7f`fa5d0000        0`00011000 MEM_PRIVATE MEM_COMMIT
PAGE_READWRITE                     Stack      [~1; 9024.8070]
+       7f`fa5d0000       7f`fa600000        0`00030000             MEM_FREE
PAGE_NOACCESS                      Free
...
```

The unlabeled first column of the output is a + if this region is the start of an allocation.

Note that this listing is upside-down relative to the way we traditionally draw memory maps. Memory maps are traditionally drawn with higher addresses at the top, but the `!address` command lists the addresses in increasing order from top to bottom.

What we see here are two stacks from the process. The normal pattern for a stack is a single allocation that consists of three parts:

- A bunch of reserved memory, representing stack space that has yet to be used.
- A guard region (`PAGE_GUARD`) that is used to detect stack growth.
- A bunch of committed memory, representing stack memory already allocated.

We see that both of the above stacks fit this pattern:

- The region at `7f`fa4d0000` is the start of an allocation, and it is marked `MEM_RESERVE` .
- The next region consists of three `PAGE_GUARD` pages, representing the guard region.
- Finally, we have a bunch of regular `PAGE_READWRITE` pages, representing the stack in use.

One thing I just learned from looking at this is that the guard page is no longer a single page, like it was in the early days of Windows NT, but rather is a block of three pages. My guess is that this was done to reduce the number of guard page exceptions.

If the leading `MEM_RESERVE` region is missing, then what happened is that you really did run out of stack. You used up your entire stack allocation.

If the middle `PAGE_GUARD` region is missing, then what happened is that you lost the guard region, perhaps due to a rogue pointer from another thread.

In the customer's crash dump, the stacks for all threads looked good, except that the thread which encountered the stack overflow itself did not have a stack guard region, even though it still had a generous reserve. This would normally indicate that the guard region was lost. However, it is expected to see this situation when a stack overflow is reported: The stack overflow exception is raised as part of handling the guard page exception if a new guard region cannot be allocated. You're getting the exception either because the system cannot create a new guard region, or because the old guard region got corrupted.

The customer did some more investigation on their side and discovered that even though their process was running in a job with plenty of remaining memory, that job was itself running inside *another job* that also had its own memory cap. Other sibling jobs had consumed all the memory allotted to the parent job, causing their process to crash due to the inability to allocate a new guard region.

So the initial theory was correct after all: The process ran out of memory. It just was due to external factors that weren't captured in the crash dump.

[1] A useful command is `knf` [2] which asks for an attributed stack trace, where each frame is annotated by its size in bytes. You can use this to find functions that have a lot of local variables that are causing excessive stack consumption.

[2] An early version of the `knf` command went by the name `stackpig` because it dumped a stack trace, annotating the size of the frame, and highlighting the large ones with the word `OINK`.

Raymond Chen

Follow