How to view the stack of a user-mode thread when its kernel stack has been paged out

devblogs.microsoft.com/oldnewthing/20141114-00

November 14, 2014

Raymond Chen

Suppose you have a machine that has crashed, and your investigation shows that the reason is that there is a critical section that everybody is waiting for. While waiting for that critical section, work piles up, and eventually the machine keels over. Suppose further that this crash is given to you in the form of a kernel debugger.

In case it wasn't obvious, by "you" I mean "me".

Okay, so the critical section that is the cause of the logjam is this one:

```
1: kd> !cs CONTOSO!g_csDataLock
-----------------------------------------
Critical section   = 0x00007ff7f0ed2f68 (CONTOSO!g_csDataLock+0x0)
DebugInfo          = 0x0000000022f2efd0
LOCKED
LockCount          = 0x5D
WaiterWoken        = No
OwningThread       = 0x0000000000004228
RecursionCount     = 0x1
LockSemaphore      = 0x17A0
SpinCount          = 0x00000000020007cb
```

"Great," you say. "I just need to look at thread 0x4228 to see why it is stuck.

```
1: kd> !process -1 4
PROCESS ffffe000047ae900
    SessionId: 1  Cid: 0604    Peb: 7ff74ecfa000  ParentCid: 05cc
    DirBase: 0eb07000  ObjectTable: ffffc000014c5680  HandleCount: 7003.
    Image: contoso.exe
        ...
        THREAD ffffe0000c136080  Cid 0604.4228  Teb: 00007ff74e94c000 Win32Thread:
fffff90144edea60 WAIT
        ...
```

Woo-hoo, there's the thread. Now I just need to switch to its context to see what it is stuck on.

```
1: kd> .thread ffffe0000c136080
Can't retrieve thread context, Win32 error 0n30
```

Okay, that didn't work out too well. Now what?

Even though the kernel stack is paged out, the user-mode stack may still be available.

```
1: kd> !thread ffffe0000c136080
THREAD ffffe0000c136080  Cid 0604.4228  Teb: 00007ff74e94c000
       Win32Thread: fffff90144edea60 WAIT: (UserRequest) UserMode Non-Alertable
    ffffe000077a7830  NotificationEvent
Not impersonating
DeviceMap                 ffffc00000e89c80
Owning Process            ffffe000047ae900       Image:         contoso.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      12735890       Ticks: 328715 (0:01:25:36.171)
Context Switch Count      75             IdealProcessor: 2
UserTime                  00:00:00.000
KernelTime                00:00:00.031
Kernel stack not resident.
```

The limits of the user-mode stack are kept in the Teb.

```
2: kd> !teb 00007ff74e94c000
TEB at 00007ff74e94c000
    ExceptionList:        0000000000000000
    StackBase:            00000001027d0000
    StackLimit:           00000001027c2000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 00007ff74e94c000
    EnvironmentPointer:   0000000000000000
    ClientId:             0000000000000604 . 0000000000004228
    RpcHandle:            0000000000000000
    Tls Storage:          000000010132dfc0
    PEB Address:          00007ff74ecfa000
    LastErrorValue:       1008
    LastStatusValue:      c0000034
    Count Owned Locks:    2
    HardErrorMode:        0
```

We now use the trick we learned some time ago where we grovel the stack of a thread without knowing where its stack pointer is.

In this case, the groveling is made easier because we already know that everybody is waiting on the data lock. The data lock is taken in only two functions, so it's a matter ot looking for any occurrences of one of those two functions. And here it is: `DataWrapper::VerifyData`.

```
00000001`027cef88  00007ffe`f1bbac9a NTDLL!NtWaitForSingleObject+0xa
00000001`027cef90  00000000`00006ac8
00000001`027cef98  00007ffe`ef1bd085 KERNELBASE!WaitForSingleObjectEx+0xa5
00000001`027cefa0  00000000`00006ac8
00000001`027cefa8  00000000`00000000
00000001`027cefb0  00000000`026d0000
00000001`027cefb8  00000000`00000000
00000001`027cefc0  00000001`01739ee0
00000001`027cefc8  00000000`00000001
00000001`027cefd0  00000000`00000048
00000001`027cefd8  00000001`00000001
00000001`027cefe0  00000000`00000000
00000001`027cefe8  00000000`00000000
00000001`027ceff0  00000000`00000000
00000001`027ceff8  00000000`00000000
00000001`027cf000  00000000`00000000
00000001`027cf008  00000000`00000000
00000001`027cf010  00000000`00000000
00000001`027cf018  00007ff7`eefe6540 FABRIKAM!Lock::IsInitialized+0xfc
00000001`027cf020  00000000`00000000
00000001`027cf028  00000000`3b803fa0
00000001`027cf030  00000000`00006ac8
00000001`027cf038  00007ff7`eeff3c43 FABRIKAM!AccessRequest::WaitTimeout+0x87
00000001`027cf040  00000000`ffffffff
00000001`027cf048  10f513ec`6a161fb8
00000001`027cf050  00000000`00000000
00000001`027cf058  00000000`00006ac8
00000001`027cf060  00000000`3b803fb8
00000001`027cf068  00007ff7`eeff3cc0 FABRIKAM!AccessRequest::Wait+0x18
00000001`027cf070  00000000`8007000e
00000001`027cf078  00000000`fd416ff0
00000001`027cf080  4fe80c4a`51236583
00000001`027cf088  00000000`3b803fb8
00000001`027cf090  00000000`ffffffff
00000001`027cf098  00000000`00000000
00000001`027cf0a0  00000000`3b803fa0
00000001`027cf0a8  00007ff7`da8375b7 FABRIKAM!DataAccess::RequestAccess+0x93
00000001`027cf0b0  00000000`3b803fa0
00000001`027cf0b8  00000000`3b803fb8
00000001`027cf0c0  4fe80c4a`51236583
00000001`027cf0c8  10f513ec`6a161fb8
00000001`027cf0d0  00000000`00000076
00000001`027cf0d8  00007ff7`f07a8619 CONTOSO!Widget::SetColor+0x9
00000001`027cf0e0  000095dc`90897985
00000001`027cf0e8  00000000`00000110
00000001`027cf0f0  00000000`00000000
00000001`027cf0f8  00000000`00000000
00000001`027cf100  00007ff7`4dd8000c
00000001`027cf108  00007ff7`f07a85e5 CONTOSO!Widget::UpdateColor+0x39
00000001`027cf110  00000000`ffc03f90
00000001`027cf118  00000001`027cf1e8
00000001`027cf120  00000000`ffc03fc8
```

```
00000001`027cf128  00000000`00000000
00000001`027cf130  00000000`00000000
00000001`027cf138  00007ff7`4dd8000c
00000001`027cf140  00007ff7`da5bc420 FABRIKAM!DataAccess::`vftable'+0x18
00000001`027cf148  00000000`ffc03f90
00000001`027cf150  00000001`027cf260
00000001`027cf158  00007ff7`f0b3459c CONTOSO!DataWrapper::VerifyData+0x428
00000001`027cf160  00000000`3b803fa0
00000001`027cf168  00007ff7`f0ed1d30 CONTOSO!g_DataManager
00000001`027cf170  00000000`fe196f10
00000001`027cf188  00000000`00000001
00000001`027cf190  00000000`00000000
00000001`027cf198  00000001`027cf270
00000001`027cf1a0  00000001`027cf480
00000001`027cf1a8  00007ff7`00000001
00000001`027cf1b0  00000001`027cf220
00000001`027cf1b8  00000000`00000000
00000001`027cf1c0  00000000`00000000
```

I left the red herrings in place just to make things a little more interesting.

The `DataWrapper::VerifyData` method enters the critical section and then calls `DataAccess::RequestAccess` via a virtual method call:

```
00007ff7`f0b3458f mov     dword ptr [rsp+28h],eax
00007ff7`f0b34593 mov     eax,dword ptr [rsi+10h]
00007ff7`f0b34596 mov     dword ptr [rsp+20h],eax
00007ff7`f0b3459a call    qword ptr [rdi] ←
```

Let's disassemble the start of `DataAccess::RequestAccess` to see how it sets up its stack. This will help us interpret the other values in the stack dump.

```
0: kd> u 00007ff7`da8375b7-93
FABRIKAM!DataAccess::RequestAccess
;; function prologue
00007ff7`da837524 mov     rax,rsp
00007ff7`da837527 mov     qword ptr [rax+8],rbx
00007ff7`da83752b mov     qword ptr [rax+20h],r9
00007ff7`da83752f mov     qword ptr [rax+18h],r8
00007ff7`da837533 mov     qword ptr [rax+10h],rdx
00007ff7`da837537 push    rbp
00007ff7`da837538 push    rsi
00007ff7`da837539 push    rdi
00007ff7`da83753a push    r12
00007ff7`da83753c push    r13
00007ff7`da83753e push    r14
00007ff7`da837540 push    r15
00007ff7`da837542 sub     rsp,70h
;; function body
00007ff7`da837546 lea     rsi,[rcx+18h]
00007ff7`da83754a mov     rdi,rcx
00007ff7`da83754d mov     rbp,r9
00007ff7`da837550 mov     rcx,rsi
00007ff7`da837553 call    qword ptr [FABRIKAM!_imp_EnterCriticalSection]
00007ff7`da837559 xor     r13b,r13b
00007ff7`da83755c mov     ebx,8007000Eh
...
00007ffe`da8375b1 call    FABRIKAM!AccessRequest::Wait
```

We can replay the above code in our head and annotate the stack trace accordingly. On entry to the function, the stack pointer is `00000001`027cf158` (the return address). The function stashes some registers in the caller-provided spill area and it pushes others onto the stack, and then it subtracts some space for local variables as well as for outbound parameters of functions it intends to call.

```
/-00000001`027cf0b0  00000000`3b803fa0
| 00000001`027cf0b8  00000000`3b803fb8
| 00000001`027cf0c0  4fe80c4a`51236583
| 00000001`027cf0c8  10f513ec`6a161fb8
| 00000001`027cf0d0  00000000`00000076
| 00000001`027cf0d8  00007ff7`f07a8619 CONTOSO!Widget::SetColor+0x9
| 00000001`027cf0e0  000095dc`90897985
| 00000001`027cf0e8  00000000`00000110
| 00000001`027cf0f0  00000000`00000000
| 00000001`027cf0f8  00000000`00000000
| 00000001`027cf100  00007ff7`4dd8000c
| 00000001`027cf108  00007ff7`f07a85e5 CONTOSO!Widget::UpdateColor+0x39
| 00000001`027cf110  00000000`ffc03f90
\-00000001`027cf118  00000001`027cf1e8
  00000001`027cf120  00000000`ffc03fc8 // VerifyData's r15
  00000001`027cf128  00000000`00000000 // VerifyData's r14
  00000001`027cf130  00000000`00000000 // VerifyData's r13
  00000001`027cf138  00007ff7`4dd8000c // VerifyData's r12
  00000001`027cf140  00007ff7`da5bc420 FABRIKAM!DataAccess::`vftable'+0x18 //
VerifyData's rdi
  00000001`027cf148  00000000`ffc03f90 // VerifyData's rsi
  00000001`027cf150  00000001`027cf260 // VerifyData's rbp
  00000001`027cf158  00007ff7`f0b3459c CONTOSO!DataWrapper::VerifyData+0x428 ← ESP is
here
  00000001`027cf160  00000000`3b803fa0 // VerifyData's rbx
  00000001`027cf168  00007ff7`f0ed1d30 CONTOSO!g_DataManager // VerifyData's rdx
  00000001`027cf170  00000000`fe196f10 // VerifyData's r8
  00000001`027cf188  00000000`00000001 // VerifyData's r9
  00000001`027cf190  00000000`00000000
  00000001`027cf198  00000001`027cf270
  00000001`027cf1a0  00000001`027cf480
  00000001`027cf1a8  00007ff7`00000001
  00000001`027cf1b0  00000001`027cf220
  00000001`027cf1b8  00000000`00000000
  00000001`027cf1c0  00000000`00000000
```

The region marked in brackets is the 0x70 bytes of space for local variables and outbound parameters. Notice that some red herring function pointers are in that space. Those are probably variables that haven't been initialized yet, and the memory happened previously to have been used to hold some return addresses.

A reassuring observation is that the `rdx` coming from `VerifyData` is the address of `CONTOSO!g_DataManager`. That is the second function parameter (or first, if you aren't counting the hidden `this`) to `RequestAccess`.

Another reassuring observation is that that `VerifyData`'s `rdi` points into the vtable for `DataAccess`, since that matches the code we saw at the call point: `call qword ptr [rdi]`.

The `mov rdi, rcx` instruction in the function body tells us that the function stashed its `this` pointer in `rdi`. That's good info to keep track of, because that will let us look at the `DataAccess` object once we figure out what is in `rdi`.

The next function on the stack is `AccessRequest::Wait`.

```
FABRIKAM!AccessRequest::Wait:
00007ff7`eeff3ca8 sub     rsp,38h
00007ffe`eeff3cb3 mov     dword ptr [rsp+20h],0FFFFFFFFh
00007ffe`eeff3cbb call    FABRIKAM!AccessRequest::WaitTimeout
00007ffe`eeff3cc0 add     rsp,38h
00007ffe`eeff3cc4 ret
```

This function doesn't bother saving any registers; it just reserves space for local variables and outbound parameters. From inspection, you can see that this is a simple wrapper that passes all its parameters onward to `WaitTimeout`, with an `INFINITE` tacked onto the end, so this function has no local variables at all. Everything is just for outbound parameters.

We can annotate some more entries in our stack trace.

```
00000001`027cf070  00000000`8007000e // spill space for WaitTimeout
00000001`027cf078  00000000`fd416ff0 // spill space for WaitTimeout
00000001`027cf080  4fe80c4a`51236583 // spill space for WaitTimeout
00000001`027cf088  00000000`3b803fb8 // spill space for WaitTimeout
00000001`027cf090  00000000`ffffffff // INFINITE parameter
00000001`027cf098  00000000`00000000 // unused
00000001`027cf0a0  00000000`3b803fa0 // unused
00000001`027cf0a8  00007ff7`da8375b7 FABRIKAM!DataAccess::RequestAccess+0x93
```

The next function on the list is `AccessRequest::WaitTimeout`.

```
FABRIKAM!AccessRequest::WaitTimeout:
00007ff7`eeff3bbc mov     qword ptr [rsp+8],rbx
00007ff7`eeff3bc1 mov     qword ptr [rsp+10h],rbp
00007ff7`eeff3bc6 push    rsi
00007ff7`eeff3bc7 sub     rsp,20h
00007ff7`eeff3bcb mov     ebx,edx
00007ff7`eeff3bcd mov     rsi,rcx
00007ff7`eeff3bd0 mov     edx,0Bh
00007ff7`eeff3bd5 mov     rcx,r8
```

This function stashes two registers in the parameter spill space, pushes one onto the stack, and reserves another 0x20 bytes for local use (outbound parameters).

```
00000001`027cf040  00000000`ffffffff // spill space for WaitForSingleObjectEx
00000001`027cf048  10f513ec`6a161fb8 // spill space for WaitForSingleObjectEx
00000001`027cf050  00000000`00000000 // spill space for WaitForSingleObjectEx
00000001`027cf058  00000000`00006ac8 // spill space for WaitForSingleObjectEx
00000001`027cf060  00000000`3b803fb8 // Wait's rsi
00000001`027cf068  00007ff7`eeff3cc0 FABRIKAM!AccessRequest::Wait+0x18
00000001`027cf070  00000000`8007000e // Wait's rbx
00000001`027cf078  00000000`fd416ff0
00000001`027cf080  4fe80c4a`51236583
00000001`027cf088  00000000`3b803fb8
```

Notice that the stashed `rbx` value is `8007000E`, which conveniently lines up with the `mov ebx,8007000Eh` instruction in `DataAccess::RequestAccess`. That's a bit reassuring, since it's another sign that we're on the right track.

Next up is `WaitForSingleObjectEx`.

```
0: kd> u 00007ffe`ef1bd085 -a5
KERNELBASE!WaitForSingleObjectEx
00007ffe`ef1bcfe0 mov     r11,rsp
00007ffe`ef1bcfe3 mov     qword ptr [r11+8],rbx
00007ffe`ef1bcfe7 mov     dword ptr [r11+18h],r8d
00007ffe`ef1bcfeb push    rsi
00007ffe`ef1bcfec push    rdi
00007ffe`ef1bcfed push    r14
00007ffe`ef1bcfef sub     rsp,80h
00007ffe`ef1bcff6 mov     ebx,r8d
```

Incorporating this prologue into our stack annotation yields

```
/-00000001`027cefa0  00000000`00006ac8 // spill space for NtWaitForSingleObject
| 00000001`027cefa8  00000000`00000000 // spill space for NtWaitForSingleObject
| 00000001`027cefb0  00000000`026d0000 // spill space for NtWaitForSingleObject
| 00000001`027cefb8  00000000`00000000 // spill space for NtWaitForSingleObject
| 00000001`027cefc0  00000001`01739ee0
| 00000001`027cefc8  00000000`00000001
| 00000001`027cefd0  00000000`00000048
| 00000001`027cefd8  00000001`00000001
| 00000001`027cefe0  00000000`00000000
| 00000001`027cefe8  00000000`00000000
| 00000001`027ceff0  00000000`00000000
| 00000001`027ceff8  00000000`00000000
| 00000001`027cf000  00000000`00000000
| 00000001`027cf008  00000000`00000000
| 00000001`027cf010  00000000`00000000
\-00000001`027cf018  00007ff7`eefe6540 FABRIKAM!Lock::IsInitialized+0xfc
  00000001`027cf020  00000000`00000000 // WaitTimeout's r14
  00000001`027cf028  00000000`3b803fa0 // WaitTimeout's rdi
  00000001`027cf030  00000000`00006ac8 // WaitTimeout's rsi
  00000001`027cf038  00007ff7`eeff3c43 FABRIKAM!AccessRequest::WaitTimeout+0x87
  00000001`027cf040  00000000`ffffffff // WaitTimeout's rbx
  00000001`027cf048  10f513ec`6a161fb8
  00000001`027cf050  00000000`00000000 // WaitTimeout's r8
  00000001`027cf058  00000000`00006ac8
```

Ooh, another red herring function pointer got caught in the local variables.

Putting everything together results in the following annotated stack, with red herrings removed.

```
00000001`027cef88  00007ffe`f1bbac9a NTDLL!NtWaitForSingleObject+0xa
00000001`027cef90  00000000`00006ac8
00000001`027cef98  00007ffe`ef1bd085 KERNELBASE!WaitForSingleObjectEx+0xa5
00000001`027cefa0  00000000`00006ac8
00000001`027cefa8  00000000`00000000
00000001`027cefb0  00000000`026d0000
00000001`027cefb8  00000000`00000000
00000001`027cefc0  00000001`01739ee0
00000001`027cefc8  00000000`00000001
00000001`027cefd0  00000000`00000048
00000001`027cefd8  00000001`00000001
00000001`027cefe0  00000000`00000000
00000001`027cefe8  00000000`00000000
00000001`027ceff0  00000000`00000000
00000001`027ceff8  00000000`00000000
00000001`027cf000  00000000`00000000
00000001`027cf008  00000000`00000000
00000001`027cf010  00000000`00000000
00000001`027cf018  00007ff7`eefe6540
00000001`027cf020  00000000`00000000 // WaitTimeout's r14
00000001`027cf028  00000000`3b803fa0 // WaitTimeout's rdi
00000001`027cf030  00000000`00006ac8 // WaitTimeout's rsi
00000001`027cf038  00007ff7`eeff3c43 FABRIKAM!AccessRequest::WaitTimeout+0x87
00000001`027cf040  00000000`ffffffff // WaitTimeout's rbx
00000001`027cf048  10f513ec`6a161fb8
00000001`027cf050  00000000`00000000 // WaitTimeout's r8
00000001`027cf058  00000000`00006ac8
00000001`027cf060  00000000`3b803fb8 // Wait's rsi
00000001`027cf068  00007ff7`eeff3cc0 FABRIKAM!AccessRequest::Wait+0x18
00000001`027cf070  00000000`8007000e // Wait's rbx
00000001`027cf078  00000000`fd416ff0
00000001`027cf080  4fe80c4a`51236583
00000001`027cf088  00000000`3b803fb8
00000001`027cf090  00000000`ffffffff // INFINITE parameter
00000001`027cf098  00000000`00000000
00000001`027cf0a0  00000000`3b803fa0
00000001`027cf0a8  00007ff7`da8375b7 FABRIKAM!DataAccess::RequestAccess+0x93
00000001`027cf0b0  00000000`3b803fa0
00000001`027cf0b8  00000000`3b803fb8
00000001`027cf0c0  4fe80c4a`51236583
00000001`027cf0c8  10f513ec`6a161fb8
00000001`027cf0d0  00000000`00000076
00000001`027cf0d8  00007ff7`f07a8619
00000001`027cf0e0  000095dc`90897985
00000001`027cf0e8  00000000`00000110
00000001`027cf0f0  00000000`00000000
00000001`027cf0f8  00000000`00000000
00000001`027cf100  00007ff7`4dd8000c
00000001`027cf108  00007ff7`f07a85e5
00000001`027cf110  00000000`ffc03f90
00000001`027cf118  00000001`027cf1e8
00000001`027cf120  00000000`ffc03fc8 // VerifyData's r15
```

```
00000001`027cf128  00000000`00000000 // VerifyData's r14
00000001`027cf130  00000000`00000000 // VerifyData's r13
00000001`027cf138  00007ff7`4dd8000c // VerifyData's r12
00000001`027cf140  00007ff7`da5bc420 FABRIKAM!DataAccess::`vftable'+0x18 //
VerifyData's rdi
00000001`027cf148  00000000`ffc03f90 // VerifyData's rsi
00000001`027cf150  00000001`027cf260 // VerifyData's rbp
00000001`027cf158  00007ff7`f0b3459c CONTOSO!DataWrapper::VerifyData+0x428
00000001`027cf160  00000000`3b803fa0 // VerifyData's rbx
00000001`027cf168  00007ff7`f0ed1d30 CONTOSO!g_DataManager // VerifyData's rdx
00000001`027cf170  00000000`fe196f10 // VerifyData's r8
00000001`027cf188  00000000`00000001 // VerifyData's r9
00000001`027cf190  00000000`00000000
00000001`027cf198  00000001`027cf270
00000001`027cf1a0  00000001`027cf480
00000001`027cf1a8  00007ff7`00000001
00000001`027cf1b0  00000001`027cf220
00000001`027cf1b8  00000000`00000000
00000001`027cf1c0  00000000`00000000
```

From this, we can also suck out the `this` pointer passed to `DataAccess::RequestAccess`. We saw that it was stashed in `rdi`. The `Wait` function doesn't use `rdi` (because if it did, it would have saved the old value), so its `rdi` is the same as `RequestAccess`'s `rdi`. Similarly, the `WaitTimeout` function does not use `rdi`. Therefore, when `WaitForSingleObject` saves the `rdi` register, it is saving the value from `DataAccess::RequestAccess`.

```
00000001`027cf028  00000000`3b803fa0 // ~~WaitTimeout~~ DataAccess's rdi
```

And that is the `this` pointer that lets us study the `DataAccess` object to figure out why its access request is not completing.

Raymond Chen

Follow