



Development Lifecycle » Debug Tips » General **Advanced**

ASM, VC6Win2K, WinXP, Win2003,
Dev, QA

Debug Tutorial Part 7: Locks and Synchronization Objects

By **Toby Opferman**

Posted: **8 Aug 2004**

Views: **63,294**

Bookmarked: **83 times**

Learn the basics of debugging deadlocks and other issues.

17 votes for this article.

Popularity: 5.52 Rating: 4.49 out of 5

Introduction

Welcome to part 7 of the Debug Tutorial series. In this section, we will learn about locks and synchronization objects in Windows. In this tutorial, we will do something a little different, we will be using both the Usermode debugger and the Kernel mode debugger since I have given the introduction to both debuggers already. This way, we get the best of both worlds.

What is a DeadLock?

This is one of the most basic multithreading questions. A deadlock occurs when a thread will never be able to acquire a resource, critical section, lock, or some type of synchronization object, due to circumstances that have caused the object to become "locked" indefinitely. The most simple example of this would be with two threads and two locks. Let's say that Lock A is used to read from the database and Lock B is used to write to the database.

Thread 1 acquires Lock A and reads from the database. Thread 2 at the same time acquires Lock B and writes to the database. Now, without releasing the locks, Thread 1 wants Lock B to write and Thread 2 wants Lock A to read. Neither thread will ever finish since they both want a resource that the other is holding. This is just an example.

This is not the only form of a "deadlock". Perhaps a thread waits on an event to become signaled before performing some action, even clean up. All threads of the application, let's say, must exit in order for the process to be allowed to exit. The thread that signals the event exits without issuing the clean up signal. The process is technically deadlocked since it will never have the last thread signaled.

Another example could be that a thread holding a lock is terminated, using `TerminateThread`. This causes the thread to be deadlocked indefinitely since the thread holding the object is no longer around and thus can never release it.

These are just variations of the problem. The basic steps to solve this problem would be to do the following.

1. Find out who owns the resource the thread wants.
2. Find out if the owning thread is waiting on a resource (Yes, goto 1).
3. Find out what resources the current thread owns.

You basically need to map out what threads own what resources and what threads are waiting to acquire those resources. Once you have plotted this, you can simply connect the dots.

Objects Available in User Mode

What types of objects are available in usermode? In this section, we will explore them and find out how to deal with them using the usermode debugger.

Events

An event is simply an object that can become signaled. No one "owns" an event, but it can be used for synchronization. Events can be named which means they are "global" and many processes can open them by name. They can also just be unnamed and as such would only be seen within that process space.

Events can be manual reset or automatic. Automatic means as soon as a waiting thread gets signaled, the system automatically resets the event to 'not signaled'. If the event is manual, then the thread or any thread must eventually reset the event in order for it to become non signaled. The kernel creates events using "KeInitializeEvent", and events can be "Notification" type or "Synchronization" type, and you may see this type when viewing handle information on events. "Notification" type is manual reset and "Synchronization" type is automatic reset. For more information on events, look in MSDN for "CreateEvent" and "KeInitializeEvent".

As with most things in user mode, an event is simply a handle, so you can use *!handle*.

```
0:001> !handle 7e4 ff
Handle 7e4
  Type          Event
  Attributes    0
  GrantedAccess 0x1f0003:
                Delete,ReadControl,WriteDac,WriteOwner,Synch
                QueryState,ModifyState
  HandleCount   2
  PointerCount  4
  Name          <NONE>
  Object Specific Information
    Event Type Auto Reset
    Event is Set
```

As you can see, this event is set and it's an automatic reset event. A real life example of how events can be used as synchronization is in "DBGVIEW", how "OutputDebugString" works with "DBGVIEW".

```
Application
1. Acquire Mutex
2. Open Memory Mapped File
3. Wait for Buffer is Ready event
4. Write to Memory Mapped File
5. Signal Buffer Data Available event
6. Close Handles

DbgView
1. Wait for Buffer Data Available event
2. Read Buffer Data.
3. Signal Buffer is ready event
4. Goto 1
```

As you can see, the mutex is simply to protect that thread against other threads in its own and other processes from writing to the memory mapped file. However, the double events are used to synchronize read/write access to the file between the application and DBGVIEW.

Mutex

A mutex is a synchronization object that can also be used globally by specifying a name. This means that multiple applications may also use the same mutex, and thus again it exists in the kernel. A mutex will only allow one thread to acquire it at a time.

```
0:005> !handle 2c0 ff
Handle 2c0
Type           Mutant
Attributes     0
GrantedAccess  0x1f0001:
               Delete,ReadControl,WriteDac,WriteOwner,Synch
               QueryState
HandleCount    2
PointerCount   3
Name           <NONE>
Object Specific Information
               Mutex is Free
0:005> !handle 2b0 ff
Handle 2b0
Type           Mutant
Attributes     0
GrantedAccess  0x120001:
               ReadControl,Synch
               QueryState
HandleCount    17
PointerCount   19
Name           \BaseNamedObjects\ShimCacheMutex
Object Specific Information
               Mutex is Free
```

Mutexes are listed as "Mutant" types. These mutexes are free. One is named and the other is not named.

Note that Windows 2003 displays more handle information than Window 2000, and due to certain problems such as dead locking, when querying information, not all information will always be displayed. So, a kernel debugger or a program like *handle.exe* or *oh.exe* that uses a driver to read the kernel object can be used to get more information. Please refer to "[Debug Tutorial Part 5](#)" for more information on handles.

So, what does a mutex look like when it's being used? There are two situations that could occur. The mutex is being used by a thread in your process or the mutex is being used by a thread in another process.

```
0:005> !handle 50 ff
Handle 50
Type           Mutant
Attributes     0
GrantedAccess  0x1f0001:
               Delete,ReadControl,WriteDac,WriteOwner,Synch
               QueryState
HandleCount    2
PointerCount   3
Name           <NONE>
Object Specific Information
               Mutex is Owned
```

This is taken from within the same process. It simply states "Mutex is Owned". There are other Mutexes in the process that have no Object Specific Information. With those Mutexes, you don't know if they're owned or free. So how do we figure out who owns this mutex?

It appears that even *handle.exe* fails us when attempting to locate the owner. We may need to get into the kernel.

So, I have created a mutex and called "[WaitForSingleObject](#)". I then set a breakpoint in the kernel on "[NtWaitForSingleObject](#)" and step through it. After stepping through everything, it maps the handle to an object and called "[KeWaitForSingleObject](#)". This function checks if it's already owned, and then we get down to two pieces of magic.

```

eax=00000001 ebx=fcc724a0 ecx=00000000 edx=00000000 esi=fcd19860 edi=fcd198cc
eip=8042d697 esp=fb72bcc0 ebp=fb72bce0 iopl=0          ov up ei ng nz na pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000a83
nt!KeWaitForSingleObject+0x1d5:
8042d697 ff4b04          dec     dword ptr [ebx+0x4] ds:0023:fcc724a4=00000001
...
eax=00000000 ebx=fcc724a0 ecx=00000000 edx=00000000 esi=fcd19860 edi=fcd198cc
eip=8042d6aa esp=fb72bcc0 ebp=fb72bce0 iopl=0          nv up ei ng nz ac po cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000297
nt!KeWaitForSingleObject+0x1e8:
8042d6aa 897318          mov     [ebx+0x18],esi  ds:0023:fcc724b8=00000000
...
kd> !object fcc724b8
Object: fcc724b8 Type: (fcc724a8)
  ObjectHeader: fcc724a0
    HandleCount: 0  PointerCount: 524290
    Directory Object: 00000001 Name: (** Name not accessible **)
kd> dd fcc724b8 l1
fcc724b8 fcd19860
kd> !thread fcd19860 1
THREAD fcd19860 Cid 214.418 Teb: 7ffdc000 Win32Thread: 00000000 RUNNING

```

EBX is pointing to the object header. So, we know that ObjectHeader + 4 = Free count, and ObjectHeader + 18 = Owner Thread.

Semaphore

Just like the rest, these are really kernel objects. The only real difference between a Semaphore and a Mutex is that a Semaphore can have a count > 1. While a mutex will only let one owner attempt access, a Semaphore can be assigned a number and allow "x" number of threads access.

```

0:002> !handle 0 f semaphore
Handle 90
Type           Semaphore
Attributes     0
GrantedAccess  0x1f0003:
                Delete,ReadControl,WriteDac,WriteOwner,Synch
                QueryState,ModifyState
HandleCount    2
PointerCount   3
Name           <NONE>
Object Specific Information
  Semaphore Count 15
  Semaphore Limit 16
1 handles of type Semaphore

```

So, from the usermode debugger, we can find how many semaphores are open and what the limit is. Remember, a value of 0 means they are all taken. The usermode debugger also treats semaphores and mutexes as "seperate" objects. In any case, let's see what we can find in the object.

```
kd> !handle 90 ff fccde020
processor number 0
PROCESS fccde020 SessionId: 0 Cid: 03cc Peb: 7ffdf000 ParentCid: 03d8
  DirBase: 00e7b000 ObjectTable: fccbfae8 TableSize: 36.
  Image: mspaint.exe
```

```
Handle Table at e1e62000 with 36 Entries in use
0090: Object: fcec1680 GrantedAccess: 001f0003
Object: fcec1680 Type: (fceb620) Semaphore
  ObjectHeader: fcec1668
    HandleCount: 1 PointerCount: 1
```

```
kd> dd fcec1668
fcec1668 00000001 00000001 fceb620 00000000
fcec1678 fcc75888 00000000 00050005 00000010
fcec1688 fcec1688 fcec1688 00000010 7ffddfff
fcec1698 00000000 00000000 03018002 644c6d4d
fcec16a8 fcec19a8 fce73b48 00650074 0052006d
fcec16b8 006f006f 005c0074 fc8f5000 fc90b87c
fcec16c8 00025000 00500050 e134a588 00160016
fcec16d8 fcec87a8 09104000 004b0002 ffffffff
```

The above is a dump of the object header before we make the call to acquire the object. Now, below, we will acquire the object and see what occurs.

```
kd> !handle 90 ff fccde020
processor number 0
PROCESS fccde020 SessionId: 0 Cid: 03cc Peb: 7ffdf000 ParentCid: 03d8
  DirBase: 00e7b000 ObjectTable: fccbfae8 TableSize: 36.
  Image: mspaint.exe
```

```
Handle Table at e1e62000 with 36 Entries in use
0090: Object: fcec1680 GrantedAccess: 001f0003
Object: fcec1680 Type: (fceb620) Semaphore
  ObjectHeader: fcec1668
    HandleCount: 1 PointerCount: 1
```

```
kd> dd fcec1668
fcec1668 00000001 00000001 fceb620 00000000
fcec1678 fcc75888 00000000 00050005 0000000f
fcec1688 fcec1688 fcec1688 00000010 7ffddfff
fcec1698 00000000 00000000 03018002 644c6d4d
fcec16a8 fcec19a8 fce73b48 00650074 0052006d
fcec16b8 006f006f 005c0074 fc8f5000 fc90b87c
fcec16c8 00025000 00500050 e134a588 00160016
fcec16d8 fcec87a8 09104000 004b0002 ffffffff
```

The count was decremented, so we now know where the semaphore limit and count is stored. The other problem is that we do not know what thread acquired the semaphore. How do we find this out?

After a little bit of debugging, we find that the same code path isn't taken for a semaphore, and that address does not contain the owning thread, and the thread's context is not saved. Unfortunately, semaphores aren't "owned" by a thread or a number of threads, they work based on a count.

In fact, we notice that if you call "[WaitForSingleObject](#)" a couple of times on the same thread, the resource decrements. This means that you must be responsible when programming, since you could probably just randomly release a semaphore even if you didn't decrement its count. It also means that a recursive thread is not saved using a semaphore as it is with a mutex or critical section.

Critical Section

These are actually usermode data structures. They prevent multiple threads from attempting to access a single resource from within a single process. The fact that they exist in usermode makes them very easy to debug with just a usermode debugger.

This is good because as mentioned, we can find out who the owner is, from usermode. You can also

use `!locks` and `!locks -v` to get information about critical sections in your own process.

```
0:000> !critsec ntdll!LdrpLoaderLock

CritSec ntdll!LdrpLoaderLock+0 at 77FC1774
LockCount          0
RecursionCount     1
OwningThread       9ac
EntryCount         0
ContentionCount    0
*** Locked
```

The most famous lock in Windows is the loader lock. It's locked while loading a library, creating or destroying a thread among other things. As we can see, the loader lock is now locked. We used the `!critsec` on the address of the lock to get this information. Who is the owning thread?

```
0:001> ~*
  0 Id: e28.9ac Suspend: 1 Teb: 7ffde000 Unfrozen
    Start: notepad!WinMainCRTStartup (01006ae0)
    Priority: 0 Priority class: 32
.  1 Id: e28.aa0 Suspend: 1 Teb: 7ffdd000 Unfrozen
    Start: ntdll!DbgUiRemoteBreakin (77f5f2f2)
    Priority: 0 Priority class: 32
```

While there is only two threads in the process, you can see how to match up the thread with the lock.

```
0:001> !critsec ntdll!LdrpLoaderLock

CritSec ntdll!LdrpLoaderLock+0 at 77FC1774
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0
```

This is the output when the critical section is not locked. What other advantages do critical sections have over mutexes? Since they are in usermode, they don't need to enter the kernel to become locked or have contention with other processes. This makes them faster and as you can see, easier to work with when debugging your user mode application. Doing "dd" on the address, you will also see how simple the data structure really is, and what information the debugger is displaying from where.

If the thread is no longer there, then obviously it has either been terminated (`TerminateThread`), or exited on its own, without releasing the lock.

Objects Available in Kernel Mode

I actually covered some of the objects available in kernel mode. Events, Semaphores, Mutexes are all available in kernel mode. In fact, the user mode applications simply call into the kernel to create the same objects using the same functions the kernel uses to create them. There are a few other types of objects that I would like to talk about in the kernel though. These are spinlocks and ERESOURCES.

Spinlocks

A spinlock is a synchronization object that is used to keep multiprocessor systems from accessing the same resource at the same time. The difference between a spinlock and a critical section is that the second processor will spin on this lock until it is able to acquire it, instead of allowing other threads to be scheduled to run.

On a single processor system, a spinlock will simply raise the IRQL level so that nothing else is scheduled in the time this code is executing. This means that you cannot access pageable memory, and you should perform only a small operation as you do not want to hog the processor.

```

hal!KfAcquireSpinLock:
80069850 33c0          xor     eax,eax
80069852 a024f0dfff   mov     al,[ffdff024]
80069857 c60524f0dfff02  mov     byte ptr [ffdff024],0x2
8006985e c3           ret

```

The address is a global that points to the IRQL level. Calling `KeAcquireSpinLock` simply puts the IRQL level at 2. It then simply saves the old IRQL level in the spinlock data structure passed into the function. This is used to restore the previous level upon calling the `KeReleaseSpinlock`.

An IRQL is an IRQ Level that the Operating System is operating at. Below is the level as defined in the NTDDK header files.

```

#define PASSIVE_LEVEL 0           // Passive release level
#define LOW_LEVEL 0             // Lowest interrupt level
#define APC_LEVEL 1           // APC interrupt level
#define DISPATCH_LEVEL 2       // Dispatcher level

```

So, a spinlock brings the Operating System up to `DISPATCH_LEVEL`. For more information on IRQLs, please refer to the MSDN documentation.

The spinlock function is a little different on a multiprocessor machine as you can see below. They "spin" on a byte attempting to "lock" it and continue doing this until it's "locked".

Basically, to describe the below assembly, the "LOCK" instruction locks the bus for protection against multiple processors writing or reading to the same memory area. The `BTS` instruction with 0 specified moves bit 0 into the carry flag and then sets bit 0 to 1.

So, "JB" will jump if the carry flag is 1 which means it was previously 1. It then does a test if bit 0 is 1. If bit 0 is not one, it jumps back and tries again. If bit 0 is 1, it means it's still being held, so it does a "pause" before trying again.

```

hal!KfAcquireSpinLock:
80065420 8b158000feff   mov     edx,[fffe0080]
80065426 c7058000feff41000000  mov     dword ptr [fffe0080],0x41
80065430 c1ea04        shr     edx,0x4
80065433 0fb68280a30680  movzx  eax,byte ptr [edx+0x8006a380]
8006543a f00fba2900    lock  bts  dword ptr [ecx],0x0
8006543f 7203         jb     hal!KfAcquireSpinLock+0x24 (80065444)
80065441 c3           ret
80065442 8bff        mov     edi,edi
0: kd> u
hal!KfAcquireSpinLock+0x24:
80065444 f70101000000  test   dword ptr [ecx],0x1
8006544a 74ee        jz     hal!KfAcquireSpinLock+0x1a (8006543a)
8006544c f390        pause
8006544e ebf4        jmp    hal!KfAcquireSpinLock+0x24 (80065444)

```

This is the essence of how spinlocks work, there's not much to them, and in general, most applications would never need a spinlock. In most cases, a semaphore or mutex should work fine. If you want to use spinlocks, read the documentation in MSDN. There are also "queued" spinlocks available that are supposed to provide better performance.

The ERESOURCE

I am going to keep the explanation of ERESOURCE down to a minimum since:

- A. You can read about them on MSDN and I don't want to regurgitate information, and
- B. If you don't know what they are, you probably aren't using them and don't need to debug them. This is an article about debugging not programming.

However, I will give a quick overview. The `ERESOURCE` is a data structure you can use in the kernel to allow shared or exclusive access. Shared means many threads can acquire it, and Exclusive

obviously means only one thread may acquire it.

One thing to be aware of is that **ERESOURCES** are in a globally linked list in non-paged memory. This means that if you free this memory or overwrite this data structure without deleting the resource first, you will corrupt this list.

In the kernel debugger, you can use a command called `!locks` to dump all locks on the system.

```
kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.....

Resource @ 0xfceba0c0   Shared 1 owning threads
      Threads: fcebeda3-01<*> *** Actual Thread FCEBEDA0
KD: Scanning for held locks.
1814 total locks, 1 locks currently held
```

When doing this, it will show you the threads which own the lock, and it will also show you a list of threads waiting on the lock. You can also do `!locks <ADDRESS>` to get the information. There are also some flags that can be used, `!locks -v`, for example.

The great thing about these locks is that `!locks` will list owning **ETHREAD** addresses as well as waiting **ETHREAD** addresses, so it becomes very simple to debug contention. The lock above only has one owner and no waiters listed. In Windows XP/2003, I believe you can even do `dt _ERESOURCE` on the data to display all of the fields.

Conclusion

We have covered the commonly used synchronization objects in Windows usermode and kernel. There are other methods such as `InterlockedDecrement`, Locking a byte in a file (`LockFile`), and other operations/methods that the programmer themselves could actually use to implement their own synchronization. Those would be beyond this tutorial as it would depend upon the implementation how you would go about debugging them.

License

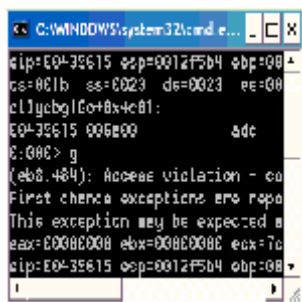
This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

About the Author

Toby Opferman

Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.



He has also played a variety of roles professionally on a wide range of projects. This has included pure researching roles, architect roles and developer roles. He also was also solely responsible for debugging traps and blue screens for a number of years.

Previously of Citrix Systems he is very experienced in the area of Terminal Services. He currently works on Operating Systems and low level architecture at Intel.


Occupation: Engineer

Company: Intel

Location:  United States

Member

Discussions and Feedback

 **6 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/debug/cdbntsd7.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)
Last Updated: 8 Aug 2004
Editor: [Smitha Vijayan](#)

Copyright 2004 by Toby Opferman
Everything else Copyright © [CodeProject](#), 1999-2009
Web16 | [Advertise on the Code Project](#)