Synchronization and deadlock

Effective Debugging

This lab focuses on finding the reason why an application would stop responding.

Setup: download https://github.com/chrisnas/EffectiveDebugging/tree/master/SourceCode and open the .sln file.

Presentation of the application

The **StockMarket** application is a simulation of marketplace for cats. The protagonists can place offers for feline commodities or buy from an existing offer. If an offer doesn't get a buyer after 5 seconds, then it expires and is removed.

For every cat, a thread is created and runs the `CatThread` method. The logic is an infinite loop where the desires of the cat are rolled randomly, then buy or sell actions are taken accordingly.

To make the behavior of the application deterministic, actions are synchronized through the `Clock` object, and particularly the `_clock.WaitForNextCycle` method which will block the thread until the next second.

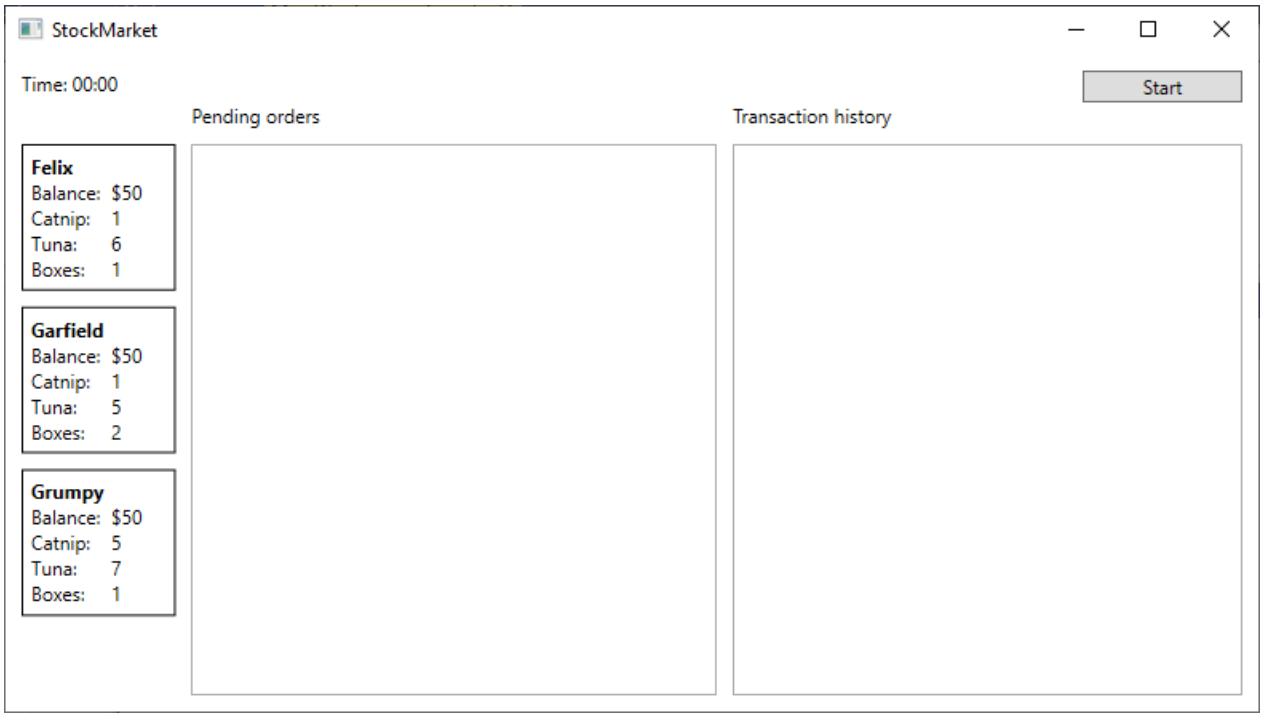
Also, every second, a task is spawned to check for expired orders.

Lastly, because of the multiple threads, the application makes heavy use of locks to synchronize all operations.

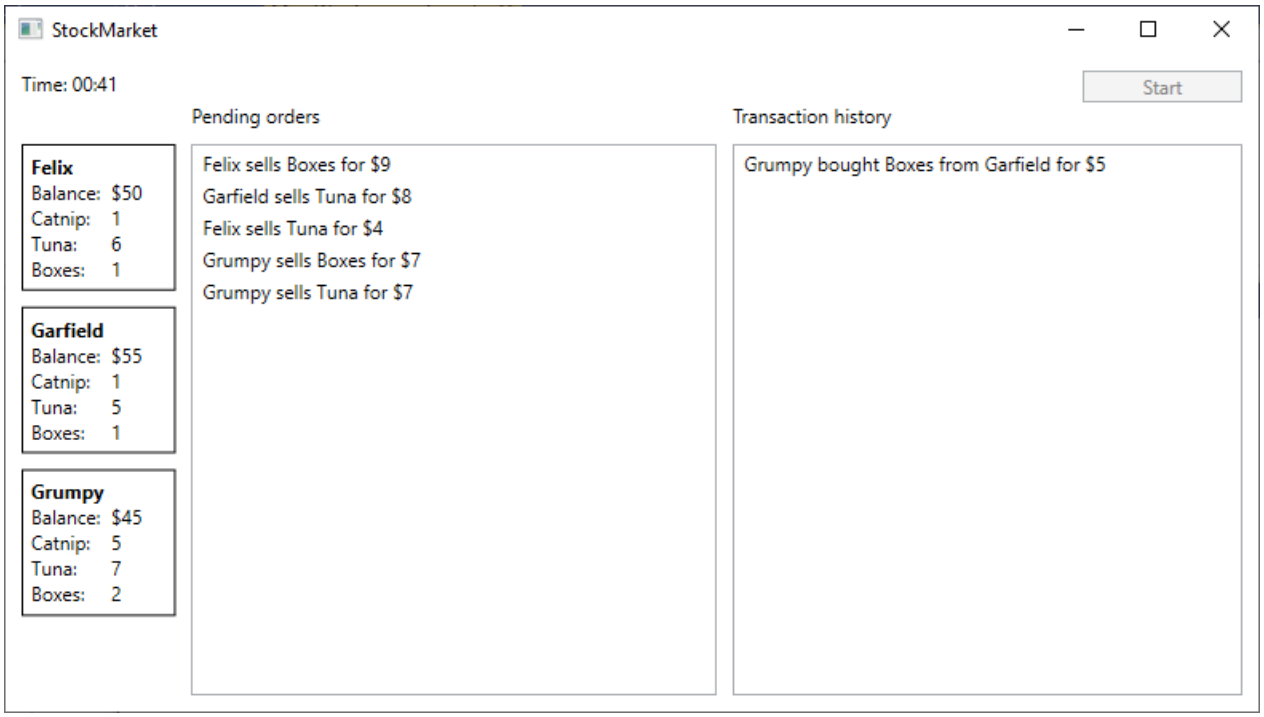
Note: because multithreaded issues are dependent on timing, and despite the use of the `Clock` to make the execution more deterministic, you might see a different set of callstacks than the ones shown in this document. This does not change the core issue, so you should be able to solve it by following the same methodology.

Reproducing the problem

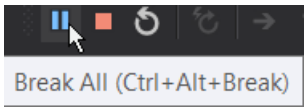
Start by debugging the **StockMarket** project, click the "Start" button on the top right corner to begin the auction:



After a few seconds, you'll see no more activity. Plus, orders aren't cleared after 5 seconds, which seems to indicate that something is wrong:



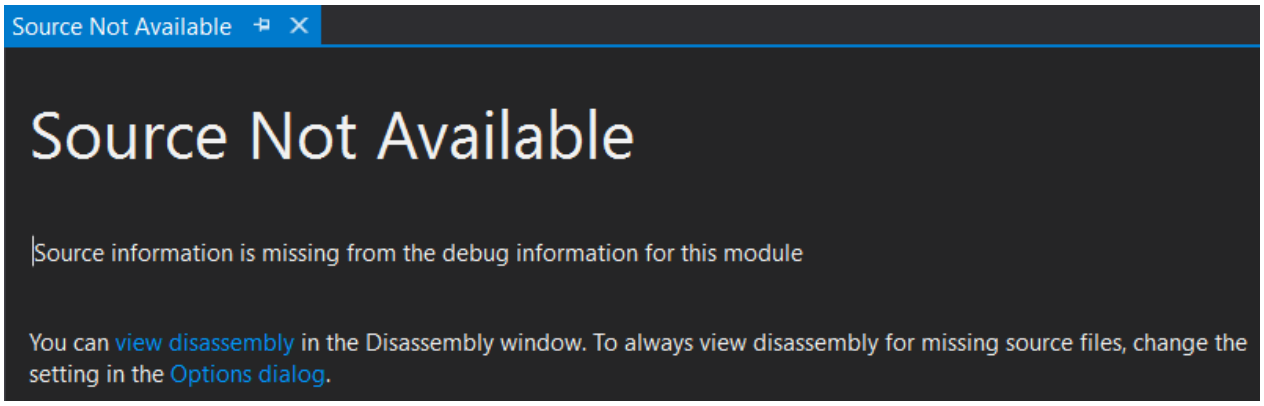
Investigating with Visual Studio

1. When the application reaches the aforementioned state, click 

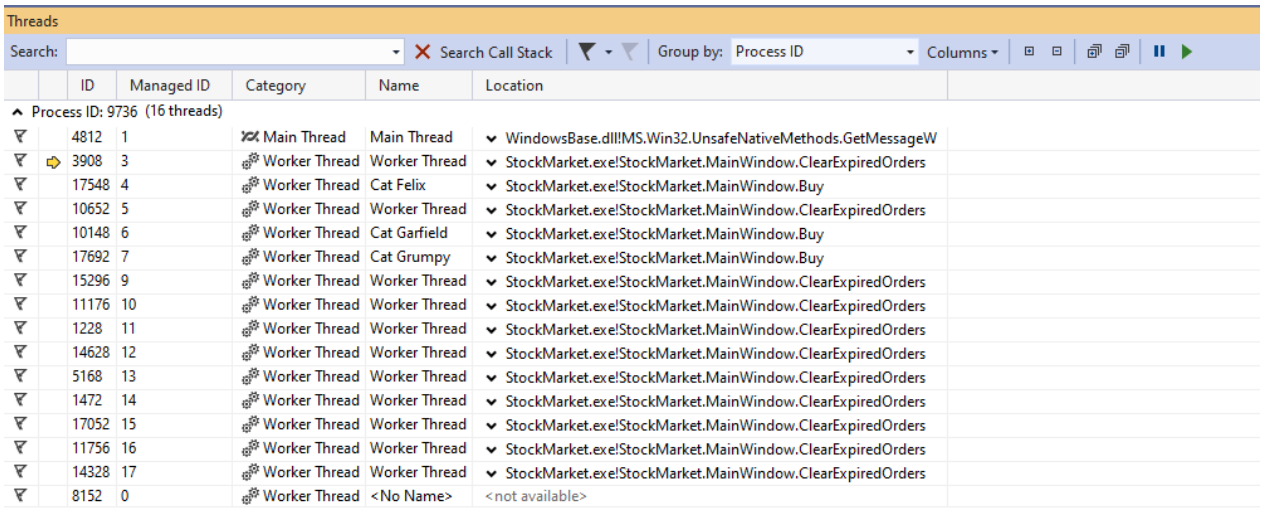
to suspend the execution of the application. This will automatically bring you to the code that one of the thread was executing:

```
1 reference
private void ClearExpiredOrders()
{
    lock (Orders)
    {
        var expiredOrders = Orders.Where(o => (Time - o.Timestamp).TotalMinutes > 5).ToList();

        foreach (var order in expiredOrders)
        {
            lock (order.Seller)
            {
                Orders.Remove(order);
                order.Seller.PendingOrders -= 1;
            }
        }
    }
}
```

Note: it is possible that the debugger picks a thread for which you don't have source code and in that case, the following window will be displayed:



2. In the case of a deadlock, multiple threads are involved, so you need a way to see the state of each of those threads.
Open the **Debug** | **Windows** | **Threads panel**



Here you can see all the threads. The yellow arrow in the second column indicates the active thread (the one executing the code seen in step 1). The name of the thread, displayed in the 6th column, can be changed by calling `Thread.CurrentThread.Name = "My thread"`. This makes debugging easier when you manage your own threads: you can see here the threads associated with cats for example. However, you shouldn't do that when using the threadpool, as the thread will keep its custom name when executing other unrelated workitems.

Note: you can only set a thread name once; an exception will be thrown if you try to set a thread name more than once.

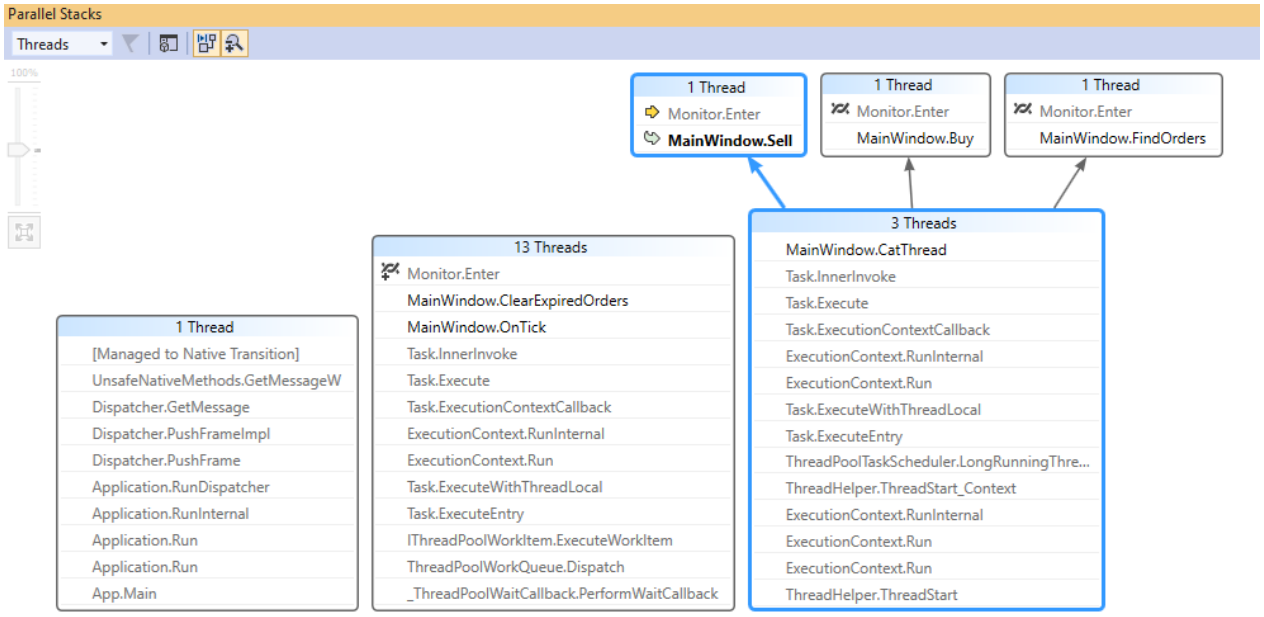
The last column shows the method that was being executed by the thread when you broke the execution.

3. Double-click on another thread, for instance « Cat Felix » :

```
1 reference
private void Sell(string item, Cat seller, int price)  item = "Catnip",  seller = {Cat},  price = 1
{
    lock (seller)  seller = {Cat}
    {
        if (seller.PendingOrders < 2)
        {
            lock (Orders)
            {
                Orders.Add(new Order(seller, item, price, Time));
                seller.PendingOrders += 1;
            }
        }
    }
}
```

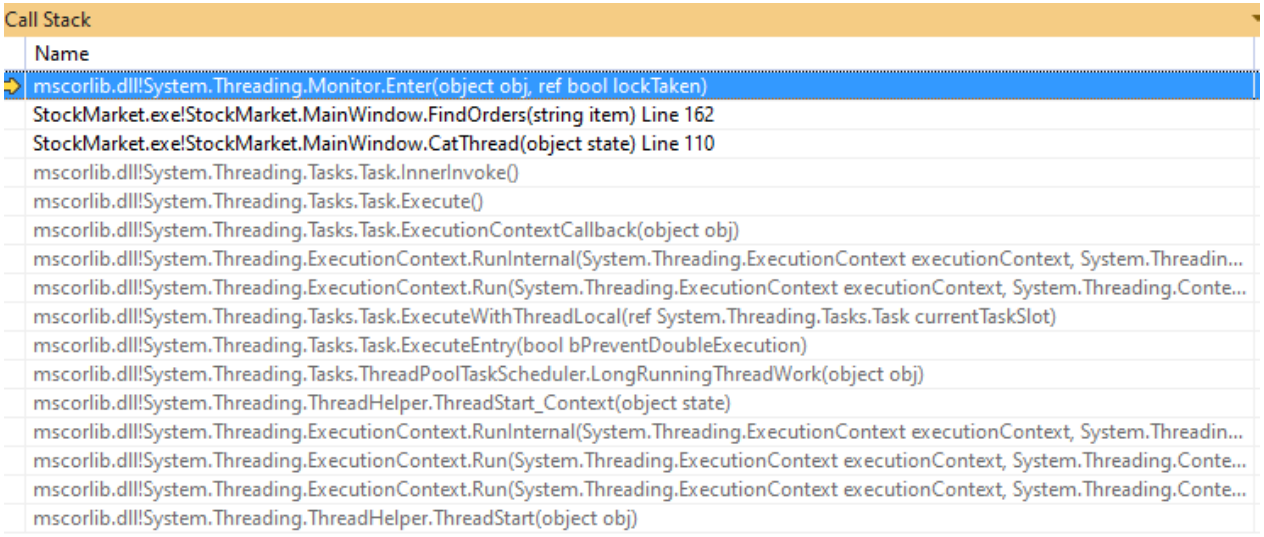
This is how you switch from on thread to another to understand what they're all doing. You can inspect the value of the local variables as seen by that thread.

4. The **Threads** panel is nice, but it's a bit difficult to get the big picture when a lot of threads are running. Fortunately there is another, more visual representation. Open the **Debug** | **Windows** | **Parallel Stacks** panel

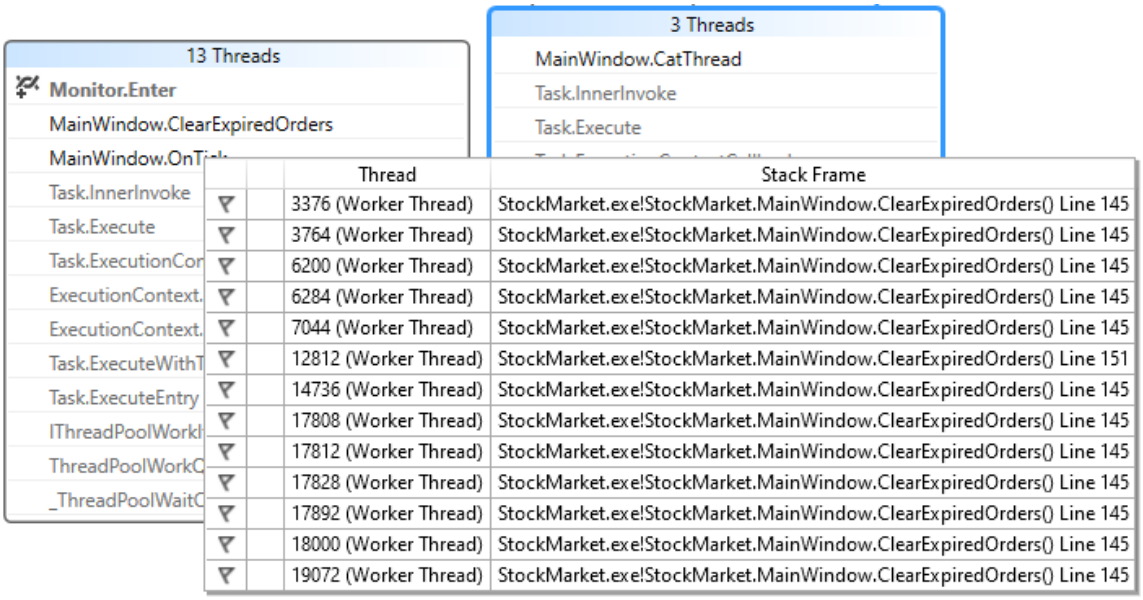


The boxes represent the different callstacks of the running threads.
At a glance, we can see where the different threads are blocked. You can double-click on any frame of a callstack to display the source code, if available.

The **Call Stack** panel (**Debug** | **Windows** | **Call Stack**) is synchronized with the thread currently selected, so it's a nice help when forgetting where you are:

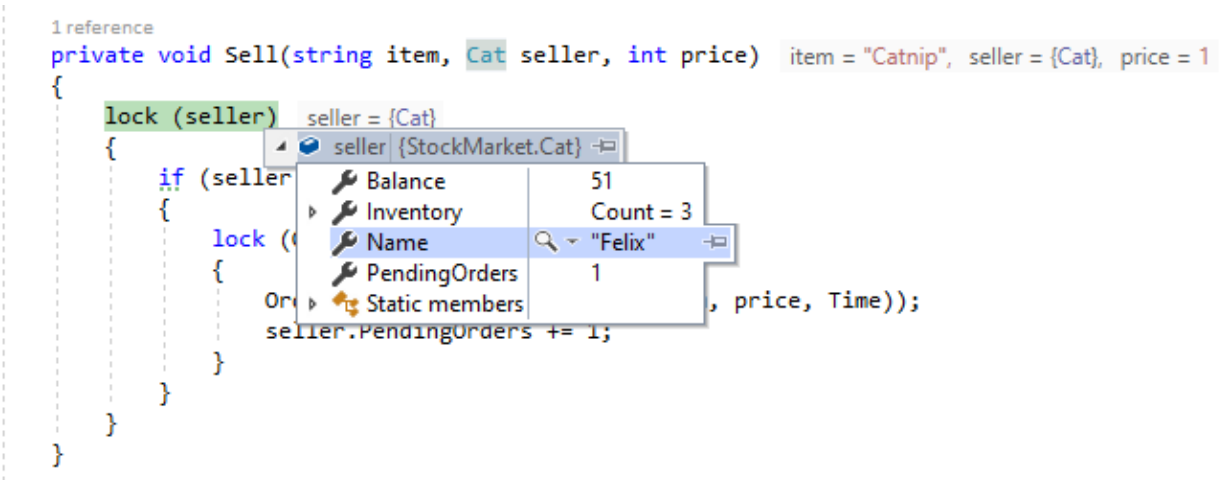


5. When there are multiple threads running the same method, you can hover the mouse on top of a stack frames to see the list of threads with the same call stack



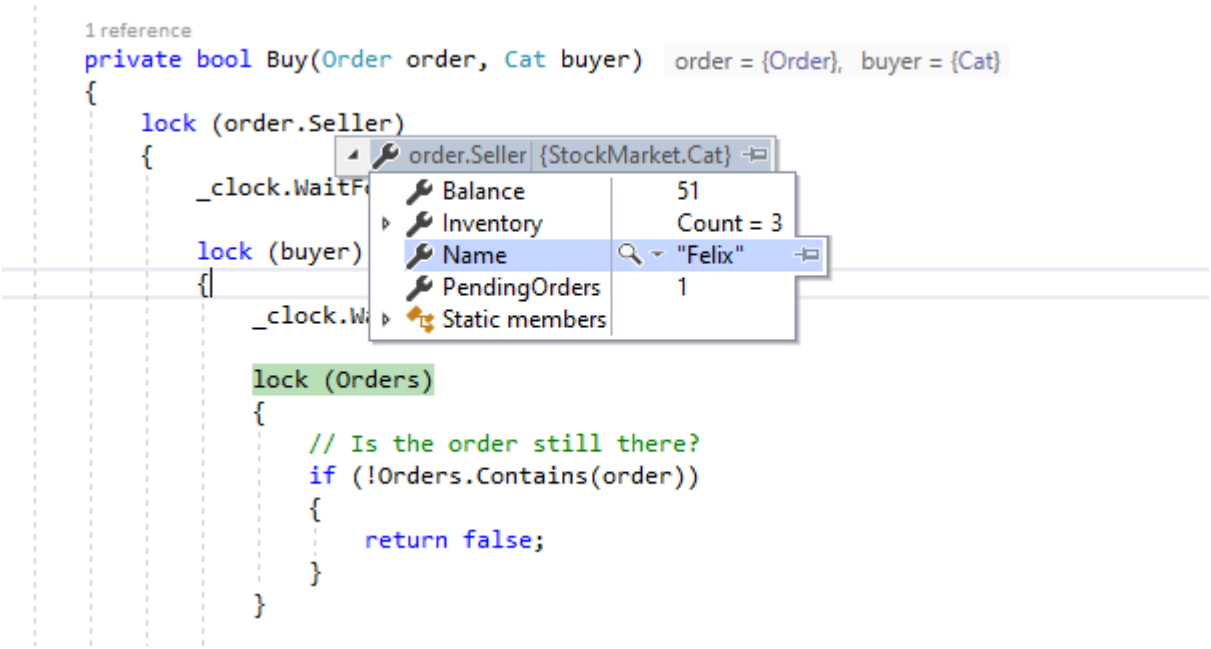
You can set any of those threads as the active thread by double-clicking it.

6. Now it's time to understand what's going on. Here, our three cat threads are blocked, respectively in **MainWindow.Sell**, **MainWindow.Buy**, and **MainWindow.FindOrders** (it might be different in your case, for instance you could have all 3 threads in **MainWindow.Buy**). The top frame of each of those callstacks shows "Monitor.Enter", which means that the thread is waiting to acquire a lock.
   Inspect each of those 3 threads to understand on what lock they're blocked.
   The first one if blocked on `lock (seller)`. By inspecting the value of **seller**, we see that it's the cat object representing « Felix » by looking at its **Name** property.

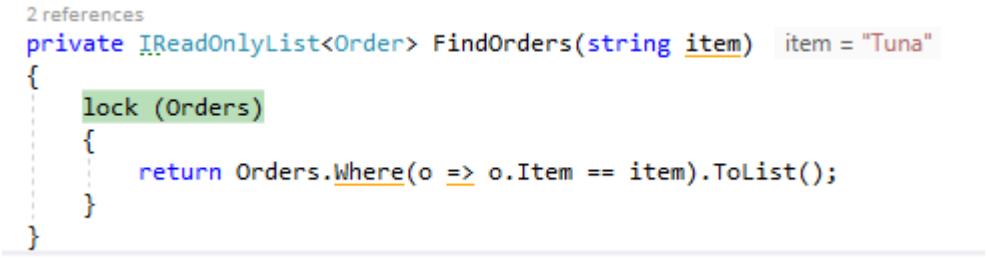


Note : you may want to apply the [DebuggerDisplayAttribute] on the `Cat` class to display the name of the cat by just hovering the mouse (without having to dig into the properties) and make debugging easier.

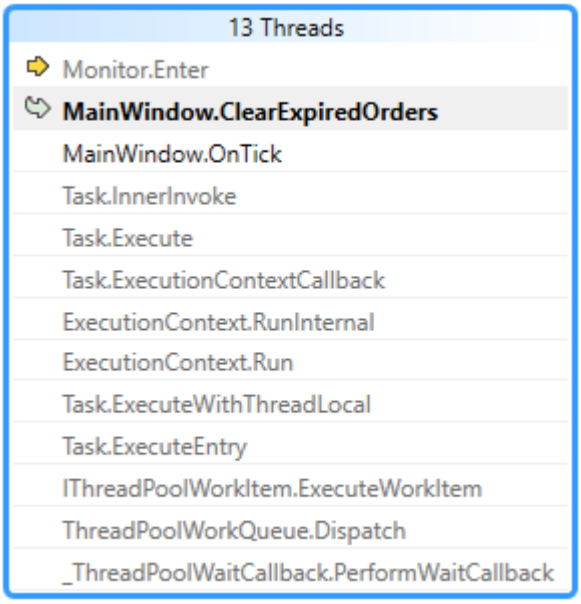
The second one is blocked on `lock (Orders)`. If we look at the method, we can see that the thread has already acquired two locks : `order.Seller` (in this case, “Felix”), and `buyer` (in this case, “Garfield”).



We now know what thread owns the lock on “Felix”, blocking the first thread. And that thread is blocked waiting on **Orders**, so we need to find who owns this lock.
The third cat thread is also blocked on `lock (Orders)`. It doesn’t look like it has acquired any lock previously, so this isn’t our culprit.



7. So far we’ve only inspected the cat threads. We haven’t looked yet at all the threads blocked on `ClearExpiredOrders`

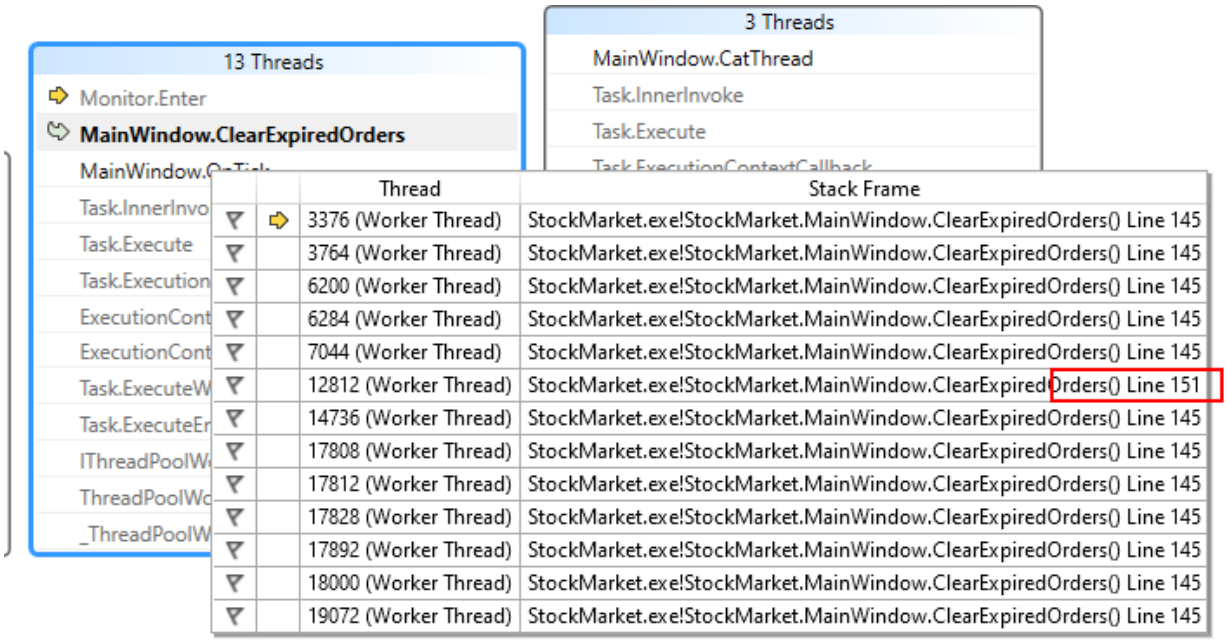


Selecting the first one, we can see it's waiting to acquire the lock on **Orders**. So it's not our culprit either:

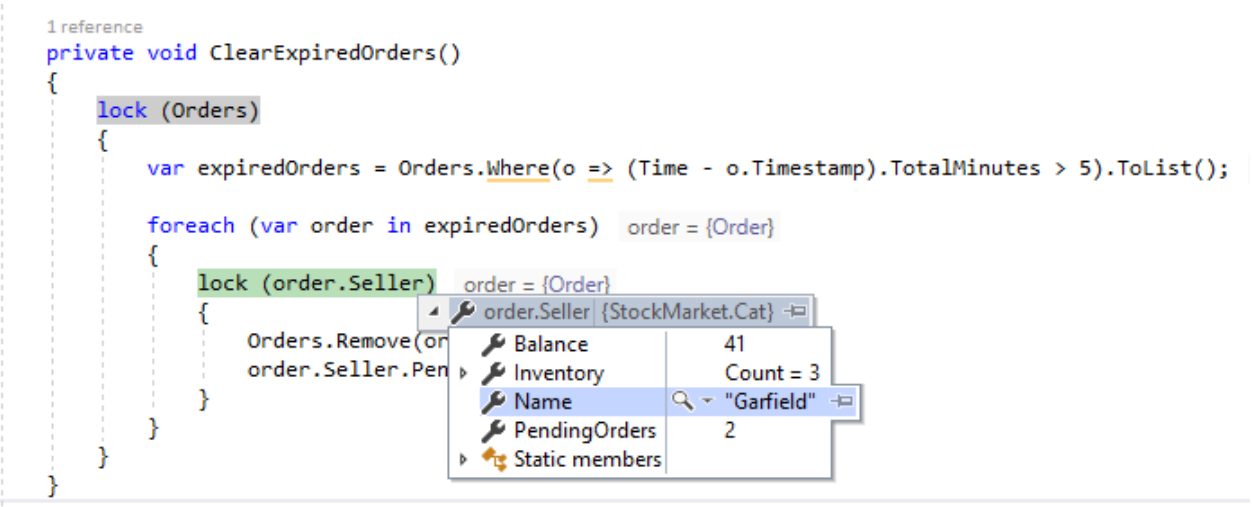
```
1 reference
private void ClearExpiredOrders()
{
    lock (Orders)
    {
        var expiredOrders = Orders.Where(o => (Time - o.Timestamp).TotalMinutes > 5).ToList();

        foreach (var order in expiredOrders)
        {
            lock (order.Seller)
            {
                Orders.Remove(order);
                order.Seller.PendingOrders -= 1;
            }
        }
    }
}
```

But there's still 12 other threads (possibly more in your case) blocked on the same method. Plus, by graying out the `lock (order.Seller)` line, Visual Studio is telling us that at least one thread has reached that point. How to find which one? Do we have to inspect all 12 of them manually?

8. Fortunately, we don't. In the **Parallel Stacks** panel, hover the mouse on **MainWindow.ClearExpiredOrders** to display the list of threads:



You can see the line number at the end of the **Stack Frame** column. All of them are on line 145 except one: double-click it to switch to line 151 in the source code.

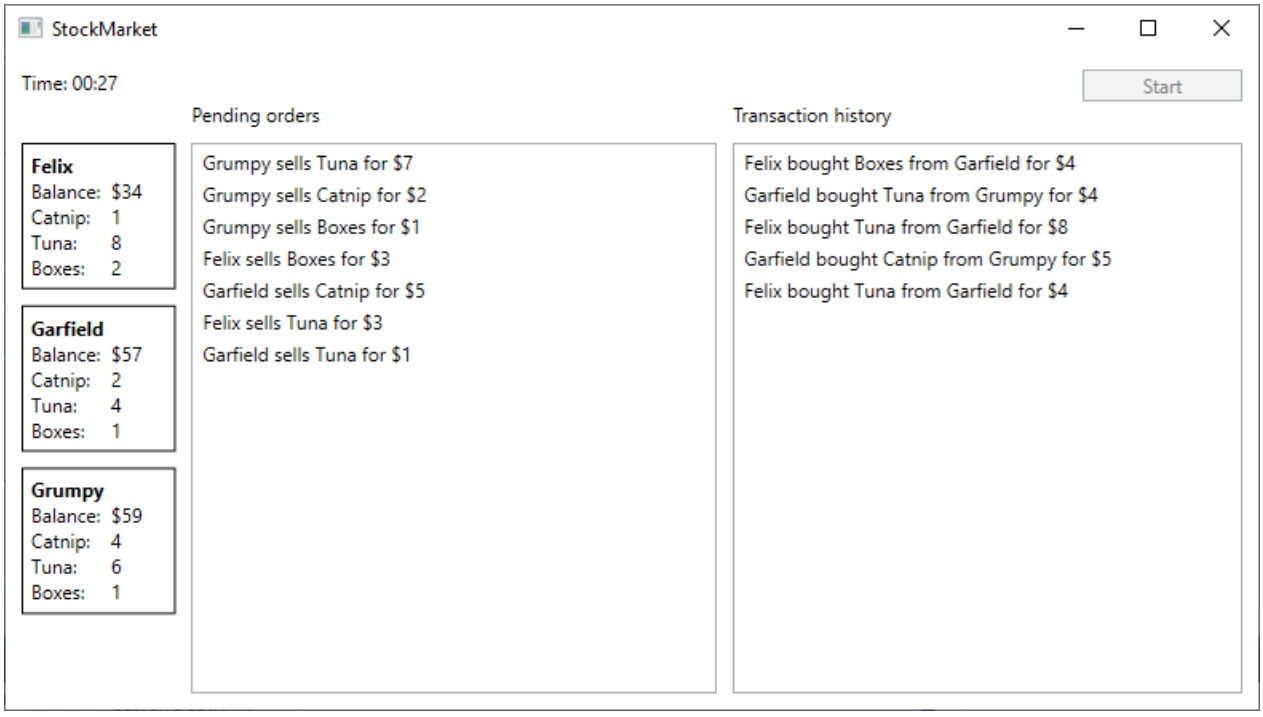


We can see that this thread has acquired the **Orders** lock, and is waiting to acquire the lock on "Garfield".

9. Piecing everything together, we have one thread that acquired the lock on "Felix" and "Garfield", and is waiting for **Orders**. We also have one thread that acquired **Orders**, and is waiting for "Garfield". This is a deadlock situation.
   How can we fix it? As much as possible, when dealing with multiple locks, you must make sure that every thread acquires them in the same order. Here, we have a deadlock because one thread tries to acquire "Garfield" then **Orders**, and the other tries to acquire **Orders** then "Garfield". To fix that, we can change the `ClearExpiredOrders` method to always lock the seller before **Orders**:

```
1 reference
private void ClearExpiredOrders()
{
    List<Order> expiredOrders;

    lock (Orders)
    {
        expiredOrders = Orders.Where(o => (Time - o.Timestamp).TotalMinutes > 5).ToList();
    }

    foreach (var order in expiredOrders)
    {
        lock (order.Seller)
        {
            lock (Orders)
            {
                Orders.Remove(order);
                order.Seller.PendingOrders -= 1;
            }
        }
    }
}
```

10. After applying the fix and running the application again, it gets stuck once more after a while:



Can you apply what you just learn to fix this one?

Post-mortem analysis

If Visual Studio cannot be used (secured server environment or customer machines for example), it is still possible to troubleshoot this kind of problem with the following steps:

1. Save the application memory to a dump file
2. Open the dump file in Visual Studio
3. Do the same investigation as presented in the previous paragraphs

Dumping application memory

Even though it is possible to create a memory dump with Task Manager, it is recommended to use **procdump** from SysInternals.

1. Download **procdump** from https://docs.microsoft.com/en-us/sysinternals/downloads/procdump

2. Start a Prompt, create and move to a **dumps** folder in your hard drive

3. Once the application is hang, type the following command line:
   procdump -ma <process ID>
   Note: the first time Sysinternals tools are started, it is required to accept the EULA form.

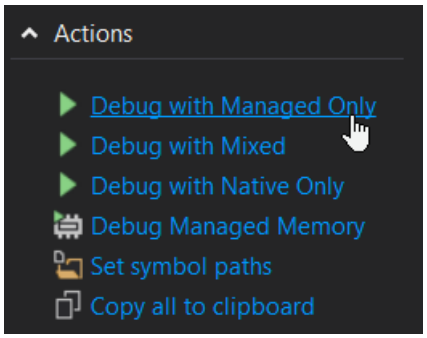
   *Note: by default, a .dmp file will be created with a name following the pattern **<process name>_<date>_<time>.dmp** but you can pass a filename as last parameter if you prefer.*

Opening a dump with Visual Studio

Visual Studio is able to open a memory dump and provides a few debugging services helping troubleshoot hang issues.

1. Select the .dmp file from **File** | **Open** | **File…** or drag and drop it on Visual Studio title bar from Windows Explorer

2. In the **Minidump File Summary** window, click the **Debug with Managed Only** link in the upper right **Actions** section



3. Once this is done, Visual Studio puts its debugger into the same state as if you had set a breakpoint at the time the dump was taken. Therefore, you are able to use the same **Parallel Stacks** panel and investigate the possible cause of deadlock the same way at it was described in the previous sections.

Using WinDBG to easily find deadlocks

Even though Visual Studio provides great help to investigate hangs scenarios, WinDBG and its extensions also brings great value by automating the search for deadlocks for critical sections, locks and **ReaderWriterLock**s.

1. Select your .dmp file via **File** | **Open Dump File** in WinDBG
2. Load sos extension: **.loadby sos clr**
3. Download the sosex.dll extension from http://www.stevestechspot.com/downloads/sosex_64.zip
   Use http://www.stevestechspot.com/downloads/sosex_32.zip if you have a 32bits memory dump
4. Load sosex extension: **.load <path>\sosex**
5. Let sosex find the deadlocks: **!dlk**

```
*DEADLOCK DETECTED*

CLR thread 0x5 holds the lock on SyncBlock 01700f10
OBJ:036f0110[System.Collections.ObjectModel.ObservableCollection`1[[StockMa
rket.Order, StockMarket]]]

...and is waiting for the lock on SyncBlock 063785bc
OBJ:036eff24[StockMarket.Cat]

CLR thread 0x6 holds the lock on SyncBlock 063785bc
OBJ:036eff24[StockMarket.Cat]

...and is waiting for the lock on SyncBlock 01700f10
OBJ:036f0110[System.Collections.ObjectModel.ObservableCollection`1[[StockMa
rket.Order, StockMarket]]]
```

```
CLR Thread 0x5 is waiting at System.Threading.Monitor.Enter(System.Object,
Boolean ByRef)(+0x18 Native)
[f:\dd\ndp\clr\src\BCL\system\threading\monitor.cs @ 62,9]

CLR Thread 0x6 is waiting at System.Threading.Monitor.Enter(System.Object,
Boolean ByRef)(+0x18 Native)
[f:\dd\ndp\clr\src\BCL\system\threading\monitor.cs @ 62,9]

1 deadlock detected.
```

6. The output provides:
   - The id of the deadlocked threads (0x5 and 0x6)
   - The objects and their type used as locks (`036f0110[System.Collections.ObjectModel.ObservableCollection`1[[StockMarket.Order, StockMarket]]]` and `036eff24[StockMarket.Cat]`)
   - The top of each thread callstack

7. With !do sos command, you are able to look at the fields value of the objects used as locks. For example, **!dlk** shows that thread 0x6 tries to acquire the instance of **StockMarket.Cat**; referenced by 036eff24. This is interesting but it would be better to know its name: just type the following command:

```
!do 0x036eff24
```

```
0:000> !do 0x036eff24
Name:        StockMarket.Cat
MethodTable: 00007ffb1a977df8
EEClass:     00007ffb1a8e4408
Size:        48(0x30) bytes
File:        C:\github\chrisnas\EffectiveDebugging\SourceCode\StockMarket\bin\x64\Release\StockMarket.exe
Fields:
              MT    Field   Offset                 Type VT     Attr            Value Name
00007ffb59168ea0  4000001        8 ...angedEventHandler  0 instance 000002ce00174170 PropertyChanged
00007ffb78f69808  4000002       10        System.String  0 instance 000002ce000683f0 <Name>k__BackingField
00007ffb78852bf8  4000003       18 ...Int32, mscorlib]]  0 instance 000002ce00107f78 <Inventory>k__BackingField
00007ffb78f6c158  4000004       20         System.Int32  1 instance               52 <Balance>k__BackingField
00007ffb78f6c158  4000005       24         System.Int32  1 instance                2 <PendingOrders>k__BackingField
```

```
0:000> 0x036eff24
```

Since we are using C# automatic properties, the fields names are automatically generated following the **<property name>k__BackingField** so the name can be found by looking at the string instance referenced by the **<Name>k__BackingField** value:

```
0:000> !do 000002ce000683f0
Name:        System.String
MethodTable: 00007ffb78f69808
EEClass:     00007ffb78846cb8
Size:        38(0x26) bytes
File:        C:\WINDOWS\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
String:      Grumpy
Fields:
              MT    Field   Offset                 Type VT     Attr            Value Name
00007ffb78f6c158  400027b        8         System.Int32  1 instance                6 m_stringLength
00007ffb78f6a9c0  400027c        c          System.Char  1 instance               47 m_firstChar
00007ffb78f69808  4000280       c0        System.String  0   shared           static Empty
                                 >> Domain:Value  000002ce7d1bb8c0:NotInit  <<
```

So the cat is “Grumpy”.

You might want to investigate more to figure out what the threads were doing before ending up deadlocked.

1. The **!threads** command allows you to find the mapping between the 0x5 and 0x6 identifiers and the corresponding threads ids that will appear in other commands

```
0:009> ~
   0  Id: a2c.c64 Suspend: 0 Teb: 01199000 Unfrozen
   1  Id: a2c.3f24 Suspend: 0 Teb: 011a8000 Unfrozen
   2  Id: a2c.47ac Suspend: 0 Teb: 011ab000 Unfrozen
   3  Id: a2c.3f40 Suspend: 0 Teb: 011ae000 Unfrozen
   4  Id: a2c.481c Suspend: 0 Teb: 011b1000 Unfrozen
   5  Id: a2c.2148 Suspend: 0 Teb: 011bd000 Unfrozen
   6  Id: a2c.29f4 Suspend: 0 Teb: 011c0000 Unfrozen
   7  Id: a2c.b80 Suspend: 0 Teb: 011cf000 Unfrozen
   8  Id: a2c.38cc Suspend: 0 Teb: 011d2000 Unfrozen
   9  Id: a2c.45e4 Suspend: 0 Teb: 011d5000 Unfrozen
  10  Id: a2c.46c0 Suspend: 0 Teb: 011db000 Unfrozen
  11  Id: a2c.320c Suspend: 0 Teb: 011de000 Unfrozen
  12  Id: a2c.4bb8 Suspend: 0 Teb: 011e1000 Unfrozen
  13  Id: a2c.4434 Suspend: 0 Teb: 011e4000 Unfrozen
  14  Id: a2c.1ebc Suspend: 0 Teb: 011e7000 Unfrozen
  15  Id: a2c.4a80 Suspend: 0 Teb: 011ea000 Unfrozen
  16  Id: a2c.4590 Suspend: 0 Teb: 011f6000 Unfrozen
  17  Id: a2c.1838 Suspend: 0 Teb: 011fc000 Unfrozen
  18  Id: a2c.3990 Suspend: 0 Teb: 011ff000 Unfrozen
  19  Id: a2c.eb4 Suspend: 0 Teb: 01005000 Unfrozen
  20  Id: a2c.45a4 Suspend: 0 Teb: 01008000 Unfrozen
  21  Id: a2c.d30 Suspend: 0 Teb: 0100e000 Unfrozen
  22  Id: a2c.1b84 Suspend: 0 Teb: 01011000 Unfrozen
  23  Id: a2c.188c Suspend: 0 Teb: 01014000 Unfrozen
  24  Id: a2c.4594 Suspend: 0 Teb: 0101a000 Unfrozen
  25  Id: a2c.2cb8 Suspend: 0 Teb: 0101d000 Unfrozen
  26  Id: a2c.4650 Suspend: 0 Teb: 01020000 Unfrozen
```

The **~** WinDBG native command lists all threads (both native and managed) with an ID in the first column.

```
0:009> !threads
ThreadCount:      19
UnstartedThread:  0
BackgroundThread: 18
PendingThread:    0
DeadThread:       0
Hosted Runtime:   no
                                                                                  Lock
       ID OSID ThreadOBJ    State GC Mode     GC Alloc Context  Domain   Count Apt Exception
   0    1  c64 0168a678     a6028 Preemptive  039A31F4:00000000 0164d248 0     STA
   3    2 3f40 01697220     2b228 Preemptive  00000000:00000000 0164d248 0     MTA (Finalizer)
   9    3 45e4 0ed3e2d0  3029228 Preemptive  03831940:00000000 0164d248 0     MTA (Threadpool Worker)
  10    4 46c0 0ed68528   202b228 Preemptive  03834BC4:00000000 0164d248 0     MTA
  11    5 320c 06377a58  3029228 Preemptive  03832B9C:00000000 0164d248 1     MTA (Threadpool Worker)
  12    6 4bb8 06377fa0   202b228 Preemptive  03837684:00000000 0164d248 2     MTA
  13    7 4434 0637af00   202b228 Preemptive  03839A04:00000000 0164d248 0     MTA
  14    8 1ebc 0edc0f48   102a228 Preemptive  00000000:00000000 0164d248 0     MTA (Threadpool Worker)
  15    9 4a80 0edc88f0  3029228 Preemptive  03897170:00000000 0164d248 0     MTA (Threadpool Worker)
  16   10 4590 0edd0198  3029228 Preemptive  0394C424:00000000 0164d248 0     MTA (Threadpool Worker)
  17   11 1838 0edcd520  3029228 Preemptive  03950A14:00000000 0164d248 0     MTA (Threadpool Worker)
  18   12 3990 0edcda68  3029228 Preemptive  03954424:00000000 0164d248 0     MTA (Threadpool Worker)
  19   13  eb4 0eded7d0  3029228 Preemptive  03958A14:00000000 0164d248 0     MTA (Threadpool Worker)
  20   14 45a4 0edeff28  3029228 Preemptive  0395C424:00000000 0164d248 0     MTA (Threadpool Worker)
  21   15  d30 0edd64c0  3029228 Preemptive  03960424:00000000 0164d248 0     MTA (Threadpool Worker)
  22   16 1b84 0edd6a08  3029228 Preemptive  03964D0C:00000000 0164d248 0     MTA (Threadpool Worker)
  23   17 188c 0ee00af8  3029228 Preemptive  03968424:00000000 0164d248 0     MTA (Threadpool Worker)
  24   18 4594 0edfe600  3029228 Preemptive  0396CA18:00000000 0164d248 0     MTA (Threadpool Worker)
  26   19 4650 0ee01040  3029228 Preemptive  039A071C:00000000 0164d248 0     MTA (Threadpool Worker)
```

The **!threads** sos command lists only the managed threads with the same ID as **~** in the first column and a “managed” ID in the second. The latter is the one we are interested in to match which deadlocked threads (0x5 and 0x6) were found by the **!dlk** sosex command.

Use the id highlighted in red with the **~** command to change the active thread to that one. The prompt indicates you the id of the current active thread :

```
0:012> ~11s
eax=00000000 ebx=00000001 ecx=1013f2c8 edx=1013f2d8 esi=00000001 edi=00000001
eip=77c8232c esp=1013eac8 ebp=1013ec58 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll!NtWaitForMultipleObjects+0xc:
77c8232c c21400          ret     14h
```

```
0:011>
```

Then use the **!clrstack** command to display the callstack of the active thread. You can also use **!clrstack -a** to display the value of the method parameters and local variables when available.

```
0:011> !clrstack
OS Thread Id: 0x320c (11)
Child SP       IP Call Site
1013ee24 77c8232c [GCFrame: 1013ee24]
1013efc4 77c8232c [GCFrame: 1013efc4]
1013ef74 77c8232c [HelperMethodFrame: 1013ef74] System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
1013f004 55567458 System.Threading.Monitor.Enter(System.Object, Boolean ByRef) [f:\dd\ndp\clr\src\BCL\system\threading\monitor.cs @ 62]
1013f014 0329a5e4 StockMarket.MainWindow.ClearExpiredOrders()
1013f0c0 0329a281 StockMarket.MainWindow.OnTick()
1013f120 555cc29b System.Threading.Tasks.Task.InnerInvoke() [f:\dd\ndp\clr\src\BCL\system\threading\Tasks\Task.cs @ 2884]
1013f12c 555ca511 System.Threading.Tasks.Task.Execute() [f:\dd\ndp\clr\src\BCL\system\threading\Tasks\Task.cs @ 2498]
1013f150 555ca4dc System.Threading.Tasks.Task.ExecutionContextCallback(System.Object) [f:\dd\ndp\clr\src\BCL\system\threading\Tasks\Task.cs @ 2861]
1013f154 555675f4 System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object, Boole
1013f1c0 55567527 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object, Boolean) [f:\
1013f1d4 555ca292 System.Threading.Tasks.Task.ExecuteWithThreadLocal(System.Threading.Tasks.Task ByRef) [f:\dd\ndp\clr\src\BCL\system\threading\Tasks\Tas
1013f238 555ca137 System.Threading.Tasks.Task.ExecuteEntry(Boolean) [f:\dd\ndp\clr\src\BCL\system\threading\Tasks\Task.cs @ 2767]
1013f248 555ca07d System.Threading.Tasks.Task.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem() [f:\dd\ndp\clr\src\BCL\system\threading\Tasks\Task.c
1013f24c 5553db7d System.Threading.ThreadPoolWorkQueue.Dispatch() [f:\dd\ndp\clr\src\BCL\system\threading\threadpool.cs @ 820]
1013f29c 5553d9db System.Threading._ThreadPoolWaitCallback.PerformWaitCallback() [f:\dd\ndp\clr\src\BCL\system\threading\threadpool.cs @ 1161]
1013f4bc 5658f056 [DebuggerU2MCatchHandlerFrame: 1013f4bc]
```

2. If you want more details about each thread callstack, you could use **!clrstack -a** on each thread with the following cryptic command: **~*e!clrstack -a**

- **~** : target threads (both native and managed ones)
- ***** : all threads
- **e**: execute

*Note: Since **~** will execute the command on all threads, including non .NET threads, **clrstack** might not work on all threads: expect the following type of error*

```
OS Thread Id: 0x59bc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
```

How to avoid deadlocks?

Deadlocks issues are complicated to investigate because usually hard to reproduce and locks dependencies could span more than simply 2 threads like in this Lab. However, there is an easy way to avoid deadlocks: always try to acquire the different locks in the same order. That way, even if several locks need to be taken, you won't end up in a situation where two threads will hold the lock that the other one needs. Easier said than done (especially when using third party code) but keeping it in mind will always help… before trying writing lock-free kind of code. But this is another story.

Last but not least, as you've seen in this Lab, it is possible to get the reference of the objects used as locks. Instead of simple **Object** instances, feel free to use typed instances to more easily identify which locks and resources are involved in a deadlock.