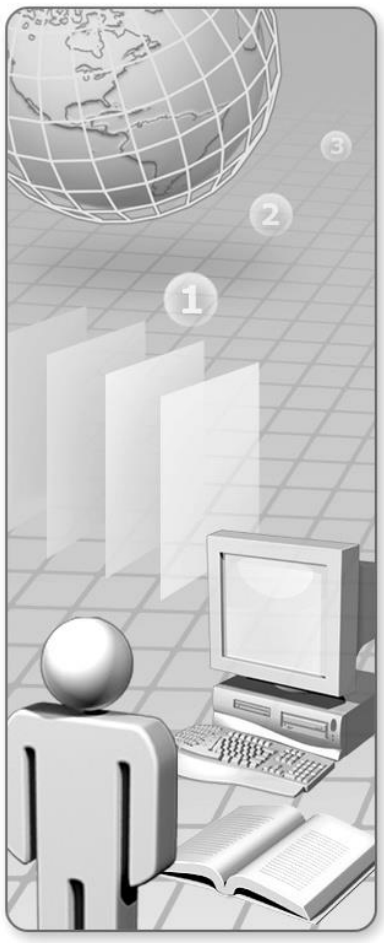
Microsoft | Services

Module 4: Troubleshooting Hangs



Lab 4: Troubleshooting Hangs

Objectives

After completing this lab, you will be able to:

- Use Perfmon, WinDbg, and SOS to troubleshoot a high CPU hang in a managed application.
- Use Perfmon, WinDbg, and SOS to troubleshoot a low CPU hang in a managed application.
- Understand how to investigate managed synchronization primitives within a debug session.
- Acquire a basic understanding of the common interop challenges that can hang applications

Estimated time to complete this lab: 105 minutes

Note: This lab focuses on the concepts in this module and as a result might not comply with Microsoft® security recommendations.

Exercise 1: High CPU hang

In this exercise, you will use Perfmon, Procdump, WinDbg, and the SOS debugger extension to diagnose and troubleshoot a high CPU hang. When you get close to pinpointing the problem, you will use the source code to identify the problem.

Task Description

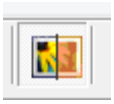
1. Here are the steps to reproduce the problem and we will repeat the steps a number of times during throughout the debugging process to capture relevant data that will help us identify the root cause.

   A. Open **FotoVision.exe** from the **<FotoVisionDir> \C#\Desktop\Bin** directory.

   B. In the left-hand pane, click the **Flowers** album.

   C. Click the picture of the **Wine Cup** flower.

   D. On the toolbar, click the **Photo Actions** button located sixth from the right.

   In the **Photo Actions** pane that comes up on the **Adjust** tab click the **Sepia** button.

   E. What you **should** see is that the picture immediately loses most of its color. If it takes more than one second, there is a problem.
   Does the picture lose its color?

2. Attempt to define the problem - While the application is not responding, do some preliminary troubleshooting and data gathering to see if you can get a problem definition. If you can do this, then you will be able to get the right tools configured in the right way to gather better data in preparation for the next time the problem occurs.

   A. While the application is unresponsive and the picture of the flower still appears in full color, open Task Manager.

   B. On the **Processes** tab, sort by the **Image Name** column so that you can find FotoVision easily. In Task Manager, look at the columns for the FotoVision record—*which one stands out as signifying a problem?*

   C. Once the Wine Cup picture loses its color, close FotoVision (do not save the change you made to the image) and proceed to the next step. The picture should lose its color eventually after you click the Sepia button. The exact time will vary depending on the hardware specifications on the computer you're using.

3. Set up Perfmon. - Now that you have a better definition of the problem (high CPU hang), set up Perfmon to record current activity.

> **Note:** If you prefer to skip steps A to F in the following performance counter collection steps, use the Data Collector Set template provided in the C:\LabFiles\Lab_Hang\HighCPUHang.xml. Refer to your instructor for further guidance if needed on how to load the template.

   A. On the Start, click **Run**, and then type **Perfmon**

   B. In Perfmon, expand **Data Collector Sets** and right-click **User Defined** and click on New > Data Collector Set.

   C. Name it "**HighCPUHang**" and select the radio button "Create manually (Advanced) option. Click Next to continue.

   D. Select the checkbox "Performance counter" under the Create data logs radio button. Click Next to continue.

   E. Click Add… and add the following objects

      i. Select the **Process** object and then **<All Instances>**. Click **Add >>** so that the counter gets added in the Added counters list.

      ii. Repeat the previous steps for **.NET CLR Memory** object.

      iii. Click OK close the dialog.

   F. Change the Sample interval to 1 second and then click Next to continue.

   G. Specify the directory %systemdrive%\PerfLogs\Admin\HighCPUHang if it is not already done so as default.

   H. Leave the **Run as** setting on **<Default>** and click Finish.

   I. Click on the **HighCPUHang** collector set on the left pane that you have just defined. Up on the Toolbar you will see a Play button. Start logging by clicking this button. Wait for the log to start before continuing to the next step (make sure the icon next to the collector set shows a small play button).

4. Now we will again reproduce the problem but this time we will also collect additional memory dumps of the FotoVision process.

   A. Start **FotoVision.exe** from the <FotoVisionDir>\C#\Desktop\Bin directory.

   B. In the left-hand pane, click the **Flowers** album.

   C. Click the picture of the **Wine Cup** flower.

D. On the toolbar, click the **Photo Actions** button located sixth from the right.

E. In order to get a dump while FotoVision is hanging, launch an administrator command prompt and type

procdump -c 90 -u -ma FotoVision.exe c:\temp

This will create a mini dump with heap **(-ma)** of the target process if the CPU of one Core is above 90% **(-c 90 –u)** for more than 10 seconds (default) and store it in c:\temp.
procdump can be found within **c:\labfiles\tools\sysinternals\suite** and is part of the Microsoft Sysinternals Suite.
Make sure that the target directory c:\temp exist.

F. In the **Photo Actions** pane that comes up on the **Adjust** tab click the **Sepia** button. You should an output like below:

```
procdump -c 90 -u -ma FotoVision.exe c:\temp

..

Process:               FotoVision.exe (4512)
CPU threshold:         >= 90% of single core
Performance counter:   n/a
Commit threshold:      n/a
Threshold seconds:     10
Number of dumps:       1
Hung window check:     Disabled
Exception monitor:     Disabled
Exception filter:      *
Terminate monitor:     Disabled
Dump file:             c:\temp\FotoVision_YYMMDD_HHMMSS.dmp

[12:07:29] CPU:            96%    1s
[12:07:30] CPU:            96%    2s
..
[12:07:38] CPU:            96%   10s

Process has hit CPU spike threshold.
Writing dump file c:\temp\FotoVision_150118_120738.dmp ...
Writing 182MB. Estimated time (less than) 6 seconds.
Dump written.

Dump count reached.
```

G. Wait 30 seconds to one minute after the dump completes and take another dump of the process:

procdump –ma FotoVision.exe c:\temp

This will give context and something to compare what values are changing.

H. Close the command window that was created in step 4F above.

I. Once the flower in FotoVision loses its color, close FotoVision and do not save the changes you made to the image.

J. Stop the Perfmon log.

5. Open and review the Perfmon log.

   We will examine the Perfmon log before we analyze the dump file. This is a good practice because this allows us to confirm that the dump was actually produced during the high CPU problem—not before or after. If Perfmon shows that the dump was not performed during the problem, then analyzing the dump file is a waste of time.

   A. In Perfmon, click **Performance Monitor** on the left-hand side.

   B. Right-click anywhere in the graph area and then click **Properties**.

   C. On the **Graph** tab, add Horizontal grid lines.

   D. On the **Source** tab, click **Log Files** and then click **Add**

   E. Browse to the location where you saved the Perfmon log (by default, c:\Perflogs) and click the log that corresponds to your reproduction of the problem.

   F. Click the **Time Range** button.

> **Note:** The Perfmon log may only be a few minutes long. When troubleshooting a production issue, it's very unlikely (and often undesirable) to have such a short log. But for our purposes in this lab, a few minutes is okay.

   G. On the **Data** tab, remove any counters that are included by default and click **Add** to add the following counters:

      i. From the **Process** object, add the **%Processor Time** counter for the **FotoVision**.

      ii. From the **.NET CLR Memory** object, add the **%Time in GC** counter for **FotoVision**.

      iii. When you click OK you will see all the counters you have added.

   H. Click **OK** to display the graph.

6. Examine the Perfmon log.

A. The first thing to do is to verify that the dump was performed during the high CPU hang. In the list of counters below the graph area, click the **%Processor Time** for the **FotoVision** instance and then press **Ctrl+H** to highlight the counter in the graph.

7. In the graph, you should see the highlighted counter spike to 80% or more, then dip down, then spike back up. The dip corresponds to the time in which the dump was performed.Examine the dump file.

   A. Open WinDbg.

   B. On the **File** menu, click **Open Crash Dump**. Browse to your first dump file, click it, and then click **Open**.

   C. If prompted to save your workspace, click **No**.

   D. Load SOS.

   E. Use the "**~**" command to see all the native threads within the process (the "**!Threads**" command is used to display the managed threads):

```
0:005> ~
   0  Id: 2c34.3468 Suspend: 0 Teb: 7eb9f000 Unfrozen
   1  Id: 2c34.30f4 Suspend: 0 Teb: 7eb9c000 Unfrozen
   2  Id: 2c34.37dc Suspend: 0 Teb: 7eb99000 Unfrozen
   3  Id: 2c34.30fc Suspend: 0 Teb: 7ea6f000 Unfrozen
   4  Id: 2c34.3558 Suspend: 0 Teb: 7ea6c000 Unfrozen
   5  Id: 2c34.37b4 Suspend: 0 Teb: 7ea69000 Unfrozen
   6  Id: 2c34.374c Suspend: 0 Teb: 7ea66000 Unfrozen
   7  Id: 2c34.15e8 Suspend: 0 Teb: 7ea63000 Unfrozen
   8  Id: 2c34.2f3c Suspend: 0 Teb: 7ea60000 Unfrozen
.  9  Id: 2c34.3500 Suspend: 0 Teb: 7ea5a000 Unfrozen
```

   F. WinDbg lists the threads in hexadecimal format. To convert this to hexadecimal which is what WinDbg uses just use the ? command:
   ? 0n<decimalValue>

   The third column in the "~" command output gives the process and thread ID. For example, the third column has a value of **2c34**.**3468** . Here **2c34**is the process ID and **3468** is the thread ID.

Note: Threads have a unique OS Thread ID. This value is recorded in the **ID Thread** counter in Perfmon. The debugger and Perfmon both enumerate the threads in a process using by counting them off (0,1,2,etc). This means that for the life of a thread it will have the same OS Thread ID but the count that it is associated with may change. For example if there are 10 threads and thread 5 terminates then the thread that was thread 6 will become thread 5 and so on. That is why it is always important to check the ID Thread value when verifying that you are looking at the same thread for a given time interval in Perfmon or back in the debugger.

In conclusion, the ID Thread value can change during the run of an application as threads stop. For instance, you may see the high CPU thread change from one thread to another in the Perfmon output. Notice that the ID Thread value changes at the same point as the high CPU spike.

Even if we did not have the Perfmon logs the **!runaway** debugger command displays information about the time consumed by each thread. In this example of output from the **!runaway** debugger command, notice that the threads are sorted by the amount of time they have consumed since they started. Open the second dump file and compare the **!runaway** output from both. Which thread has the highest delta? In this case the threads with the highest values and the largest delta should be the same but it is important to remember that this might not always be the case. In the output below, the call stacks of threads 3 and 0 would be of interest if there were a high CPU hang.

```
0:000> !runaway
 User Mode Time
  Thread       Time
   9:3500      0 days 0:00:07.265
   0:3468      0 days 0:00:00.640
   8:2f3c      0 days 0:00:00.062
   7:15e8      0 days 0:00:00.000
   6:374c      0 days 0:00:00.000
   5:37b4      0 days 0:00:00.000
   4:3558      0 days 0:00:00.000
   3:30fc      0 days 0:00:00.000
   2:37dc      0 days 0:00:00.000
   1:30f4      0 days 0:00:00.000
```

8. Switch to the problem thread and view its call stack. The first few frames may look something like:

```
0:000> ~9s
eax=00000000 ebx=0a43ee78 ecx=0000908d edx=00000000 esi=03c997a8 edi=03cd7000
eip=7454d8e8 esp=0a43ed50 ebp=0a43ed64 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
clr!WKS::gc_heap::adjust_limit_clr+0xf7:
7454d8e8 f3ab            rep stos dword ptr es:[edi]

0:009> k
ChildEBP RetAddr
06a7ec9c 74822a54 clr!WKS::gc_heap::adjust_limit_clr+0xf7
06a7ecdc 7455ac7d clr!WKS::gc_heap::a_fit_free_list_large_p+0x241
```

```
06a7ecf8 7455abce clr!WKS::gc_heap::loh_try_fit+0x1b
06a7ed58 7455aa89 clr!WKS::gc_heap::allocate_large+0x235
06a7ed7c 7454dbe0 clr!WKS::gc_heap::try_allocate_more_space+0x17a
06a7ed94 7455aa51 clr!WKS::gc_heap::allocate_more_space+0x18
06a7eddc 74552c1e clr!WKS::gc_heap::allocate_large_object+0x7b
06a7edf0 74453433 clr!WKS::GCHeap::Alloc+0x72
06a7ee10 7445b029 clr!Alloc+0x54
06a7eee0 7445a6b5 clr!AllocateArrayEx+0x2ce
06a7ef88 0a23482a clr!JIT_NewArr1+0x149
…
```

9. *What does it appear the thread was doing at the time the dump was performed?*
10. *How does this help you find the root cause of the problem?*

    Depending on what the process was doing at the exact moment in time when the dump was performed, your results may appear similar to the preceding results—a thread doing GC work.

The thread that has been using the majority of the CPU time was performing GC work, but this information does not help us identify the root cause.. When we suspect that the GC may be causing the problem we can check the **% time in GC** Perfmon counter. Before we do that let's take a quick look at another way to gauge the CPU usage of our threads in the dump file.

A. Our thread appears to be a managed thread even though the GC work is going on at the top of the thread. The last GC Frame (clr!JIT_NewArr1 is the function in the last frame in the sample output in step 7G) is followed by the message "WARNING: Frame IP not in any known module. Following frames may be wrong." You should see something similar in your callstack. Since we believe that managed code is executing on this callstack let us take a look at that code.

B. Check to see what managed calls are on the stack by using the "**!clrstack**" command

If the line of code is missing, make sure that your symbol path points to the directory with FotoVision.pdb:

.sympath+ C:\LabFiles\FotoVision\4.0\C#\Desktop\Bin
.reload

```
0:005!clrstack
OS Thread Id: 0x3500 (9)
Child SP       IP Call Site
0a43efd0 7454d8e8 [HelperMethodFrame: 0a43efd0]
0a43f05c 09ae60a1 *** WARNING: Unable to verify checksum for FotoVision.exe
FotoVisionDesktop.PhotoHelper.GetSepiaMatrix()
[c:\LabFiles\FotoVision\4.0\C#\Desktop\Source\util\PhotoHelper.cs @ 346]
0a43f0f8 09ae46da FotoVisionDesktop.OptimizeActions+<>c__DisplayClass0.<AdjustColor>b__2()
[c:\LabFiles\FotoVision\4.0\C#\Desktop\Source\util\OptimizeActions.cs @ 180]
0a43f180 736404e5 System.Threading.Tasks.Task.InnerInvoke()
```

```
0a43f18c 7364044a System.Threading.Tasks.Task.Execute()
0a43f1b0 73640419 System.Threading.Tasks.Task.ExecutionContextCallback(System.Object)
0a43f1b4 7360df17
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)
0a43f220 7360de66 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)
0a43f234 73640238
System.Threading.Tasks.Task.ExecuteWithThreadLocal(System.Threading.Tasks.Task ByRef)
0a43f298 73640157 System.Threading.Tasks.Task.ExecuteEntry(Boolean)
0a43f2a8 736400a3
System.Threading.Tasks.Task.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem()
0a43f2ac 7366894a System.Threading.ThreadPoolWorkQueue.Dispatch()
0a43f2fc 73668835 System.Threading._ThreadPoolWaitCallback.PerformWaitCallback()
0a43f524 74452652 [DebuggerU2MCatchHandlerFrame: 0a43f524]
```

C. The name of the function at the very top of the stack should look familiar. It looks like the call that occurs when you click the **Sepia** button.

D. So where do we go from here? It seems we are running a task on a worker thread (System.Threading.Tasks.Task.ExecuteEntry)
The most easy way to find out what thread has spawned the Task is using !gcroot.
This will tell us "who" has a reference to our task.

Use **!dso** to dump the Task Object on the thread (should be on the bottom) and **!gcroot** on the address to where we are coming from.

```
0:009> !dso
OS Thread Id: 0x3500 (9)
ESP/REG  Object   Name
0A43F07C 03f45208 System.Object[]    (System.Text.StringBuilder[])
0A43F084 03f45208 System.Object[]    (System.Text.StringBuilder[])
0A43F10C 02ce7174 System.Single[][]
..
0A43F150 02ce7f18 FotoVisionDesktop.OptimizeActions+<>c__DisplayClass0
0A43F190 02ce7f58 System.Threading.Tasks.Task
0A43F1B4 02ce7140 System.Threading.Thread
0A43F1C4 02ce7140 System.Threading.Thread
0A43F208 02ce7f58 System.Threading.Tasks.Task
0A43F21C 02ce7f58 System.Threading.Tasks.Task
0A43F230 02ce7f58 System.Threading.Tasks.Task
0A43F238 02ce6630 System.Threading.Tasks.TplEtwProvider
0A43F244 02ce7f58 System.Threading.Tasks.Task
0A43F2B4 02ce6f80 System.Threading.ThreadPoolWorkQueueThreadLocals
0A43F2B8 02ce6840 System.Threading.ThreadPoolWorkQueue
0A43F2CC 02ce7f58 System.Threading.Tasks.Task

0:009> !gcroot 02ce7f58
Thread 3468:
    00ede0f8 73643ebf System.Threading.Tasks.Task.SpinThenBlockingWait(Int32,
System.Threading.CancellationToken)
        ebp+30: 00ede0fc
            ->  02ce7f58 System.Threading.Tasks.Task
```

```
    00ede138 73674bd4 System.Threading.Tasks.Task.InternalWait(Int32,
System.Threading.CancellationToken)
        ebp+30: 00ede154
            ->  02ce7f58 System.Threading.Tasks.Task

    00ede190 73643d72 System.Threading.Tasks.Task.Wait(Int32,
System.Threading.CancellationToken)
        esi:
            ->  02ce7f58 System.Threading.Tasks.Task

    00ede1a4 09ae43d7
FotoVisionDesktop.OptimizeActions.AdjustColor(System.Drawing.Bitmap)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\util\OptimizeActions.cs @ 215]
        ebp+4c: 00ede1e0
            ->  02ce7f58 System.Threading.Tasks.Task

Thread 3500:
    0a43f18c 7364044a System.Threading.Tasks.Task.Execute()
        ebp+18: 0a43f190
            ->  02ce7f58 System.Threading.Tasks.Task
```

E. The first root is pointing to our main thread **Thread 3468**. Switch to the thread using the thread ID and dump the callstack to see the initial call:
~~[Thread ID]s
!clrstack

```
0:000> ~~[3468]s
eax=00000094 ebx=00000001 ecx=00000000 edx=00000000 esi=00000002 edi=00000002
eip=776eceec esp=00edda68 ebp=00eddbf0 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:
776eceec c21400          ret     14h
0:000> !clrstack
OS Thread Id: 0x3468 (0)
Child SP       IP Call Site
00eddf5c 776eceec [GCFrame: 00eddf5c]
00ede00c 776eceec [HelperMethodFrame_1OBJ: 00ede00c]
System.Threading.Monitor.ObjWait(Boolean, Int32, System.Object)
00ede090 736298ea System.Threading.Monitor.Wait(System.Object, Int32, Boolean)
00ede0a0 7363bcdb System.Threading.Monitor.Wait(System.Object, Int32)
00ede0a4 736441c6 System.Threading.ManualResetEventSlim.Wait(Int32,
System.Threading.CancellationToken)
00ede0f8 73643ebf System.Threading.Tasks.Task.SpinThenBlockingWait(Int32,
System.Threading.CancellationToken)
00ede138 73674bd4 System.Threading.Tasks.Task.InternalWait(Int32,
System.Threading.CancellationToken)
00ede190 73643d72 System.Threading.Tasks.Task.Wait(Int32,
System.Threading.CancellationToken)
00ede1a0 73643d47 System.Threading.Tasks.Task.Wait()
00ede1a4 09ae43d7
FotoVisionDesktop.OptimizeActions.AdjustColor(System.Drawing.Bitmap)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\util\OptimizeActions.cs @ 215]
00ede234 09ae3a8e FotoVisionDesktop.OptimizeActions.Apply(System.Drawing.Bitmap
ByRef, Single)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\util\OptimizeActions.cs @ 80]
```

```
00ede268 09ae379c FotoVisionDesktop.PhotoViewer.ApplyActionsToWorking()
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\controls\PhotoViewer.cs @ 513]
00ede2e8 09ae2fea
FotoVisionDesktop.PhotoViewer.ApplyPhotoAction(FotoVisionDesktop.ActionItem)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\controls\PhotoViewer.cs @ 223]
00ede300 09ae2f81
FotoVisionDesktop.PhotosPane.ApplyPhotoAction(FotoVisionDesktop.ActionItem)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\panes\PhotosPane.cs @ 645]
00ede338 09ae2ddb FotoVisionDesktop.MainForm.paneDetails_Action(System.Object,
FotoVisionDesktop.ActionEventArgs)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\MainForm.cs @ 2875]
00ede370 09ae2d4e FotoVisionDesktop.DetailsPane.photoActions_Action(System.Object,
FotoVisionDesktop.ActionEventArgs)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\panes\DetailsPane.cs @ 268]
00ede38c 09ae2c0b
FotoVisionDesktop.DetailsActions.OnNewAction(FotoVisionDesktop.ActionItem)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\controls\DetailsActions.cs @ 872]
00ede3c0 09ae6005
FotoVisionDesktop.DetailsActions.buttonSepia_Click(System.Object,
System.EventArgs)
[C:\LabFiles\FotoVision\4.0\C#\Desktop\Source\controls\DetailsActions.cs @ 686]
00ede3d8 53ab113e *** WARNING: Unable to verify checksum for
System.Windows.Forms.ni.dll
…
```

Clicking on **buttonSepia_Click** we have created a Task running on a worker thread causing a lot of CPU.

F. Looking into the sources of the Task can you see for the reason for the High CPU issue?

Did you find the root cause?

What type of work was taking up most of the CPU? Was this work the root cause, or was it a victim of the root cause?

Was your first lead—that is, the thread that was consuming the most CPU time—the one that revealed the root cause?

Note: It's common to pursue something in your data in hopes of finding the source of the problem. But sometimes your lead turns out to be a dead end and you have to find another clue that will lead you to the source of the problem. So if at first you don't succeed, try again!

Exercise 2: Low CPU Hang

In this exercise, you will use some of the skills and commands learned in the first exercise to help you diagnose and find the root cause of this problem. You will focus primarily on using SOS and WinDbg, and then you will find out the best way to resolve the problem without losing functionality of the application. Finally, you will test your problem resolution.

Task Description

1. Crop a picture.
   A. Close any open instances of FotoVision.exe and open a new instance.
   B. In the **Animals** album, click the picture of the **Easter Iguana**.
   C. On the toolbar, click the **Photo Actions** button.
   D. Click the **Crop** tab.
   E. In the picture, drag to make a selection that you'd like to crop to.
   F. Click the **Crop** button.
   G. You should see that your selection is now the only visible area of the picture and that the pixel dimensions next to the Original label have changed.
   Is that your experience?
2. Test other application functions and gather more information about the problem symptoms.
   A. On the **Adjust** tab, click the **Grayscale** button. Does it seem to work?
   B. Undo this change and then adjust some of the other settings on the **Adjust** tab. *Do they seem to work?*

      Other application functionality seems to be working, so the entire application isn't hung. Only the **Crop** button isn't responding appropriately (the selection you made isn't even reset), so the problem appears to be confined to that button.
   C. In Task Manager, view the **CPU** value for FotoVision.exe. You may want to click the **Crop** button once more while looking at Task Manager to see if the CPU usage spikes.

      Since Task Manager didn't reveal anything about CPU usage let us gather some addition data with Perfmon.
3. Gather information with Perfmon.
   A. Open Perfmon.
   B. Add all counters for the **Process** object, but only for the FotoVision.exe instance.

C. Keep the default settings that Perfmon puts in place (scale, graph max, and so forth).

D. Click the **Crop** button several times to see if any of the counters in Perfmon change with each click. You should notice one counter that changes—and keeps its new value (that is, it doesn't just spike up and down)—with each click.
Which counter is it?
Is it the Thread Count? .

4. Dump FotoVision.exe.

   A. Using procdump (or any other tool you like), dump FotoVision.exe.

5. Open the dump file and do some preliminary analysis.

   A. Open the dump file in WinDbg.

   B. Load SOS.

   C. The improper use of SyncBlocks can result in a low CPU hang. Only one feature is nonresponsive. Nevertheless, try running the "**!syncblk**" command just to see if it yields some insight into the root cause.

6. View the stacks.

   A. View the native call stack for each thread (**~*kb**). The number of threads you have in the dump will depend on the number of times you clicked the Crop button before dumping the process (see Step 3).

   Do you see a pattern in any of the stacks? What do many of them appear to be doing?
   Use **!uniqstack** to get a better overview

> The **!uniqstack** extension displays all of the stacks for all of the threads in the current process, excluding stacks that appear to have duplicates.

   B. Using the knowledge disclosed in this chapter about the WaitForMultipleObjectsEx API call, dump out some of the objects that some of the threads are waiting on to see if you can find a deeper clue as to what the problem is.

> The first argument to WaitForMultipleObjectsEx is the number of objects (possibly events) on which we're waiting. In this case, we are waiting on 2. The second argument is a pointer to the array of object handles for the objects on which we're waiting. If we dump them out, we can get the handle addresses.

   C. Below is some sample output:

```
0:045> kb
ChildEBP RetAddr  Args to Child
0777e82c 7537112f 00000002 0777eb30 00000001 ntdll!NtWaitForMultipleObjects+0xc
```

```
0777e9b8 74596e26 00000002 0777eb30 00000000
KERNELBASE!WaitForMultipleObjectsEx+0xcc
0777ea08 74596b85 00000000 ffffffff 00000001
clr!WaitForMultipleObjectsEx_SO_TOLERANT+0x3c
0777ea94 74596c76 00000002 0777eb30 00000000
clr!Thread::DoAppropriateWaitWorker+0x237
0777eb00 7452f32a 00000002 0777eb30 00000000 clr!Thread::DoAppropriateWait+0x64
0777ec5c 7365a242 00000000 00000001 02d71acc
clr!WaitHandleNative::CorWaitMultipleNative+0x26d
0777ec84 73d303ba 00000001 09ae7e84 00000000 mscorlib_ni+0x38a242
0777ecc0 7363d93b 02d6c75c 0777ed30 7360df17 mscorlib_ni+0xa603ba
0777eccc 7360df17 02d6c6f4 00000000 00000000 mscorlib_ni+0x36d93b
0777ed30 7360de66 00000000 02d6c728 00000000 mscorlib_ni+0x33df17
0777ed44 7360de31 00000000 02d6c728 00000000 mscorlib_ni+0x33de66
0777ed60 7363d8c4 02d6c728 00000000 00000000 mscorlib_ni+0x33de31
0777ed78 74452652 09cddff0 0777edd8 74461580 mscorlib_ni+0x36d8c4
0777ed84 74461580 0777ee24 0777edc8 745f619f clr!CallDescrWorkerInternal+0x34
0777edd8 7446e670 736d426c 0777edfc 7445df7b clr!CallDescrWorkerWithHandler+0x6b
0777ee58 7452e98d 0777ee90 2cc8a131 0777f0a4
clr!MethodDescCallSite::CallTargetWorker+0x152
0777efcc 745957a1 0777f148 09cddff0 0777f0ec
clr!ThreadNative::KickOffThread_Worker+0x173
0777efe4 7459580f 2cc8bfd1 0777f0ec 00000000
clr!ManagedThreadBase_DispatchInner+0x67
0777f088 745958dc 2cc8bfbd 7452e796 09cddff0
clr!ManagedThreadBase_DispatchMiddle+0x82
0777f0e4 74595984 00000001 00000000 00000001
clr!ManagedThreadBase_DispatchOuter+0x5b
0777f108 7452e849 00000001 00000002 2cc8bedd
clr!ManagedThreadBase_FullTransitionWithAD+0x2f
0777f184 7454814d 09cddff0 776f8d6c 0777f280 clr!ThreadNative::KickOffThread+0x1d1
0777fb24 754e7c04 0124ebf0 754e7be0 2d3eb7e7
clr!Thread::intermediateThreadProc+0x4d
0777fb38 7770b90f 0124ebf0 2f18acc2 00000000 kernel32!BaseThreadInitThunk+0x24
0777fb80 7770b8da ffffffff 776f06f1 00000000 ntdll!__RtlUserThreadStart+0x2f
0777fb90 00000000 74548104 0124ebf0 00000000 ntdll!_RtlUserThreadStart+0x1b
```

```
0:045> dd 0777eb30  L2
0777eb30  000003b0 000003b4
```

```
0:045> !handle 000003b0 f
Handle 000003b0
  Type                 Event
  Attributes           0
  GrantedAccess        0x1f0003:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         QueryState,ModifyState
  HandleCount          2
  PointerCount         28
  Name                 <none>
  No object specific information available
```

```
0:045> !handle 000003b4f
Handle 000003b4
  Type                 Event
```

```
  Attributes          0
  GrantedAccess       0x1f0003:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         QueryState,ModifyState
  HandleCount         2
  PointerCount        53
  Name                <none>
  No object specific information available
```

Were you able to find any helpful information (for example, common named events) after dumping the objects from more than one thread?

7. View the managed stacks for all managed threads. *Do some of them seem to share a pattern? What is it?* Keep in mind what you learned about managed synchronization objects in this module.

```
0:045> !clrstack
OS Thread Id: 0xf2c (16)
Child SP       IP Call Site
0777eb58 776eceec [HelperMethodFrame_1OBJ: 0777eb58]
System.Threading.WaitHandle.WaitMultiple(System.Threading.WaitHandle[], Int32,
Boolean, Boolean)
0777ec6c 7365a242
System.Threading.WaitHandle.WaitAny(System.Threading.WaitHandle[], Int32, Boolean)
0777ec90 73d303ba
System.Threading.WaitHandle.WaitAny(System.Threading.WaitHandle[])
0777ec94 09ae7e84 FotoVisionDesktop.DetailsActions.BackgroundSave()
[c:\LabFiles\FotoVision\4.0\C#\Desktop\Source\controls\DetailsActions.cs @ 725]
0777ecc8 7363d93b System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
0777ecd4 7360df17
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)
0777ed40 7360de66
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)
0777ed54 7360de31
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0777ed6c 7363d8c4 System.Threading.ThreadHelper.ThreadStart()
```

A. If you dump the objects on thread using **!dso** is there something interesting? – could the **WaitHandle** help you? Who is referencing it? Who should fire it?

```
0:045> !dso
OS Thread Id: 0xf2c (16)
ESP/REG  Object   Name
0777EB48 02cb02d0 System.Threading.ManualResetEvent
0777EB74 02cb02d0 System.Threading.ManualResetEvent
0777EC10 02d70648 System.String    C:\Users\wolfk\Documents\FotoVision\tmp.jpg
0777EC38 02d71acc System.Object[]    (System.Threading.WaitHandle[])
0777EC6C 02d71acc System.Object[]    (System.Threading.WaitHandle[])
0777EC70 02d71ab4 System.Object[]    (System.Threading.WaitHandle[])
0777EC78 02cd0cb0 System.Threading.ContextCallback
```

```
0777EC7C 02cb00e0 FotoVisionDesktop.DetailsActions
0777EC98 02d71ab4 System.Object[]    (System.Threading.WaitHandle[])
0777EC9C 02d70648 System.String    C:\Users\wolfk\Documents\FotoVision\tmp.jpg
0777ECA0 02c92610 System.String    FotoVision
0777ECA4 02d70414 System.String    C:\Users\wolfk\Documents
0777ECA8 02d6dfb8 System.Drawing.Bitmap
0777ECAC 02cb00e0 FotoVisionDesktop.DetailsActions
```

B. to get the ThreadHandle you can use **!da -details** to dump the array and afterwards **!gcroot** to find the references

```
0:045>!da -details 02d71acc
Name:        System.Threading.WaitHandle[]
MethodTable: 736a5738
EEClass:     7333e43c
Size:        24(0x18) bytes
Array:       Rank 1, Number of elements 2, Type CLASS
Element Methodtable: 736e7738
[0] 02cb0278
    Name:        System.Threading.ManualResetEvent
    MethodTable: 736d421c
    EEClass:     73338ed8
    Size:        24(0x18) bytes
    File:
…<deleted>
[1] 02cb02d0
    Name:        System.Threading.ManualResetEvent
    MethodTable: 736d421c
    EEClass:     73338ed8
    Size:        24(0x18) bytes
    File:
…<deleted>

0:045>!gcroot 02cb0278
Thread 3468:
    00edeffc 53ad4cb8
System.Windows.Forms.Application+ComponentManager.System.Windows.Forms.UnsafeNativ
eMethods.IMsoComponentManager.FPushMessageLoop(IntPtr, Int32, Int32)
        ebp+50: 00edf028
            ->  02cd25b4 System.Windows.Forms.Application+ComponentManager
            ->  02cd2614 System.Collections.Hashtable
            ->  02cd2648 System.Collections.Hashtable+bucket[]
            ->  02cd2604
System.Windows.Forms.Application+ComponentManager+ComponentHashtableEntry
            ->  02c96168 System.Windows.Forms.Application+ThreadContext
            ->  02cc3df8 System.Windows.Forms.ApplicationContext
            ->  02c94f7c FotoVisionDesktop.MainForm
            ->  02c95700 System.Windows.Forms.PropertyStore
            ->  02cd1310 System.Windows.Forms.PropertyStore+ObjectEntry[]
            ->  02cbfef4 System.Windows.Forms.Form+ControlCollection
            ->  02cbff0c System.Collections.ArrayList
            ->  02cbff24 System.Object[]
            ->  02caa770 System.Windows.Forms.Panel
            ->  02caa820 System.Windows.Forms.PropertyStore
            ->  02cc3b44 System.Windows.Forms.PropertyStore+ObjectEntry[]
            ->  02cbe88c System.Windows.Forms.Control+ControlCollection
```

```
            ->  02cbe8a0 System.Collections.ArrayList
            ->  02cbe8b8 System.Object[]
            ->  02caf2b4 FotoVisionDesktop.DetailsPane
            ->  02caf3c0 System.Windows.Forms.PropertyStore
            ->  02cc3bb8 System.Windows.Forms.PropertyStore+ObjectEntry[]
            ->  02caf7d0 System.Windows.Forms.Control+ControlCollection
            ->  02caf7e4 System.Collections.ArrayList
            ->  02caf7fc System.Object[]
            ->  02cb00e0 FotoVisionDesktop.DetailsActions
            ->  02cb0278 System.Threading.ManualResetEvent

Thread e9c:
    06abeee8 [HelperMethodFrame_1OBJ: 06abeee8]
System.Threading.WaitHandle.WaitMultiple(System.Threading.WaitHandle[], Int32,
Boolean, Boolean)
        06abefc8
            ->  02d39b1c System.Object[]
            ->  02cb0278 System.Threading.ManualResetEvent

Thread 1010:
    06d7ece8 [HelperMethodFrame_1OBJ: 06d7ece8]
System.Threading.WaitHandle.WaitMultiple(System.Threading.WaitHandle[], Int32,
Boolean, Boolean)
        06d7edc8
            ->  02d45acc System.Object[]
            ->  02cb0278 System.Threading.ManualResetEvent

Thread 2ee8:
    06ffeab8 [HelperMethodFrame_1OBJ: 06ffeab8]
System.Threading.WaitHandle.WaitMultiple(System.Threading.WaitHandle[], Int32,
Boolean, Boolean)
        06ffeb98
            ->  02d4facc System.Object[]
            ->  02cb0278 System.Threading.ManualResetEvent)
And so on…
```

It seems that a lot of threads are using the same ManualResetEvent (and waiting for it?)

8. Open the solution.

    A. Sometimes it's easier to view the whole source file so that you can see member variables, the definition and implementation of surrounding functions, and so forth. So let's stop pretending that we don't have access to the source and open the solution in Visual Studio.

    B. Open the .sln file for FotoVision's desktop application.

    C. Go to the source file that contains the functions found in the managed call stacks from the preceding steps.

```
private void BackgroundCopy()
{
          eT1.Reset();
          imgCopy = (Image)pictCropCoord.Image.Clone();
          WaitHandle.WaitAny(new WaitHandle[] {    eT2,    MRE  });
}
```

```
private void BackgroundSave()
{
    eT2.Reset();
    //Make a copy and save it in case the user accidentally edits the original
    imgCopy.Save(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.
Personal), Application.ProductName, "tmp.jpg"));
    WaitHandle.WaitAny(new WaitHandle[] { eT1,  MRE  });
    //Display the cropped image
    OnNewAction(new ActionItem(PhotoAction.Crop, _cropBounds));
}
```

D. If the functions seem harmless to you, then see if you can determine who called each function by searching the source and using the callstack from WinDbg.

E. The dump file indicates that you are waiting on events, and you see events in the source code.
Where are event used in the code?
Are calls on the call stack using events?
What is causing the problem? Why are the events blocking our thread?
What might fix this problem? How can we signal the events?

9. Implement a resolution and test it.

A. There are several different ways to solve the problem, ranging from editing to deleting to moving code. Theorize on a solution and implement it.

B. Save the project and recompile. You'll have to shut down the existing instance of FotoVision.exe before recompiling.

C. Test your solution by cropping an image. Note that you don't have to save the image you edited to test your theory. All you're trying to do is get the crop functionality to work.

D. If you can't find a solution, group together with a teammate and discuss some other possibilities about how to resolve this problem.

E. Confirm the solution that you choose with the Instructor.

Exercise 3: Troubleshooting Intermittent Hangs

In this exercise, you will examine a very common hang problem with WinForm applications where the GUI becomes unresponsive but only very briefly and it does not hang completely. The frequent lag in UI responses often leads to a detrimental user experience over time.

Task Description

1. Intermittent hangs are arguably the hardest issues to troubleshoot since it is difficult to capture proper data for accurate diagnosis of the issue. Fortunately in this scenario, we have already localized scope of the hang and it is associated with a specific action performed in the application. Here are the steps to reproduce the problem.

   A. Open **FotoVision.exe** from the **<FotoVisionDir>\C#\Desktop\Bin** directory.

   B. In the left-hand pane, click the **Baby** album.

   C. Click the picture of the **Pink** baby.

   D. On the toolbar, click the **Photo Actions** button located sixth from the right.

   E. In the **Photo Actions** pane and on the **Adjust** tab, click and drag the slider values to the following:

Contrast	100
Brightness	-100
Midtones (gamma)	100
Saturation	-100

   Do you notice the lag in slider movement? If you drag one of the sliders back and forth continuously, does the slider follow your mouse drag?

2. Try and apply what you have learned in Exercise 1 & 2 to determine what type of hang this is. *Hint: Refer to Exercise 1 Step 2. Is this a high CPU hang?*

3. In the previous exercises, we have only shown you post mortem debugging by collecting memory dumps. For brief intermittent hangs, it is almost impossible to catch a memory dump during the hang. Hence, we will live debug the process which will allow us to inspect the state of the process more quickly.

> **Note:** Live debugging a server process is almost always a bad idea because one wrong mistake can lead to crashing the process affecting hundreds if not thousands of production users.

   A. Start **Windbg** and snap the window to the left by using the keystroke sequence Start + Left arrow.

B. Start **FotoVision.exe** and snap the window to the right by using the keystroke sequence Start + Right arrow.

C. Attach **Windbg** to the **FotoVision.exe** process. *Now you are setup to switch efficiently between Windbg & FotoVision which is essentially to break in quickly when the hang occurs. Read ahead the following instructions and consult with your instructor if you need help.*

D. In Windbg, type in 'g' and hit enter to allow FotoVision to continue running.

E. In FotoVision and on the left-hand pane, click the **Baby** album.

F. Click the picture of the **Pink** baby.

G. On the toolbar, click the **Photo Actions** button located sixth from the right.

H. In the **Photo Actions** pane that comes up on the **Adjust** tab, click and drag **any of the slider** to value **100**.

I. Quickly hit the **Break** button in **Windbg** to break into FotoVision.exe.

J. Type in the command "~0s" and then Enter.

K. Type in the command "k" to inspect the callstack of thread 0 which is the GUI thread.

L. If you see the following callstack, repeat steps H to J. That means you have broken in after the hang. Remember to use the "g" command before repeating. *This hang example is written to last about 2 seconds. You'll need to be quick with your hand eye coordination.*

```
   0  Id: 2154.17c0 Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr
003bebcc 58999aba USER32!NtUserWaitMessage+0x15
003bec60 5899956c
System_Windows_Forms_ni!System.Windows.Forms.Application+ComponentManager.System.W
indows.Forms.UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(IntPtr,
Int32, Int32)+0x402
003becbc 589993c1
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageL
oopInner(Int32, System.Windows.Forms.ApplicationContext)+0x16c
003becec 589236dd
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageL
oop(Int32, System.Windows.Forms.ApplicationContext)+0x61
003bed04 002a03f5
System_Windows_Forms_ni!System.Windows.Forms.Application.Run(System.Windows.Forms.
Form)+0x31
003bed2c 002a01ae FotoVision!FotoVisionDesktop.MainForm.Run()+0x95
003bed64 703f21bb FotoVision!FotoVisionDesktop.MainForm.Main()+0xce
003bed74 70414be2 clr!CallDescrWorker+0x33
003bedf0 70414d84 clr!CallDescrWorkerWithHandler+0x8e
003bef34 70414db9 clr!MethodDesc::CallDescr+0x194
003bef50 70414dd9 clr!MethodDesc::CallTargetWorker+0x21
003bef68 70517e1d clr!MethodDescCallSite::Call_RetArgSlot+0x1c
003bf0cc 70517f28 clr!ClassLoader::RunMain+0x24c
003bf334 70517d3f clr!Assembly::ExecuteMainMethod+0xc1
```

```
003bf818 70518131 clr!SystemDomain::ExecuteMainMethod+0x4ec
003bf86c 70518032 clr!ExecuteEXE+0x58
003bf8b8 705568b0 clr!_CorExeMainInternal+0x19f
003bf8f0 70a655ab clr!_CorExeMain+0x4e
003bf8fc 70b97f16 mscoreei!_CorExeMain+0x38
003bf90c 70b94de3 MSCOREE!ShellShim__CorExeMain+0x99
003bf914 750f339a MSCOREE!_CorExeMain_Exported+0x8
003bf920 773a9ed2 KERNEL32!BaseThreadInitThunk+0xe
003bf960 773a9ea5 ntdll!__RtlUserThreadStart+0x70
003bf978 00000000 ntdll!_RtlUserThreadStart+0x1b
```

M. However if you see the following callstack instead that means you have broken in just in time and are ready to proceed to the next steps.

```
0:000> k
ChildEBP RetAddr
003be208 753631bb ntdll!ZwDelayExecution+0x15
003be270 75363a8b KERNELBASE!SleepEx+0x65
003be280 0ff31dba KERNELBASE!Sleep+0xf
003be294 74b8586c CommonUtilSTA!CLog::Write+0x1a
003be2b0 74c005f1 RPCRT4!Invoke+0x2a
003be6b4 767faec1 RPCRT4!NdrStubCall2+0x2ea
003be6fc 7682ffd3 ole32!CStdStubBuffer_Invoke+0x3c
003be720 767fd876 OLEAUT32!CUnivStubWrapper::Invoke+0xcb
003be768 767fddd0 ole32!SyncStubInvoke+0x3c
003be7b4 76718a43 ole32!StubInvoke+0xb9
003be890 76718938 ole32!CCtxComChnl::ContextInvoke+0xfa
003be8ac 7671950a ole32!MTAInvoke+0x1a
003be8d8 767fdccd ole32!STAInvoke+0x46
003be90c 767fdb41 ole32!AppInvoke+0xab
003be9ec 767fe1fd ole32!ComInvokeWithLockAndIPID+0x372
003bea14 76719367 ole32!ComInvoke+0xc5
003bea28 76719326 ole32!ThreadDispatch+0x23
003bea6c 752162fa ole32!ThreadWndProc+0x161
003bea98 75216d3a USER32!InternalCallWinProc+0x23
003beb10 752177c4 USER32!UserCallWinProcCheckWow+0x109
003beb70 7521788a USER32!DispatchMessageWorker+0x3bc
003beb80 589a56dc USER32!DispatchMessageW+0xf
003bebcc 5899993f
System_Windows_Forms_ni!DomainBoundILStubClass.IL_STUB_PInvoke(MSG ByRef)+0x3c
003bec60 5899956c
System_Windows_Forms_ni!System.Windows.Forms.Application+ComponentManager.System.W
indows.Forms.UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(IntPtr,
Int32, Int32)+0x287
003becbc 589993c1
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageL
oopInner(Int32, System.Windows.Forms.ApplicationContext)+0x16c
003becec 589236dd
System_Windows_Forms_ni!System.Windows.Forms.Application+ThreadContext.RunMessageL
oop(Int32, System.Windows.Forms.ApplicationContext)+0x61
003bed04 002a03f5
System_Windows_Forms_ni!System.Windows.Forms.Application.Run(System.Windows.Forms.
Form)+0x31
003bed2c 002a01ae FotoVision!FotoVisionDesktop.MainForm.Run()+0x95
003bed64 703f21bb FotoVision!FotoVisionDesktop.MainForm.Main()+0xce
003bed74 70414be2 clr!CallDescrWorker+0x33
003bedf0 70414d84 clr!CallDescrWorkerWithHandler+0x8e
003bef34 70414db9 clr!MethodDesc::CallDescr+0x194
```

```
003bef50 70414dd9 clr!MethodDesc::CallTargetWorker+0x21
003bef68 70517e1d clr!MethodDescCallSite::Call_RetArgSlot+0x1c
003bf0cc 70517f28 clr!ClassLoader::RunMain+0x24c
003bf334 70517d3f clr!Assembly::ExecuteMainMethod+0xc1
003bf818 70518131 clr!SystemDomain::ExecuteMainMethod+0x4ec
003bf86c 70518032 clr!ExecuteEXE+0x58
003bf8b8 705568b0 clr!_CorExeMainInternal+0x19f
003bf8f0 70a655ab clr!_CorExeMain+0x4e
003bf8fc 70b97f16 mscoreei!_CorExeMain+0x38
003bf90c 70b94de3 MSCOREE!ShellShim__CorExeMain+0x99
003bf914 750f339a MSCOREE!_CorExeMain_Exported+0x8
003bf920 773a9ed2 KERNEL32!BaseThreadInitThunk+0xe
003bf960 773a9ea5 ntdll!__RtlUserThreadStart+0x70
003bf978 00000000 ntdll!_RtlUserThreadStart+0x1b
```

N. Review the callstack above carefully and see if you can spot the why thread 0 hang for about 2 seconds. *Did you see any code that went to sleep?What is the function name below it?*

Note: Thread 0 is the very 1st thread created when a process is born and it is also the default thread that handles window messages for a GUI application. We frequently nickname this thread the "GUI thread" and also it is often said this thread handles the "message pump".

Blocking the message pump is the main reason why GUI windows appear unresponsive to mouse click and keyboard typing. This usually leads to the window getting grayed out or looking like washed out. It is actually another window created by the Window Manager in kernel and is referred to as the ghost window appearing on top of the original hung window. Some of the common scenarios how a message pump gets blocked are for example long running database queries and forcing the thread to wait for an event.

4. So it would appear that the hang was caused by the common logging component that ran longer than expected hence blocking the message pump. However, you as the current application owner have specifically written code to perform logging in a background thread which should have prevented this in the first place.

   A. Open the logging source file **<FotoVisionDir>\C#\Desktop\Source\util\Logger.cs** to inspect the code:

```
    private static void Init()
    {
            workerThread = new Thread(new ThreadStart(Logger.DoLog));
            workerThread.Start();
            IsReady = true;
            mreIsReady.Set();
    }
```

```
    private static void DoLog()
    {
            string msg;
            string ThreadID;
            string ThreadApt;
            while ((!IsClosing)) {
                areLogIt.WaitOne();
                if ((IsClosing)) {
                    break; // TODO: might not be correct. Was : Exit While
                }
                ThreadID = string.Concat("Caller ThreadID ", Thread.CurrentThread.M
anagedThreadId, "\t");
                ThreadApt = string.Concat("Caller ThreadApt ", Thread.CurrentThread
.GetApartmentState(), "\t");
                lock (logq) {
                    msg = logq.Dequeue();
                }
                _u.Write(string.Concat(ThreadID, ThreadApt, msg));
            }
    }
```

   B. The function DoLog is where logging is done on a dedicated background thread created in the Init function when the Logger class is instantiated.

      So why is the logging still done in the main GUI thread then? Formulate your hypothesis and apply a code fix to see if you can remediate the problem.

> **Note:** Usually logging is done right at the same thread where the log function is called. In an effort to reduce execution time, a common strategy is to "offload" the responsibility such as logging to another thread.

5. Now try the following code changes and then recompile the FotoVisionDesktop project. Rerun the slider change scenario again to observe the difference.

   A. Close Windbg if you have not already done so. This will also kill FotoVision.

   B. Comment out the lines below "STA Logging component" and uncomment the lines after "MTA Logging component" and "using CommonUtilMTALib" such as the following:

```
//STA Logging component
//using CommonUtilSTALib;
//MTA Logging component
using CommonUtilMTALib;
 static Logger()
 {
     //STA Logging component
     //Log u = (Log)System.Activator.CreateInstance(System.Type.GetTypeFromCLSID(n
ew System.Guid("A4CFEE87-74F9-453D-99C3-DE15669EFE61")));

     //MTA Logging component
     Log u = (Log)System.Activator.CreateInstance(System.Type.GetTypeFromCLSID(new
 System.Guid("125B07AE-4A82-4C00-8CBA-B1503D44C714")));
```

   C. Hit F5 to build and debug FotoVision.

   D. Now repeat steps 1B to 1E or simply open a photo and the Photo Action pane to drag and change any one of the sliders. *Do you still see the lag while dragging continuously back and forth the sliders?*

6. To understand how the code changes the application's behavior, let's break in on the logging component. Earlier in this exercise (Step 3M), we found out that the delay was caused by the Write method in the logging component. Let's break on that function again to review the changes.

   A. Attach Windbg to the FotoVision process again

   B. Identify the exact Write method name in the logging component so that we can set a breakpoint on it. Run the command "x CommonUtilMTA!*Write*" such as the following:

```
0:011> x CommonUtilMTA!*Write*
62d31e00          CommonUtilMTA!CLog::Write (wchar_t *)
62d34040          CommonUtilMTA!_imp__WriteFile = <no type information>
```

   C. Set a breakpoint at the write function of the logging component.

```
0:011> bp CommonUtilMTA!CLog::Write
```

   D. Verify that the breakpoint has been set by listing all current breakpoints.

```
0:011> bl
 0 e 62d31e00     0001 (0001)  0:**** CommonUtilMTA!CLog::Write
```

E. Type in the command "g" and Enter to relinquish the debugger control back to FotoVision.

F. Now repeat steps 1B to 1E or simply open a photo and the Photo Action pane to drag and change any of the sliders. You should see the breakpoint hit immediately.

```
Breakpoint 0 hit
eax=60851e00 ebx=00000000 ecx=60854330 edx=005ba600 esi=02af3170 edi=085df280
eip=60851e00 esp=085df1d4 ebp=085df330 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
CommonUtilMTA!CLog::Write:
60851e00 55              push    ebp
```

G. Run the "k" command to inspect the callstack

> You could also use **!sosex.mk** for a complete (native + .NET) callstack (after loading sosex.dll)

```
0:010> k
ChildEBP RetAddr
0896f170 04826217 CommonUtilMTA!CLog::Write [c:\labfiles\lab1
(hang)\src\commonutilmta\commonutilmta\log.cpp @ 29]
WARNING: Frame IP not in any known module. Following frames may be wrong.
0896f320 7363d93b 0x4826217
0896f32c 7360df17 mscorlib_ni+0x36d93b
0896f390 7360de66 mscorlib_ni+0x33df17
0896f3a4 7360de31 mscorlib_ni+0x33de66
0896f3c0 7363d8c4 mscorlib_ni+0x33de31
0896f3d8 74452652 mscorlib_ni+0x36d8c4
0896f3e4 74461580 clr!CallDescrWorkerInternal+0x34
0896f438 7446e670 clr!CallDescrWorkerWithHandler+0x6b
0896f4b8 7452e98d clr!MethodDescCallSite::CallTargetWorker+0x152
0896f62c 745957a1 clr!ThreadNative::KickOffThread_Worker+0x173
0896f644 7459580f clr!ManagedThreadBase_DispatchInner+0x67
0896f6e8 745958dc clr!ManagedThreadBase_DispatchMiddle+0x82
0896f744 74595984 clr!ManagedThreadBase_DispatchOuter+0x5b
0896f768 7452e849 clr!ManagedThreadBase_FullTransitionWithAD+0x2f
0896f7e4 7454814d clr!ThreadNative::KickOffThread+0x1d1
0896f904 754e7c04 clr!Thread::intermediateThreadProc+0x4d
0896f918 7770b90f KERNEL32!BaseThreadInitThunk+0x24
0896f960 7770b8da ntdll!__RtlUserThreadStart+0x2f
0896f970 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Can you tell what thread the Write method is now running on? How is this different from before the code changes?

H. Now switch to the GUI thread and review the callstack there.

Does the stack have the Write method running on it?

I. Run the "g" command and repeat steps 6F to 6H a few times to observe the callstack of the GUI & Logger thread.

Discuss your findings with the instructor and other members in the class to find out what their conclusions are.

Note: A marked difference between.NET and COM is that .NET objects are basically unsafe and synchronized access to objects is specifically implemented by application developers. COM components however can be designed entirely without synchronization in mind but still leverage the synchronized access. This is the STA model and overwhelmingly components are built to run in STA apartments. STA objects created on a given thread can only be accessed by the same thread. Hence, access to the COM object is effectively serialized.

In this exercise, even though the .NET implementation of logging was designed to run on a background thread, the native Log component however uses STA apartment. Because logging was initially instantiated from the GUI thread which by convention is an STA thread, the native log component created conveniently forces all subsequent access only through the GUI thread. Hence, it effectively negates the benefit of the .NET background logging thread since every write has to be marshall back to the GUI thread. In conclusion, it is the slow write method combined with running it on the GUI thread that causes the brief hangs.

Final note, the code introduced to fix the issue essentially uses an MTA version of the native log component. Hence shifting the native logging write method to be callable right on the .NET background thread itself. It is also worth noting that the code in the native log component is essentially the same, just that the project type is different. The STA project was compiled with an apartment model of "Apartment" and the MTA with "Free". Compare the projects in C:\LabFiles\lab1 (hang)\src.