.NET Debugging – Exercises

Please note: All the exercises in this document are built in a sequential style. In other words, you should perform the exercise steps in order, and answer the questions that appear in the exercise text before proceeding to the subsequent steps. If you get stuck or don't remember what a command does, consult the documentation (WinDbg Help | Contents or the SOS *!help* command) or ask the instructor for assistance.

Exercise 0: Configuring Symbols

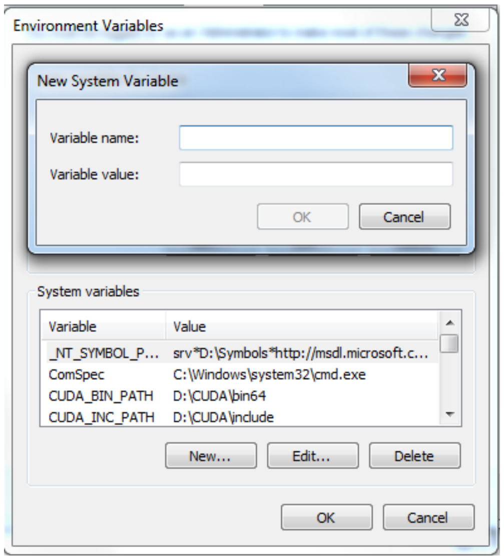
Before you can start debugging applications on your machine, you need to configure debugging symbols correctly. Follow these steps to configure symbols:

1. On a Windows XP machine, navigate to My Computer → Properties → Advanced → Environment Variables.

   On a Windows Vista or Windows 7 machine, navigate to My Computer → Properties → Advanced System Settings → Environment Variables.
2. Create a new system environment variable. Configure it as follows:

   (**Note**: In a closed environment, i.e. where Internet access to the Microsoft symbol servers is not available, the instructor will provide you with another value if necessary.)

```
Name: _NT_SYMBOL_PATH
Value: srv*C:\Symbols*http://msdl.microsoft.com/download/symbols
```



Variable	Value
_NT_SYMBOL_P...	srv*D:\Symbols*http://msdl.microsoft.c...
ComSpec	C:\Windows\system32\cmd.exe
CUDA_BIN_PATH	D:\CUDA\bin64
CUDA_INC_PATH	D:\CUDA\include

3. Close any debugger processes or windows you might have open.

Exercise 1: Diagnosing an Application Crash

In this exercise, you will obtain a crash dump of an application and determine the root cause for the crash using WinDbg and SOS.

1. Run the *FileExplorer.exe* application from the *1_Crash\Binaries* folder.

   The application crashes during its initialization sequence.

2. Use ADPlus (from the Debugging Tools for Windows package) to capture a crash dump of the application. Use the –crash command line switch and the –sc parameter to configure the executable to launch, as follows:

   ```
   ADPlus -crash -o <dump directory> -sc <path to FileExplorer.exe>
   ```

   The dump file will be stored in a directory under the dump directory path that you passed to the *-o* command-line switch.

   For example, if your command line included:

   ```
   … -crash -o D:\Temp …
   ```

   Then the dump files will appear in a directory similar to the following:

   ```
   D:\Temp\20100328_093635_CrashMode
   ```

3. Use Visual Studio 2010 to open the dump file generated by ADPlus. You can drag the file to the Visual Studio window.

   After the Minidump Summary window renders, click the “Debug with Mixed” green button on the right. You might be prompted for source code locations, in which case direct Visual Studio to the *1_Crash\Sources\FileExplorer* folder.

   You should see the offending line of code and understand the root cause of the exception. If code does not appear, you might have to configure the symbol path to *FileExplorer.pdb* (Tools → Options → Debugging → Symbols).

4. Use WinDbg (from the Debugging Tools for Windows Package) to open the dump file generated by ADPlus.

   Open WinDbg, and then use the File → Open Crash Dump menu command and navigate to the location of the crash dump file. If you see more than one crash dump, ignore the crash dumps that have Process_Shutdown as part of their name (these are dumps of the process shutdown sequence, and they are “too late” in the crash process to be diagnosed). Note that the name of the dump file might include the exception that occurred.

5. Use the following WinDbg and SOS commands to view the call stack of the main thread:

```
~0s
.loadby sos clr
!CLRStack -a
!PrintException
```

The *~0s* command switches to the first thread (main thread) in the debugger.

The *.loadby* command loads the SOS debugging extension into your debugger session.

The *!CLRStack* command displays the managed call stack of the current thread. This command might take a long time to complete in this particular case (and generally if the stack trace is very long). You can interrupt any debugger command by pressing *Ctrl+Break* repeatedly. **Note**: See below (Step 9) for instructions in the case *!CLRStack* does not display a complete stack trace.

The *!PrintException* command displays the current managed exception of the thread, if present. This command will not work for all types of exceptions, particularly the exception type you're seeing in this case.

6. Inspect the parameters passed to each invocation of the *RecursivelyFillTreeview* method on the call stack. Specifically, you might be interested in the *path* parameter. After identifying the address of the parameter, pass it to the *!do* command. For example, if the address of the *path* parameter were *0x019fa3fc* then you should use the following command to inspect its value:

```
!do 0x019fa3fc
```

7. What is the exception that was thrown? What is the call stack?

   Even though *!PrintException* might not report accurate information, go back to the beginning of the WinDbg debugging session and try to locate the exception information in the debugger output. Look for a section that begins with the following text:

```
This dump file has an exception of interest stored in it.
```

8. **Optional:** Use the *!help* command to see some other SOS commands. Feel free to experiment with these commands.
9. **Optional:** Use .NET Reflector to open the FileExplorer.exe application and look for the underlying cause for the exception you've experienced.
10. If *!CLRStack* displays a "trimmed" stack trace with only one or two stack frames, you can try the *k* command to view the unmanaged stack trace. To match an unmanaged stack frame with a managed function name, pass the third address on each row of the output to the *!u* command. For example, if the *k* output was:

```
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be
wrong.
00413008 003007f4 0x3007b6
00413044 003007f4 0x3007f4
00413080 003007f4 0x3007f4
004130bc 003007f4 0x3007f4
004130f8 003007f4 0x3007f4
00413134 003007f4 0x3007f4
…trimmed for brevity…
```

…then you can use the *!u* command to view the disassembly of the method surrounding the address 0x3007f4. At the top of the disassembly output you will see the name of the managed method:

```
0:000> !u 0x3007f4
Normal JIT generated code
FileExplorer.MainForm.RecursivelyFillTreeview(
             System.Windows.Forms.TreeNode, System.String)
Begin 003007b0, size de
003007b0 55              push    ebp
003007b1 8bec            mov     ebp,esp
003007b3 83ec30          sub     esp,30h
…trimmed for brevity…
```

If the *k* command displays a trimmed call stack with only one frame, you have to perform a manual stack walk. Recall that the EBP register (usually) points to the saved value of EBP from the previous frame, and that the memory location immediately following the saved EBP value is the return address.

Locate the value of the EBP register using the *r ebp* command. Pass this value to the *dd* command to view the memory contents to which EBP is pointing. The following value is (supposedly) the return address which you can pass to the *!u* command, similarly to what you did above. This process can be repeated to reconstruct the entire stack trace. For example:

```
0:000> r ebp
ebp=00413008
0:000> dd 00413008 L2
```

```
00413008  00413044 003007f4
0:000> dd 00413044 L2
00413044  00413080 003007f4
0:000> dd 00413080 L2
00413080  004130bc 003007f4
```

Finally, it might be the case that the EBP register itself is corrupted. In this case, you will have to look at the stack manually (e.g. in the WinDbg View | Memory window) starting from the ESP register and try to find something that looks like a return address (that *!u* would recognize). When you find such a value, the preceding memory location is typically the saved EBP value—you can use this value to perform the above reconstruction procedure. This process is rather tricky, so if you get stuck consult the instructor or wait for the exercise review.

Exercise 2: Opening a Post-Mortem Crash Dump

In this exercise, you will open an existing crash dump provided by the instructor and perform basic diagnostics to determine the source of the crash.

1. Use Visual Studio 2010 to open the dump file located in the *2_PostmortemCrash* folder. (There may be multiple files for different Windows versions—pick the one appropriate to your version.) You can drag the file to the Visual Studio window.
   After the Minidump Summary window renders, click the "Debug with Mixed" green button on the right. You might be prompted for source code locations, in which case direct Visual Studio to the *2_PostmortemCrash\Sources\FileExplorer* folder.
   You should see the offending line of code and understand the root cause of the exception. If code does not appear, you might have to configure the symbol path to *FileExplorer.pdb* (Tools → Options → Debugging → Symbols).
2. Use WinDbg to open the crash dump located in the *2_PostmortemCrash* folder. (There may be multiple files for different Windows versions—pick the one appropriate to your version.)
   Open WinDbg, and then use the File → Open Crash Dump menu command and navigate to the location of the crash dump file.
3. Use the following WinDbg and SOS commands to inspect the offending thread and the exception that occurred in it:

```
.loadby sos mscorwks
!Threads
~<thread number>s
!CLRStack -a
!PrintException
```

   The *.loadby* command loads the SOS debugging extension into your debugger session.
   The *!Threads* command displays a list of all managed threads you have in your application. The first number in each row is the debugger thread number—you should use this number for all other thread-related commands.
   The *~<thread number>s* command switches to the specified thread in the debugger. Note that you have to use the thread number as displayed by the debugger by the *!Threads* command, and not the Windows or CLR thread ID.

The *!CLRStack* command displays the managed call stack of the current thread. This command might take a long time to complete. You can interrupt any debugger command by pressing *Ctrl+Break* repeatedly.

The *!PrintException* command displays the current managed exception of the thread, if present.

4. What is the exception that occurred in the application? In which thread did the exception occur? What would you recommend to do to further diagnose the problem?
5. Use the *!u* command and pass to it the address displayed in the *IP* column in the *!PrintException* command output:

```
StackTrace (generated):
      SP          IP          Function
      <address>   <address>   <function name>
```

In the resulting annotated disassembly output, try to locate the instruction referred to by the *IP* column (by its address) and understand precisely what part of the code causes the exception.

6. **Optional:** Use .NET Reflector to open the FileExplorer.exe application and look for the underlying cause for the exception you've experienced.

Exercise 3: Diagnosing a Managed Deadlock

In this exercise, you will use a hang dump to diagnose a managed deadlock in an application.

1. Run the *FileExplorer.exe* application from the *3_Hang\Binaries* folder.
2. Click on a directory in the directory tree on the left. If the list view on the right is empty, choose another directory (specifically, one with files in it).
3. Double click one of the files in the list view on the right. A Notepad window should open with the contents of the file. However, the main application hangs—the UI stops responding and there is no visible progress.
4. Use ADPlus to capture a hang dump of the application. Use the –hang command line switch and the –pn parameter to configure the process name to capture. For example:

   ```
   ADPlus -hang -o D:\Temp -pn FileExplorer.exe
   ```

   The dump file will be stored in a directory under the dump directory path that you passed to the *-o* command-line switch.

   For example, if your command line included:

   ```
   … -hang -o D:\Temp …
   ```

   Then the dump files will appear in a directory similar to the following:

   ```
   D:\Temp\20100328_093635_HangMode
   ```

5. Use WinDbg (from the Debugging Tools for Windows Package) to open the dump file generated by ADPlus.

   Open WinDbg, and then use the File → Open Crash Dump menu command and navigate to the location of the dump file.

6. Use the following WinDbg and SOS commands to inspect the application's state:

   ```
   .loadby sos mscorwks
   !Threads
   ~<thread number>s
   !CLRStack -a
   !SyncBlk
   kb
   dd <memory address>
   !u <code address>
   ```

   The *.loadby* command loads the SOS debugging extension into your debugger session.

The *!Threads* command displays a list of all managed threads you have in your application. The first number in each row is the debugger thread number—you should use this number for all other thread-related commands.

The *~<thread number>s* command switches to the specified thread in the debugger. Note that you have to use the thread number as displayed by the debugger by the *!Threads* command, and not the Windows or CLR thread ID.

The *!CLRStack* command displays the managed call stack of the current thread. This command might take a long time to complete. You can interrupt any debugger command by pressing *Ctrl+Break* repeatedly.

The *!SyncBlk* command displays the list of "sync blocks", the CLR synchronization mechanisms that implement the *Monitor* class (and the C# *lock* keyword).

The *kb* command displays the current thread's unmanaged call stack.

The *dd* command displays a dump of memory as double-word values starting from the specified location.

The *!u* command displays annotated disassembly output for the specified code address. If the address points to managed code, additional information, such as managed method names, will be available.

7. Answer the following questions before you proceed:

   What are the application's threads doing? (**Hint:** There are two threads that participate in the deadlock. Identify them.)

   List the locks used by the application and the thread that owns each lock. (**Hint:** Look at the output of the *!SyncBlk* command.)

   What is the next step you need to perform to proceed with your diagnostics?

   Note: The *!CLRStack* command will not always report full call parameters, especially for methods such as *Monitor.Enter* that are implemented as JIT-intrinsics. The majority of this exercise is devoted to finding the value of the parameter passed to *Monitor.Enter*, which is the lock a thread is waiting for.

8. Switch to the main thread. Inspect the unmanaged call stack. You should see something similar to the following:

```
ChildEBP     RetAddr      Args to Child
…            …            …            …
<address>    <address>    <addresses>  mscorwks!JIT_MonEnterWorker_Portable…
```

```
<address>     <address>     <addresses>   <unresolved address>
<address>     <address>     <addresses>   System_Windows_Forms_ni…
```

Write down the first highlighted address—this is the *EBP* register value for the frame that calls into the CLR to perform the lock.

9. Use the *!CLRStack* command and locate the frame that has the second highlighted address in the *EIP* column of the command's output. It should be the *listbox1_DoubleClick* method.
10. Use the *!u* command and pass to it the second highlighted address. This will display an annotated disassembly of the method's code. Look for the region similar to the following (use the address and the >>> sign to guide you):

```
<address>     <opcode>      mov ecx,dword ptr [ebp-40h]
<address>     <opcode>      call mscorwks!JIT_MonEnterWorker (<address>)
>>>
```

11. Note that the address of the object to synchronize on (i.e., the object the thread waits for) is stored in the *ECX* register and retrieved from the thread's stack—the *EBP* register points to the thread's stack, and *-40h* is the memory offset.

    Unfortunately, if you attempt to view the value of the *ECX* register or the value of the *EBP* register, these values might be out of date because there are more stack frames on the call stack that could have overwritten their values.
12. Recall the highlighted *EBP* address from one of the previous steps. Use the *dd* command and pass to it this address with the *-40h* offset, e.g.:

```
dd 0025efd8-0x40 L1
```

The output should contain two addresses, e.g:

```
0025edb8     01ac6a5c
```

The second address displayed is the address of the object used for synchronization, i.e. the object the thread waits on.

13. Run the *!SyncBlk* command again. The output should be similar to the following:

```
…
16     00435dcc   3   1      003cce20 130   0        01ac6a38   System.String
17     00435dfc   3   1      0473a028 154c 5         01ac6a5c   System.String
```

Locate the row that contains the address from the previous step. This is the synchronization object your thread is waiting on. (You can use the *!do* command to inspect its value.)

14. Repeat steps 8-13 for the second participating thread.

15. Draw a wait chain by using the information obtained in the previous steps.

    In your wait chain, there should be two kinds of nodes: Thread nodes and SyncObject nodes. There should be a link between a Thread node and the SyncObject node it waits on, and there should be a link between a SyncObject node and the Thread node that owns this object.

    After completing the wait chain, you should see a cycle, indicating a deadlock.

16. **Optional:** Suggest a deadlock avoidance (or prevention) mechanism that alleviates deadlocks such as the one you encountered in this application.

17. **Optional:** Try another approach to finding the value of the parameter passed to *Monitor.Enter* by using the *!dso* (Dump Stack Objects) command. If you are lucky, it may be one of the last parameters on the stack.

18. **Optional:** Download the SOSEX.DLL debugging extension from the Internet.

    Its 32-bit version is available at:

    ```
    http://www.stevestechspot.com/downloads/sosex_32.zip
    ```

    Its 64-bit version is available at:

    ```
    http://www.stevestechspot.com/downloads/sosex_64.zip
    ```

    Load this extension into your debugging session (using the *.load* command) and use its *!dlk* command to perform deadlock diagnostics. This command streamlines the process performed in this exercise and displays the managed deadlock information immediately.

19. **Optional:** Use .NET Reflector to open the FileExplorer.exe application and look for the underlying cause for the hang you've experienced.

Exercise 4: Diagnosing a Managed Memory Leak

In this exercise, you will use SOS.DLL to diagnose a managed memory leak.

1. Run the *FileExplorer.exe* application from the *4_MemoryLeak\Binaries* folder.
2. Run the Windows Performance Monitor by clicking Start → Run, typing *perfmon* into the text box and clicking Run.

   Use the + toolbar button to add the counter *# Bytes in All Heaps* from the *.NET CLR Memory* performance category. Select the *FileExplorer.exe* process in the instances list.

   Optional: You can also use Sysinternals Process Explorer to monitor managed memory usage. When double-clicking a managed process in Process Explorer, you can inspect its memory usage in the .NET tab.
3. Click on a directory in the directory tree on the left. If the list view on the right is empty, choose another directory (specifically, one with files in it).

   Repeat this step multiple times for different directories and watch the memory usage performance counter in the meantime. Conclude that the application is leaking managed memory.
4. Use WinDbg to attach to the application's process (File → Attach to a Process, or press *F6*)

   Note: The techniques used in the rest of this exercise are perfectly applicable to debugging a dump and not a live process. You can capture a hang dump of the application and perform your analysis on the hang dump. (See Exercise 3 for the steps you need to perform to capture a hang dump.)
5. Use the following commands to determine which objects are consuming most managed memory:

```
.loadby sos mscorwks
!dumpheap -stat
```

   The *.loadby* command loads the SOS debugging extension into your debugger session.

   The *!dumpheap* command displays memory usage statistics sorted by the total size of the object instances. The output should be similar to the following:

```
<address>  <count>  <size>     FileExplorer.MainForm+FileInformation
…
<address>  <count>  <size>     System.Object[]
…
<address>  <count>  <size>     System.String
```

6. Most of the memory is probably consumed by strings. However, it's quite difficult to diagnose a memory leak that is caused by strings alone, because strings are ubiquitously used by GUI applications for their internal bookkeeping.
   Let the application continue running (by using the *g* command or the *F5* keyboard shortcut), repeat step 3 multiple times so that the memory usage climbs even higher, and then break into the application again (use the *Ctrl+Break* shortcut in WinDbg) and repeat the previous commands. Compare the size and count information to what you obtained in the previous run. This should give you some insight as to the source of the memory leak.
   Note: If in step 4 you chose to perform your analysis on a dump and not a live process, then you should capture another dump of the application and compare the two dumps in two different WinDbg windows.
7. Answer this question before proceeding:
   What object instances should you focus on when trying to identify the source of the leak?
8. Pass the name of the type to the *!dumpheap -type* command to view all instances of *FileExplorer.MainForm+FileInformation* that are present on your GC heap. The output should be similar to the following:

```
…
<address>   <MT address>     <size>
<address>   <MT address>     <size>
<address>   <MT address>     <size>
Statistics: …
```

9. Use the *!gcroot* command to view the GC root chain for instances of this type. Pass several addresses obtained from the listing above to the *!gcroot* command, e.g:

```
!gcroot 09cc1810
```

   There should be two categories of outputs for some of the instances. Most instances should have only the second type of output, while some instances will also have the first type of output.

```
…09cc46e8(System.Windows.Forms.ListBox+ItemArray+Entry)->
09cc1810(FileExplorer.MainForm+FileInformation)
```

   and

```
…09cc46a8(System.EventHandler)->
09cc1810(FileExplorer.MainForm+FileInformation)
```

The command output contains a chain of references from a GC root to the object instance you requested. For example, in the previous two snippets, you see that the object is held by the items array of a WinForms list box and by an EventHandler delegate.
What's special about the instances that have the first type of output? You might want to inspect the objects using the *!do* command and look at the application's UI.
Hint: To view the file path string contained in the *FileInformation* objects, issue the *!do* command. In the resulting output, locate the highlighted address and pass it to the *!do* command.

```
MT          Field       Offset    Value    Name
6de688c0    400000a     4      …  094139c8 <Path>k__BackingField
6de688c0    400000b     8      …  09cc1824 <Name>k__BackingField
6de44eec    400000c     c      …  09cc468c <FirstFewLines>…
6de4914c    4000009     14     …  09cc46c8 FileInformationNeedsRefresh
```

10. Analyze at least ten samples from various addresses and attempt to reach a conclusion as to why the instances of *FileExplorer.MainForm+FileInformation* are not released even though they are no longer displayed in the UI.
    Note: This part of memory leak diagnostics might be fairly difficult. Often you would have to perform educated guesses and then verify them by looking at the application's source code. If you're stuck, don't despair—try asking someone else for an idea, ask the instructor for a hint, or wait for the exercise review.
11. **Optional:** Use .NET Reflector to open the FileExplorer.exe application and look for the underlying cause for the memory leak you've experienced.

Exercise 5: Diagnosing a Difficult Memory Leak

In this exercise, you will use SOS.DLL to diagnose a more difficult memory leak.

1. Run the *MemoryLeak.exe* application from the *5_DifficultMemoryLeak\Binaries* folder.
2. Run the Windows Performance Monitor by clicking Start → Run, typing *perfmon* into the text box, and clicking Run.

   Use the + toolbar button to add the counter *# Bytes in All Heaps* from the *.NET CLR Memory* performance category. Select the *MemoryLeak.exe* process in the instances list.

   Optional: You can also use Sysinternals Process Explorer to monitor managed memory usage. When double-clicking a managed process in Process Explorer, you can inspect its memory usage in the .NET tab.
3. Wait for a few seconds to understand the memory usage pattern of this application. Conclude that it leaks managed memory, but there are sometimes large chunks of memory that are being freed. Overall, however, the memory usage goes up.
4. Use WinDbg to attach to the application's process (File → Attach to a Process, or press *F6*)

   Note: The techniques used in the rest of this exercise are perfectly applicable to debugging a dump and not a live process. You can capture a hang dump of the application and perform your analysis on the hang dump. (See Exercise 2 for the steps you need to perform to capture a hang dump.)
5. Use the following commands to determine which objects are consuming most managed memory:

   ```
   .loadby sos mscorwks
   !dumpheap -stat
   ```

   The *.loadby* command loads the SOS debugging extension into your debugger session.

   The *!dumpheap* command displays memory usage statistics sorted by the total size of the object instances. The output should be similar to the following:

   ```
   <address>  <count>  <size>    MemoryLeak.Schedule
   <address>  <count>  <size>    MemoryLeak.Employee
   …
   <address>  <count>  <size>    System.Byte[]
   ```
6. Most of the memory is probably consumed by arrays of bytes.

   Let the application continue running (by using the *g* command or the *F5* keyboard shortcut), repeat step 3 multiple times so that the memory usage climbs even higher, and then break into the application again (use the *Ctrl+Break* shortcut in WinDbg) and repeat the previous commands. Compare the size and count information to what you obtained in the previous run. This should give you some insight as to the source of the memory leak.

Note: If in step 4 you chose to perform your analysis on a dump and not a live process, then you should capture another dump of the application and compare the two dumps in two different WinDbg windows.

7. Use the following command to view all instances of *byte[]* that are present on your GC heap:

```
!dumpheap -type System.Byte[]
```

The output should be similar to the following:

```
…
<address>   <MT address>      <size>
<address>   <MT address>      <size>
<address>   <MT address>      <size>
Statistics: …
```

8. Use the *!gcroot* command to view the GC root chain for byte array instances. Pass several addresses obtained from the listing above to the *!gcroot* command, e.g:

```
!gcroot 09cc1810
```

9. Analyze at least ten samples from various addresses and attempt to reach a conclusion as to why the instances of *byte[]* are not released.

   Hint: To streamline the analysis, you might want to use the following command, that runs a loop in the debugger itself and displays GC root information for each of the *byte[]* instances:

```
.foreach (obj {!dumpheap -type System.Byte[] -short}) ↳
{!gcroot obj; .echo ---}
```

   Note: There should be no line break in the command that you type into the debugger.

10. Answer the following questions before proceeding:

    What is stopping the byte array instances from being collected? (**Hint:** Refer to the slides on finalization in the GC module of this course.)

    What should be the next step to diagnose the root cause of the memory leak?

11. Use the *!FinalizeQueue* command to inspect the finalization queue. Determine how many object instances are currently waiting for finalization.

12. Use the *!Threads* command to determine which thread is the finalizer thread. The finalizer thread has the string *(Finalizer)* in the *Exception* column of the *!Threads* command output:

```
   ID OSID  ThreadOBJ   …     APT   Exception
0      …                      MTA
2      …                      MTA   (Finalizer)
```

13. Switch to the finalizer thread using the *~2s* command (substitute the actual thread number in your debugger session if necessary).
14. Issue the *!CLRStack* command to see the call stack of the finalizer thread. What is the finalizer thread doing?
15. Formulate the root cause of the memory leak by completing the following sentence. (The answer appears in a tiny font under the box.)

The application ____________________ *at a faster rate than* ______________
____________________ *is able to run their finalizers.*

(Answer: The application creates objects at a faster rate than the finalizer thread is able to run their finalizers.)

16. **Optional:** Confirm your diagnosis of the root cause by inspecting the IL of the *Employee* class that is responsible for the memory leak. Run the following command to obtain the method table address of the *Employee* class:

```
!Name2EE *!MemoryLeak.Employee
```

In the resulting output, copy the address that appears after the *MethodTable* string. Pass that address to the *!DumpMT -md* command to inspect the method table. For example:

```
!DumpMT -MD 00183370
```

In the resulting output, look for the *Employee.Finalize()* method and note its method descriptor—the address that appears in the *MethodDesc* column. Pass that address to the *!DumpIL* command to inspect the method's IL. For example:

```
!DumpIL 00183354
```

The IL stream should show you that the finalizer of this class calls the *Schedule.FreeData()* method. To inspect the IL of this method, repeat the previous steps for the *MemoryLeak.Schedule* class and its *FreeData* method.

Eventually, you will locate a direct call to the *Thread.Sleep* method. What is the timeout duration of the *Sleep* call?

17. **Optional:** Use .NET Reflector to open the FileExplorer.exe application and look for the underlying cause for the memory leak you've experienced.