Debugging

Estimated time for completion: 30-90 minutes

Overview:

There are three different labs in this module for debugging exceptions, threads and memory issyes. Select the lab(s) that best fit your interests – it is not expected that you will be able to complete all three in class.

Goals:

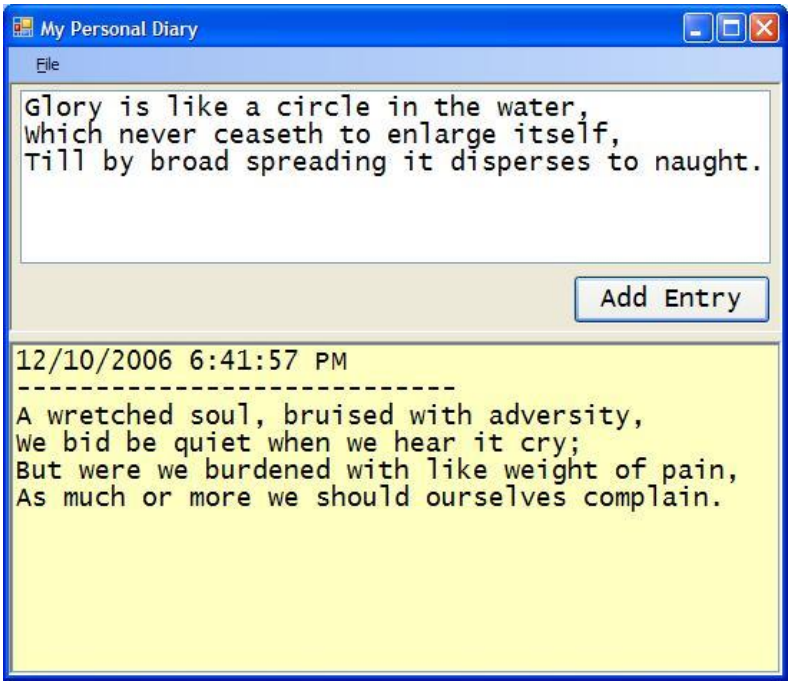
- Learn how to use debugging tool outside of Visual Studio
- Introduce memory dump analysis

Lab Notes

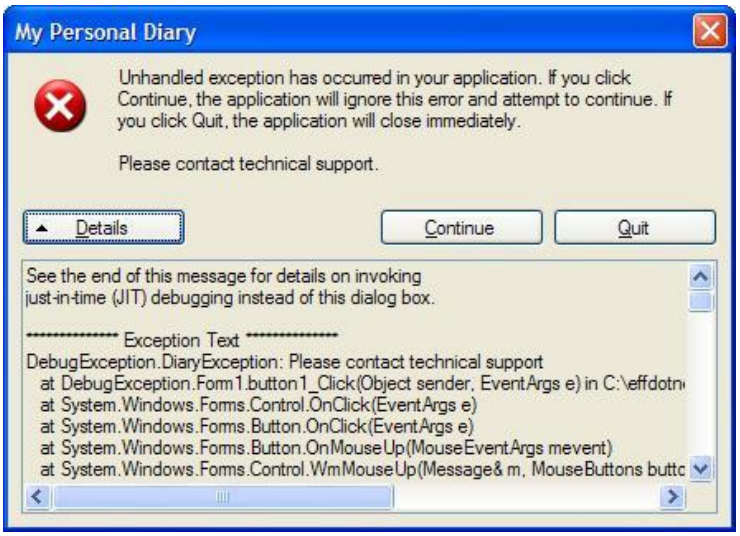
If you do not have debugging tools for windows on your machine you will need to download and install them fom Microsoft

Lab 1 – Debugging Exceptions

In this activity you will be working as a troubleshooter for the Personal Diary application shown here:



A user has submitted a bug report saying that if they clear the entries (File | Clear Entries) and then try to add a new entry they get the dialog box shown below:



For the purposes of this lab we will assume you don't have access to the source code or to Visual Studio .NET. Your task is to reproduce and fix this error using ADPLUS, WinDBG, and SOS

Part 1- Reproduce the problem

First you need to verify the problem

1. Run the **DebugException** application in the **DebugException\bin\debug** directory and add a few entries to the personal diary.

2. Clear the Entries (**File | Clear Entry**)
3. Try to add a new entry and verify that you get the same dialog box the user is reporting
4. Notice that the Exception text isn't very helpful.

Part 2 - Disable Application.ThreadException

Windows Forms applications suppresses unhandled exceptions by default using the Application.ThreadException handler. You will need to disable that so that unhandled exceptions are allowed to terminate the process, which will allow you to capture exception dumps.

1. Add an Application Configuration file either manually to the bin\debug directory or to the Visual Studio project (okay, you can use VS.NET for this part) so that it resembles the file below:

```
 <configuration>
   <system.windows.forms jitDebugging="true" />
</configuration>
```

2. Run the program outside of the debugger (**Shift-F5**) and verify that the program crashes now when the exception is thrown

---

Part 3 - Getting a memory dump

Now you're ready to capture a memory dump

1. Run the application again (in release mode). This time before you generate the exception start ADPLUS monitoring the application and generate a full memory dump when the exception occurs. You can do this by running

```
   C:\> ADPLUS -crash -o c:\temp -pn debugException.exe
```

2. ADPLUS will show a dialog or two and you should see another command window appear minimized.
3. Now press **Enter** and run the application. This time when it hits the exception you should not see the usual dialog box but rather you should see the debugger work for a bit and disappear. At this point adplus has generated a dump of your process suitable for loading into windbg.
4. Run windbg and choose File | Open CrashDump. You should find a new folder under c:\Program Files\Debugging Tools For Windows containing your crash dump files.
5. Inside that folder you should find a few files. Open whichever one has 2nd_Chance_NET_CLR_Full in the file name.
6. Windbg won't do a whole lot to indicate that the file is open, so select View | Command Window to get to the command window. This is where you will spend all your time. Once in the command window and type the command:

```
.loadby sos clr
```

7. Nothing will really happen here but you should now have SOS running inside of WinDbg. To verify this run **!help** and you should see help for SOS commands about like so:

```
0:003> .load sos
0:003> !help
-------------------------------------------------------------
SOS is a debugger extension DLL designed to aid in the debugging of managed
   programs. Functions
are listed by category, then roughly in order of importance. Shortcut names
   for popular functions
are listed in parenthesis. Type "!help <functionname>" for detailed info on
   that
function.
```

8. That's it! You can always follow this process to capture a memory dump of a misbehaving application. Now that you've done the work of capturing the dump you can analyze it to see what is causing the problem.

Part 5 - Analyzing a crashdump

The final step is to analyze the exception stored in the crashdump.

1. Use the **!pe** SOS command to dump the exception stored in the crash dump. You should see output similar to this:

```
0:000> !pe
Exception object: 023b1814
Exception type: DebugException.DiaryException
Message: Please contact technical support
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0012EF78 02231133 DebugException.Form1.button1_Click
    0012F044 7B060A6B System.Windows.Forms.Control.OnClick
    0012F054 7B105379 System.Windows.Forms.Button.OnClick
    0012F060 7B10547F System.Windows.Forms.Button.OnMouseUp
    0012F084 7B0D02D2 System.Windows.Forms.Control.WmMouseUp
    0012F0D0 7B072C74 System.Windows.Forms.Control.WndProc
...

StackTraceString: <none>
HResult: 80131500
There are nested exceptions on this thread. Run with -nested for details
```

2. Note that SOS is reporting that there is a nested exception stored as well - something the error dialog box shown to the user does not disclose. Run **!pe** again, this time specifying the **-nested** parameter to reveal the nested exception:

```
Nested exception -------------------------------------------------------------
   -
Exception object: 023b13f4
```

```
Exception type: System.IO.FileNotFoundException
Message: Could not find file
   'C:\labs\DebuggingExceptions\work\bin\Debug\diary.txt'.
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0012EE80 796536A2 System.IO.__Error.WinIOError
    0012EEE0 7936F43B System.IO.FileStream.Init
    0012EFD4 7937240A System.IO.FileStream..ctor
    0012EFF0 02230FA4 DebugException.Form1.button1_Click
StackTraceString: <none>
HResult: 80070002
```

3. This gives you a lot more to go on - in this case you can see that the file "diary.txt" appears to be missing when the app tries to write a new entry, and the stack trace pretty clearly shows that the system is trying to open the file in response to the button click.
4. Finally, take a look at the code and see if you can determine the problem. Can you fix the application?
5.
6.
7.

Lab 2 – Debugging Threads

Open runaway.sln in the DebuggingThreads folder. This application consists of two projects. `TimeServer` is a console-mode application that serves up time information (you don't really need to know how just yet). `Clock` is an application that gets data from the server and displays the current time in a label. There is a problem with this system though. If you let it run for a bit and then stop TimeServer.exe you'll see your CPU utilization go to, in effect, 100% of one core in Task Manager and the culprit will be Clock.exe. (Note: Be sure to kill the server by pressing Enter - killing it with the close button on the caption bar will not have the desired effect). The purpose of this lab is to figure out the problem. Don't look at at the source of the application just yet - instead you'll work with dumps you capture with ADPLUS.

Part 1 - Finding and Fixing a Runaway Thread

steps:

1. It's probably worth opening up perfmon and looking at the number of exceptions being generated. Applications that suddenly start hogging the CPU are often failing some operation in a loop somewhere. How many exceptions a second is this app generating?
2. First you will need to use ADPLUS to generate a hang dump instead of a crash dump. The difference here is that in hang mode ADPLUS will take an immediate snapshot instead of waiting for an exception. Let Clock.exe suck up CPU for 30 seconds or so, then take a hang snapshot with the ADPLUS:

```
C:\> adplus -hang -o c:\temp -pn clock.exe
```

3. This will generate a hang dump in the windbg folder just like a crash dump. You'll probably notice that it takes a little longer that a crash dump and when it's done clock.exe is still running, merrily sucking up CPU.
4. Load your hang dump into windbg and load sos. Use the `!runaway` command to see which threads are in your process and how much CPU time they've received:

```
0:000> !runaway
User Mode Time
Thread       Time
4:12810      0 days 0:00:11.316
2:11d94      0 days 0:00:00.050
0:12684      0 days 0:00:00.040
6:584c       0 days 0:00:00.000
5:7400       0 days 0:00:00.000
3:1281c      0 days 0:00:00.000
1:18084      0 days 0:00:00.000
```

5. Here it's pretty obvious that thread 4 is the culprit – it has used vastly more CPU time than any other thread. But what is it and what is it doing? First lets see if it's running managed code: run !Threads to see the managed threads in the process.

```
0:000> !Threads
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 2
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
PreEmptive   GC Alloc           Lock
ID OSID ThreadOBJ    State     GC       Context       Domain   Count APT Exception
0    1 12684 001528d8      6020 Enabled  00000000:00000000 0014b168     0 STA
2    2 11d94 0015ef40      b220 Enabled  00000000:00000000 0014b168     0 MTA
   (Finalizer)
4    3 12810 00193a70      b220 Enabled  00000000:00000000 0014b168     0 MTA
```

6. Yep – it looks like thread 4 is in fact executing managed code, which means you probably want to look at its stack:

```
0:000> ~4e!CLRStack
OS Thread Id: 0x12810 (4)
ESP       EIP
045ef770 7c90eb94 [NDirectMethodFrameSlim: 045ef770]
   System.Messaging.Interop.SafeNativeMethods.IntMQPathNameToFormatName(Syste
   m.String, System.Text.StringBuilder, Int32 ByRef)
045ef784 012212b6
   System.Messaging.Interop.SafeNativeMethods.MQPathNameToFormatName(System.S
   tring, System.Text.StringBuilder, Int32 ByRef)
045ef7b4 0122120f
   System.Messaging.MessageQueue.ResolveFormatNameFromQueuePath(System.String
   , Boolean)
045ef7e0 01220f07 System.Messaging.MessageQueue.get_FormatName()
```

```
045ef7f4 01220a73
   System.Messaging.MessageQueue.ReceiveCurrent(System.TimeSpan, Int32,
   System.Messaging.Interop.CursorHandle,
   System.Messaging.MessagePropertyFilter,
   System.Messaging.MessageQueueTransaction,
   System.Messaging.MessageQueueTransactionType)
045ef858 01220916 System.Messaging.MessageQueue.Receive(System.TimeSpan)
045ef868 012203d3 Clock.ClockForm.ReadThread()
045ef8b4 793d7a7b
   System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
045ef8bc 793683dd
   System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
   System.Threading.ContextCallback, System.Object)
045ef8d4 793d7b5c System.Threading.ThreadHelper.ThreadStart()
045efaf8 79e88f63 [GCFrame: 045efaf8]
```

7. Looks like this thread is trying to receive from a queue. Perhaps this call is failing? Check the heap for Exception objects with !DumpHeap –type Exception. You'll probably see something like the following:

```
0:000> !DumpHeap -stat -type Exception
total 27 objects
Statistics:
MT    Count    TotalSize Class Name
790fac70        1           72 ExecutionEngineException
790fabcc        1           72 StackOverflowException
790fab28        1           72 OutOfMemoryException
790fad14        2          144 ThreadAbortException
046019f4       20         1520 MessageQueueException
Total 27 objects
```

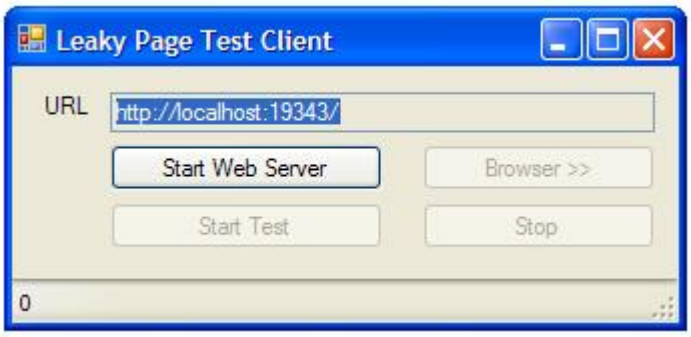
8. Hmm. Many MessageQueueExceptions are being thrown. See if you can recover the text out of one of the Exception objects (hint: you'll want to use a combination of !dumpheap –MT and !do to drill in). You should be able to recover the [somewhat unhelpful] string "External component has thrown an exception".
9. At this point you have something to start with. Clearly something is breaking when the server closes. Look at the server code to see what the server is doing when it shuts down. Anything there that could break the client?
10. The answer here is that the server is closing the queue, so instead of receiving a message and sleeping for 1 second Clock.exe is getting exceptions. What mistake did the author of clock.exe make here?
11. Fix the client code and verify that your fix works.

Lab 2 – Debugging Threads

In this activity you will troubleshoot a leaking ASP.NET application.

Part 1 - See the bug

1. In this activity you will be debugging an ASP.NET page that is suspected of a memory leak. To help show the leak you will work with a test client application shown below. To make things a little easier this application uses the ASP.NET Development Server to host the site so you don't need to do anything with IIS.
2. Go ahead and load TestClient.sln and run the application – it should look like this



3. When it first comes up the only enabled button will be Start Web Server. Click it and you should see the ASP.NET Development Server web server icon appear in the task notification area of the toolbar:

Now you can bring the page in question up in a web browser by pressing the Browser button. You'll see a simple page monitoring the operating temperature of a fictional device:

The only interesting behavior is that if the operating temperature goes above 500 degrees then the page displays a warning

4. If you press Refresh (or submit) enough times you'll see that the page leaks memory. This is more easily demonstrated with the test client– click Start Test and the client will start repeatedly requesting this page. You can monitor the memory usage by monitoring the memory usage of WebDev.WebServer.EXE in Task Manager – you should see it grow without limit as long as the test client is running.
5. Your job now is to find the memory leak without looking at the source code of default.aspx. You can look at the code later to implement a fix.

Part 2 – Getting a Memory Dump

1. First verify that this is a problem with managed code at all (as opposed to some native library somewhere). Bring up Perfmon, go to the .NET CLR Memory heap Performance object, select the #Bytes in all heaps counter for WebDev.WebServer. You should be able to see that the size of the managed heap is indeed growing.
2. Now add the Gen 0 heap size, Gen 1 heap size, Gen 2 heap size and Large Object Heap size counters as well. Which heap seems to be taking the brunt of the leak? (you should see Gen 2 growing while the other heap size counters stay relatively static).
3. This sure looks like a leak – as if some objects are being allocated and never freed. But which objects? The best way to figure this out here is to use adplus to generate a hang-mode dump of webdev.webserver.exe and inspect the managed objects. Stop the test client and get a dump of the server process by running

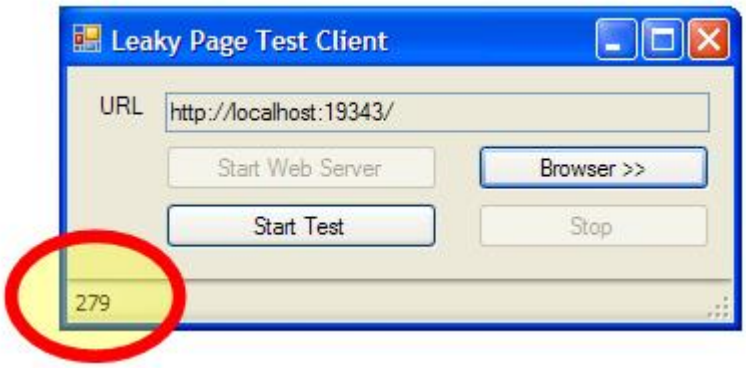
```
ADPLUS -hang -pn -o c:\temp webdev.webserver20.exe
```

Part 3 - Analyzing the crash dump

1. Open the crash dump in WinDBG and load SOS. From what you saw in performance monitor objects are piling up in Gen2, so take a look at what's on the heap by running !DumpHeap:

```
    !DumpHeap -stat
You'll get scads of output but the objects commanding the most memory will
   appear near the bottom of the list, so that's a good place to look. You'll
   likely see something like this:
  058e9e34      279        49104 System.Web.HttpRequest
  058eaa64      279        51336 System.Web.HttpResponse
  790fea70     1572        88032 System.Collections.Hashtable
  791036b0     3790        90960 System.Collections.ArrayList
  05bc9d7c     1692       101520 System.Web.UI.LiteralControl
  05bc490c      279       106020 ASP.default_aspx
  79124418     1151       115640 System.Byte[]
  791242ec     1576       277800 System.Collections.Hashtable+bucket[]
  79124670      749       597600 System.Char[]
  0015bed0       35       772832      Free
  79124228    11277       875660 System.Object[]
  790fa3e0    15875      1288004 System.String
```

   Chances are you'll see lines like those shown above – there are 279 living pages as well as 279 HttpRequest / HttpResponse objects. Take a look at your test client – the number of requests the client has mode probably will look familiar here:



2. So now you've established that the pages you're requesting aren't being collected but are living forever. But why? If you could find a page object in memory then you could figure out what is keeping it alive. Luckily this isn't hard. You can get a dump of the page objects in memory by passing the address of the metadata table (05bc490c in the snapshot above) to the dumpheap command

```
    !dumpheap -mt 05bc490c
```

   You'll get loads of output like this:

```
Address       MT     Size
0179d34c 05bc490c      380
```

```
017a817c 05bc490c      380
017b2ec8 05bc490c      380
017bdc14 05bc490c      380
017c8960 05bc490c      380
017d37d4 05bc490c      380
017de4b8 05bc490c      380
017e91ac 05bc490c      380
total 279 objects
Statistics:
MT    Count    TotalSize Class Name
05bc490c      279       106020 ASP.default_aspx
Total 279 objects
```

3. Now you can pick one of the objects at random and inspect its roots:

```
!gcroot 01730bd4

Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Scan Thread 0 OSTHread 99f4

   DOMAIN(001BEA88):HANDLE(Pinned):13712e8:Root:023aeb18(System.Object[])->
        017ea234(System.EventHandler)->
        01721280(System.Object[])->
        01731c3c(System.EventHandler)->
        01730bd4(ASP.default_aspx)
```

4. This shows us that the default_aspx page is being kept alive by an EventHandler delegate that in turn is part of a multicast delegate. Inspecting the originating event handler yields something interesting:

```
  !do 017ea234

  Name: System.EventHandler
  MethodTable: 7910d61c
  EEClass: 790c3a7c
  Size: 32(0x20) bytes

  Fields:
  MT    Field   Offset                 Type VT     Attr    Value Name
  790f9c18  40000f9        4        System.Object  0 instance 017ea234
   _target
  79109208  40000fa        8 ...ection.MethodBase  0 instance 00000000
   _methodBase
  790fe160  40000fb        c        System.IntPtr  0 instance 19728468
   _methodPtr
  790fe160  40000fc       10        System.IntPtr  0 instance 2030847496
   _methodPtrAux
  790f9c18  4000106       14        System.Object  0 instance 01721280
   _invocationList
  790fe160  4000107       18        System.IntPtr  0 instance      279
   _invocationCount
```

This delegate is pointing at 279 targets (!). It should be fairly obvious what the problem is here – the page class (default_aspx) is registering for an event and is probably forgetting to unregister.

5. This is enough of a lead to consider the code. Open the code in Default.aspx.cs and look for any place where it's registering for an event. Can you explain what's wrong with the code? Fix it and test that you have fixed the bug.