Debugging Multi-threaded Applications in Windows

Abstract

One of the most complex aspects of software development is the process of debugging. This process becomes especially challenging with the increased technicality of applications with multiple threads of execution. Different operating systems require different development environments as well as a different set of skills and tools to create a functional application. Microsoft Visual Studio, the primary integrated development environment for Windows programmers, simplifies and streamlines the debugging procedure. It allows developers to observe all current threads in the system, both to gain specific information and to manipulate threads to pinpoint where errors occur. This article explores the features that Microsoft Visual Studio has and its methods for debugging, including concrete examples demonstrating the setup of thread management and the execution of debugging procedures.

Introduction

This article presents the basic features of Microsoft Visual Studio. It will explain the features and how to utilize Visual Studio to create a functional, bug-free application. Visual Studio's language services allow the debugger to support (to varying degrees) almost any programming language. Built-in languages include C/C++, VB.NET, C#, and F#. Support for other languages such as M, Python, and Ruby is available via separately installed language services. By the end of this article, a developer should have a number of new techniques to apply to their own work immediately and the foundation necessary for approaching more advanced multi-threaded debugging projects.

Debugging Multi-threaded Applications in Windows

Most Windows programmers use Microsoft Visual Studio as their primary integrated development environment (IDE). As part of the IDE, Microsoft includes a debugger with multi-threaded debug support. This section examines the different multi-threaded debug capabilities of Visual Studio, and then demonstrates how they are used.

Threads Window

As part of the debugger, Visual Studio provides a "Threads" window that lists all of the current threads in the system. From this window, you can:

- *Freeze (suspend) or thaw (resume) a thread*. This is useful when you want to observe the behavior of your application without a certain thread running.
- *Switch the current active thread*. This allows you to manually perform a context switch and make another thread active in the application.

- *Examine thread state*. When you double-click an entry in the Threads window, the source window jumps to the source line that the thread is currently executing. This tells you the thread's current program counter. You will be able to examine the state of local variables within the thread.

The Threads window acts as the command center for examining and controlling the different threads in an application.

Tracepoints

As previously discussed, determining the sequence of events that lead to a race condition or deadlock situation is critical in determining the root cause of any multi-thread related bug. In order to facilitate the logging of events, Microsoft has implemented *tracepoints* as part of the debugger for Visual Studio 2005.

Most developers are familiar with the concept of a breakpoint. A tracepoint is similar to a breakpoint except that instead of stopping program execution when the applications program counter reaches that point, the debugger takes some other action. This action can be printing a message or running a Visual Studio macro.

Enabling tracepoints can be done in one of two ways. To create a new tracepoint, set the cursor to the source line of code and select "Insert Tracepoint." If you want to convert an existing breakpoint to a tracepoint, simply select the breakpoint and pick the "When Hit" option from the Breakpoint submenu. At this point, the tracepoint dialog appears.

When a tracepoint is hit, one of two actions is taken based on the information specified by the user. The simplest action is to print a message. The programmer may customize the message based on a set of predefined keywords. These keywords, along with a synopsis of what gets printed, are shown in Table 1.1. All values are taken at the time the tracepoint is hit.

Table 1.1 Tracepoint Keywords

KEYWORD	EVALUATES TO
$ADDRESS	The address of the instruction
$CALLER	The name of the function that called this function
$CALLSTACK	The state of the callstack
$FUNCTION	The name of the current function
$PID	The ID of the process
$PNAME	The name of the process
$TID	The ID of the thread
$TNAME	The name of the thread

In addition to the predefined values in Table 1.1, tracepoints also give you the ability to evaluate expressions inside the message. In order to do this, simply enclose the variable or expression in curly braces. For example, assume your thread has a local variable threadLocalVar that you'd like to have displayed when a tracepoint is hit. The expression you'd use might look something like this:

```
Thread: $TNAME local variables value is {threadLocalVar}.
```

Breakpoint Filters

Breakpoint filters allow developers to trigger breakpoints only when certain conditions are triggered. Breakpoints may be filtered by machine name, process, and thread. The list of different breakpoint filters is shown in Table 1.2.

Table 1.2 Breakpoint Filter Options

FILTER	DESCRIPTION
MachineName	Specifies that the breakpoint should only be triggered on certain machines
ProcessId	Limit breakpoint to process with the matching ID
ProcessName	Limit breakpoint to process with matching name
ThreadId	Limit breakpoint to thread with matching ID
ThreadName	Limit breakpoint to thread with matching name

Breakpoint filters can be combined to form compound statements. Three logic operators are supported: !(NOT), &(AND), and ||(OR).

Naming Threads

When debugging a multi-threaded application, it is often useful to assign unique names to the threads that are used in the application. In Chapter 5 of *Multi-Core Programming*, you read that assigning a name to a thread in a managed application was as simple as setting a property on the thread object. In this environment, it is highly recommended that you set the name field when creating the thread, because managed code provides no way to identify a thread by its ID.

In native Windows code, a thread ID can be directly matched to an individual thread. Nonetheless, keeping track of different thread IDs makes the job of debugging more difficult; it can be hard to keep track of individual thread IDs. An astute reader might have noticed in Chapter 5 the conspicuous absence of any sort of name parameter in the methods used to create threads. In addition, there was no function provided to get or set a thread name. It turns out that the standard thread APIs in Win32 lack the ability to associate a name with a thread. As a result, this association must be made by an external debugging tool.

Microsoft has enabled this capability through predefined exceptions built into their debugging tools. Applications that want to see a thread referred to by name need to implement a small function that

raises an exception. The exception is caught by the debugger, which then takes the specified name and assigns it to the associated ID. Once the exception handler completes, the debugger will use the user-supplied name from then on.

The implementation of this function can be found on the Microsoft Developer Network (MSDN) Web site at msdn.microsoft.com by searching for: "setting a thread name (unmanaged)." The function, named SetThreadName(), takes two arguments. The first argument is the thread ID. The recommended way of specifying the thread ID is to send the value -1, indicating that the ID of the calling thread should be used. The second parameter is the name of the thread. The SetThreadName() function calls RaiseException(), passing in a special 'thread exception' code and a structure that includes the thread ID and name parameters specified by the programmer.

Once the application has the SetThreadName() function defined, the developer may call the function to name a thread. This is shown in 228 Multi-Core Programming Listing 1.1. The function Thread1 is given the name Producer,[1] indicating that it is producing data for a consumer. Note that the function is called at the start of the thread, and that the thread ID is specified as -1. This indicates to the debugger that it should associate the calling thread with the associated ID.

```
unsigned __stdcall Thread1(void *)
{
        int i, x = 0; // arbitrary local variable declarations SetThreadName(-1,
        "Producer");
        // Thread logic follows
}
```

Listing 1.1 Using SetThreadName to Name a Thread

Naming a thread in this fashion has a couple of limitations. This technique is a debugger construct; the OS is not in any way aware of the name of the thread. Therefore, the thread name is not available to anyone other than the debugger. You cannot programmatically query a thread for its name using this mechanism. Assigning a name to a thread using this technique requires a debugger that supports exception number 0x406D1388. Both Microsoft's Visual Studio and WinDbg debuggers support this exception. Despite these limitations, it is generally advisable to use this technique where supported as it makes using the debugger and tracking down multi-threaded bugs much easier.

Putting It All Together

Let's stop for a minute and take a look at applying the previously discussed principles to a simplified real-world example. Assume that you are writing a data acquisition application. Your design calls for a producer thread that samples data from a device every second and stores the reading in a global variable for subsequent processing. A consumer thread periodically runs and processes the data from

[1] Admittedly the function name Thread1 should be renamed to Producer as well, but is left somewhat ambiguous for illustration purposes.

the producer. In order to prevent data corruption, the global variable shared by the producer and consumer is protected with a Critical Section. An example of a simple implementation of the producer and consumer threads is shown in Listing 1.2. Note that error handling is omitted for readability.

```
1       static int m_global = 0;
2       static CRITICAL_SECTION hLock; // protect m_global
3
4               // Simple simulation of data acquisition
5       void sample_data()
6        {
7               EnterCriticalSection(&hLock);
8               m_global = rand();
9               LeaveCriticalSection(&hLock);
10      }
11
12              // This function is an example
13              // of what can be done to data
14              // after collection
15              // In this case, you update the display
16              // in real time
17      void process_data()
18      {
19              EnterCriticalSection(&hLock);
20              printf("m_global = 0x%x\n", m_global);
21              LeaveCriticalSection(&hLock);
22      }
23
24              // Producer thread to simulate real time
25              // data acquisition. Collect 30 s
26              // worth of data
27              unsigned __stdcall Thread1(void *)
28              {
29                      int count = 0;
30                      SetThreadName(-1, "Producer");
```

```
31                  while (1)
32                  {
33                          // update the data
34                          sample_data();
35
36                          Sleep(1000);
37                          count++;
38                          if (count > 30)
39                                  break;
40                  }
41                  return 0;
42          }
43
44          // Consumer thread
45          // Collect data when scheduled and
46          // process it. Read 30 s worth of data
47          unsigned __stdcall Thread2(void *)
48          {
49                  int count = 0;
50                  SetThreadName(-1, "Consumer");
51                  while (1)
52                  {
53                          process_data();
54
55                          Sleep(1000);
56                          count++;
57                          if (count > 30)
58                                  break;
59                  }
60                  return 0;
61          }
```

Listing 1.2 Simple Data Acquisition Device

The producer samples data on line 34 and the consumer processes the data in line 53. Given this relatively simple situation, it is easy to verify that the program is correct and free of race conditions and deadlocks. Now assume that the programmer wants to take advantage of an error detection mechanism on the data acquisition device that indicates to the user that the data sample collected has a problem. The changes made to the producer thread by the programmer are shown in Listing 1.3.

```
void sample_data()
{
        EnterCriticalSection(&hLock);
        m_global = rand();
        if ((m_global % 0xC5F) == 0)
        {
                // handle error
                return;
        }
        LeaveCriticalSection(&hLock);
}
```

Listing 1.3 Sampling Data with Error Checking

After making these changes and rebuilding, the application becomes unstable. In most instances, the application runs without any problems. However, in certain circumstances, the application stops printing data. How do you determine what's going on?

The key to isolating the problem is capturing a trace of the sequence of events that occurred prior to the system hanging. This can be done with a custom trace buffer manager or with tracepoints. This example uses the trace buffer implemented in Listing 1.1.

Now armed with a logging mechanism, you are ready to run the program until the error case is triggered. Once the system fails, you can stop the debugger and examine the state of the system. To do this, run the application until the point of failure. Then, using the debugger, stop the program from executing. At this point, you'll be able bring up the Threads window to see the state information for each thread, such as the one shown in Figure 1.1.

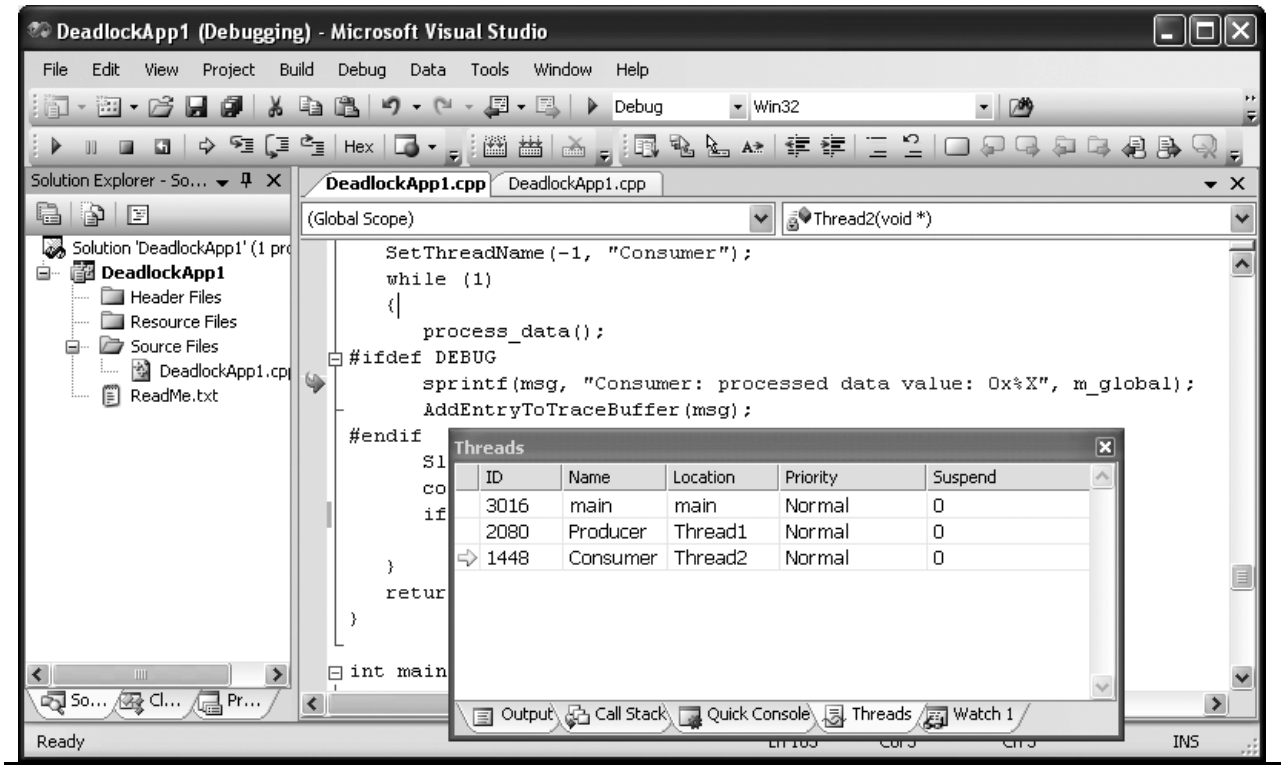

Figure 1.1 Examining Thread State Information Using Visual Studio 2005

When you examine the state of the application, you can see that the consumer thread is blocked, waiting for the process_data() call to return. To see what occurred prior to this failure, access the trace buffer. With the application stopped, call the PrintTraceBuffer() method directly from Visual Studio's debugger. The output of this call in this sample run is shown in Figure 1.2.

```
1   Thread ID | Timestamp   Msg
2    ----------------|----------------|----------------------------------------------------
3   0x0000728|1137395188|Producer: sampled data value: 0x29
4   0x00005a8|1137395188|Consumer: processed data value: 0x29
5   0x0000728|1137395189|Producer: sampled data value: 0x78
6   0x00005a8|1137395189|Consumer: processed data value: 0x78
7   0x0000728|1137395190|Producer: sampled data value: 0x18BE
8   0x0000728|1137395190|Producer: sampled data value: 0x6784
9   0x0000728|1137395190|Producer: sampled data value: 0x4AE1
10  0x0000728|1137395191|Producer: sampled data value: 0x3D6C
```

Figure 1.2 Output from trace buffer after Error Condition Occurs

Examination of the trace buffer log shows that the producer thread is still making forward progress. However, no data values after the first two make it to the consumer. This coupled with the fact that the

thread state for the consumer thread indicates that the thread is stuck, points to an error where the critical section is not properly released. Upon closer inspection, it appears that the data value in line 7 of the trace buffer log is an error value. This leads up back to your new handling code, which handles the error but forgets to release the mutex. This causes the consumer thread to be blocked indefinitely, which leads to the consumer thread being starved. Technically this isn't a deadlock situation, as the producer thread is not waiting on a resource that the consumer thread holds.

The complete data acquisition sample application is provided on this book's Web site, http://noggin.intel.com/intelpress/categories/books/multi-core-programming.

Conclusion

This article introduced debugging for multi-threaded applications using Microsoft Visual Studio. Comprehensive testing is an essential component of writing and developing multi-threaded applications. Microsoft Visual Studio offers a simplified approach that allows developers to view, manipulate and test individual threads. Through tracepoints and breakpoint filters, Visual Studio helps developers to pinpoint and log the sequence of events that lead to a race condition or a deadlock situation. Microsoft has also enabled thread naming capability through predefined exceptions raised by the implementation of a small function. For advanced debugging, consider using the Intel software tools, specifically, the Intel Debugger, the Intel Thread Checker, and the Intel Thread Profiler. The book from which this article was sampled supplies additional information on effective software debugging and provides an excellent foundation for developers interested in learning more about multi-core processing and software optimization.

This article is based on material found in the book *Multi-Core Programming: Increasing Performance through Software Multi-threading* by Shameem Akhter and Jason Roberts. Visit the Intel Press website to learn more about this book:

http://noggin.intel.com/intelpress/categories/books/multi-core-programming

Also see our Recommended Reading List for similar topics:

http://noggin.intel.com/rr

About the Authors

Shameem Akhter is a platform architect at Intel, focusing on single socket multi-core architecture and performance analysis. He has also worked as a senior software engineer with the Intel Software and Solutions Group, designing application optimizations for desktop and server platforms. Shameem holds a patent on a threading interface for constraint programming, developed as a part of his master's thesis in computer science.

Jason Roberts is a senior software engineer at Intel Corporation. Over the past 10 years, Jason has worked on a number of different multi-threaded software products that span a wide range of applications targeting desktop, handheld, and embedded DSP platforms.
