An Introduction to Multi-threaded Debugging Techniques

Abstract

Although debugging multi-threaded applications can seem daunting, the additional complexity of the testing process is simply a product of the complexity of the software, and the background and principles of debugging remain constant. A solid understanding of these principles allows developers to debug software as it is being developed, resulting in more efficient software creation and execution. This article explores key elements of development that minimize disruption, including designing with debugging in mind, utilizing code review to eliminate errors early, and extending applications via trace buffers. The article also prepares developers to take on increasingly complex debugging with confidence and experience.

Introduction

This article presents the general background and principles of debugging multiple threads of execution, a complex process that requires often specialized knowledge and tools. Developers that begin their projects with careful consideration of debugging will eliminate unnecessary revision and save time. Developers that are able to use simple tools to trace different threads and are able to monitor the sequence in which they access shared code resources can effectively narrow down problematic areas. the By the end of this article, developers will have the necessary knowledge to begin making their software development efficient and have the background necessary to understand and implement more complex debugging methods.

Multi-threaded Debugging Techniques

Debugging multi-threaded applications can be a challenging task. The increased complexity of multi-threaded programs results in a large number of possible states that the program may be in at any given time. Determining the state of the program at the time of failure can be difficult; understanding why a particular state is troublesome can be even more difficult. Multi-threaded programs often fail in unexpected ways, and often in a nondeterministic fashion. Bugs may manifest themselves in a sporadic fashion, frustrating developers who are accustomed to troubleshooting issues that are consistently reproducible and predictable. Finally, multi-threaded applications can fail in a drastic fashion—deadlocks cause an application or worse yet, the entire system, to hang.

Users tend to find these types of failures to be unacceptable. This article examines general purpose techniques that are useful when debugging multi-threaded applications. Intel has developed a number of tools, including the Intel® Thread Checker, the Intel Thread Profiler, and the Intel Debugger that help debug and profile multithreaded applications. These tools are discussed in Chapter 11.

General Debug Techniques

Regardless of which library or platform that you are developing on, several general principles can be applied to debugging multi-threaded software applications.

Designing with Debugging in Mind

The first technique for eliminating bugs in multi-threaded code is to avoid introducing the bug in the first place. Many software defects can be prevented by using proper software development practices.[1] Then later a problem is found in the product development lifecycle, the more expensive it is to fix. Given the complexity of multi-threaded programs, it is critical that multi-threaded applications are properly designed up front.

How often have you, as a software developer, experienced the following situation? Someone on the team that you're working on gets a great idea for a new product or feature. A quick prototype that illustrates the idea is implemented and a quick demo, using a trivial use-case, is presented to management. Management loves the idea and immediately informs sales and marketing of the new product or feature. Marketing then informs the customer of the feature, and in order to make a sale, promises the customer the feature in the next release. Meanwhile, the engineering team, whose original intent of presenting the idea was to get resources to properly implement the product or feature sometime in the future, is now faced with the task of delivering on a customer commitment immediately. As a result of time constraints, it is often the case that the only option is to take the prototype, and try to turn it into production code.

While this example illustrates a case where marketing and management may be to blame for the lack of following an appropriate process, software developers are often at fault in this regard as well. For many developers, writing software is the most interesting part of the job. There's a sense of instant gratification when you finish writing your application and press the run button. The results of all the effort and hard work appear instantly. In addition, modern debuggers provide a wide range of tools that allow developers to quickly identify and fix simple bugs. As a result, many programmers fall into the trap of coding now, deferring design and testing work to a later time. Taking this approach on a multi-threaded application is a recipe for disaster for several reasons:

- *Multi-threaded applications are inherently more complicated than single-threaded applications.* Hacking out a reliable, scalable implementation of a multi-threaded application is hard; even for experienced parallel programmers. The primary reason for this is the large number of corner cases that can occur and the wide range of possible paths of the application. Another consideration is the type of run-time environment the application is running on. The access patterns may vary wildly depending on whether or not the application is running on a single-core or multi-core platform, and whether or not the platform supports simultaneous

[1] There are a number of different software development methodologies that are applicable to parallel programming. For example, parallel programming can be done using traditional or rapid prototyping (Extreme Programming) techniques.

multithreading hardware. These different run-time scenarios need to be thoroughly thought out and handled to guarantee reliability in a wide range of environments and use cases.

- *Multi-threaded bugs may not surface when running under the debugger*. Multi-threading bugs are very sensitive to the timing of events in an application. Running the application under the debugger changes the timing, and as a result, may mask problems. When your application fails in a test or worse, the customer environment, but runs reliably under the debugger, it is almost certainly a timing issue in the code.

While following a software process can feel like a nuisance at times, taking the wrong approach and not following any process at all is a perilous path when writing all but the most trivial applications. This holds true for parallel programs.

While designing your multi-threaded applications, you should keep these points in mind.

- *Design the application so that it can run sequentially*. An application should always have a valid means of sequential execution. The application should be validated in this run mode first. This allows developers to eliminate bugs in the code that are not related to threading. If a problem is still present in this mode of execution, then the task of debugging reduces to single-threaded debugging.

- In many circumstances, it is very easy to generate a sequential version of an application. For example, an OpenMP application compiled with one of the Intel compilers can use the openmpstubs option to tell the compiler to generate sequential OpenMP code.

- *Use established parallel programming patterns*. The best defense against defects is to use parallel patterns that are known to be safe. Established patterns solve many of the common parallel programming problems in a robust manner. Reinventing the wheel is not necessary in many cases.

- *Include built-in debug support in the application*. When trying to root cause an application fault, it is often useful for programmers to be able to examine the state of the system at any arbitrary point in time. Consider adding functions that display the state of a thread—or all active threads. Trace buffers, described in the next section, may be used to record the sequence of accesses to a shared resource. Many modern debuggers support the capability of calling a function while stopped at a breakpoint.

This mechanism allows developers to extend the capabilities of the debugger to suit their particular application's needs.

Code Reviews

Many software processes suggest frequent code reviews as a means of improving software quality. The complexity of parallel programming makes this task challenging. While not a replacement for using well

established parallel programming design patterns, code reviews may, in many cases, help catch bugs in the early stages of development.

One technique for these types of code reviews is to have individual reviewers examine the code from the perspective of one of the threads in the system. During the review, each reviewer steps through the sequence of events as the actual thread would. Have objects that represent the shared resources of the system available and have the individual reviewers (threads) take and release these resources. This technique will help you visualize the interaction between different threads in your system and hopefully help you find bugs before they manifest themselves in code.

As a developer, when you get the urge to immediately jump into coding and disregard any preplanning or preparation, you should consider the following scenarios and ask yourself which situation you'd rather be in. Would you rather spend a few weeks of work up front to validate and verify the design and architecture of your application, or would you rather deal with having to redesign your product when you find it doesn't scale? Would you rather hold code reviews during development or deal with the stress of trying to solve mysterious, unpredictable showstopper bugs a week before your scheduled ship date? Good software engineering practices are the key to writing reliable software applications. Nothing is new, mysterious, or magical about writing multi-threaded applications. The complexity of this class of applications means that developers must be conscious of these fundamental software engineering principles and be diligent in following them.

Extending your Application—Using Trace Buffers

Chapter 7 identified two categories of bugs found in multi-threaded applications: synchronization bugs and performance bugs. Synchronization bugs include race conditions and deadlocks that cause unexpected and incorrect behavior. Performance bugs arise from unnecessary thread overhead due to thread creation or context switch overhead, and memory access patterns that are suboptimal for a given processor's memory hierarchy. The application returns the correct results, but often takes too long to be usable. This article focuses on debugging synchronization bugs that cause applications to fail.

In order to find the cause of these types of bugs, two pieces of information are needed:

1. Which threads are accessing the shared resource at the time of the failure.
2. When the access to the shared resource took place.

In many cases, finding and fixing synchronization bugs involves code inspection. A log or trace of the different threads in the application and the pattern in which they accessed the shared resources of the code helps narrow down the problematic code sections. One simple data structure that collects this information is the trace buffer.

A trace buffer is simply a mechanism for logging events that the developer is interested in monitoring. It uses an atomic counter that keeps track of the current empty slot in the array of event records. The type

of information that each event can store is largely up to the developer. A sample implementation of a trace buffer, using the Win32 threading APIs, is shown in Listing 1.1.[2]

```
1       // Circular 1K Trace buffer
2       #define TRACE_BUFFER_SIZE 1024
3
4       typedef struct traceBufferElement
5       {
6               DWORD threadId;
7               time_t timestamp;
8               const char *msg;
9       } traceBufferElement;
10
11      static LONG m_TraceBufferIdx = -1;
12      static traceBufferElement traceBuffer[TRACE_BUFFER_SIZE];
13
14      void InitializeTraceBuffer()
15      {
16              m_TraceBufferIdx = -1;
17
18              /* initialize all entries to {0, 0, NULL} */
19              memset(traceBuffer, 0,
20              TRACE_BUFFER_SIZE*sizeof(traceBufferElement));
21      }
22
23      void AddEntryToTraceBuffer(const char *msg)
24      {
25              LONG idx = 0;
26
27              // Get the index into the trace buffer that this
```

[2] In the interest of making the code more readable, Listing 1.1 uses the time() system call to record system time. Due to the coarse granularity of this timer, most applications should use a high performance counter instead to keep track of the time in which events occurred.

```
28              // thread should use
29              idx = InterlockedIncrement(&m_TraceBufferIdx) %
30              TRACE_BUFFER_SIZE;
31
32              // Enter the data into the Trace Buffer
33              traceBuffer[idx].threadId = GetCurrentThreadId();
34              traceBuffer[idx].timestamp = time(NULL);
35              traceBuffer[idx].msg = msg;
36      }
37
38      void PrintTraceBuffer()
39      {
40              int i;
41              printf("Thread ID Timestamp Msg\n");
42              printf("----------|----------|----------------------“
43              "-----------------\n");
44
45              // sort by timestamp before printing
46              SortTraceBufferByTimestamp();
47              for (i = 0; i < TRACE_BUFFER_SIZE; i++)
48              {
49                      if (traceBuffer[i].timestamp == 0)
50                      {
51                              break;
52                      }
53                      printf("0x%8.8x|0x%8.8x| %s\n",
54                      traceBuffer[i].threadId,
55                      traceBuffer[i].timestamp,
56                      traceBuffer[i].msg);
57              }
58      }
```

Listing 1.1 Sample Implementation of a Trace Buffer

Listing 1.1 creates a trace buffer that can store 1,024 events. It stores these events in a circular buffer. As you'll see shortly, once the circular buffer is full, your atomic index will wrap around and replace the oldest event. This simplifies your implementation as it doesn't require dynamically resizing the trace buffer or storing the data to disk. In some instances, these operations may be desirable, but in general, a circular buffer should suffice.

Lines 1–13 define the data structures used in this implementation. The event descriptor traceBufferElement is defined in lines 4–9. It contains three fields: a field to store the thread ID, a timestamp value that indicates when the event occurred, and a generic message string that is associated with the event. This structure could include a number of additional parameters, including the name of the thread.

The trace buffer in Listing 1.1 defines three operations. The first method, InitializeTraceBuffer(), initializes the resources used by the trace buffer. The initialization of the atomic counter occurs on line 16. The atomic counter is initialized to –1. The initial value of this counter is –1 because adding a new entry in the trace buffer requires us to first increment (line 29) the atomic counter. The first entry should be stored in slot 0. Once the trace buffer is initialized, threads may call AddEntryToTraceBuffer() to update the trace buffers with events as they occur. PrintTraceBuffer() dumps a listing of all the events that the trace buffer has logged to the screen. This function is very useful when combined with a debugger that allows users to execute code at a breakpoint. Both Microsoft Visual Studio and GDB support this capability. With a single command, the developer can see a log of all the recent events being monitored, without having to parse a data structure using the command line or a watch window.

Note that the implementation of the trace buffer in Listing 1.1 logs events as they are passed into the buffer. This doesn't necessarily guarantee that the trace buffer will log events exactly as they occur in time. To illustrate this point, consider the two threads shown in Listing 1.2.

```
unsigned __stdcall Thread1(void *)
{
        // … thread initialization
        // write global data
        m_global = do_work();
        AddEntryToTraceBuffer(msg);
        // … finish thread
}
```

```
unsigned __stdcall Thread2(void *)
{
        // … thread initialization
        // read global data
        Thread_local_data = m_global;
        AddEntryToTraceBuffer(msg);
        // … finish thread
}
```

Listing 1.2 Two Threads Logging Events to a Trace Buffer

By now it should be clear what the problem is. A race condition exists between the two threads and the access to the trace buffer. Thread1 may write to the global data value and then start logging that write event in the trace buffer. Meanwhile, Thread2 may read that same global value after the write, but log this read event before the write event. Thus, the data in the buffer may not be an accurate reflection of the actual sequence of events as they occurred in the system.

One potential solution to this problem is to protect the operation that you want to log and the subsequent trace buffer access with a synchronization object. A thread, when logging the event, could request exclusive access to the trace buffer. Once the thread has completed logging the event, it would then unlock the trace buffer, allowing other threads to access the buffer. This is shown in Listing 1.3.

```
// This is NOT RECOMMENDED
unsigned __stdcall Thread1(void *)
{
        // … thread initialization
        // write global data
        LockTraceBuffer();
        m_global = do_work();
        AddEntryToTraceBuffer(msg);
        UnlockTraceBuffer();
        // … finish thread
}
unsigned __stdcall Thread2(void *)
{
        // … thread initialization
```

```
        // read global data
        LockTraceBuffer();
        Thread_local_data = m_global;
        AddEntryToTraceBuffer(msg);
        UnlockTraceBuffer();
        // … finish thread
}
```

Listing 1.3 Incorrectly Synchronizing Access to the Trace Buffer

There are a number of drawbacks to this technique. Using a synchronization primitive to protect access to a trace buffer may actually mask bugs in the code, defeating the purpose of using the trace buffer for debug. Assume that the bug the developer is tracking down is related to a missing lock around the read or write access in the thread. By locking access to the trace buffer, the developer is protecting a critical section of code that may be incorrectly unprotected. Generally speaking, when tracking down a race condition, the programmer should avoid synchronizing access to the trace buffer. If you synchronize access and your application works, it's a clue that there may be a problem in the synchronization mechanism between those threads.

The preferred method to overcoming this limitation is to log a message before and after the event occurs. This is demonstrated in Listing 1.4.

```
unsigned __stdcall Thread1(void *)
{
        // … thread initialization
        // write global data
        AddEntryToTraceBuffer(before_msg);
        m_global = do_work();
        AddEntryToTraceBuffer(after_msg);
        // … finish thread
}
unsigned __stdcall Thread2(void *)
{
        // … thread initialization
        // read global data
        AddEntryToTraceBuffer(before_msg2);
```

```
            Thread_local_data = m_global;
            AddEntryToTraceBuffer(after_msg2);
            // … finish thread
        }
```

Listing 1.4 Preferred Method of Logging Messages with a Trace buffer

By logging a before and after message, a programmer can determine whether or not the events occurred as expected. If the before and after messages between the two threads occur in sequence, then the developer can safely assume that the event was ordered. If the before and after messages are interleaved, then the order of events is indeterminate; the events may have happened in either order.

A trace buffer can be used to gather useful data about the sequence of operations occurring in a multi-threaded application. For other more difficult problems, more advanced threading debug tools may be required. These tools are discussed in Chapter 11.

Conclusion

This article introduced foundational knowledge and skills for the complex topic of multi-threaded debugging. To maximize efficiency and minimize error, developers need to begin their software development with the concept of debugging in mind. Code review can catch bugs early in development and save an enormous amount of time and effort. A trace buffer can be utilized to log useful event data about the sequence of operations occurring in a multi-threaded application. Though knowledge of these general techniques is valuable, more complex debugging problems must sometimes be addressed with more advanced tools. For more advanced debugging, consider employing Intel software tools, specifically, the Intel Debugger, the Intel Thread Checker, and the Intel Thread Profiler. To learn more about multi-core processing and software optimization, consult the book from which this article was sampled.

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.
