Debugging and Defect Management

Dr. Dobb's Journal

August 2013

Making Issue Tracking Drive Development

Transforming defect- and issue-trackers into workflow tools

Next

ALSO INSIDE

UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

August 2013

COVER ARTICLE

FEATURES

More on DrDobbs.com

The Misplaced Obsession with Simplicity

The false dichotomy between simplicity and complexity leads us to chase after baubles and value the wrong things.

http://www.drdobbs.com/240157265

GNU Awk: This is Not Your Father's Awk

Awk's features have advanced considerably in the last decade — such as the addition of a debugger and a profiler — all without removing any of the elegance or terseness of the fabled little language.

http://www.drdobbs.com/240158351

Advice to a New Programmer

So much advice is heaped upon beginners that it can be hard to know where to start. However, these five practices are the foundation upon which everything else is built.

http://www.drdobbs.com/240158341

Scala for C# Developers: The Magic

Implicit conversions, avoiding nulls, default and named parameters, and using mixins and traits to borrow functionality from other classes are all part of what makes Scala a magically powerful language.

http://www.drdobbs.com/240157380

The Best of the First Half

The most popular articles of the first half of the year, sprinkled with a few pieces carefully chosen by the editors.

http://www.drdobbs.com/240157430

Mailbag

Reconsidering Our Obsession with Simplicity

In response to our editorial on the need to reconsider our commitment to simplicity (http://www.drdobbs.com/240157265), we received many thoughtful comments and these interesting letters.

"I agree with much of the thrust of your article, and you are right to attempt to redefine simplicity. The terms simplicity and complexity have been misused and their definitions blurred at the edges.

However, there seems to be a flaw here. Simplicity is not a state, it is a description of the code's value in terms of clarity and maintainability. Whereas all code has complexity. The level of complexity is described as simple if it is understandable (and therefore maintainable) by your average software engineer. If it is not maintainable by his/her peers, where is the value in it? It does what its supposed to till it falls over? Where is the contribution to the body of human knowledge in that?

Complexity that is not understood is a chain round our necks. It may be mistakenly so, or deliberately so.

The first is born from isolated personalities who believe their thought processes are the same as others in their profession. If they were to understand the uniqueness of the human mind they would recognize that there is *always* the need to communicate their thought processes at a level that others can understand through explanation and dialogue.

Some developers believe rightly that their thought processes are unique, and they want to keep it that way. They use lack of communication to confuse, and take delight in boosting their ego by the results. They are complimented by the fact that the code is not understood. This amounts to criminal neglect of their duty to their peers and they should be ashamed!

So,to understand a complex system, it needs to be described at a level of complexity that is simple (clear) enough for their profession to understand. That is not to say that the code itself should be dumbed down, but that every effort should be made to describe the thought processes that led to that generated that code."

— Richard Taylor
New Milton, UK

"A colleague and I recently wrestled with this question when we were presented with a large, decades-old legacy piece of software and asked for advice in identifying parts that would be economically-attractive candidates for redesign. Although we had tools to measure the complexity of the existing code, we had only intuition to guide us as to the inherent complexity of the tasks being performed by the software. A little research on inherent complexity suggested that we had a chicken-and-egg problem: the only ways to measure inherent complexity of a task is to reduce the task to an algorithm and then measure the complexity of the algorithm. We lacked the budget/schedule to create our own algorithmic expression of the underlying task.

In the end, we allowed ourselves to be guided by intuition, which left us vulnerable to our own incomplete understanding of the problem's underlying complexity. I think your reference to readability is crucial —

when readability is compromised (particularly by superficial complexity, for example, multiple layers of wrappers), then we're left with a mismatch between what a human sees when looking at the solution and what a human sees when looking at the underlying task. It's this mismatch that becomes the 'bad.'"

— Mark J. Munkacsy
Portsmouth, R.I.

"In fact, your editorial argues for the use of the term 'irreducible complexity' to just 'complexity', or as Einstein already said: Everything should be explained as simply as possible, but no simpler. Replace explained by coded and you have it…And yes, coding is explaining, not only to machines. But also to human readers.

I fully agree."

— Peter Sels

Andrew Binstock responds: "I was tempted to use the Einstein quotation when writing the editorial, but it doesn't quite align with my point in that making problems too simple is not really an issue in the context we're discussing. Excessive simplicity is generally the mark of those who have not entirely grasped the nature of the problem. That being said, though, it's hard to disagree with the quotation."

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

News Briefs

By Adrian Bridgwater

Forget DevOps, Try Continuous Dynamic DevOps

OpenMake Software has announced the 7.5 release of its Dynamic DevOps suite comprised of the Meister build automation, Mojo workflow management, and Cloudbuilder provisioning products. Also featuring the Deploy + tool, this collection is what OpenMake describes as "a consolidated toolchain for process automation" to handle continuous build and continuous deployment tasks.
http://www.drdobbs.com/240157729

Taking .NET Libraries To Mobile On Xamarin

Xamarin has announced a new online service designed to help developers move apps to mobile. The browser-based service scans existing .NET libraries to determine "by device platform" the percentage of code that can immediately be incorporated into mobile apps using Xamarin. The service also provides a roadmap of the additional work required to migrate the remaining code to create fully native iOS, Android, Mac, Windows Phone, and Windows Store apps.
http://www.drdobbs.com/240157658

Annual Eclipse "Kepler" Release Train Ready

The Eclipse Foundation has released Kepler for 2013 in its normal annual "predictable coordinated release" summer timeframe. Highlights of the Kepler release include support for Java EE 7 inside the Eclipse Web Tools Project (WTP) 3.5 — and support for JPA 2.1, JSF 2.2, JAX-RS 2.0, Servlet 3.1, EJB 3.2, Connector 1.7, App Client 7.0, and EAR 7.0. Eclipse WTP project and code generation wizards, content assist, and validation have been updated so Java developers can easily create, debug, and deploy Java EE 7 compliant applications.
http://www.drdobbs.com/240155794

Scalability Is OK, Tunable Persistence Engines Are Better

Couchbase Server 2.1 arrives this week as the latest release of the firm's NoSQL document-oriented database. While the company lists scalability, performance, and always-on reliability (don't they always?) among the new features, more interesting is this release's multi-threaded persistence engine to improve disk utilization and latency, a function that will work on both commodity hardware and specialized servers. This product also has a good set of monitoring and management capabilities. The newly designed and tunable persistence engine increases resource efficiency and joins cross datacenter replication optimizations, resulting in improved latency for data operations across clusters.
http://www.drdobbs.com/240157597

Why Isn't Test Connected To Release?

Atlassian has announced the Bamboo 5 continuous integration and delivery server. The firm declares that this software "connects the processes" of preparing and testing software with the process of releasing it. Not just for running builds and tests, the Atlassian team says that Bamboo 5 connects issues, commits, test results, and deploys. In this way a more holistic picture is available to the entire product team: from project managers and devs to testers and system administrators. Atlassian Bamboo aims to address lack of traceability between build and deployment processes, and lack of visibility into the release-preparation process. Streamlined workflows with fine-grained access controls mean that developers and QA can deploy to their own environments on-demand, while production remains tightly regulated.
http://www.drdobbs.com/240158132

Making Issue Tracking Drive the Development Process

Issue trackers contain the list of defects and the inventory of new features to be developed. As they also capture progress on these items, they emerge as the one tool that can drive and manage the project workflow. This central view of issue tracking is becoming widely adopted, using the practices presented here.

By Dan Radigan

The right use of tools can radically improve a development organization's culture and its ability to work well. Products such as defect- and issue-trackers can be transformed into workflow tools if used to their fullest advantage. Here are fundamental good practices for setting up a tracker tool at your organization to be the centerpiece of your project workflow.

Make Contributing Simple

Reducing friction is key to getting rapid adoption from a user base. Start with a URL that is simple for everyone to remember. Good URLs in the form of `http://go/<yourapp>` make it easy for users to quickly find the system. To make login a seamless experience, find a solution that integrates with your existing directory infrastructure such as LDAP, Active Directory, or Google Apps.

When laying out the issue creation form, every field needs to "earn the right to be seen." It's easy to just add fields for data that a user might "want" to enter. However, extra, unnecessary fields make the system more cumbersome — they're just more clutter to read/scroll through and can be confusing for the user. Don't attempt to future proof; make every field count.

Whether the type of issue being tracked is a bug, a feature request, or a to-do item, try to capture as much information as possible to understand the issue. Engineers, for example, want environment-specific information from inside a product at runtime. For that reason, many

issue trackers now have RESTful interfaces that make filing issues in-product easier. Developers can reliably get information about the state of the product in each and every bug when issues are filed RESTfully from the product.

Minimize Ramp-Up and Engage Everyone

Teams that engage their full ecosystem in an issue tracker are better connected to their market. With many issue trackers now running in a hosted service, it's easy to allow beta users and customers to log issues directly into your system. Shortening the distance between an issue and its solution encourages issues to get resolved faster. If customer service issues are logged in one system, then manually transferred to another system for engineering review, it takes more work for a customer's issue to get fixed.

As new users engage with the issue tracker, take some time to get first impressions. Processes — as well as the tools themselves — need to be agile. Proactively seek feedback on how things are working. Often, existing employees suffer from "it's always been that way" syndrome; new users have a short but valuable window where they can give an outside perspective. Take advantage of that window.

Today, many software teams work cross-functionally. When people cross team boundaries, it's helpful to use the same language across teams within the same company. For example, priority schemes (such as Blocker, Critical, Major, etc., or P1–P4) create a field that conveys relative importance of issues. Some teams have a policy that blockers need to be resolved by end-of-day. Other teams have a less strict definition. Standardizing terminology across teams makes it easier and faster to work together because expectations are consistent.

Dashboards Done Well Incite Action

Everyone on the team wants to know whether the project is going well. One key benefit of a distributed work management tool is that everyone can update his or her part of the project. Project managers can then focus on using the tool to plan forward direction as the team pushes status, rather than pestering everyone on the team for an update. Effective managers can build dashboards that pull real-time stats from work management tools and keep the team in the know. Finally, make the dashboard visible. Link it from the team's intranet home page, and consider creating a simple go URL as well (for example, `http:/go/teamstatus`).

"Each organization has a unique process that it naturally wants to follow. You shouldn't have to build your own issue tracker to create a custom solution."

Dashboards should always incite an action. Team members and stakeholders should know at a glance if things are going well. Resourceful dashboard authors ask project participants for feedback to ensure that all agree on the message of the dashboard. Finally, a user tends to silo his or her "part," focusing on the fact that "my bugs are OK" rather than taking a holistic view of the program. Much like the build team creates the official build, consider tasking the program manager to "own" the project status and reporting for the full team.

Make Your Issue Tracker Work for You!

I've seen many teams struggle with work management solutions as they try to build their process around the tool, rather than build the tool around the process. Each organization has a unique process that it naturally wants to follow. You shouldn't have to build your own issue tracker to create a custom solution. Unfortunately, many teams don't stray far from the vendor's defaults. Spend the time to customize your tool around how your organization gets work done; it's a valuable investment.

Effective workflows help the organization to work together more fluidly. States (new, open, resolved, etc.) indicate work status. The assignee makes it clear across the company who is responsible. Transitions between states inform everyone how work can get done. When the team has the right workflow, the burden of issue tracking fades to the background because it's a natural part of the culture of the organization.

Integrate with Other Systems

Choosing a system with a flexible API makes it easy to integrate with other systems such as source control or the build system. Integrating systems increases transparency across the engineering process. For example, the tool should integrate with other code repositories, as well as the continuous integration (CI) server, and the project wiki to track an issue's progress into working software. Engineers can check-in new code to source control, and the issue tracker will pick-up on that update. The CI server can then update the issue tracker noting that the referenced issue has been built into working software. When related systems automate manual processes, everyone wins.

Categorize to Optimize

As an industry we've grown beyond bug tracking. It's now all about work management. We don't just fix bugs, we innovate through software. Some of that is writing new features. Other times, it's rearchitecting

existing code. And yes, of course, we still fix defects. When we commit all of our work to our tools, we can see the entirety of what we do.

Modern work management tools can categorize issues by sprint, release, component, or theme, to name a few categories. In practice, it's much easier to implement related tickets at the same time. When issues are organized by component, it's easier to see all of the work that needs to be done in a particular area. The team can then plan when that work needs to be performed over the next several iterations, thus optimizing time spent in a certain area. Issues can also be categorized by priority. It's always helpful in retrospect to see which high-priority issues had to be dealt with in an unexpected way during an iteration.

Remain Lean

One of the most important things to keep your issue tracker healthy is regular triage of incoming issues and grooming the backlog. Software is dynamic. Priorities change. Regular triage keeps the team in tune with the current state of the product. The backlog is the list of work that outlines the future direction of the product in priority order. When new feature requests or bugs come in, they need to be prioritized against all of the existing work. Keeping the backlog current gives the engineering team confidence that they are working on the highest priority items.

As a QA engineer early in my career, I hated closing bugs that were not fixed. A user reported i, so it must be relevant, right? At some level, yes; but feedback has a shelf life. As more and more issues get logged in the system, it becomes harder to focus on the issues that matter. Bugs that were filed five years ago are, generally speaking, less relevant than ones filed last week.

If an issue has no path to closure in the foreseeable future, close it. Closed issues are not lost. When closing an issue, always include a resolution so you can search it later. At a minimum, resolutions should include: fixed, will not fix, cannot reproduce, and obsolete. Obsolete issues those that are no longer relevant to the direction of the program. Product managers can then search closed issues in the same way that they search open ones. In the closed case, issues that got marked will not fix and obsolete represent real work that was deferred.

Going Forward

For years, companies have viewed the defect tracker as a tool apart. During the last decade, it's become evident that the addition of feature requests has converted the tracker into a planning tool for team sprints and individual work. Thus, it is the central product for capturing project status accurately.

The guidelines I list here should help you move from a traditional siloed view of defect tracking to a broader view that enables team coordination, better planning, and improved delivery.

— Dan Radigan is an Agile Evangelist at Atlassian (https://www.atlassian.com/), makers of JIRA (https://www.atlassian.com/software/jira). Prior to Atlassian, he was an engineering manager at Netflix.

Comment

Debugging Multithreaded Applications in Windows

End the frustration of tracking down thread-specific bugs with a few simple options in Visual Studio.

By Shameem Akhter and Jason Roberts

Debugging applications in which several threads are active can be challenging to the point of great frustration. This article aims to relieve a bit of that frustration by explaining how to utilize Visual Studio to debug multithreaded applications. As part of Visual Studio, Microsoft includes a debugger with multithreaded debug support. We'll start by examining the different multithreaded debug capabilities of Visual Studio, and then demonstrate how to use them.

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

Determining the sequence of events that lead to a race condition or deadlock is critical in determining the root cause of any multithread-related bug. To facilitate the logging of events, Microsoft has implemented tracepoints as part of the debugger for Visual Studio for many years.

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.

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

Table 1: Tracepoint keywords.

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. You can 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. All values are taken at the time the tracepoint is hit.

In addition to these predefined values, tracepoints also give you the ability to evaluate expressions ins the message. 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 enable developers to trigger breakpoints only when certain conditions are triggered. Breakpoints can be filtered by machine name, process, and thread. A list of different breakpoint filters is shown in Table 2.

Breakpoint filters can be combined to form compound statements. Three logic operators are supported: `!` (not), `&` (and), and `||` (or).

Naming Threads

When debugging a multithreaded application, it is often useful to assign unique names to the threads that are used in the application. Assigning a name to a thread in a managed application is as simple as

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

Table 2: Breakpoint filter options.

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, as it is hard to keep track of individual thread IDs. Unfortunately, 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) website 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 Listing One. The function Thread1 is given the name `Producer`, 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.

Listing One: Using SetThreadName to name a thread.

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

Naming a thread in this fashion has a few 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 it much easier to use the debugger and track down multithreaded bugs.

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 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 Two. (Note that error handling is omitted for readability.)

Listing Two: Simple data acquisition device.

```
static int m_global = 0;
static CRITICAL_SECTION hLock; // protect m_global

// Simple simulation of data acquisition
void sample_data()
{
    EnterCriticalSection(&hLock);
    m_global = rand();
    LeaveCriticalSection(&hLock);
}

    // This function is an example
    // of what can be done to data
    // after collection
    // In this case, you update the display
    // in real time
void process_data()
{
    EnterCriticalSection(&hLock);
    printf("m_global = 0x%x\n", m_global);
    LeaveCriticalSection(&hLock);
}

    // Producer thread to simulate real time
    // data acquisition. Collect 30 s
    // worth of data
    unsigned __stdcall Thread1(void *)
    {
        int count = 0;
        SetThreadName(-1, "Producer");
        while (1)
        {
            // update the data
            sample_data();

            Sleep(1000);
            count++;
            if (count > 30)
                break;
        }
        return 0;
    }

    // Consumer thread
    // Collect data when scheduled and
    // process it. Read 30 s worth of data
    unsigned __stdcall Thread2(void *)
    {
        int count = 0;
        SetThreadName(-1, "Consumer");
        while (1)
        {
```

```
            process_data();

            Sleep(1000);
            count++;
            if (count > 30)
                break;
        }
        return 0;
    }
```

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 Three.

Listing Three: Sampling data with error checking.

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

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 One.

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.

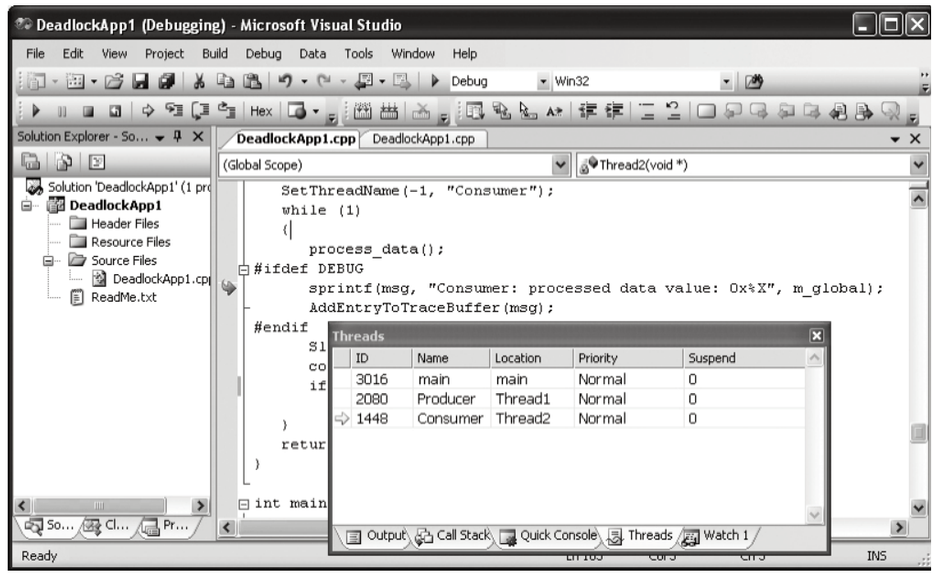

Figure 1: Examining thread state information using Visual Studio.

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 2.

```
Thread ID| Timestamp|Msg
 ---------|----------|------------------------------------
 0x0000728|1137395188|Producer: sampled data value: 0x29
 0x00005a8|1137395188|Consumer: processed data value: 0x29
 0x0000728|1137395189|Producer: sampled data value: 0x78
 0x00005a8|1137395189|Consumer: processed data value: 0x78
 0x0000728|1137395190|Producer: sampled data value: 0x18BE
 0x0000728|1137395190|Producer: sampled data value: 0x6784
 0x0000728|1137395190|Producer: sampled data value: 0x4AE1
 0x0000728|1137395191|Producer: sampled data value: 0x3D6C
```

Figure 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.)

What this example shows is that by combining several features — the ability to name threads, tracepoints, and logging — it becomes possible to zero in on the problem without having to try to do all the diagnostic work after the fact. Developing other specialized activities that are tied to tracepoints can provide further assistance in bug hunting. Good luck!

— Shameem Akhter is a platform architect at Intel, focusing on single socket multi-core architecture and performance analysis. Jason Roberts is a senior software engineer at Intel Corporation. This article is adapted from in the book **Multi-Core Programming: Increasing Performance through Software Multi-threading** *(http://is.gd/hOifZT), published by Intel Press, Copyright 2011 Intel Corporation.*

Comment

From the Vault

Post-Mortem Debugging Revisited

Debugging after the fact

By Stefan Worthmüller

Several years ago, I wrote an article entitled "Post-Mortem Debugging" (http://www.drdobbs.com/185300443) that included source code to create stack dumps with function names from C/C++ programs that crashed on client sites without any modification on the client's machine. This was done in two steps (available on Windows and Linux/UNIX):

1. Install a fault handler in your application that dumps the call stack to a text file if an error has occurred.
2. Run a program (MapAddr.exe) to add function names to the memory addresses of the stack dump by matching them to a map file from the program's build.

Since then, I've received feedback that led me to a number of different approaches, which I summarize in this article.

Retrieving Line Numbers from the Stack Dump (Windows Only)

Visual Studio stores all debug information in a program database which resides in a file with the extension ".pdb." This database file is created for debug builds by default, but it is easy to create them for optimized release builds as well:

In the Project Setup dialog select the following 2 Options:

- In the C/C++ Section
  Debug Information Format: **Program Database (/Zi)**
- In the Linker Section
  Generate Debug Info. **YES**

This will generate a .pdb file for the selected build, which should be archived for each released version. It also enables the possibility to debug the release version of your program just as you would do with the debug version.

.pdb files are stored in a binary format that is not documented, but it's easy to access it through a COM API. The Debug Information API: DIA (http://is.gd/H3EiOF) is provided by Microsoft starting from Visual Studio 2008. Using this API, it is straightforward to get the following information for every code address:

- Unmangled function name (human readable including the list of parameters)
- Source file name
- Line number within that source file

Figure 1 shows a stack dump created from the same FaultApp as used for demonstration in "Post-Mortem Debugging." While adding the line number to the stack dump may only seem like a "nice to have" feature, it ultimately proved to be very valuable once I started using it. Before having line numbers, I always had to examine the entire function. Now I only have to check a single line of code, and the problem often becomes obvious immediately.

Accessing the pdb file involves a number of COM-Calls bundled together in a class file. Adding the usage of this class to `MapAddr` was simple and straightforward. `MapAddr` can now be called with pdb files as well (as before) using map files. The full source code can be downloaded at http://is.gd/otS6qT.

```
*****************************************************
*** A Programm Fault occured:
*** Error code C0000005: ACCESS VIOLATION
*****************************************************
***   Address: 004398E8
***     Flags: 00000000
*****************************************************
*** CallStack:
*****************************************************

  Fault Occured At $ADDRESS:004398E8
                   ======== 000398E8 == belongs to FunctionC(int,int)
    File d:\projects\mapaddr\faultapp.cpp Line 29
                            with 64 00 00 00 10 00 00 00 80 FE 12 00 00 00 00 00 E0 FD 7E

***  0 called from $ADDRESS:00439969
                   ======== 00039969 == belongs to FunctionB(int)
    File d:\projects\mapaddr\faultapp.cpp Line 41
                            with 64 00 00 00 54 FF 12 00 00 00 00 00 E0 FD 7E CC CC CC CC

***  1 called from $ADDRESS:004399E5
                   ======== 000399E5 == belongs to FunctionA()
    File d:\projects\mapaddr\faultapp.cpp Line 51
                            with 00 00 00 00 00 00 00 00 E0 FD 7E CC CC CC CC CC CC CC

***  2 called from $ADDRESS:00439A68
                   ======== 00039A68 == belongs to main
    File d:\projects\mapaddr\faultapp.cpp Line 64
                            with 01 00 00 00 70 34 4D 00 B8 34 4D 00 84 5C FF C1 00 00 00 00

***  3 called from $ADDRESS:0044B573
                   ======== 0004B573 == belongs to __tmainCRTStartup
    File f:\dd\vctools\crt_bld\self_x86\crt\src\crt0.c Line 327
                            with

***  4 called from $ADDRESS:0044B32D
                   ======== 0004B32D == belongs to mainCRTStartup
    File f:\dd\vctools\crt_bld\self_x86\crt\src\crt0.c Line 195
                            with 00 00 00 00 00 00 00 00 00 00 E0 FD 7E 05 00 00 C0 C8 FF 12 00

***  5 called from $ADDRESS:7D4E7D42
         with

***  6 called from $ADDRESS:00000000
*************************************************************
```

Figure 1.

64-Bit Windows

The fault handler of the first version used x86 assembler calls to unwind the stack. But when porting to 64 bits, I found no way to write stack frames reliable with assembler instructions with optimization enabled. While there are a number of Win32 API functions to unwind the stack, only one (`RtlCaptureStackBackTrace` at http://is.gd/gcq6wU) seems to work on x64 as expected. It can only retrieve 62 stack frames, which is sufficient in most cases, although this limit seems questionable.

Other Post-Mortem Techniques: Windows Mini Dumps

Minidumps store the current state of a program in a binary file that can be loaded into Visual Studio or WinDbg. When loaded into a debugger, a minidump shows the full symbolized stack and all local variables just as it would during a debug session. That is, of course, more comfortable than symbolizing a logfile.

Minidumps can be created a number of ways:

- Using Dr.Watson (http://support.microsoft.com/kb/275481/en-us) to be called on a program fault.
- Calling it explicitly from within the program itself (http://www.debuginfo.com/articles/effminidumps.html); for example, by an installed Structured Exception handler, as in the approach described in the original iteration of this article (http://www.drdobbs.com/185300443).
- Using a tool such as WinDbg (http://www.windbg.org/) or a MiniDump tool (http://is.gd/awC30q).

The disadvantages of minidumps are that they are Windows specific and they cannot be batch processed (as is possible using a utility such as MapAdrr.exe).

Other Post-Mortem Techniques: Windows Error Reporting

You have probably seen a dialog similar to Figure 2, which was introduced with Windows XP.

To tell the truth, I was surprised to learn that Microsoft will send error reports even for programs that I have written but never published. However, I realized that I could use this mechanism for my personal applications. Windows Error Reporting (WER) is a service offered by

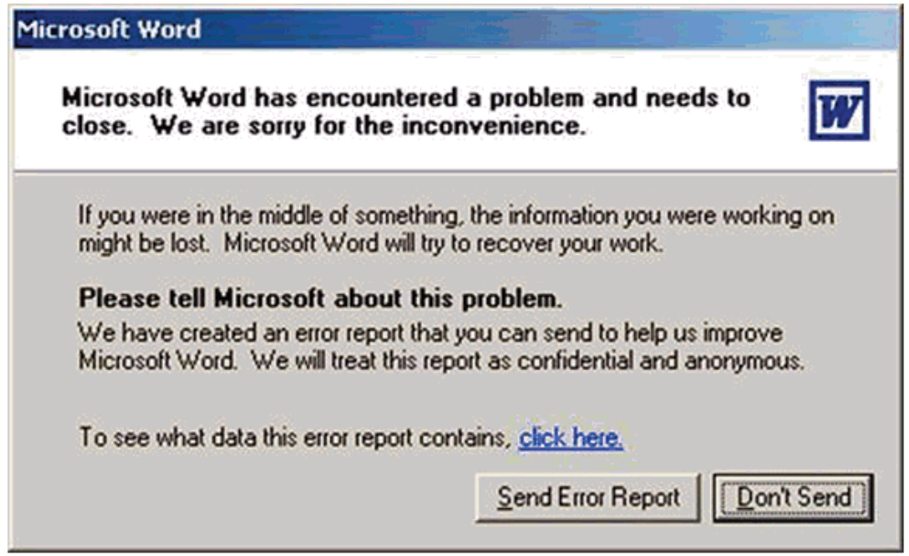

Figure 2.

Microsoft basically for free (http://is.gd/OgkfVo). To sign up, you need a VeriSign certificate. With the certificate you can sign up for WER and obtain the error reports that users have accepted to upload using the dialog box above. These reports are just minidumps as described above.

Some considerations about WER:

- Applications that have installed an exception handler using `SetUnhandledExceptionFilter` (as in the first article) sometimes crash without calling the installed exception filter. In such cases, WER still can be used, thus providing an additional last chance to find bugs.
- WER does not offer the ability to add any additional data or description by users. When using its own mechanism for error re-

porting (possibly a small separate utility), it may ask users to report information about how to reproduce the problem as well as sending additional data files to reproduce the bug. Such a utility can also be called when other problems occur that do not lead to a crash.
- It seems best to use WER as an additional service to report problems. Note also that there has been a number of reports about problems signing up to WER, especially when using a certificate from a different provider than VeriSign.

Other Post-Mortem Techniques: Google Breakpad

According to Google, breakpad "is a library and tool suite that allows you to distribute an application to users with compiler-provided debugging information removed, record crashes in compact "minidump" files, send them back to your server, and produce C and C++ stack traces from these minidumps. Breakpad can also write minidumps on request for programs that have not crashed. Breakpad is currently in use by Google Chrome, Firefox, Google Picasa, Camino, Google Earth, and other projects as well."

This is basically the same mechanism as propagated by `MapAddr` in my first article, but of course, Google has enhanced the functionality:

- Breakpad works on multiple platforms (Windows, Linux, Mac). The minidump format has been adapted on other platforms.
- It includes functions to transmit a crash report directly to your HTTP server
- It can also handle other errors such as C++ pure virtual function calls or invalid parameter calls, and it can be triggered explicitly to create a crash report.

Breakpad (http://code.google.com/p/google-breakpad/) seems to be well developed and can probably handle a number of conflicts that a simple utility like `MapAddr` cannot (i.e., loading DLLs to a different base address than the preferred one). Unfortunately, its documentation is only fragmented. Even the documentation about installing and compiling is not correct. To compile breakpad for Windows, you first must install Python, then run the script:

```
src\tools\gyp\gyp.bat
src\client\windows\breakpad_client.gyp
```

while the current directory is the root directory of breakpad. This will create Visual Studio Solution and project files; for other platforms, makefiles are provided.

Summary

There are a number of ways to capture stack dumps from program crashes and add symbol information to them. Such information is very helpful to resolve bugs and to improve the quality of a program. Post-mortem debugging saves time in finding bugs, as a stack dump often is sufficient to reproduce and solve a bug. The effort needed to acquire this information is minimal.

— Stefan is founder of www.RED-SOFT-ADAIR.com, a company that rescues software development projects in technical crisis.

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

MOVING AN OBJECT DOES NOT DESTROY THE ORIGINAL

Andrew Koenig presents a problem and you may ask: Why can't the compiler recognize `std::move` as a special case that causes `t` to be destroyed? It could notice this statement and simply not destroy `t` afterward, knowing that `std::move` had already done so.

http://www.drdobbs.com/240157784

JDK 8 NOW FEATURE COMPLETE

There are still a few milestones to go before JDK 8 is released as Java SE 8, but we're inching closer.

http://www.drdobbs.com/240157637

SCALA FOR C# DEVELOPERS: USEFUL FEATURES

Scala's immutable values and mutable variables, classes and constructors, and its use of operators as method names.

http://www.drdobbs.com/240157009

JAVA EE 7: AND THEN WHAT?

With little fanfare, Oracle released Java EE 7 last week. This is the most recent release of the enterprise bundle of services you might recall as J2EE. It has come a long way from its early days as an oversized, tangled hairball. Today, it is a slimmed-down, redesigned, responsive, and more usable product than it's ever been. The question, though, is whether anyone cares.

http://www.drdobbs.com/240156899

APIS FOR IMAGE-CAPTURE APPLICATIONS

Online image-scanning applications continue to grow in importance. More organizations need to digitize documents for recordkeeping, safekeeping, and archiving among other reasons. As the practice of document digitizing grows, so does the need for greater options in image capturing. TWAIN, WIA, and DirectShow are three popular APIs for imaging apps. How do they differ and which one should you pick?

http://www.drdobbs.com/240156834

IN THIS ISSUE

Dr.Dobb's