GEORGE MASON UNIVERSITY

The Limit of DTrace

A Failed Attempt at Deadlock Detection

Khemara Chuon

2008-03-27

Solaris operating system is missing crucial probe implementations required for deadlock detection using DTrace. Java deadlock detection is made possible because the Java Virtual Machine implements its own probes that cover different semantic situations. Future implementations of Solaris must include probes that fire on a mutex block regardless of whether the process/thread eventually acquires the lock or not. Otherwise the Solaris operating system is doomed to never support general deadlock detection.

Table of Contents

Abstract

Solaris operating system is missing crucial probe implementations required for deadlock detection using DTrace. Java deadlock detection is made possible because the Java Virtual Machine implements its own probes that cover different semantic situations. Future implementations of Solaris must include probes that fire on a mutex block regardless of whether the process/thread eventually acquires the lock or not. Otherwise the Solaris operating system is doomed to never support general deadlock detection.

Introduction

Deadlocks are a common problem in multithreaded applications and operating systems which result lost resources and impaired performance. Although there are methods available to detect and prevent deadlocks, a majority of all applications and operating systems use the *Ostrich Algorithm* to "just ignore [deadlocks] and hope it doesn't happen."[1] The decision to ignore this issue instead of making the application or operating system more robust stems from the fact that deadlocks rarely occur in practice. A production system may run for years without ever experiencing a deadlock. Adding deadlock prevention code to such a system would slow down all synchronization calls since every call must check whether the synchronization lock may be granted without introducing a deadlock. It is a great waste to slow down an entire application or operating system to handle a case that might not even occur within the year. Therefore many developers simply ignore the problem and expect end-users to detect and resolve deadlocks on their own.

This idea works well for production systems when applications are rigorously tested for errors before being installed and run, but on development systems where developers have incomplete and bug-ridden source code the idea fails to hold. Development systems will frequently have deadlock situations that spawn from multiple developers making incorrect assumptions about their peer's source code. Because the frequency of deadlock situations in a development system is so high, it places a huge burden on the application developer to determine where and how a deadlock is occurring. A deadlock is defined to exist among a set of processes/threads if "every [process/thread] is waiting for an event that can be caused only by another process/thread in the set."[2] This means that the application developer would be required to examine all processes/threads and to piece together the minimal clique of processes/threads involved in the deadlock. Then for each process/thread in the clique the developer would need to inspect the stack trace and discover the flawed timing assumptions that led to this particular deadlock.

In the particular case of the Solaris operating system, intuition tells us that we can easily write a deadlock detector based on the output of Sun's dynamic tracing tool (DTrace). However closer investigation into the semantics of the probe definitions tell us that detecting deadlocks will be impossible in all but a few very specific cases.

[1] Ingalls, Robert P. 2004. CSCI.4210 Operating Systems Deadlock. http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html (accessed March 27, 2008).
[2] Snoeren, Alex C. 2006. CSE 120: Principles of Operating Systems, Lecture 8: Scheduling & Deadlock. http://www.cs.ucsd.edu/classes/fa06/cse120/lectures/120-fa06-l8.pdf (accessed March 27, 2008).

Initial Deadlock Detection Design

DTrace is a tool to observe probes instrumented in an application or operating system. The Solaris Dynamic Tracing (DTrace) Guide offers us some promising looking probes provided by *lockstat* and *plockstat*. The *-block and *-spin probes seem to give us the information we need. They both tell us when a process/thread is waiting on a lock. Deadlock detection is dependent on the fact that processes/threads wait on locks that are owned by other processes/threads that transitively require locks that the initial process/thread owns. Because of this we are assured that when there is a deadlock, there will be at least two block or spin events. Then we merely post-process the information to determine if a cycle existed among the owners of the blocked processes/threads.

We run the *Blocked and Spun Mutex Detection Script* defined in Appendix A and drive it with the *Deadlock Generator Code* also defined in Appendix A. By passing in '2' as an argument into the *Deadlock Generator Code* we expect to see two blocked threads corresponding to the generated deadlock in the output of the *Blocked and Spun Mutex Detection Script*, but surprisingly we will only see one blocked thread or perhaps none at all depending on how the threads were scheduled to run.

A closer inspection into the semantics of the *-block and *-spin probes tell us that they only fire after a blocked or spun thread has "reawakened and successfully acquired the mutex."[3] This will never happen in a deadlock situation because the threads are deadlocked by definition! We need a different set of probes that fire when a process/thread is blocked or spun without the additional requirement that they eventually acquire the lock. Another look at the Solaris Dynamic Tracing (DTrace) Guide tells us that no such probes exist. Thus we find that DTrace is limited by the strict semantics of its probes and cannot detect deadlocks in the general case.

[3] Sun Microsystems Inc. 2005. *Solaris(TM) Dynamic Tracing Guide*. iUniverse. http://docs.sun.com/app/docs/doc/817-6223 (accessed March 27, 2008)

Java Deadlock Detection Design

Although deadlock detection cannot be performed generically throughout the whole operating system, deadlock detection may still be performed within the subset of Java applications. Java Virtual Machines (JVMs) are instrumented separately from the operating system and give us access to new probes such as monitor-contended-enter and monitor-contended-entered. The monitor-contended-enter probe fires as a thread attempts to enter a contended monitor and the monitor-contended-entered probe fires as a probe successfully enters a contended monitor. This is exactly what we need to perform deadlock detection.

We run the *Hotspot Monitor Detection* script defined in Appendix A and drive it with the *Deadlock Generator Code* also defined in Appendix A only to find that no output is being produced! Another look at the hotspot provider documentation states that although the probes are present in the JVM, the probes are dormant by default and must be statically enabled with the '-XX:+ExtendedDtraceProbes' flag on JVM initialization or dynamically enabled using the jinfo utility.[4] However attempts to enable this flag on Solaris Express Developer Edition (1/08, snv79b, x86) running Java SE Runtime Environment 1.6.0_03-b05 were futile. The JVM did not recognize the '-XX:+ExtendedDtraceProbes' flag and failed to initialize, crippling any tests to verify whether the *Hotspot Monitor Detection Script* produces the output necessary to perform deadlock detection.

Although we were unable to perform Java deadlock detection with DTrace under our specific software builds, Java deadlock detection is possible and is already implemented in other JVM monitoring tools like JConsole.[5] Since other tools already exist to detect deadlocks in JVMs, the disconnect between DTrace and the JVM flags seen here is a minor inconvenience. If the '-XX:+ExtendedDtraceProbes' were enabled, the *Hotspot Monitor Detection Script* produced the expected output and the post-processing code correctly identified the deadlocked threads, we still would have only have reproduced work that is already in existence.

[4] Sun Microsystems Inc. 2006. DTrace Probes in HotSpot VM. http://java.sun.com/javase/6/docs/technotes/guides/vm/dtrace.html (accessed March 27, 2008)
[5] Chung, Mandy. 2004. Using JConsole to Monitor Applications. http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html (accessed March 27, 2008)

Summary

The problem of deadlocks still plagues developers on the Solaris operating system. The probes necessary for deadlock detection are not implemented in the current Solaris build so application developers must still dredge through a swamp of processes and threads to locate a deadlock. However future work to add in these missing probes in later Solaris builds would eliminate this problem. Java developers do not suffer from the problem of deadlocks since the Java Virtual Machine implements its own probes and provides monitoring tools like JConsole to detect deadlocks.

Appendix A

Deadlock Generator Code

```
package edu.gmu.cs671.test;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;


public class Test1 {

   /**
    * Called to start this class
    *
    * @param pArguments Ordered list of command line arguments
    */
   public static void main(String[] pArguments) {
      final int aNumThreads = Integer.parseInt(pArguments[0]);
      final Thread[] aThreadList = new Thread[aNumThreads];

      for (int i=0; i<aNumThreads; ++i) {
         final int aCurrentThread = i;
         aThreadList[i] = new Thread("TestThread"+aCurrentThread) {

            /* (non-Javadoc)
             * @see java.lang.Thread#run()
             */
            public void run() {
               acquire(aCurrentThread);
            }

            private void acquire(int pThreadNum) {
               StringBuilder aStringBuilder = new StringBuilder();
               aStringBuilder.append(Thread.currentThread().getName());
               aStringBuilder.append(" attempting to acquire lock ");
               aStringBuilder.append(pThreadNum);
               aStringBuilder.append(".\n");
               System.out.println(aStringBuilder.toString());

               synchronized (aThreadList[pThreadNum]) {
                  aStringBuilder = new StringBuilder();
                  aStringBuilder.append(Thread.currentThread().getName());
                  aStringBuilder.append(" acquired lock ");
                  aStringBuilder.append(pThreadNum);
                  aStringBuilder.append(".\n");
                  System.out.println(aStringBuilder.toString());

                  acquireRecursive(pThreadNum, (pThreadNum+1)%aNumThreads);
               }
            }

            private void acquireRecursive(int pStartThreadNum, int pThreadNum) {
               if (pStartThreadNum == pThreadNum) {
                  StringBuilder aStringBuilder = new StringBuilder();
                  aStringBuilder.append(Thread.currentThread().getName());
                  aStringBuilder.append(" acquired all locks!\n");
                  System.out.println(aStringBuilder.toString());
               } else {
                  StringBuilder aStringBuilder = new StringBuilder();
                  aStringBuilder.append(Thread.currentThread().getName());
                  aStringBuilder.append(" attempting to acquire lock ");
                  aStringBuilder.append(pThreadNum);
                  aStringBuilder.append(".\n");
                  System.out.println(aStringBuilder.toString());

                  synchronized (aThreadList[pThreadNum]) {
```

```
                    aStringBuilder = new StringBuilder();
                    aStringBuilder.append(Thread.currentThread().getName());
                    aStringBuilder.append(" acquired lock ");
                    aStringBuilder.append(pThreadNum);
                    aStringBuilder.append(".\n");
                    System.out.println(aStringBuilder.toString());

                    acquireRecursive(pStartThreadNum, (pThreadNum+1)%aNumThreads);
                }
            }
        }
      };
   }

   BufferedReader System_in = new BufferedReader(new InputStreamReader(
            System.in));
   try {
      System.out.println("Press [Enter] to start threads...");
      String aInput = System_in.readLine();
   } catch (IOException e) {
      e.printStackTrace();
   }

   for (int i=0; i<aNumThreads; ++i) {
      System.out.println("Starting " + aThreadList[i].getName() + "...");
      aThreadList[i].start();
   }
 }

}
```

Blocked and Spun Mutex Detection Script

```
#!/usr/sbin/dtrace -s

/*****************************************************************************
 * DTrace Script: Deadlock Detection
 * Class:         Advanced Operating Systems (cs671-001)
 * Author:        Khemara Chuon
 * Date:          March 27, 2008
 *****************************************************************************/

:::*acquire
{
        self->mutex = (kmutex_t *) arg0;
        self->mutex_owner = mutex_owner(self->mutex);

        printf("\nACQUIRE_START mutex=0x%x mutex_owner=0x%x acquire_thread=%s(%d):0x%x",
                (int64_t) self->mutex,
                (int64_t) self->mutex_owner,
                execname,
                pid,
                (int64_t) curthread);
        jstack(50, 500);
        printf("ACQUIRE_END\n");
}

:::*release
{
        self->mutex = (kmutex_t *) arg0;
        self->mutex_owner = mutex_owner(self->mutex);

        printf("\nRELEASE_START mutex=0x%x mutex_owner=0x%x release_thread=%s(%d):0x%x",
                (int64_t) self->mutex,
```

```
                (int64_t) self->mutex_owner,
                execname,
                pid,
                (int64_t) curthread);
        jstack(50, 500);
        printf("RELEASE_END\n");
}

:::*block
{
        self->mutex = (kmutex_t *) arg0;
        self->mutex_owner = mutex_owner(self->mutex);

        printf("\nBLOCK_START mutex=0x%x mutex_owner=0x%x blocking_thread=%s(%d):0x%x",
                (int64_t) self->mutex,
                (int64_t) self->mutex_owner,
                execname,
                pid,
                (int64_t) curthread);
        jstack(50, 500);
        printf("BLOCK_END\n");
}

:::*spin
{
        self->mutex = (kmutex_t *) arg0;
        self->mutex_owner = mutex_owner(self->mutex);

        printf("\nSPIN_START mutex=0x%x mutex_owner=0x%x spinning_thread=%s(%d):0x%x",
                (int64_t) self->mutex,
                (int64_t) self->mutex_owner,
                execname,
                pid,
                (int64_t) curthread);
        jstack(50, 500);
        printf("SPIN_END\n");
}
```

Hotspot Monitor Detection Script

```
#!/usr/sbin/dtrace -s

/*****************************************************************************
 * DTrace Script: Deadlock Detection
 * Class:         Advanced Operating Systems (cs671-001)
 * Author:        Khemara Chuon
 * Date:          March 27, 2008
 *
 * Note:          Requires –XX:+ExtendedDtraceProbes flag to be enabled traced
 *                JVMs.
 *****************************************************************************/

:::monitor-contended-enter
{
        self->thread_name = copyinstr(arg0);
        self->mutex = copyinstr(arg1);
        self->mutex_owner = "unknown";

        printf("\nMONITOR_START mutex=%s mutex_owner=%s monitor_thread=%s(%d/%d):%s",
                self->mutex,
```

```
                self->mutex_owner,
                execname,
                pid,
                tid,
                self->thread_name);
        jstack(50, 500);
        printf("MONITOR_END\n");
}

:::monitor-contended-entered
{
        self->thread_name = copyinstr(arg0);
        self->mutex = copyinstr(arg1);
        self->mutex_owner = "unknown";

        printf("\nMONITOR_ENTERED_START mutex=%s mutex_owner=%s monitor_thread=%s(%d/%d):%s",
                self->mutex,
                self->mutex_owner,
                execname,
                pid,
                tid,
                self->thread_name);
        jstack(50, 500);
        printf("MONITOR_ENTERED_END\n");
}
```