



Ensuring Code Quality in Multi-threaded Applications

How to Eliminate Concurrency Defects with Static Analysis

Ben Chelf, Coverity CTO
John Kodumal, Coverity Advanced Technology Team

Introduction

Most developers would agree that consumers of software today continually demand more from their applications. Because of its pace of evolution to date, the world now anticipates a seemingly endless expansion of capabilities from their software, regardless of where that software is applied. For the last 40 years of computing, Moore's Law had held true, with the processing power of computers doubling every two years. This constant pace of innovation provided software developers with critical hardware resources, enabling them to expand the features and functionalities of their applications without concern for performance. However, Moore's Law became less law-like after single-core processing reached its performance ceiling. In response to this, hardware manufacturers developed new multi-core (or multi-processor) devices to accomplish the speed gains to which the world had grown accustomed.

Today, the world of software development is presented with a new challenge. To fully leverage this new class of multi-core hardware, software developers must change the way they create applications. By turning their focus to multi-threaded applications, developers will be able to take full advantage of multi-core devices and deliver software that meets the demands of the world. But this paradigm of multi-threaded software development adds a new wrinkle of complexity for those who care the utmost about software quality. Concurrency defects such as race conditions and deadlocks are software defect types that are unique to multi-threaded applications. Complex and hard-to-find, these defects can quickly derail a software project. To avoid catastrophic failures in multi-threaded applications, software development organizations must understand how to identify and eliminate these deadly problems early in the application development lifecycle.

Did You Use Multi-core Today?

The most advertised use of multi-core processing has been in consumer markets on devices ranging from PCs that leverage processors from Intel or AMD to gaming consoles such as Sony's Play Station 3. In these markets, the number of processors is a clear competitive advantage to which consumers are becoming attuned. In enterprise markets, multi-core technology has been adopted in a number of processor types including those used in the digital signal, network and embedded markets.

Moving to Multi-threaded Application Development

Most software development over time has been predicated on single-core hardware. Therefore, the collective knowledge of software developers across universities, companies, organizations and countries has been based primarily on single processor hardware platforms. Despite the fact that multi-processor systems have existed for a long time, they have not been the focus of general purpose software development. Instead, early multi-processor hardware was preserved for computationally-intensive applications. As a result, multi-threaded software development has not been much of a focus in mainstream software development.

Now that the world is moving to multi-core on nearly everything, every software developer must become versed in multi-threaded development. There are armies of developers that are now challenged with learning how to take advantage of multi-core on the fly, because they have no choice in delivering on the demands of faster software release after release.

The bad news for software developers is that multi-threaded applications introduce a host of concurrency-related errors, which historically, have been impossible to effectively test for. When they occur in the field, concurrency defects can be catastrophic for software, requiring days, if not weeks or months, to diagnose and repair.

Developers responsible for creating high reliability applications, such as those in the embedded space, inherit this dangerous new blind spot regarding code quality when they move from single- to multi-threaded applications. This paper will review the most common pitfalls that software developers face when creating multi-threaded applications, and how innovative new software analysis capabilities are emerging that will help development organizations capitalize on the exciting new world of multi-core hardware.

Why Concurrency (Multi-core) Defects Are So Dangerous

To take advantage of multi-core hardware, software developers are required to create multi-threaded applications that can execute multiple operations concurrently. Concurrency creates new complexities in the software development process that can introduce hard-to-find, crash-causing software defects. Because widespread multi-threaded development is still in its relative infancy, many of today's existing defect detection tools are ill prepared to help identify and eliminate this new class of defects.

Common Concurrency Defects

Race Condition Multiple threads access the same shared data without the appropriate locks to protect access points. When this defect occurs, one thread may inadvertently overwrite data used by another thread, leading to both loss of information and data corruption.

Thread Block A thread calls a long-running operation while holding a lock thereby preventing the progress of other threads. When this defect occurs, application performance can drop dramatically due to a single bottleneck for all threads.

Deadlock Two or more threads wait for locks in a circular chain such that the locks can never be acquired. When this defect occurs, the entire software system may halt, as none of the threads can either proceed along their current execution paths or exit.

Testing software code is notoriously difficult, even for single-threaded applications. One reason this is a challenge, is that properly testing an application requires the simulation of all possible inputs to the program. Many organizations are satisfied when they achieve 95% line coverage in their programs (meaning 95% of all lines of code in their program are executed at least once in their test suite). However, this metric of coverage does not even come close to the total number of possible execution paths through a code base. Each decision point in a program means at least a factor of two in the number of paths through that program.

Multi-threaded applications add a new level of complexity to the program that results in an exponential increase in the number of possible run time scenarios. See Figure 1 below for an example as to how turning a simple single-threaded application into a multi-threaded application increases the complexity tremendously.

Figure 1: Simple single-threaded program

```
void main() {
    int x = input_from_user();

    if (x < 100) {
        printf("x is less than 100!\n");
    } else {
        printf("x is at least 100 or greater!\n");
    }
}
```

To completely test the program in Figure 1, the software developer would simply need to make sure that they fed the program a value that was less than 100 as well as a value that was greater than or equal to 100. Then the two paths through the program would be explored and the program would be tested entirely. (NOTE: This assumes that nothing interesting happens in the `input_from_user` function - admittedly a bad assumption in software development!). Now imagine that the end-user wanted this program to input two values instead of one with the same logic after the input. This change would increase the complexity of the single-threaded application in such a way that the software developer would need to explore four different paths to fully test the program.

Now imagine that a requirement comes in from the field to make this application multi-threaded to take advantage of a dual-core processor. This change would require that the code in Figure 1 would be executed simultaneously by two different threads to take in the two inputs. Assuming that each statement is made up of three instructions (likely an underestimate), the number of possible interleavings of the instructions in the two threads leads to a mind boggling 194,480 different execution possibilities.

By moving to multi-threaded code in this simplified example, complexity explodes by five orders of magnitude, creating a tremendous challenge for any tester of software. If this type of defect does happen to slip into the field, developers are then faced with a dramatic increase in the number of avenues that require exploration before they can determine the root cause of the problem. Worse, for the first time, developers may not be able to reproduce problems that occur in multi-threaded applications because they are not in control of how their application will be executed when it runs. The operating system and thread scheduler together decide when each thread will be executed as multiple threads execute.

Fortunately new technology in Coverity Prevent is helping developers avoid costly concurrency defects. Leveraging breakthroughs in static analysis, Coverity's solution can analyze code prior to run-time enabling developers to identify and eliminate onerous multi-threaded defects including race conditions, thread blocks and deadlocks.

Deadlocks

Deadlocks (situations where a multi-threaded program cannot make any progress) are a particularly difficult concurrency issue to debug. They arise when the order of lock acquisition used by the program is inconsistent. A simple example is lock inversion, which happens when one thread attempts to acquire a lock **(a)**, then holds on to the lock while attempting to acquire lock **(b)** . If another thread tries to acquire the locks in the opposite order (acquiring **b** before **a**), neither thread can make progress, and the program is deadlocked. A sample of this type of software defect can be found in Figure 2. (Note: While the remaining code examples in this paper are in Java, all the defects discussed herein can occur in any programming language. Coverity Prevent discovers concurrency defects in C, C++, and Java code.)

Figure 2: Deadlock Defect in Java Code

```
public class Deadlock {
    static Object a;
    static Object b;

    public static void lock1() {
        synchronized(a) {
            ...
            synchronized(b) {
                ...
            }
        }
    }

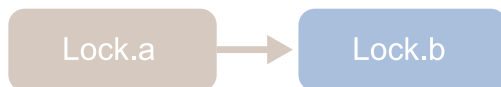
    public static void lock2() {
        synchronized(b) {
            ...
            synchronized(a) {
                ...
            }
        }
    }
}
```

If two distinct threads call methods **lock1** and **lock2**, an unlucky scheduling might have the first thread acquire lock **(a)**, while the second thread acquires lock **(b)**. In this state, it is impossible for either thread to acquire the lock it needs to continue execution or release the lock it holds. More complicated deadlocks exist that involve the acquisition of multiple locks across multiple threads.

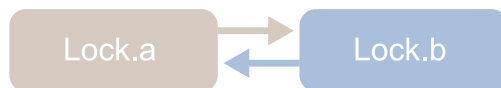
One way that advanced static analysis technology detects deadlocks is to ensure that all locks in the program are acquired and released in a consistent order. More concretely, the static analysis can construct an acyclic lock graph for a program. Nodes in a lock graph represent lock names, and an edge between nodes **(a)** and **(b)** denotes that at some point, lock **(a)** was held while lock **(b)** was acquired.

If a lock graph contains a cycle, the program may have a lock ordering or lock inversion deadlock. Innovative static analysis is now capable of finding these lock ordering defects inter-procedurally (through method calls), as these are likely to be missed by conventional code review or other defect detection techniques.

In Figure 2 above, Coverity Prevent would create the following lock graph for method **lock1**:



The edge from **Lock.a** to **Lock.b** indicates that **Lock.a** should always be acquired before **Lock.b**. Analysis of method **lock2**, however, causes the analysis to add an edge from **Lock.b** to **Lock.a**:



This violates the requirement that the lock graph be acyclic. Therefore, a defect would be reported in **lock2**.

One of the biggest difficulties in leveraging static analysis to detect deadlocks is the issue of naming locks. In Figure 2, it is clear that “synchronized(a)” locks the static field “a” declared in the “Lock” class. But naming is not always so obvious. Consider the following refactoring of **lock2**.

Figure 3: Refactoring 'lock2'

```
public static void lock2() {
    synchronized(b) {
        ...
        f(a);
    }
}
public static void f(Object x) {
    synchronized(x) {
        ...
    }
}
```

Coverity Prevent is sophisticated enough to determine that the call to method (**f**) has the same effect as before. It should attempt to add an edge from **Lock.b** to **Lock.a**, fail, and report a defect. The details behind the lock naming algorithm are part of the innovation that makes Coverity Prevent so powerful at finding complex, inter-procedural deadlock defects in C, C++ and Java code.

Race Conditions

Race conditions are another potentially disastrous type of concurrency defect that is endemic to multi-threaded applications. A race condition occurs when multiple threads access the same piece of data concurrently, at least one thread accessing the data is writing to the data (as opposed to simply reading the data), and the execution depends critically on the relative timing of events.

This is as concrete a definition as can be achieved in languages like C and C++ that do not have a formally defined memory model. In Java, a more precise (but not necessarily equivalent) definition can be formulated based on the Java memory model. A data race occurs when a variable is read by one or more threads and written by at least one thread, where the read and write events are not ordered by the 'happens-before' relationship. The happens-before relationship precisely defines when events are made visible to other threads (check the Java Language Specification for a complete definition of happens-before). See Figure 4 below for an example of this.

Figure 4: Race Condition

```
public class Race {
    int count = 10;
    Object lock;

    public void increment() {
        count++;
    }
    ...
}
```

Suppose that an instance of class **Race** is used to keep a counter of active events across multiple threads. One problem with **Race** is that the ++ operator is not atomic. If **increment** is called 10 times from two different threads, the final value of count could be less than 10, because some of the writes to **count** can be clobbered by writes from the second thread. One possible fix is to synchronize the increment method as shown in Figure 5.

Figure 5: Race Condition with Synchronized Increment Method

```
public class Race {
    int count = 10;
    Object lock;

    public void increment() {
        synchronized(lock) {
            count++;
        }
    }
    ...
}
```

With **count** accessed in a synchronized block, interleaved calls to increment on the same instance object are eliminated. Coverity Prevent helps programmers avoid race conditions by inferring and enforcing ‘guarded-by’ relationships on shared fields in the program. A field (**f**) is guarded-by lock (**l**) if accesses to (**f**) must be protected by holding lock (**l**). In Figure 5, the count variable is guarded-by the **lock** field of the **Race** class. The set of guarded-by relationships essentially define the locking discipline for each class in the program.

Coverity Prevent is the first static analysis solution to infer guarded-by relationships. When a preponderance of accesses to a field occur with the same lock held, a guarded-by relationship is inferred between the accessed field and the held lock. This may sound simple, but a number of sophisticated techniques are necessary to make this approach tenable.

Most importantly, the same lock-naming algorithm used in the context of deadlock detection is applied to associate fields with the specific lock used to protect access. This serves two purposes. First, it yields stronger correlations, since fields accessed incidentally with different locks held are not used to infer guarded-by relationships. Second, it allows Coverity Prevent to suggest a possible way to correct the discovered defect, such as enclosing the access in a synchronized scope with the suggested lock.

Another important technique in discovering race conditions with static analysis is to infer information about the program’s thread execution model. This includes information such as where threads are created, which functions can be called by multiple threads simultaneously, and which pieces of data are shared among threads. Because Coverity Prevent understands a program’s thread execution model, it is able to eliminate false positives results due to idioms such as data handoff between threads.

A final, important technique in discovering race conditions is the use of dataflow analysis to track values both intra- and inter-procedurally. Tracking data flow makes the analysis less brittle to simple refactorings such as common subexpression elimination that may preserve the semantics of a given program, but could still confuse the statistical inference. Dataflow analysis provides a more semantic representation of the program, making code analysis less sensitive to syntactic changes.

Once guarded-by relationships have been inferred, a best practice that can be leveraged by Java developers is to permanently enforce the discovered relationships using annotations. Annotations are a Java language-supported mechanism for associating metadata with program elements. In this case, Coverity Prevent provides an annotation **@GuardedBy(String lockname)** that can be placed on fields to denote the lock that should be held when accessing the annotated field. Note the change in Figure 6 on the next page from our previous example in Figure 5.

Figure 6: Updated Race Condition with Synchronized Increment Method

```
public class Race {
    @GuardedBy("Race.lock")
    int count = 10;
    Object lock;
    public void increment() {
        count++;
    }
    ...
}
```

In the example in Figure 6, Coverity Prevent will report a defect in `increment` since the access to `count` is not protected by the appropriate **lock**.

Conclusion

Multi-core hardware is clearly increasing software complexity by driving the need for multi-threaded applications. Based on the rising rate of multi-core hardware adoption in both enterprise and consumer devices, the challenge of creating multi-threaded applications is here to stay for software developers. In the coming years, multi-threaded application development will most likely become the dominant paradigm in software.

As this shift continues, many development organizations will transition to multi-threaded application development on the fly. This creates new liabilities for development teams in terms of application quality and security, because their code is now vulnerable to a host of concurrency defects – a class of defect that can easily slip past manual code review and traditional testing techniques.

Developers moving to multi-threaded applications need advanced new testing capabilities to help them control this new cause of software complexity. Because they are nearly impossible to replicate in dynamic testing environments, static analysis is uniquely suited to play an important role in eliminating concurrency defects early in the software development lifecycle. However, due to their underlying complexity, some concurrency defect types (such as race conditions) have historically escaped detection by conventional static analysis tools.

Today, sophisticated new technology from Coverity is advancing the science of static analysis to help developers meet the challenge of creating multi-threaded applications. By arming developers with static analysis capabilities that detect complex concurrency defects early in the development lifecycle, organizations will accelerate their ability to produce and release multi-threaded applications with greater confidence.

About Coverity

Coverity (www.coverity.com), the leader in improving software quality and security, is a privately held company headquartered in San Francisco. Coverity's groundbreaking technology removes the barriers to writing and delivering complex software by automatically finding and helping to fix critical software defects and security vulnerabilities as software is written. More than 350 leading companies choose Coverity because it scales to tens of millions of lines of code, has the lowest false positive rate while providing 100 percent path and value coverage. Companies like Juniper Networks, Symantec, McAfee, Synopsys, NASA, Palm and Wind River rely on Coverity's tools to find and eliminate critical defects from their mission-critical code.



185 Berry Street, Suite 3600
San Francisco, CA 94107
Phone: (800) 873-8193

<http://www.coverity.com/>
sales@coverity.com