Dr. Dobb's
Software Tools for the Professional Programmer

Dr. Dobb's Journal, January 2006

Detecting Potential Deadlocks

Uncovering problems before they occur

By Tomer Abramson

Tomer is a software engineer for Verint Systems. He can be contacted at tomer.abramson@verint.com.

One of the hardest problems to solve in multithreaded programming is that of deadlocks. While there are numerous applications that detect and analyze deadlocks, most (if not all) of them perform a "post" deadlock analysis. In other words, they analyze deadlocks after they occur.

There's a problem with this approach. For instance, the following code (where *T* is for Thread, *R* for Resource) contains a potential deadlock. However, you can run it 100 times and it will not deadlock. This means that tools that perform post-deadlock analysis will not report on the problem.

```
T1: T2:
Lock(R1) Lock(R2)
Lock(R2) Lock(R1)

Main()
Run(T1);
Run(T2);
```

Post-deadlock analysis forces you to have deadlock detection applications constantly running. If deadlock did not occur during the debug session, you must continue using the deadlock detection application in release versions as well, something that can be problematic in terms of performance issues, stability, and the like.

The solution is to use tools that detect potential deadlocks every time the code is executed, even if no deadlock occurred during execution time. In this article, I present the Deadlock Detection Application (DDA), which detects potential deadlocks before they occur. (The complete source code for the DDA is available electronically; see "Resource Center," page 4.)

As Figure 1 illustrates, the DDA contains the following components:

- DDA Agent. Initializes and terminates the DDA. This is the only component from the DDA used directly by the application.
- DDA Manager. Manages the main flow of the DDA.
- DDA View. Implements a simple GUI and notifies the DDA Manager about user actions.
- Locks Logger. Keeps a log of each lock/unlock operation made in the application.
- Lock Data Queue. The queue is the object through which the application communicates with the DDA. The application passes to the queue information about each lock/unlock operation. The DDA Logger retrieves

the information from the queue and stores it for future use.
- Lock Sequence Builder. Receives as input the locks information in the format that the logger holds, and translates it to a new format called "lock sequence."
- Lock Sequence Analyzer. Takes as input a list of lock sequences, analyzes them, and decides if the application contains a potential deadlock. If so, the analyzer returns the deadlock cycle.

The DDA program flow is: When the application starts to run, the DDA hooks all synchronization functions (*WaitForSingleObject(), CreateMutex()*, and so on). Once functions are hooked, each lock operation triggers an event that inserts the lock's info into the queue from which the DDA's logger pulls data. This way, the DDA tracks each and every lock operation in the application.

You then perform a few tests. When you finish running the tests, you tell the DDA (via a simple GUI) to start analyzing the collected information. The DDA analyzes the locks and searches for potential deadlocks. If a potential deadlock is detected, then the DDA tells you and displays a detailed description of the test that leads to a deadlock.

The Algorithm

Example 1 contains a potential deadlock that occurs with the following scenario: *T1* executes lines 1 to 3. At this point, *T1* locks *R1*. *T2* executes line 1. *T2* then locks *R3*. Now *T1* tries to lock *R3* (in line 4) while *R3* is locked by *T2,* and *T2* tries to lock *R1* (in line 2) while *R1* is locked by *T1*. The application is deadlocked.

Now assume a different scenario: *T2* starts to run only after *T1* has completed. In this scenario no deadlock occurs.

In the first phase, the DDA tracks and logs all lock/unlock operations in the application. In the second phase, the DDA has a list for each thread, where each list contains the lock/unlock operations performed by the thread (*L*=Lock, *U*=Unlock, *T*=Thread, *R*=Resource):

```
T1: L(R1), L(R2), U(R2), L(R3), L(R4), U(R1), U(R3)
T2: L(R3), L(R1), U(R3), L(R2), U(R2), U(R1)
```

In this second phase, the DDA uses logger information to find all the lock sequences in the application. A lock sequence represents the current resources locked by a specific thread at a given time. If you have a lock sequence of (*R1*, *R2*), it means that at a specific time during execution there was a thread that locked resources *R1* and *R2* (in this specific order!). For example:

```
Lock(R1) // curr lock sequence: R1
Lock(R2), // curr lock sequence: R1, R2
Unlock(R2) // curr lock sequence: R1
Lock(R3) // curr lock sequence: R1, R3
```

Lock sequences that are subsequences of other sequences are dropped.

This example has four lock sequences:

```
1. T1: R1, R2
2. T1: R1, R3, R4
3. T2: R3, R1
4. T2: R1, R2
```

Now you are ready to search for potential deadlocks, which exist if the lock sequences contain cyclic dependencies. The most intuitive way to check for such cycles is to represent the lock sequences as a directed graph. A node in the graph is a resource, while a directed vertex is drawn from a resource in the lock sequence to the next resource in the sequence. For a simple lock sequence such as *R1, R2*, the graph looks like *R1->R2*. The graph that corresponds to the lock sequences in the example is:

```
R1 -> R2
```

```
^|
|v

R3 -> R4
```

It is easy to see that the graph contains the cycle *R1->R3->R1*, which represents a potential deadlock upon which the DDA reports.

Implementation

The DDA, which was developed and tested with Visual C++ 6 under Windows 2000, was implemented in a separate DLL. The DDA is initialized by defining in the application a global object of type *DDAAgent*, a simple class that starts the DDA in the constructor and terminates it in the destructor. This is the only class that the DDA DLL exports. The rest of the communication between the application and the DDA is done via the Windows messaging queue.

When the application performs lock/unlock, a notification is sent to the queue. To make the operation of sending a notification to the queue transparent to the application, the DDA wraps all synchronization functions by hooking them. The hooking function technique (introduced and implemented by John Robbins in his *MSDN* October 1998 "BugSlayer" column) lets you take control over a function (or set of functions) so that each call to the function is routed to another function that you defined.

To illustrate how hooked functions are used by the DDA, I focus on *WaitForSingleObject()*. Listing One (available electronically) presents the function *MyWaitForSingleObject()*, which acts as a wrapper for *WaitForSingleObject()* that is called every time the application calls *WaitForSingleObject()*.

MyWaitForSingleObject() gets the same parameters as *WaitForSingleObject()*. What this function does is collect information about the lock action (such as the ID of the locking thread, the handle to the locked resource, the stack trace, and so on), encapsulates the information into the structure *LockData*, and sends the information to the DDA via the queue. Finally, it calls the original *WaitForSingleObject()*. This flow is similar for all the hooked functions: getting lock information, sending information to the DDA via the messaging queue, and calling the original function. The complete set of functions can be found in the file hookedFunctions.cpp (also available electronically).

Again, *LockData* is the structure that encapsulates all the information about the lock. If you want to add additional information about each lock, just add new fields to the *LockData* (implemented in lockData.h; available electronically).

On the DDA's side of the queue, the logger runs in a separate thread, retrieves lock data from the queue, and stores it in a list. There is a separate list for each thread: If a lock operation was made by thread *X*, its lock data is saved in a different list than if the operation was made by thread *Y*. The *locksLogger* class (available electronically) is implemented as a Singleton, meaning there could be only one instance of it in the application. This is done to assure that all information about the locks is stored in one place.

The main flow of the DDA is determined by *DDAMgr*, which is responsible for hooking the functions at start time and unhooking them on termination. It also controls the logger and notifies it when to start/stop. To analyze the information stored in the logger, *DDAMgr* uses the *LockSequenceBuilder* and *LockSequenceAnalyzer* helper classes. The builder transforms the info stored in the logger into a lock sequence, and the analyzer looks for cycles in the lock sequences. I used STL lists to implement the lock sequences and the lists of lock data.

Integrating the DDA

As Example 2 illustrates, to integrate the DDA into your application, you need to perform three steps:

1. Add the lib DAA.lib to your project settings (under the link tab, in the object/library modules field).

2. Add to the file that contains your *main()* function an *#include* to the DDAAgent.h file.
3. Declare a global object of type *DDAAgent*.

To benefit from the DDA, you must produce .pdb files for your application. This is necessary to get the stack trace for problematic locks. I've also included a directory named "Integration" that contains all the components needed for the integration: DDA.lib, DDA.dll, and DDAAgent.h (the dll and lib were compiled in debug mode).

Using the DDA

Once you've integrated the DDA into your application, it runs automatically when you launch the program. Once the application is running, you will see the DDA GUI (Figure 2) in the upper left corner of the screen. Exercise your application by running a few scenarios. When you want the DDA to analyze the scenarios, push the Stop button from the GUI. If there is a potential deadlock, the DDA pops up a message box with a detailed description about the deadlock cycle (Figure 4). If you do not press Stop, the DDA automatically checks for potential deadlocks when you exit the application. If you see "Error-Disabled!," the DDA has failed to initialize and is not functional. Also, the DDA works in debug and release mode.

Example

To demonstrate the power of the DDA, I've provided a demo program (see Listing Two; also available electronically) that includes three mutexes and three threads. Each thread performs the following locks:

Thread1: Lock(Mutex1), Lock(Mutex2)
Thread2: Lock(Mutex2), Lock(Mutex3)
Thread3: Lock(Mutex3), Lock(Mutex1)

Figure 3 is the output of the demo program. Although this scenario contains a potential deadlock (the cycle: *Mutex1*-> *Mutex2*-> *Mutex3*-> *Mutex1*), the application terminates successfully (no deadlock occurred). The DDA detects the potential deadlock and prompts you.

The DDA's output (Figure 4) shows you each lock that is part of the deadlock cycle (all locks, in this case). For each lock it shows the handle to the locked object, the ID of the locking thread, and the most important thing—where in the code the lock was performed (if desired, the DDA can display a full stack trace for each lock). As part of the example, I've included a directory named "Test" that contains the compiled test application. (The test was compiled in debug mode. You must have Visual C++ installed on your computer in order to run it.)

Caveats

The DDA still needs some work to execute perfectly. For now, it does not support *WaitForMultipleObjects()* or events. While the DDA can be extended to support *WaitForMultipleObjects()*, I did not do it because it makes the basic idea and the code much harder to understand. Events are a special type of synchronization object that have a unique behavior, so handling them correctly is not straightforward.

Another problem is a case where the DDA can report about a false deadlock. Think of the following scenario:

T1 T2
Lock(R1) Lock(R1)
Lock(R2) Lock(R3)
Lock(R3) Lock(R2)

The DDA reports about a potential deadlock cycle: *R2*-> *R3*-> *R2*, while the aforementioned code never deadlocks because each thread first tries to acquire a lock on *R1*. This bug can be easily fixed. However, leaving it as-is may be a good idea because the code may cause problems in other scenarios.

References

Robbins, John. "BugSlayer," *MSDN*, October 1998.

"Tech Tips (Identifying Your Caller)," *Windows Developer's Journal*, December 2000.

DDJ