Microsoft Windows

Best Practices for Creating DLLs

May 17, 2006

Abstract

A dynamic-link library (DLL) is shared code and data that an application can load and call at run time. Advantages of using DLLs include reduced code footprint, lower memory utilization due to single-copy-sharing, flexible development and testing, modularity, and functional isolation. This paper provides guidelines for DLL developers to help in building more robust, portable, and extensible DLLs.

The current version of this paper is maintained on the Web at:
http://www.microsoft.com/whdc/driver/kernel/DLL_bestprac.mspx

Contents

Introduction 3
General Best Practices 4
Deadlocks Caused by Lock Order Inversion 6
Best Practices for Synchronization 7
Recommendations 8
References 8

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2006 Microsoft Corporation. All rights reserved.

Microsoft, Windows, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

© 2006 Microsoft Corporation. All rights reserved.

Introduction

A dynamic-link library (DLL) is shared code and data that an application can load and call at run time. A DLL typically exports a set of routines for applications to use and contains other routines for internal use. This technique enables code reuse by allowing multiple applications to share common functionality in a library and load it on demand. Advantages of using DLLs include reduced code footprint, lower memory utilization due to single-copy-sharing, flexible development and testing, modularity, and functional isolation.

Creating DLLs presents a number of challenges for developers. DLLs do not have system-enforced versioning. When multiple versions of a DLL exist on a system, the ease of being overwritten coupled with the lack of a versioning schema creates dependency and API conflicts. Complexity in the development environment, the loader implementation, and the DLL dependencies has created fragility in load order and application behavior. Lastly, many applications rely on DLLs and have complex sets of dependencies that must be honored for the applications to function properly. This document provides guidelines for DLL developers to help in building more robust, portable, and extensible DLLs.

The three main components of the DLL development model are:

- **The library loader**. DLLs often have complex interdependencies that implicitly define the order in which they must be loaded. The library loader efficiently analyzes these dependencies, calculates the correct load order, and loads the DLLs in that order.
- **The DllMain entry-point function**. This function is called by the loader when it loads or unloads a DLL. The loader serializes calls to **DllMain** so that only a single **DllMain** function is run at a time. For more information, see http://msdn.microsoft.com/library/en-us/dllproc/base/dllmain.asp.
- **The loader lock**. This is a process-wide synchronization primitive that the loader uses to ensure serialized loading of DLLs. Any function that must read or modify the per-process library-loader data structures must acquire this lock before performing such an operation. The loader lock is recursive, which means that it can be acquired again by the same thread.

© 2006 Microsoft Corporation. All rights reserved.

Figure 1 illustrates what happens when a library is loaded.

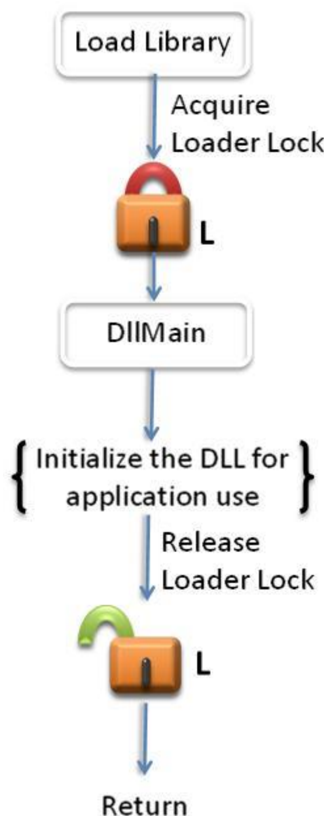

Figure 1. What Happens When a Library Is Loaded

Improper synchronization within **DllMain** can cause an application to deadlock or access data or code in an uninitialized DLL. Calling certain functions from within **DllMain** causes such problems.

General Best Practices

DllMain is called while the loader-lock is held. Therefore, significant restrictions are imposed on the functions that can be called within **DllMain**. As such, **DllMain** is designed to perform minimal initialization tasks, by using a small subset of the Microsoft® Windows® API. You cannot call any function in **DllMain** that directly or indirectly tries to acquire the loader lock. Otherwise, you will introduce the possibility that your application deadlocks or crashes. An error in a **DllMain** implementation can jeopardize the entire process and all of its threads.

The ideal **DllMain** would be just an empty stub. However, given the complexity of many applications, this is generally too restrictive. A good rule of thumb for **DllMain** is to postpone as much initialization as possible. Lazy initialization increases robustness of the application because this initialization is not performed while the loader lock is held. Also, lazy initialization enables you to safely use much more of the Windows API.

Some initialization tasks cannot be postponed. For example, a DLL that depends on a configuration file should fail to load if the file is malformed or contains garbage. For this type of initialization, the DLL should attempt the action and fail quickly rather than waste resources by completing other work.

© 2006 Microsoft Corporation. All rights reserved.

You should never perform the following tasks from within **DllMain**:

- Call **LoadLibrary** or **LoadLibraryEx** (either directly or indirectly). This can cause a deadlock or a crash.
- Synchronize with other threads. This can cause a deadlock.
- Acquire a synchronization object that is owned by code that is waiting to acquire the loader lock. This can cause a deadlock.
- Initialize COM threads by using **CoInitializeEx**. Under certain conditions, this function can call **LoadLibraryEx**.
- Call the registry functions. These functions are implemented in Advapi32.dll. If Advapi32.dll is not initialized before your DLL, the DLL can access uninitialized memory and cause the process to crash.
- Call **CreateProces**. Creating a process can load another DLL.
- Call **ExitThread**. Exiting a thread during DLL detach can cause the loader lock to be acquired again, causing a deadlock or a crash.
- Call **CreateThread**. Creating a thread can work if you do not synchronize with other threads, but it is risky.
- Create a named pipe or other named object (Windows 2000 only). In Windows 2000, named objects are provided by the Terminal Services DLL. If this DLL is not initialized, calls to the DLL can cause the process to crash.
- Use the memory management function from the dynamic C Run-Time (CRT). If the CRT DLL is not initialized, calls to these functions can cause the process to crash.
- Call functions in User32.dll or Gdi32.dll. Some functions load another DLL, which may not be initialized.
- Use managed code.

The following tasks are safe to perform within **DllMain**:

- Initialize static data structures and members at compile time.
- Create and initialize synchronization objects.
- Allocate memory and initialize dynamic data structures (avoiding the functions listed above.)
- Set up thread local storage (TLS).
- Open, read from, and write to files.
- Call functions in Kernel32.dll (except the functions that are listed above).
- Set global pointers to NULL, putting off the initialization of dynamic members. In Microsoft Windows Vista™, you can use the one-time initialization functions to ensure that a block of code is executed only once in a multithreaded environment.

© 2006 Microsoft Corporation. All rights reserved.

Deadlocks Caused by Lock Order Inversion

When you are implementing code that uses multiple synchronization objects such as locks, it is vital to respect lock order. When it is necessary to acquire more than one lock at a time, you must define an explicit precedence that is called a *lock hierarchy* or *lock order*. For example, if lock A is acquired before lock B somewhere in the code, and lock B is acquired before lock C elsewhere in the code, then the lock order is A, B, C and this order should be followed throughout the code. Lock order inversion occurs when the locking order is not followed—for example, if lock B is acquired before lock A. Lock order inversion can cause deadlocks that are difficult to debug. To avoid such problems, all threads must acquire locks in the same order.

It is important to note that the loader calls **DllMain** with the loader lock already acquired, so the loader lock should have the highest precedence in the locking hierarchy. Also note that code only has to acquire the locks it requires for proper synchronization; it does not have to acquire every single lock that is defined in the hierarchy. For example, if a section of code requires only locks A and C for proper synchronization, then the code should acquire lock A before it acquires lock C; it is not necessary for the code to also acquire lock B. Furthermore, DLL code cannot explicitly acquire the loader lock. If the code must call an API such as **GetModuleFileName** that can indirectly acquire the loader lock and the code must also acquire a private lock, then the code should call **GetModuleFileName** before it acquires lock P, thus ensuring that load order is respected.

Figure 2 is an example that illustrates lock order inversion. Consider a DLL whose main thread contains **DllMain**. The library loader acquires the loader lock L and then calls into **DllMain.** The main thread creates synchronization objects A, B, and G to serialize access to its data structures and then tries to acquire lock G. A worker thread that has already successfully acquired lock G then calls a function such as **GetModuleHandle** that attempts to acquire the loader lock L. Thus, the worker thread is blocked on L and the main thread is blocked on G, resulting in a deadlock.

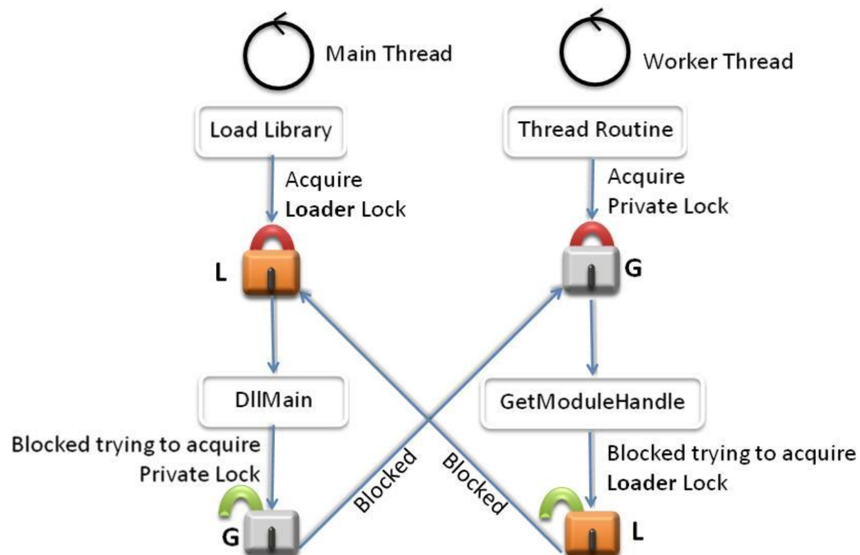

Figure 2. Deadlock Caused by Lock Order Inversion

To prevent deadlocks that are caused by lock order inversion, all threads should attempt to acquire synchronization objects in the defined load order at all times.

© 2006 Microsoft Corporation. All rights reserved.

Best Practices for Synchronization

Consider a DLL that creates worker threads as part of its initialization. Upon DLL cleanup, it is necessary to synchronize with all the worker threads to ensure that the data structures are in a consistent state and then terminate the worker threads. Today, there is no straightforward way to completely solve the problem of cleanly synchronizing and shutting down DLLs in a multithreaded environment. This section describes the current best practices for thread synchronizing during DLL shutdown.

Thread Synchronization in DllMain during Process Exit

- By the time **DllMain** is called at process exit, all the process's threads have been forcibly cleaned up and there is a chance that the address space is inconsistent. Synchronization is not required in this case. In other words, the ideal DLL_PROCESS_DETACH handler is empty.
- Windows Vista ensures that core data structures (environment variables, current directory, process heap, and so on) are in a consistent state. However, other data structures can be corrupted, so cleaning memory is not safe.
- Persistent state that needs to be saved must be flushed to permanent storage.

Thread Synchronization in DllMain for DLL_THREAD_DETACH during DLL Unload

- When the DLL is unloaded, the address space is not thrown away. Therefore, the DLL is expected to perform a clean shutdown. This includes thread synchronization, open handles, persistent state, and allocated resources.
- Thread synchronization is tricky because waiting on threads to exit in **DllMain** can cause a deadlock. For example, DLL A holds the loader lock. It signals thread T to exit and waits for the thread to exit. Thread T exits and the loader tries to acquire the loader lock to call into DLL A's **DllMain** with DLL_THREAD_DETACH. This causes a deadlock. To minimize the risk of a deadlock:
  - DLL A gets a DLL_THREAD_DETACH message in its **DllMain** and sets an event for thread T, signaling it to exit.
  - Thread T finishes its current task, brings itself to a consistent state, signals DLL A, and waits infinitely. Note that the consistency-checking routines should follow the same restrictions as **DllMain** to avoid deadlocking.
  - DLL A terminates T, knowing that it is in a consistent state.

If a DLL is unloaded after all its threads have been created, but before they begin executing, the threads may crash. If the DLL created threads in its **DllMain** as part of its initialization, some threads may not have finished initialization and their DLL_THREAD_ATTACH message is still waiting to be delivered to the DLL. In this situation, if the DLL is unloaded, it will begin terminating threads. However, some threads may be blocked behind the loader lock. Their DLL_THREAD_ATTACH messages are processed after the DLL has been unmapped, causing the process to crash.

© 2006 Microsoft Corporation. All rights reserved.

Recommendations

The following are recommended guidelines:

- Use Application Verifier to catch the most common errors in **DllMain**.
- If using a private lock inside **DllMain**, define a locking hierarchy and use it consistently. The loader lock must be at the bottom of this hierarchy.
- Verify that no calls depend on another DLL that may not have been fully loaded yet.
- Perform simple initializations statically at compile time, rather than in **DllMain**.
- Defer any calls in **DllMain** that can wait until later.
- Defer initialization tasks that can wait until later. Certain error conditions must be detected early so that the application can handle errors gracefully. However, there are tradeoffs between this early detection and the loss of robustness that can result from it. Deferring initialization is often best.

References

Blogs for Developers

The Old New Thing
    http://blogs.msdn.com/oldnewthing/

    Some reasons not to do anything scary in your DllMain
        http://blogs.msdn.com/oldnewthing/archive/2004/01/27/63401.aspx

    Another reason not to do anything scary in your DllMain: Inadvertent Deadlock
        http://blogs.msdn.com/oldnewthing/archive/2004/01/28/63880.aspx

MGrier’s WebLog
    http://blogs.msdn.com/mgrier/

Larry Osterman’s WebLog
    http://blogs.msdn.com/larryosterman/default.aspx

    Things you shouldn’t do, part 1 – DllMain is special

MSDN

DllMain
    http://msdn.microsoft.com/library/en-us/dllproc/base/dllmain.asp

Mixed DLL Loading Problem
    http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vcconMixedDLLLoadingProblem.asp

© 2006 Microsoft Corporation. All rights reserved.