Initialization of Mixed Assemblies

09/26/202013 minutes to read +2

In this article

Windows developers must always be wary of loader lock when running code during `DllMain`. However, there are some additional issues to consider when dealing with C++/CLI mixed-mode assemblies.

Code within DllMain must not access the .NET Common Language Runtime (CLR). That means that `DllMain` should make no calls to managed functions, directly or indirectly; no managed code should be declared or implemented in `DllMain`; and no garbage collection or automatic library loading should take place within `DllMain`.

Causes of Loader Lock

With the introduction of the .NET platform, there are two distinct mechanisms for loading an execution module (EXE or DLL): one for Windows, which is used for unmanaged modules, and one for the CLR, which loads .NET assemblies. The mixed DLL loading problem centers around the Microsoft Windows OS loader.

When an assembly containing only .NET constructs is loaded into a process, the CLR loader can do all of the necessary loading and initialization tasks itself. However, to load mixed assemblies that can contain native code and data, the Windows loader must be used as well.

The Windows loader guarantees that no code can access code or data in that DLL before it's been initialized. And it ensures that no code can redundantly load the DLL while it's partially initialized. To do it, the Windows loader uses a process-global critical section (often called the "loader lock") that prevents unsafe access during module initialization. As a result, the loading process is vulnerable to many classic deadlock scenarios. For mixed assemblies, the following two scenarios increase the risk of deadlock:

- First, if users attempt to execute functions compiled to Microsoft intermediate language (MSIL) when the loader lock is held (from `DllMain` or in static initializers, for example), it can cause deadlock. Consider the case in which the MSIL function references a type in an assembly that's not loaded yet. The CLR will attempt to automatically load that assembly, which may require the Windows loader to block on the loader lock. A deadlock occurs, since the loader lock is already held by code earlier in the call sequence. However, executing MSIL under loader lock doesn't guarantee that a deadlock will occur. That's what makes this scenario difficult to diagnose and fix. In some circumstances, such as when the DLL of the referenced type contains no native constructs and all of its dependencies contain no native constructs, the Windows loader isn't required to load the .NET assembly of the referenced type. Additionally, the required assembly or its mixed native/.NET dependencies may have already been loaded by other code. Consequently, the deadlocking can be difficult to predict, and can vary depending on the configuration of the target machine.

- Second, when loading DLLs in versions 1.0 and 1.1 of the .NET Framework, the CLR assumed that the loader lock wasn't held, and took several actions that are invalid under loader lock. Assuming that the loader lock isn't held is a valid assumption for purely .NET DLLs. But because mixed DLLs execute native initialization routines, they require the native Windows loader, and consequently the loader lock. So, even if the developer wasn't attempting to execute any MSIL functions during DLL initialization, there was still a small possibility of nondeterministic deadlock in .NET Framework versions 1.0 and 1.1.

All non-determinism has been removed from the mixed DLL loading process. It was accomplished with these changes:

- The CLR no longer makes false assumptions when loading mixed DLLs.
- Unmanaged and managed initialization is done in two separate and distinct stages. Unmanaged initialization takes place first (via `DllMain`), and managed initialization takes place afterwards, through a .NET-supported `.cctor` construct. The latter is completely transparent to the user unless **`/Zl`** or **`/NODEFAULTLIB`** are used. For more information, see/NODEFAULTLIB (Ignore Libraries) and /Zl (Omit Default Library Name).

Loader lock can still occur, but now it occurs reproducibly, and is detected. If `DllMain`contains MSIL instructions, the compiler generates warning Compiler Warning (level 1) C4747. Furthermore, either the CRT or the CLR will try to detect and report attempts to execute MSIL under loader lock. CRT detection results in runtime diagnostic C Run-Time Error R6033.

The rest of this article describes the remaining scenarios for which MSIL can execute under the loader lock. It shows how to resolve the issue under each of those scenarios, and debugging techniques.

Scenarios and Workarounds

There are several different situations under which user code can execute MSIL under loader lock. The developer must ensure that the user code implementation doesn't attempt to execute MSIL instructions under each of these circumstances. The following subsections describe all possibilities with a discussion of how to resolve issues in the most common cases.

DllMain

The `DllMain` function is a user-defined entry point for a DLL. Unless the user specifies otherwise, `DllMain` is invoked every time a process or thread attaches to or detaches from the containing DLL. Since this invocation can occur while the loader lock is held, no user-supplied `DllMain` function should be compiled to MSIL. Furthermore, no function in the call tree rooted at `DllMain` can be compiled to MSIL. To resolve issues here, the code block that defines `DllMain` should be modified with `#pragma unmanaged`. The same should be done for every function that `DllMain` calls.

In cases where these functions must call a function that requires an MSIL implementation for other calling contexts, you can use a duplication strategy where both a .NET and a native version of the same function are created.

As an alternative, if `DllMain` isn't required or if it doesn't need to be executed under loader lock, you can remove the user-provided `DllMain` implementation, which eliminates the problem.

If `DllMain` attempts to execute MSIL directly, Compiler Warning (level 1) C4747 will result. However, the compiler can't detect cases where `DllMain` calls a function in another module that in turn attempts to execute MSIL.

For more information on this scenario, see Impediments to Diagnosis.

Initializing Static Objects

Initializing static objects can result in deadlock if a dynamic initializer is required. Simple cases (such as when you assign a value known at compile time to a static variable) don't require dynamic initialization, so there's no risk of deadlock. However, some static variables get initialized by function calls, constructor invocations, or expressions that can't be evaluated at compile time. These variables all require code to execute during module initialization.

The code below shows examples of static initializers that require dynamic initialization: a function call, object construction, and a pointer initialization. (These examples aren't static, but are assumed to have definitions in the global scope, which has the same effect.)

C++ Copy

```cpp
// dynamic initializer function generated
int a = init();
CObject o(arg1, arg2);
CObject* op = new CObject(arg1, arg2);
```

This risk of deadlock depends on whether the containing module is compiled with **`/clr`** and whether MSIL will be executed. Specifically, if the static variable is compiled without **`/clr`** (or is in a `#pragma unmanaged` block), and the dynamic initializer required to initialize it results in the execution of MSIL instructions, deadlock may occur. It's because, for modules compiled without **`/clr`**, the initialization of static variables is performed by DllMain. In contrast, static variables compiled with **`/clr`** are initialized by the `.cctor`, after the unmanaged initialization stage has completed and the loader lock has been released.

There are a number of solutions to deadlock caused by the dynamic initialization of static variables. They're arranged here roughly in order of time required to fix the problem:

- The source file containing the static variable can be compiled with **`/clr`**.
- All functions called by the static variable can be compiled to native code using the `#pragma unmanaged` directive.
- Manually clone the code that the static variable depends upon, providing both a .NET and a native version with different names. Developers can then call the native version from native

static initializers and call the .NET version elsewhere.

User-Supplied Functions Affecting Startup

There are several user-supplied functions on which libraries depend for initialization during startup. For example, when globally overloading operators in C++ such as the **`new`** and **`delete`** operators, the user-provided versions are used everywhere, including in C++ Standard Library initialization and destruction. As a result, C++ Standard Library and user-provided static initializers will invoke any user-provided versions of these operators.

If the user-provided versions are compiled to MSIL, then these initializers will be attempting to execute MSIL instructions while the loader lock is held. A user-supplied `malloc` has the same consequences. To resolve this problem, any of these overloads or user-supplied definitions must be implemented as native code using the `#pragma unmanaged` directive.

For more information on this scenario, see Impediments to Diagnosis.

Custom Locales

If the user provides a custom global locale, this locale gets used to initialize all future I/O streams, including streams that are statically initialized. If this global locale object is compiled to MSIL, then locale-object member functions compiled to MSIL may be invoked while the loader lock is held.

There are three options for solving this problem:

The source files containing all global I/O stream definitions can be compiled using the **`/clr`** option. It prevents their static initializers from being executed under loader lock.

The custom locale function definitions can be compiled to native code by using the `#pragma unmanaged` directive.

Refrain from setting the custom locale as the global locale until after the loader lock is released. Then explicitly configure I/O streams created during initialization with the custom locale.

Impediments to Diagnosis

In some cases, it's difficult to detect the source of deadlocks. The following subsections discuss these scenarios and ways to work around these issues.

Implementation in Headers

In select cases, function implementations inside header files can complicate diagnosis. Inline functions and template code both require that functions be specified in a header file. The C++ language specifies the One Definition Rule, which forces all implementations of functions with the same name to be semantically equivalent. Consequently, the C++ linker need not make any special considerations when merging object files that have duplicate implementations of a given function.

In Visual Studio versions before Visual Studio 2005, the linker simply chooses the largest of these semantically equivalent definitions. It's done to accommodate forward declarations, and scenarios when different optimization options are used for different source files. It creates a problem for mixed native and .NET DLLs.

Because the same header may be included both by C++ files with **`/clr`** enabled and disabled, or a #include can be wrapped inside a `#pragma unmanaged` block, it's possible to have both MSIL and native versions of functions that provide implementations in headers. MSIL and native implementations have different semantics for initialization under the loader lock, which effectively violates the one definition rule. Consequently, when the linker chooses the largest implementation, it may choose the MSIL version of a function, even if it was explicitly compiled to native code elsewhere using the `#pragma unmanaged` directive. To ensure that an MSIL version of a template or inline function is never called under loader lock, every definition of every such function called under loader lock must be modified with the `#pragma unmanaged` directive. If the header file is from a third party, the easiest way to make this change is to push and pop the `#pragma unmanaged` directive around the #include directive for the offending header file. (See managed, unmanaged for an example.) However, this strategy doesn't work for headers that contain other code that must directly call .NET APIs.

As a convenience for users dealing with loader lock, the linker will choose the native implementation over the managed when presented with both. This default avoids the above issues. However, there are two exceptions to this rule in this release because of two unresolved issues with the compiler:

- The call to an inline function is through a global static function pointer. This scenario isn'table because virtual functions are called through global function pointers. For example,

C++ Copy

```
#include "definesmyObject.h"
#include "definesclassC.h"

typedef void (*function_pointer_t)();

function_pointer_t myObject_p = &myObject;

#pragma unmanaged
void DuringLoaderlock(C & c)
{
    // Either of these calls could resolve to a managed implementation,
    // at link-time, even if a native implementation also exists.
    c.VirtualMember();
    myObject_p();
}
```

Diagnosing in Debug Mode

All diagnoses of loader lock problems should be done with Debug builds. Release builds may not produce diagnostics. And, the optimizations made in Release mode may mask some of the MSIL under loader lock scenarios.

How to debug loader lock issues

The diagnostic that the CLR generates when an MSIL function is invoked causes the CLR to suspend execution. That in turn causes the Visual C++ mixed-mode debugger to be suspended as well when running the debuggee in-process. However, when attaching to the process, it'sn't possible to obtain a managed callstack for the debuggee using the mixed debugger.

To identify the specific MSIL function that was called under loader lock, developers should complete the following steps:

1. Ensure that symbols for mscoree.dll and mscorwks.dll are available.

   You can make the symbols available in two ways. First, the PDBs for mscoree.dll and mscorwks.dll can be added to the symbol search path. To add them, open the symbol search path options dialog. (From the **Tools** menu, choose **Options**. In the left pane of the **Options** dialog box, open the **Debugging** node and choose **Symbols**.) Add the path to the mscoree.dll and mscorwks.dll PDB files to the search list. These PDBs are installed to the %VSINSTALLDIR%\SDK\v2.0\symbols. Choose **OK**.

   Second, the PDBs for mscoree.dll and mscorwks.dll can be downloaded from the Microsoft Symbol Server. To configure Symbol Server, open the symbol search path options dialog. (From the **Tools** menu, choose **Options**. In the left pane of the **Options** dialog box, open the **Debugging** node and choose **Symbols**.) Add this search path to the search list: `https://msdl.microsoft.com/download/symbols`. Add a symbol cache directory to the symbol server cache text box. Choose **OK**.

2. Set debugger mode to native-only mode.

   Open the **Properties** grid for the startup project in the solution. Select **Configuration Properties** > **Debugging**. Set the **Debugger Type** property to **Native-Only**.

3. Start the debugger (F5).

4. When the **`/clr`** diagnostic is generated, choose **Retry** and then choose **Break**.

5. Open the call stack window. (On the menu bar, choose **Debug** > **Windows** > **Call Stack**.) The offending `DllMain` or static initializer is identified with a green arrow. If the offending function isn't identified, the following steps must be taken to find it.

6. Open the **Immediate** window (On the menu bar, choose **Debug** > **Windows** > **Immediate**.)

7. Enter `.load sos.dll` into the **Immediate** window to load the SOS debugging service.

8. Enter `!dumpstack` into the **Immediate** window to obtain a complete listing of the internal **`/clr`** stack.

9. Look for the first instance (closest to the bottom of the stack) of either _CorDllMain (if `DllMain` causes the issue) or _VTableBootstrapThunkInitHelperStub or GetTargetForVTableEntry (if a static initializer causes the issue). The stack entry just below this call is the invocation of the MSIL implemented function that attempted to execute under loader lock.

10. Go to the source file and line number identified in the previous step and correct the problem using the scenarios and solutions described in the Scenarios section.

Example

Description

The following sample shows how to avoid loader lock by moving code from `DllMain` into the constructor of a global object.

In this sample, there's a global managed object whose constructor contains the managed object that was originally in `DllMain`. The second part of this sample references the assembly, creating an instance of the managed object to invoke the module constructor that does the initialization.

Code

C++ Copy

```
// initializing_mixed_assemblies.cpp
// compile with: /clr /LD
#pragma once
#include <stdio.h>
#include <windows.h>
struct __declspec(dllexport) A {
   A() {
      System::Console::WriteLine("Module ctor initializing based on global instance of
class.\n");
   }

   void Test() {
      printf_s("Test called so linker doesn't throw away unused object.\n");
   }
};

#pragma unmanaged
// Global instance of object
A obj;

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved) {
   // Remove all managed code from here and put it in constructor of A.
   return true;
}
```

This example demonstrates issues in initialization of mixed assemblies:

C++ Copy

```
// initializing_mixed_assemblies_2.cpp
// compile with: /clr initializing_mixed_assemblies.lib
#include <windows.h>
using namespace System;
#include <stdio.h>
#using "initializing_mixed_assemblies.dll"
struct __declspec(dllimport) A {
   void Test();
};

int main() {
   A obj;
   obj.Test();
}
```

This code produces the following output:

Output

Copy

```
Module ctor initializing based on global instance of class.

Test called so linker doesn't throw away unused object.
```

See also

Mixed (Native and Managed) Assemblies