When you transfer control across stack frames, all the frames in between need to be in on the joke

devblogs.microsoft.com/oldnewthing/20120910-00

September 10, 2012

Raymond Chen

Chris Hill suggests discussing the use of structured exception handling as it relates to the window manager, and specifically the implications for applications which raise exceptions from a callback.

If you plan on raising an exception and handling it in a function higher up the stack, all the stack frames in between need to be be in on your little scheme, because they need to be able to unwind. (And I don't mean "unwind" in the "have a beer and watch some football" sense of "unwind".)

If you wrote all the code in between the point the exception is raised and the point it is handled, then you're in good shape, because at least then you have a chance of making sure they all unwind properly. This means either using RAII techniques (and possibly compiling with the `/EHa` flag to convert asynchronous exceptions to synchronous ones, so that Win32 exceptions will also trigger unwind; although that has its own problems since the C++ exception model is synchronous, not asynchronous) or judiciously using `try` / `finally` (or whatever equivalent exists in your programming language of choice) to clean up resources in the event of an unwind.

But if you don't control all the frames in between, then you can't really guarantee that they were written in the style you want.

In Win32, exceptions are considered to be horrific situations that usually indicate some sort of fatal error. There may be some select cases where exceptions can be handled, but those are more the unusual cases than the rule. Most of the time, an exception means that something terrible has happened and you're out of luck. The best you can hope for at this point is a controlled crash landing.

As a result of this overall mindset, Win32 code doesn't worry too much about recovering from exceptions. If an exception happens, then it means your process is already toast and there's no point trying to fix it, because that would be trying to reason about a total

breakdown of normal functioning. As a general rule generic Win32 code is not exception-safe.

Consider a function like this:

```
struct BLORP
{
    int Type;
    int Count;
    int Data;
};
CRITICAL_SECTION g_csGlobal; // assume somebody initialized this
BLORP g_Blorp; // protected by g_csGlobal
void SetCurrentBlorp(const BLORP *pBlorp)
{
    EnterCriticalSection(&g_csGlobal);
    g_Blorp = *pBlorp;
    LeaveCriticalSection(&g_csGlobal);
}
void GetCurrentBlorp(BLORP *pBlorp)
{
    EnterCriticalSection(&g_csGlobal);
    *pBlorp = g_Blorp;
    LeaveCriticalSection(&g_csGlobal);
}
```

These are perfectly fine-looking functions from a traditional Win32 standpoint. They take a critical section, copy some data, and leave the critical section. The only thing[1] that could go wrong is that the caller passed a bad pointer. In the case of `TerminateThread`, we're already in the world of "don't do that" If that happens, a `STATUS_ACCESS_VIOLATION` exception is raised, and the application dies.

But what if your program decides to handle the access violation? Maybe `pBlorp` points into a memory-mapped file, and there is an I/O error paging the memory in, say because it's a file on the network and there was a network hiccup. Now you have two problems: The critical section is orphaned, and the data is only partially copied. (The partial-copy case happens if the `pBlorp` points to a `BLORP` that straddles a page boundary, where the first page is valid but the second page isn't.) Just converting this code to RAII solves the first problem, but it doesn't solve the second, which is kind of bad because the second problem is what the critical section was trying to prevent from happening in the first place!

```
void SetCurrentBlorp(const BLORP *pBlorp)
{
    CriticalSectionLock lock(&g_csGlobal);
    g_Blorp = *pBlorp;
}
void GetCurrentBlorp(BLORP *pBlorp)
{
    CriticalSectionLock lock(&g_csGlobal);
    *pBlorp = g_Blorp;
}
```

Suppose somebody calls `SetCurrentBlorp` with a `BLORP` whose `Type` and `Count` are in readable memory, but whose `Data` is not. The code enters the critical section, copies the `Type` and `Count`, but crashes when it tries to copy the `Data`, resulting in a `STATUS_ACCESS_VIOLATION` exception. Now suppose that somebody unwisely decides to handle this exception. The RAII code releases the critical section (assuming that you compiled with `/EHa`), but there's no code to try to patch up the now-corrupted `g_Blorp`. Since the critical section was probably added to prevent `g_Blorp` from getting corrupted, the result is that the thing you tried to protect against ended up happening anyway.

Okay, that was a bit of a digression. The point is that unless everybody between the point the exception is raised and the point the exception is handled is in on the joke, you are unlikely to escape fully unscathed. This is particular true in the generalized Win32 case, since it is perfectly legal to write Win32 code in languages other than C++, as long as you adhere to the Win32 ABI. (I'm led to believe that Visual Basic is still a popular language.)

There are a lot of ways of getting stack frames beyond your control between the point the exception is raised and the point it is handled. For example, you might call `EnumWindows` and raise an exception in the callback function and try to catch it in the caller. Or you might raise an exception in a window procedure and try to catch it in your message loop. Or you might try to `longjmp` out of a window procedure. All of these end up raising an exception and catching it in another frame. And since you don't control all the frames in between, you can't guarantee that they are all prepared to resume execution in the face of an exception.

Bonus reading: My colleague Paul Betts has written up a rather detailed study of one particular instance of this phenomenon.

[1]Okay, another thing that could go wrong is that somebody calls `TerminateThread` on the thread, but whoever did that knew they were corrupting the process.

Raymond Chen

Follow