

Re: Terminating program in ProcessWndProcException

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2004-08/2116.html>

From: Joseph M. Newcomer (*newcomer_at_flounder.com*)

Date: 08/30/04

Date: Mon, 30 Aug 2004 15:33:12 -0400

New/malloc failures are actually pretty easy to handle, and normally would not involve exiting the program. Heap corruption has deep implications; what is really going to happen if the process exits (e.g., how do you handle aborts of file modifications?)

Note that calling `exit()` invokes the `onexit` handler, which will do things like call destructors, which can crash if there is enough memory corruption (a little single-stepping on exit, or doing a backtrace after a crash on exit, reveals this whole mechanism). This is going to try to do all the standard cleanup, which involves touching the heap. Except that now there is no longer an exception handler present, because you are in the last possible exception handler. Any kind of exception-throwing error at this point is going to be very unpleasant.

You're already in the context of an exception handler. The world is in an unstable state. It is not even clear that it is possible to set another exception frame at this point. This might help, if it works at all. But if you call `exit()`, you invoke the whole static-variable-destructor mechanism, which is going to end up touching the heap anyway. The API call `::ExitProcess` would be safest, but it won't terminate any threads that are blocked in the kernel, which still poses problems. `TerminateProcess` is instantaneous, but means that DLLs are not cleanly terminated, which means that if they have any shared state (e.g., decrementing reference counts of COM objects) such shared state is not guaranteed to be cleaned up properly.

DLL cleanup could touch the heap. Note that `CloseHandle` can throw an exception (exception code 8, if I recall correctly). Any of the cleanups could touch the heap, in a way that trips over the same corruption. Now you find yourself with the potential of recursively entering the exception handler if another exception is taken as a side effect of the "clean" exit, and it probably isn't set up to handle this recursion. Handling program termination in the presence of serious data structure damage is a serious problem. In my case, I actually had control of the heap allocator, and modified it so it had a "reset heap" entry point. The entire heap was erased and reinitialized, and then I reconstructed, by polling the operating system, all the state I needed to return to a somewhat sane state. I never did find the bug that caused the heap damage, but I had the ability to do things like enumerate all open handles, which is all that made it possible. If the handle had a pending I/O request coming into my component, I completed it with an error status. This surprised a lot of application programmers who thought that it was impossible to get an I/O error, and so never bothered to check for them. We do not have this luxury in

Windows; there are no means of reinitializing the process heap, enumerating handles, or anything else. Bottom line: it is very, very, very hard, if not impossible, to do a clean recovery from errors like heap damage. Frankly, as profoundly ugly as ::TerminateProcess is, it is the only thing I can think of that has any hope of actually terminating the process without the potential for recursive error. And This Is Not A Pretty Sight, because of all the other problems it can cause. But given at that point your executable image is screwed up beyond recovery, and if it had simply crashed without the handler that's what would have happened, it may be your only alternative. Perhaps calling exit() will work, as long as you don't get nailed by other heap damage reports, but at this point you are already in serious trouble. You can't even rely on the heap for your "clean" notification to the user, or anything else. The world is well and truly messed up beyond any sane recovery.

joe

On Mon, 30 Aug 2004 15:29:34 GMT, XXXMartin.Aupperle@PrimaProgrammXXX.de (Martin Aupperle) wrote:

>On Sat, 28 Aug 2004 12:46:05 -0400, Joseph M. Newcomer

><newcomer@flounder.com> wrote:

>

>>Philosophically, NO program should terminate except by user directive. When I get problems
>>like this, I simply enter a mode in which most of the menu items are disabled. One of the
>>few that is still active is Program>Exit, and I usually have a few diagnostic menu items
>>that either become enabled or appear magically.

>

>If this works for all of your exceptioanl situations, you then you are
>lucky.

>

>For all of our exceptional situations, we simply bring up a dialog
>that has lots of information about the situation, the context, etc.
>The user can dump that to file, send it via email, or print ist. This
>is clear and undisputed.

>

>The interesting thing is what happens after the user closed the
>dialog. Most of the time, the program can continue, especially when
>the problem arose in a command handler (user pressed a button, user
>selected a menu item etc.). Our command handlers are "exception
>safe", i.e. the program is in a defined state even if the command
>handler exits through an exception. That was the easy case – more than
>90% of our exceptional situations fall into this category.

>

>The difficult ones are things like "the creation of necessary central
>data structures failed" "memory shortage (i.e. new/malloc failed)",
>"heap corruption (i.e. AfxIsValidAddress failed)" and some others. We
>consider these severe enough to abort the program immediately after
>the user quits the info-dialogbox.

>

>

>>

>>Do not call _exit. Do not call _abort. The problem with all such "recovery" mechanisms is
>>that besides doing more harm than good, they aren't defined for Windows apps.

>
>Well, depends. They work well in other situations. E.G. you can call
>them from command handlers (you should not, but you can). They simply
>do not work in ProcessWndProcException.
>
>>Posting WM_QUIT won't help, because ;you probably don't have a message loop running at
>>that point.
>
>Why not? I thought I always have a message loop running – at least
>until a WM_QUIT gets processed. ProcessWndProcException gets called
>when an exception of type CException* gets thrown during message
>processing and not caught earlier.
>
>>
>>A more serious question is where are these exceptions coming from? If they are exceptions
>>you are throwing, that's fine. But if they are exceptions being thrown by the runtime in
>>some way, they should be fixed.
>
>We distinguish between "logic errors" and "runtime errors".
>
>Logic errors are failed preconditions, ASSERTs and that kind of stuff.
>If such an exception gets thrown, we know that we have made a
>programming error which must be fixed, and a new software version
>released. Nevertheless – no software of the size of ours is error
>free, so we
>1. must present the user with some information about an "internal
>error", as we like to call it.
>2. allow the user to send us information about that, so that we can
>fix it.
>
>Please note that this is more than most programs do when a programming
>error hits. They simply crash.
>
>Runtime Errors are situations of mostly transient nature that go away
>with time, or need some administrator intervention, rebooting or
>reinstallation etc. E.G when the network chokes, or the database
>becomes unreachable in the middle of a transaction. Memory exhaustion
>or out of Windows resources also belongs to that category.
>
>So, if "the runtime" throws an exception, it really depends.
>
>
>
>>
>>My exception loop was something like
>>
>>int MYApp::Run()
>> {
>> while(true)
>> { /* main loop */
>> try

```
>> {
>> return CWinApp::Run();
>> }
>> catch(...specific exception...)
>> {
>> }
>> catch(...other exception....)
>> {
>> }
>> catch(...)
>> {
>> }
>> } /* main loop */
>>}
>
>I did this once too, but a little different. I completely overloaded
>Run and had my own message loop. I wanted to have all exceptions
>handled WITHIN the loop.
>
>The advantage compared to overloading ProcessWndProcException is that
>we can catch exceptions that are not derived from CException. When
>using external libraries that throw this definitely in an advantage.
>
>But the question remains: how can I terminate the program from within
>ProcessWndProcException?
>
>
>[snip]
>
>>
>>There is nothing more confusing to a user than having a program spontaneously exit. There
>>is nothing harder for tech support to deal with than a program that merely exits.
>>
>>By continuing to run, I was able to add diagnostics and reporting mechanisms that would
>>allow tech support to diagnose the problem. This was important, because the libraries we
>>used would throw random exceptions. Putting an exception frame at every call was too hard
>>(hundreds of calls). Once I started catching exceptions like this, we could target our
>>handlers to do better recovery in key places.
>
>We are not so far apart from each other. I do this "diagnostics and
>reporting" too, but I do the diagnosing at the throw location and the
>reporting at the catch location (in an appropriate dialog).
>
>Martin
>
>-----
>Martin Aupperle
>-----
```

microsoft.public.vc.mfc: Re: Terminating program in ProcessWndProcException

Joseph M. Newcomer [MVP]

email: newcomer@flounder.com

Web: <http://www.flounder.com>

MVP Tips: http://www.flounder.com/mvp_tips.htm