Heap corruption under Win32; how to locate?

Asked 16 years, 4 months ago Modified 1 year, 6 months ago

Viewed 40k times

65

I'm working on a **multithreaded** C++ application that is corrupting the heap. The usual tools to locate this corruption seem to be inapplicable. Old builds (18 months old) of the source code exhibit the same behavior as the most recent release, so this has been around for a long time and just wasn't noticed; on the downside, source deltas can't be used to identify when the bug was introduced - there are *a lot* of code changes in the repository.

The prompt for crashing behavior is to generate throughput in this system - socket transfer of data which is munged into an internal representation. I have a set of test data that will periodically cause the app to exception (various places, various causes - including heap alloc failing, thus: heap corruption).

The behavior seems related to CPU power or memory bandwidth; the more of each the machine has, the easier it is to crash. Disabling a hyper-threading core or a dual-core core reduces the rate of (but does not eliminate) corruption. This suggests a timing-related issue.

Now here's the rub:
When it's run under a lightweight debug environment (say `Visual Studio 98 / AKA MSVC6`) the heap corruption is reasonably easy to reproduce - ten or fifteen minutes pass before something fails horrendously and exceptions, like an `alloc;` when running under a sophisticated debug environment (Rational Purify, `VS2008/MSVC9` or even Microsoft Application Verifier) the system becomes memory-speed bound and doesn't crash (Memory-bound: CPU is not getting above `50%`, disk light is not on, the program's going as fast it can, box consuming `1.3G` of 2G of RAM). So, **I've got a choice between being able to reproduce the problem (but not identifying the cause) or being able to identify the cause of a problem I can't reproduce.**

My current best guesses as to where to next is:

1. Get an insanely grunty box (to replace the current dev box: 2Gb RAM in an `E6550 Core2 Duo`); this will make it possible to repro the crash causing misbehavior when running under a powerful debug environment; or
2. Rewrite operators `new` and `delete` to use `VirtualAlloc` and `VirtualProtect` to mark memory as read-only as soon as it's done with. Run under `MSVC6` and have the OS catch the bad guy who's writing to freed memory. Yes, this is a sign of desperation: who the hell rewrites `new` and `delete`?! I wonder if this is going to make it as slow as under Purify et al.

And, no: Shipping with Purify instrumentation built in is not an option.

A colleague just walked past and asked "Stack Overflow? Are we getting stack overflows now?!?"

And now, the question: **How do I locate the heap corruptor?**

Update: balancing `new[]` and `delete[]` seems to have gotten a long way toward solving the problem. Instead of 15mins, the app now goes about two hours before crashing. Not there yet. Any further suggestions? The heap corruption persists.

Update: a release build under Visual Studio 2008 seems dramatically better; current suspicion rests on the `STL` implementation that ships with `VS98`.

> 3. Reproduce the problem. `Dr Watson` will produce a dump that might be helpful in further analysis.

I'll take note of that, but I'm concerned that Dr. Watson will only be tripped up after the fact, not when the heap is getting stomped on.

> Another try might be using `WinDebug` as a debugging tool which is a quite powerful being at

> the same time also lightweight.

Got that going at the moment, again: not much help until something goes wrong. I want to catch the vandal in the act.

> Maybe these tools will allow you at least to narrow the problem to a certain component.

I don't hold much hope, but desperate times call for...

> And are you sure that all the components of the project have correct runtime library settings (`C/C++ tab`, Code Generation category in VS 6.0 project settings)?

No, I'm not, and I'll spend a couple of hours tomorrow going through the workspace (58 projects in it) and checking they're all compiling and linking with the appropriate flags.

Update: This took 30 seconds. Select all projects in the `Settings` dialog, and unselect until you find the project(s) that don't have the right settings (they all had the right settings).

c++ **windows** **multithreading** **debugging** **memory**

Share

Improve this question

Follow

edited Jun 1, 2023 at 11:53

community wiki
12 revs, 5 users 50%
Josh

What does the failure look like exactly? You say "including heap alloc failing" - could that mean you're simply running out of memory? (I'm not up on Windows programming, but that could be a cause in the Linux world.) – svec Aug 5, 2008 at 21:03

@svec C++ says that out of memory results in std::bad_alloc being thrown. What I'm seeing is Memory exceptions ("hey, you can't read (or maybe write) there!") – Josh Aug 6, 2008 at 12:36

> Building with 2008 would have caught crazy crap like that... maybe even MSVC6, but I'm not sure. MSVC6 won't catch that but Lint would. De-linting your code might be a good place to start. It's only $250 (nought compared to the amount of time saved in debugging). Tip for first-time lint users: Turn off everything and slowly turn stuff on. I started with non-needed headers and worked my way up to about 20 items so far. When I ran it first time overnight on our product it had more errors than lines of code!! – graham.reeds Aug 6, 2008 at 12:49

2 It'd be interesting to know if you got a real solution here... – Roddy Mar 15, 2010 at 15:49

I don't think you get std::bad_alloc in VC6, I think it returns null? – paulm Jan 2, 2013 at 11:01

15 Answers Sorted by: Highest score (default)

29

My first choice would be a dedicated heap tool such as pageheap.exe.

Rewriting new and delete might be useful, but that doesn't catch the allocs committed by lower-level code. If this is what you want, better to Detour the `low-level alloc API`s using Microsoft Detours.

Also sanity checks such as: verify your run-time libraries match (release vs. debug, multi-threaded vs. single-threaded, dll vs. static lib), look for bad deletes (eg, delete where delete [] should have been used), make sure you're not mixing and matching your allocs.

Also try selectively turning off threads and see when/if the problem goes away.

What does the call stack etc look like at the time of the first exception?

Share Improve this answer

Follow

edited Sep 1, 2015 at 17:32

Michael Kelley

3,689 ● 5 ● 40 ● 41

answered Aug 4, 2008 at 7:51

user2189331

10

I have same problems in my work (we also use `VC6` sometimes). And there is no easy solution for it. I have only some hints:

- Try with automatic crash dumps on production machine (see Process Dumper). My experience says Dr. Watson is **not perfect** for dumping.
- Remove all **catch(...)** from your code. They often hide serious memory exceptions.
- Check Advanced Windows Debugging - there are lots of great tips for problems like yours. I recomend this with all my heart.
- If you use `STL` try `STLPort` and checked builds. Invalid iterator are hell.

Good luck. Problems like yours take us months to solve. Be ready for this...

Share Improve this answer

Follow

edited Feb 7, 2013 at 12:30

CloudyMarble

37.6k ● 70 ● 100 ● 132

answered Aug 6, 2008 at 12:41

Michal Sznajder

9,406 ● 4 ● 46 ● 62

8

We've had pretty good luck by writing our own malloc and free functions. In production, they just call the standard malloc and free, but in debug, they can do whatever you



want. We also have a simple base class that does nothing but override the new and delete operators to use these functions, then any class you write can simply inherit from that class. If you have a ton of code, it may be a big job to replace calls to malloc and free to the new malloc and free (don't forget realloc!), but in the long run it's very helpful.

In Steve Maguire's book Writing Solid Code (highly recommended), there are examples of debug stuff that you can do in these routines, like:

- Keep track of allocations to find leaks
- Allocate more memory than necessary and put markers at the beginning and end of memory -- during the free routine, you can ensure these markers are still there
- memset the memory with a marker on allocation (to find usage of uninitialized memory) and on free (to find usage of free'd memory)

Another good idea is to *never* use things like `strcpy`, `strcat`, or `sprintf` -- always use `strncpy`, `strncat`, and `snprintf`. We've written our own versions of these as well, to make sure we don't write off the end of a buffer, and these have caught lots of problems too.

Share Improve this answer

Follow

edited Feb 7, 2013 at 12:32

CloudyMarble

37.6k ● 70 ● 100 ● 132

answered Aug 22, 2008 at 17:11

Graeme Perrow

57.2k ● 24 ● 86 ● 125

1 "always use strncpy instead of strcpy" - in Microsoft CRT there is an even better alternative, strcpy_s. – Constantin Oct 12, 2008 at 20:37

remember to read the full msdn specifications with this kind of functions! strange things could happen if you don't read them fully! – alcor Apr 10, 2013 at 19:49

8

Run the original application with `ADplus -crash -pn appnename.exe` When the memory issue pops-up you will get a nice big dump.

You can analyze the dump to figure what memory location was corrupted. If you are lucky the overwrite memory is a unique string you can figure out where it came from. If you are not lucky, you will need to dig into `win32` heap and figure what was the orignal memory characteristics. (heap -x might help)

After you know what was messed-up, you can narrow appverifier usage with special heap settings. i.e. you can specify what `DLL` you monitor, or what allocation size to monitor.

Hopefully this will speedup the monitoring enough to catch the culprit.

In my experience, I never needed full heap verifier mode, but I spent a lot of time analyzing the crash dump(s) and browsing sources.

P.S: You can use DebugDiag to analyze the dumps. It can point out the `DLL` owning the corrupted heap, and give you other usefull details.

Share Improve this answer

Follow

edited Feb 7, 2013 at 12:33

CloudyMarble

37.6k ● 70 ● 100 ● 132

answered Sep 16, 2008 at 7:33

Tal

1,769 ● 2 ● 12 ● 22

4

You should attack this problem with both runtime and static analysis.

For static analysis consider compiling with PREfast (`cl.exe /analyze`). It detects mismatched `delete` and `delete[]`, buffer overruns and a host of other problems. Be prepared, though, to wade through many kilobytes of L6 warning, especially if your project still has `L4` not fixed.

PREfast is available with Visual Studio Team System and, apparently, as part of Windows SDK.

Share Improve this answer

Follow

edited Feb 7, 2013 at 12:32

CloudyMarble

37.6k ● 70 ● 100 ● 132

answered Oct 12, 2008 at 21:55

Constantin

28.1k ● 10 ● 63 ● 79

3

Is this in low memory conditions? If so it might be that new is returning `NULL` rather than throwing std::bad_alloc. Older `VC++` compilers didn't properly implement this. There is an article about Legacy memory allocation failures crashing `STL` apps built with `VC6`.

Share Improve this answer

Follow

edited Feb 7, 2013 at 12:34

CloudyMarble

37.6k ● 70 ● 100 ● 132

answered Sep 2, 2008 at 6:03

Steve Steiner

5,339 ● 4 ● 33 ● 45

This was very helpful! I didnt know why my new was suddenly returning NULL after switching Application Verifier on! – Vinz Jun 8, 2015 at 20:36

2

The apparent randomness of the memory corruption sounds very much like a thread synchronization issue - a bug is reproduced depending on machine speed. If objects (chuncks of memory) are shared among threads and synchronization (critical section, mutex, semaphore, other) primitives are not on per-class (per-object, per-class) basis, then it is possible to come to a situation

where class (chunk of memory) is deleted / freed while in use, or used after deleted / freed.

As a test for that, you could add synchronization primitives to each class and method. This will make your code slower because many objects will have to wait for each other, but if this eliminates the heap corruption, your heap-corruption problem will become a code optimization one.

Share Improve this answer

Follow

answered Aug 25, 2008 at 19:55

Ignas Limanauskas

2,486 ● 7 ● 30 ● 33

1

You tried old builds, but is there a reason you can't keep going further back in the repository history and seeing exactly when the bug was introduced?

Otherwise, I would suggest adding simple logging of some kind to help track down the problem, though I am at a loss of what specifically you might want to log.

If you can find out what exactly CAN cause this problem, via google and documentation of the exceptions you are getting, maybe that will give further insight on what to look for in the code.

Share Improve this answer

Follow

answered Aug 4, 2008 at 7:48

Mike Stone

44.6k ● 30 ● 114 ● 140

1

My first action would be as follows:

1. Build the binaries in "Release" version but creating debug info file (you will find this possibility in project settings).
2. Use Dr Watson as a defualt debugger (DrWtsn32 -I) on a machine on which you want to reproduce the problem.
3. Repdroduce the problem. Dr Watson will produce a dump that might be helpful in further analysis.

Another try might be using WinDebug as a debugging tool which is quite powerful being at the same time also lightweight.

Maybe these tools will allow you at least to narrow the problem to certain component.

And are you sure that all the components of the project have correct runtime library settings (C/C++ tab, Code Generation category in VS 6.0 project settings)?

Share Improve this answer Follow

answered Aug 4, 2008 at 8:26

Piotr Tyburski
651 ● 6 ● 11

1

So from the limited information you have, this can be a combination of one or more things:

- Bad heap usage, i.e., double frees, read after free, write after free, setting the HEAP_NO_SERIALIZE flag with allocs and frees from multiple threads on the same heap
- Out of memory
- Bad code (i.e., buffer overflows, buffer underflows, etc.)
- "Timing" issues

If it's at all the first two but not the last, you should have caught it by now with either pageheap.exe.

Which most likely means it is due to how the code is accessing shared memory. Unfortunately, tracking that down is going to be rather painful. Unsynchronized access to shared memory often manifests as weird "timing" issues. Things like not using acquire/release semantics for synchronizing access to shared memory with a flag, not using locks appropriately, etc.

At the very least, it would help to be able to track allocations somehow, as was suggested earlier. At least then you can view what actually happened up until the heap corruption and attempt to diagnose from that.

Also, if you can easily redirect allocations to multiple heaps, you might want to try that to see if that either fixes the problem or results in more reproduceable buggy behavior.

When you were testing with VS2008, did you run with HeapVerifier with Conserve Memory set to Yes? That might reduce the performance impact of the heap allocator. (Plus, you have to run with it Debug->Start with Application Verifier, but you may already know that.)

You can also try debugging with Windbg and various uses of the !heap command.

MSN

Share Improve this answer

Follow

answered Aug 22, 2008 at 16:51

Mat Noguchi

1,080 ● 6 ● 7

0

Graeme's suggestion of custom malloc/free is a good idea. See if you can characterize some pattern about the corruption to give you a handle to leverage.

For example, if it is always in a block of the same size (say 64 bytes) then change your malloc/free pair to always allocate 64 byte chunks in their own page. When you free a 64 byte chunk then set the memory protection bits on that page to prevent reads and wites (using VirtualQuery). Then anyone attempting to access this memory will generate an exception rather than corrupting the heap.

This does assume that the number of outstanding 64 byte chunks is only moderate or you have a lot of memory to burn in the box!

Share Improve this answer

Follow

answered Sep 2, 2008 at 4:23

Rob Walker

47.4k • 15 • 100 • 137

0

If you choose to rewrite new/delete, I have done this and have simple source code at:

http://gandolf.homelinux.org/~smhanov/blog/?id=10

This catches memory leaks and also inserts guard data before and after the memory block to capture heap corruption. You can just integrate with it by putting #include "debug.h" at the top of every CPP file, and defining DEBUG and DEBUG_MEM.

Share Improve this answer

Follow

answered Sep 17, 2008 at 13:40

Steve Hanov

11.5k • 16 • 66 • 72

0

The little time I had to solve a similar problem. If the problem still exists I suggest you do this : Monitor all calls to new/delete and malloc/calloc/realloc/free. I make single DLL exporting a function for register all calls. This function receive parameter for identifying your code source, pointer to allocated area and type of call saving this information in a table. All allocated/freed pair is eliminated. At the end or after you need you make a call to an other function for create report for left data. With this you can identify wrong calls (new/free or malloc/delete) or missing. If have any case of buffer

overwritten in your code the information saved can be wrong but each test may detect/discover/include a solution of failure identified. Many runs to help identify the errors. Good luck.

Share Improve this answer Follow

answered Dec 19, 2008 at 11:52

community wiki
lsalamon

0

Do you think this is a race condition? Are multiple threads sharing one heap? Can you give each thread a private heap with HeapCreate, then they can run fast with HEAP_NO_SERIALIZE. Otherwise, a heap should be thread safe, if you're using the multi-threaded version of the system libraries.

Share Improve this answer Follow

answered Jul 30, 2009 at 13:48

community wiki
Don

0

A couple of suggestions. You mention the copious warnings at W4 - I would suggest taking the time to fix your code to compile cleanly at warning level 4 - this will go a long way to preventing subtle hard to find bugs.

Second - for the /analyze switch - it does indeed generate copious warnings. To use this switch in my own project, what I did was to create a new header file that used #pragma warning to turn off all the additional warnings generated by /analyze. Then further down in the file, I turn on only those warnings I care about. Then use the /FI compiler switch to force this header file to be included first in all your compilation units. This should allow you to use the /analyze switch while controling the output

answered Oct 3, 2009 at 16:48

community wiki
Dan