

# Re: Memory Allocation in a Multi-Threaded Environment

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2008-06/msg00341.html>

---

- *From:* "Jeff" <someone@xxxxxxxxxxxxxx>
  - *Date:* Mon, 9 Jun 2008 16:19:09 +0100
- 

"Joseph M. Newcomer" <newcomer@xxxxxxxxxxxxxx> wrote in message <news:3fbo44pe1d25rvuotr1sj58ughh69uaqmv@xxxxxxxxxxxxxx>

See below...

On Sun, 8 Jun 2008 18:20:16 +0100, "Jeff" <someone@xxxxxxxxxxxxxx> wrote:

In general, I believe if you are sharing the same data between two threads, the design is already hopeless and needs to be redone. See, for example, my essay "the best synchronization is no synchronization", the point being that you should not create designs where synchronization is required for correct behavior.

It would not at all be surprising to see heap corruption errors if you are sharing a `std::vector` between two threads. It has nothing to do with "storage allocation" as such, but a LOT to do with the fact that none of the STL containers are "thread-safe" and therefore should not be used by one thread unless the entire container is locked against access by all other threads. This is usually a design error, because it can mean lengthy locks, and therefore is usually a Very Bad Idea.

For example, imagine you are iterating over a `std::vector` of `n` elements. While you are doing the iteration, and modifying the elements, some other thread does a `push_back`. Alas, the first thread, which cleverly has a *\*pointer\** to the array position, will find itself writing to unallocated memory, because the `push_back` could have freed the old vector

## Re: Memory Allocation in a Multi-Threaded Environment

contents after copying to the new vector.

Thanks for your response – it's really appreciated.

As I've already detailed in another response, I do now have a working system which appears to be both fast and stable. I am however very interested in your comments especially given your experience in this area – I have just about enough experience now to know that these problems can be hard.

In my specific case I want to have a streaming thread whose sole purpose is to download a file as quickly as possible – ie with as few interruptions as possible. I then have another thread which wants to process this data as soon as it is available – ie as it is being downloaded. In some cases (eg a local file system) the download will probably finish before the first bytes have been processed. In other cases (eg a slow remote file) the processor will spend a lot of time waiting for bytes to become available.

One thing that confuses me in your response is your comment that sharing the same data between threads is a bad idea. In the above scenario the two threads have to share the same data – that's the whole point. Could you describe a solution to this problem which would not involve the two threads accessing the same memory location at some point?

Since this specific problem must be encountered fairly often, I imagine there is already a well optimised stable solution. I'd be really grateful if someone could point me at it.

Thanks again.

Jeff

.