Doc No: SC22/WG21/N2917 = PL22.16/09-0107

Date: 2009-06-19

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@microsoft.com

N2880 Distilled, and a New Issue With Function Statics

This paper is an attempt to focus on two key fundamental difficulties that underlie many of the issues mentioned in N2880.

1. The trouble with detached threads and static destruction

Any thread that can continue running during static destruction is inherently nearly useless.

Such a thread cannot reliably use global or static objects;[1] therefore, it can't reliably call library functions, because library functions may use global or static objects; therefore, it is essentially useless. In the working paper, detached threads are already effectively required to call only code that is async-signal-safe, which is extremely restrictive and not the behavior generally expected by programmers who detach threads.

Therefore, detached threads are inherently nearly useless, *unless* they are guaranteed to end before the end of `main` before static destruction begins, by being joined manually or automatically or employing other synchronization to achieve the same end (see also N2802).

Possible resolutions include, in rough order from most to least desirable in my opinion:

- *1A: Require automatic joining.* Require that the end of `main` automatically join with all still-running non-`main` threads before static destruction begins.

  Only this option guarantees safety and elimination of undefined behavior if a thread is not joined with in time to avoid running during static destruction. It does not address non-`std::thread` threads, but those are beyond our domain.

[1] This applies only after the point in its lifetime when static destruction begins, but the thread generally cannot know when that is and so the restriction applies to the entire thread.

An argument against this option is that some implementers want to write std::thread as a zero-overhead wrapper around another C-based API, such as pthreads. C-based APIs support detaching much more readily because they are designed for a language where shutdown is almost trivial: run atexit handlers then flush and close streams/files. However, any objection to 1A based on overhead should quantify what the overhead actually is, and weigh it against the consideration that not joining will essentially result in undefined behavior.

- *1B: Disallow detaching, and say that failure to join before static destruction begins is undefined behavior.* Remove thread::detach from the standard, and require that a program must join with all non-main threads before the end of main; failure to do so is undefined behavior.

  This closes the most obvious way of accidentally creating a too-long-running thread that encounters undefined behavior. It leaves most of the burden on the programmer, who must track all threads and ensure they are joined with in time in order to avoid undefined behavior.

  An argument against this option is that Boost.Thread provides thread::detach. Nevertheless, the feature has problems. Note that Boost.Thread likewise provides a thread::~thread with detach semantics, which has also been recognized as flawed and already changed in C++0x for similar reasons (see N2802, whose Alternative 2 was adopted at the Summit meeting which changes the semantics from detach to terminate, though I personally believe the change should be to join; at any rate, we agree that detach is problematic).

- *1C: Just say that failure to join before static destruction begins is undefined behavior.* Require that a program must join with all non-main threads before the end of main; failure to do so is undefined behavior.

  This option has all the drawbacks of 1B, and differs only in that it doesn't provide the minimal guard rail of removing detach which officially becomes undefined behavior-bait.

- *1D (status quo): Just say that a program must not access a global after it has been destroyed.*

  That is slightly weaker than 1C in practice because it permits thread pools in global variables, which is desirable. However, it does not address the problem that it is in general unsafe to call libraries from detached/unjoined threads unless the user adds explicit synchronization to be sure he's still running before static destruction has begun.

2. The trouble with thread_local destruction and static destruction

C++ is one of the few languages with destructors, the rules for static destruction already have a hole (see "function local statics poisoning static destructors," below), and we're now trying to work out the rules for automatic destruction for a new storage class without existing practice (for automatic ordering rules) as a guide. (Pthreads' destructor approach does not correspond, for example because it is manual rather than automatic, and targets resurrection/retry/give-up rather than deterministic-single-destruction semantics.)

A general concern is that getting automatic destruction rules right for thread_local objects is inherently complex, because it magnifies today's existing difficulties. We already have subtle shutdown ordering rules for *sequential* shutdown of *a single* partially ordered set of global/static objects (with unsolved flaws; see "function local statics poisoning" below). We are attempting to expand the problem to *concurrent* shutdown of some or all of *arbitrarily many distinct* partially ordered sets of objects (each thread's set of thread_local objects, and the set of global/static objects).

For perspective, consider that it's already hard in C++98 to ensure graceful sequential shutdown of just a single partially ordered set of global/static objects, and at least one feature (function local statics) introduces a safety hole in global shutdown that we're just getting away with in practice because the feature is not widely used. Major issues include:

- *Ordering across translation units.* Expert programmers routinely resort to nifty counters and other subtle techniques, not all of which are actually portably reliable, to get the relative order of destruction correct for globals/statics in different translation units.
- *Function local statics poisoning static destructors (already problematic, but not widely recognized?).* Even among expert programmers, how many realize that it's actually unsafe in principle (i.e., potentially undefined behavior) for a destructor of a global or static object to call an arbitrary library function? This is because the library function might contain a function local static object in its implementation or transitively call another function that does so, that object might already have been destroyed, and if so then calling the function is undefined behavior (3.6.3/2). Theoretically, this means it's impossible to reliably call opaque library functions from the destructor of a global or static object, which means that global/static object destructors are formally nearly useless in C++ (see also §1); in practice, of course, they are useful and we're mostly getting away with this, because function local statics are not heavily used and libraries sometimes document which functions are unsafe to call during static destruction.

    Aside: Incidentally, if we want to close this hole, we could for example: (a) make "this function is guaranteed not to (transitively) access a function local static" a required part of its interface description and type, otherwise that function can't be called reliably during static destruction; or (b) add resurrection semantics to recreate the function local static object if it has already been destroyed (aka Phoenix singleton).

- *Standard library requires magic.* We already can't reliably implement the standard library's own single-threaded static shutdown requirements in portable C++ code without resorting to 'magic' not available to programmers in general. The standard requires implementations to do magic that cannot necessarily be written in portable C++

to guarantee that some key standard library facilities, such as memory allocation and iostreams, are reliably usable from static destructors.

Trying to allow thread_local objects with nontrivial destructors replicates the problem *N*-fold (for *N* threads):

- *Ordering.* Same issues as before, but with a combinatorial number of potential interactions if programmers allow references to thread_local objects to escape. Even if all non-main threads are joined with before the end of main, threads may perform thread_local destruction concurrently. An important mitigating factor is that normally those objects at least should not contain interdependencies if programmers follow the guidance to not hand out references to thread_local objects.
- *Function thread_local statics poisoning* thread_local *destructors.* The current draft permits function thread_local statics. Analogously to the global static case, destructors of thread_local objects must not try to call a function that contains a function thread_local static object because it might already have been destroyed. With this restriction, thread_local destructors can't safely call library functions because those might (transitively) use a function thread_local static. In practice, I think it's likely that we'll encounter this theoretical problem more often in practice than we do the global version we've been getting away with.

  If we're going to allow function thread_local statics in C++0x, we should somehow make "this function is guaranteed not to (transitively) access a function thread_local static" a required part of the interface of the function, or add resurrection semantics, or some other solution.

Program termination is inherently a mysterious time, difficult to understand and reason about in nearly any language even without the complication of threads. Trying to automate it by invention outside existing practice seems dangerous.

Ignoring the other difficulties, possible resolutions to the thread_local static poisoning issue include, in rough order from most to least desirable in my opinion:

- *2A (incomplete, but better than status quo): Remove the feature of function thread_local static objects.* This basically enables thread_local objects having nontrivial destructors, which otherwise couldn't safely call any library function because it might (transitively) contain function thread_local statics. It doesn't address any of the other complexity concerns though.
- *2B (bad, incomplete): Resurrection semantics.* Make function thread_local statics not poison thread_local destructors by giving them resurrection semantics: If one is accessed after it is destroyed, it is reconstructed (presumably initializaed the same way as before).

  That's a lot of work and invention to keep function thread_local statics, I doubt anyone will like it, and it has little implementation experience I know of in C++ (closest would be Alexandresu Singletons and pthreads destructors used in a resurrection style), but I suppose it could work. Even so, it would only make function thread_local statics safe(r); it doesn't address any of the complexity concerns.

- *2C (bad, incomplete): Annotations in the type system.* Make function thread_local statics not poison thread_local destructors by making "this function is guaranteed not to (transitively) access a function thread_local static" a required part of the interface of a function; enforce it by permitting such functions to call only other such functions; annotate the whole standard library to that effect; and finally require that thread_local static objects be of a type with a destructor that is either trivial or annotated not to invoke function thread_local statics.

  That's a lot of work and invention to keep function thread_local statics, I doubt anyone will like it, and it has no implementation experience I know of, but I suppose it could work. Even so, it would only make function thread_local statics safe(r); it doesn't address any of the complexity concerns.

See N2880 for broader possible resolutions of other issues mentioned in this section, including the option of disallowing nontrivial destructors for thread_local statics.

3. The trouble with thread_local data and reusable threads

No matter what we do now or in the future, with or without thread pools in the standard, thread_local variables are inherently a difficult-to-use feature on a system that reuses threads (e.g., for efficiency), including but not limited to thread pools. As illustrated in N2880, this is because cleanup is problematic (see §2 above) and programmers don't control the lifetime of the threads or which thread a task will execute on.

I'd like to note that, for a general developer audience, today's existing practice is to identify the use of thread_local objects on reusable threads such as thread pools as inherently problematic and warn programmers not to use them there.

Acknowledgments

Thanks to Lawrence Crowl and Peter Dimov for their comments on this paper, to Bill Plauger for his help with fact-checking, and to the many cpp-threads participants for extensive illuminating discussion on these issues. All remaining errors and boneheaded misunderstandings are, as always, my own.