Doc No: SC22/WG21/N2970 = PL22.16/09-0160

Date: 2009-09-26

Project: JTC1.22.32

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

A simple async() (revision 1)

This is a minor update of paper N2901 to update the proposed wording to not rely on concepts, fix a few typos, and to add informational section 2.5 with some additional background.

1. Introduction

1.1 Overview

This paper is a standalone proposal that competes with N2889/09-0079 because of what I and others believe are important details with undesirable resolutions in that proposal; these are listed and discussed in §2. However, first and foremost, I'd like to thank the author of N2889, Lawrence Crowl, for his hard work in putting together that proposal and working with a subgroup over email to iterate over it and build better understanding of the issues.

1.2 Points of agreement

This proposal and N2889 agree on the same basic design goal: to provide a simple means to perform an asynchronous function call that yields an asynchronous "future" result. Without this, the working paper provides no simple way to perform an asynchronous function call, nor any function that returns a future, which certainly seems incomplete. Today's status quo is that users must write packaged_tasks and fill futures/promises manually.

That is, just as we can make a synchronous function call that returns a synchronous result:

```
T t = f();
```

we want to be able to make an asynchronous function call that returns an asynchronous result (which here I'll just call "future<T>", but see §2 for discussion about what type this should be) :

```
future<T> t = async([]{ f(); });
```

It's essential that async *not require*, but also desirable that it *not exclude*, the implementation options of running its work on a thread pool or a work stealing runtime, which can be essential for efficient execution. To avoid precluding those implementation choices, it is sufficient to have runtime policies similar to the three basic policy options in N2889:

- *Guaranteed synchronous (N2889's "fully_synchronous", this proposal's "synchronous"):* This policy guarantees that the work will be run on the calling thread. This is primarily for debugging.
- *Guaranteed asynchronous (N2889's "fully_threaded", this proposal's "asynchronous"):* A policy that requires the work to be run on another thread. This is essential to enable scenarios where correctness relies on doing the work on a different thread. For example, when calling async() to get work off a GUI or other thread that must be highly responsive, it would defeat the purpose if the work might remain on the calling thread. For another example, the work might try to acquire a lock on a non-reentrant mutex already held by the calling thread, if it needs to use data also used by this thread. (Note: N2889 more specifically says "in a new thread" which is one of the problems because it excludes thread pool implementations; see §2.)
- *Either (proposed default; N2889's "impl_discretion", this proposal's "either"):* A policy that lets the implementation decide whether to perform the call synchronously or asynchronously. This enables orders-of-magnitude efficiency gains when the programmer knows that the program's correctness doesn't rely on running the work on a different thread, on the order of thousands or tens of thousands of cycles to run on a new or different thread vs. hundreds of cycles to run on this thread. This policy is necessary not only to permit work stealing implementations, but also necessary to permit efficient throttling in general where the system can avoid generating new tasks when there are already enough tasks pending to keep the machine saturated.

This proposal and N2889 agree on (most of) the above. However, there are several important areas where N2889 makes what I and other mailing list participants consider to be incorrect choices. §2 covers the major differences.

2 Issues and Differences

2.1 Avoid creating a new future type for use only with async

N2889 introduces a new future type (called "joining_future" in the current draft) beyond the unique_future and shared_future already present in the working draft. This is undesirable for several reasons:

- We already have future types. We should use them. (If they are not usable, we should fix them.)
- The new future type is currently a special-purpose type separate from the other future types. A bifurcated set of future types makes it hard to write general-purpose code. For example, a user should be able to write code that can accept any kind of standard

future, and the following function can indeed accept either a unique_future (with move/conversion) or a shared_future:

```
void f( shared_future<X>&& );

unique_future<X> u = ...;
f( move(u) );                   // ok

shared_future<X> s = ...;
f( s );                         // ok
```

But it cannot accept a joining_future:

```
// N2889 style
joining_future<X> s = async( ... );
f( s );                         // error
```

The user would be forced to work around this in some way. Major options include: (a) always write an overload that takes a joining_future; (b) make every such function a template with the future type as a template parameter; or (c) possibly write a custom function to "convert" a joining_future to a unique/shared future via a wrapper and a promise.

- The new future type can't be converted to a shared_future, which is necessary for all scenarios where a shared_future is desirable. For example, when sending the result to other code (e.g., in a message, or by spawning a new thread, etc.), it's important to distinguish whether that code needs the value immediately vs. eventually. Sometimes the former is fine and you would just join via .get() and send an ordinary value; other times you want the latter and you would just send a future to avoid being forced to collapse concurrency. Finally, whenever there can be multiple recipients, which is anytime the result is of interest to more than one thread, the future type would necessarily be a shared_future; as one class of example, in dataflow terms, this occurs in any example where the pipe splits two or more ways. (Note that this last point can be easily addressed by having N2889 also propose a conversion from joining_future to shared_future, but this is insufficient to address the main issues with joining_future.)

Objection 1. A major reason N2889 gives for not returning a unique_future is that the current futures interface isn't quite sufficient for someone to write their own implementation of std::async on a separately-authored implementation of unique_future that it doesn't know the internals of. This proposal disagrees, because:

- *The objection doesn't apply to async():* It's normal standard library practice to assume and require that in any given standard library implementation the same vendor implements both of two closely related components such as std::unique_future and std::async, and that one can know the internal details of the other.
- *If any correction is needed, the right place to fix is unique_future/shared_future/thread's interface (for timedness etc.) and/or thread_local destruction semantics.* If unique_future's in-

terface could be improved, then the solution isn't not to use it, it's to fix it. So if anyone considers it desirable to enable an independently authored std::async be supplied as part of a different library on top of a separately implemented std::unique_future, or otherwise that futures should be adjusted as noted in N2889, they should propose adjusting the futures interface. Any such proposed change is completely independent of this proposal, as this proposal does not depend upon such a change.

Objection 2. A major reason N2889 gives for not returning a unique_future is that the future must join with the thread used to run the task, so that the thread's thread_locals will be destroyed before the future.get() returns. The key concern seems to be cases like this:

```
int main() {
  ...
  future<T> t = async( f );
  ...
  t.get();
} // and immediately drop off the end of main
```

where it's possible that the thread_local destruction work on the thread used to run f might now run concurrently with global static destruction.

I disagree that this is a problem, because:

- The consumer of the future value doesn't care. It's sufficient that get() return once a value is available (promise is filled).
- In general, any function that accesses a global variable already has to ensure that the global variable still exists. In terms of the counter example in N2880, instead of making the counter itself a global object, each per-thread cache should contain a shared_ptr to the counter and detect when the use count goes down to one.
- What we're actually trying to achieve is to give a general guarantee that the tail end of async task (cleanup) won't slop over the end of main. But we can't do that anyway in general; as a simple example, the programmer can call async() from a static destructor and so is already past the end of main (though this could normally be considered deplorable style, as the work should be async-signal-safe, etc.); more realistically, the function that is calling async() doesn't know it's in turn being called from a static destructor. Nevertheless, if that is still a guarantee we want to give anyway for broader reasons, it needs a broader solution than adding it as a design requirement to async.

Objection 3. Another objection is performance. From N2889:

> *Furthermore, a modified unique_future would necessarily induce more overhead on the original intended uses of unique_future. … However, we have no measurements comparing that overhead to the normal cost of unique_future.*

First, as I understand it, the performance overhead surrounds the ability to join, which I do not think is necessary (see objection 2 above).

Second the point of unique_future::get() is that it might block, which already implies a large performance cost that will tend to swamp any smaller overheads. For any difference in the future object's performance to matter, the additional overhead would need to be at least of a similar order as the existing overhead of blocking.

Objection 4. Another objection is desire to leave unique_future stable and not change it. From N2889:

> *Modifying unique_future implies revisiting aspects of the working draft that we thought were stable.*

… and have found insufficient. The solution is to fix it, not make a new one.

2.2 Relax the restrictive "in a new thread" to "in another thread"

In N2889, the "guaranteed asynchronous" and "either" policies require a task to be executed "in a new thread" always. This has two performance problems:

- *It penalizes performance by precluding caching.* This wording essentially precludes implementations from caching threads in any way, including easily executing the task on a thread of an existing thread pool implementation.
- *It can penalize performance for compute-intensive async tasks by oversubscribing hardware in applications that already use thread pools for their compute-intensive work.* Applications that run their compute-intensive work on a thread pool really want all their compute-intensive work to run on the pool. The pool is already in the business of staying "rightsized" for the machine, and having compute-intensive work outside the thread pool interferes with the pool's ability to accurately match the number of ready threads to the available hardware. Each compute-intensive async task in a non-pool thread adds extra work that the thread pool doesn't know about and so results in oversubscribing the machine, providing more ready work than there is available hardware parallelism. If we mandate "in a new thread," then using async for a compute-intensive task will penalize an application that is using a thread pool to spread its compute-intensive work across the available hardware.

The wording for the "guaranteed asynchronous" policy should be "in another thread."

As rationale for "in a new thread," N2889 notes that it doesn't choose "another thread" because of the problems described in N2880. The issues raised in N2880 should be addressed, but they are not specific to async so as to justify influencing the async design. As various people and N2880 itself have pointed out, the basic issues around 'use-after-destruction of global objects from thread_local destructors of unjoined threads' has nothing to do specifically with async. The issues exist and need to be addressed no matter what we do with async. Note that none of the proposed resolutions in N2880 have anything to do with async directly and need to be considered no matter what we do with async.

The option should therefore be "in another thread," to allow the implementation to cache threads in any way.

2.3 Avoid adding overloads, and default the policy

We agree with N2889 that there is no need to provide both

```
async( [=]{ f( "xyzzy", 42, complex<float>(2.,1.) ); } );    // ok in both proposals
```

and the following (via variadic overload)

```
async( f, "xyzzy", 42, complex<float>(2.,1.) );              // ok alternative in N2889
```

Avoiding the variadic overload leaves a single function template, and permits the additional advantage that we can put the policy parameter last and default it (in this proposal, I suggest defaulting it to "either synchronous or asynchronous"), so that users can write:

```
async( [=]{ f(); }, async_policy::asynchronous );
async( [=]{ f(); } ); // defaults to "either"
```

That is what this paper recommends, with async as a single function template.

From N2889, the main reason not to do the above is consistency with std::thread which provides a variadic constructor overload:

> *We have no objection to* [the above-suggested] *approach. Indeed, it would make the referencing environment of the executed function quite explicit in the form of the lambda-capture. Should the variadic std::thread constructor be removed, we will modify the proposal to move the policy parameter to the end of the list and default it.*

For the same reasons given for async, and for consistency, the proposed text also removes std::thread's redundant variadic constructor overload. This is separable, and so the proposed text appears in a separate section (see §5).

2.4 Policy names

Other than the above issue, N2889 provides the right three options. We feel the enumerator names could be improved, and that async_policy should be a scoped enum to ensure these common readable names don't collide with anything else:

```
enum class async_policy {
   synchronous,
   asynchronous,
   either                    // proposed default
};
```

Another good alternative would be "async," "sync," and "either."

2.5 The intent of async is to support asynchronous concurrency (only)

The root of many of the foregoing misunderstandings trying to make async fit multiple uses, rather than its one simple originally intended use.

There are two major kinds of concurrent work a programmer might want to express, and async has always been about only the first one:

1. Asynchronous concurrency: To enable independent work to run independently.

The async facility is supposed to be entirely (and only) a way to do #1 without fiddling with std::packaged_tasks and explicit std::threads. For example, to get work off a GUI thread to run asynchronously so the GUI thread stays responsive. In this case the join (wait/get) operations are commonly expected to block because we expect most or all of the tasks will be run on a different thread. Because we're expecting to perform an expensive wait/block operation anyway, most of the concerns I've seen about the overhead on join performance seem to be rendered moot unless they're so enormous that they're comparable to the overhead of, say, a context switch.

2. Fine-grained parallelism: To use more cores to get the answer faster, by leveraging parallelism in algorithms and data structures to decompose the problem into independent chunks of work, run them in parallel, and then reduce the intermediate results to a final one.

The more complex N2889 proposal is in large part due to attempting to broaden async to also be usable for #2. For example, to do recursive divide-and-conquer where a key design point is to run synchronously without blocking for small tasks, particularly at the leaves of the computation. In this case the join (wait/get) operations are commonly expected to *not* block because we expect most of the tasks will end up being run sequentially on the caller's thread, and it's more important to have an efficient join because we want to enable efficient "inlining"(same-thread execution) of the dominating small leaf work.

Importantly, only #1 is in the scope of the Kona agreement and the original (and for many of us still the only) design goal for async. In the context of async, we are not interested in speculative execution and fine-grained parallelism and hyperefficient joins.

Even if #2 were in scope for C++0x (and it is not), I think it's fair to say that trying to do both #1 and #2 with a single async facility would mean repeating the mistakes of auto_ptr. Each of auto_ptr and async should be a simple facility that serves one simple purpose (stack-based lifetime and asynchronous concurrency, respectively), but we will render it complicated and compromised if we try to force it to also serve a second purpose (unique ownership with move-like semantics and fine-grained parallelism, respectively). And the difference between the concurrency and parallelism design goals #1 and #2 above is much greater than the difference between stack-based lifetime and unique ownership that got jammed into a single auto_ptr.

We need to stop trying to make async also fit the second use, which is at the root of much of the complication discussions and beyond the Kona agreement.

3. Proposed Wording for async

With grateful acknowledgment, the following is adapted from the proposed wording in N2889 to simplify it in line with the discussion above.

30.6.1 [futures.overview]

Add to the synopsis the appropriate entries from the following sections.

30.6.? Function template async [futures.async]

Add the following section.

```
enum class async_policy {
   synchronous,
   asynchronous,
   either                    // default
};
```

The enumerators synchronous, asynchronous, and either represent policies of execution.

```
template<class F>
unique_future<typename F::result_type>
async( F&& f, async_policy policy = async_policy::either );
```

Requires: F shall be CopyConstructible if an lvalue and otherwise MoveConstructible. The expression f() shall be a valid expression.

Effects: Constructs an object of type unique_future<F::result_type>. Any return value is captured by the unique_future. Any exception not caught by f is captured by the unique_future. If policy == async_policy::asynchronous, executes f() in a different thread of execution. If policy == async_policy::synchronous, the thread calling unique_future::get() executes f() in its own thread of execution. If policy == async_policy::either, the implementation may choose either policy synchronous or asynchronous.

Synchronization: The invocation of async happens before the invocation of f. *[Note:* This requirement applies even when the corresponding unique_future is moved to another thread. *—end note]*

Throws: std::system_error if policy is of type asynchronous and the system is unable to use or create another thread to execute f().

Error conditions: — resource_unavailable_try_again — if policy is of type asynchronous and the system lacked the necessary resources to use or create another thread.

[Example: Two items of work can be executed concurrently as below.

```
extern int work1(int value);
extern int work2(int value);

int work(int value) {
  auto handle = std::async( [=]{ work2(value); } );
  int tmp = work1(value);
  return tmp + handle.get();
}
```

– end example:]

4. Proposed Wording for Removing Variadic thread::thread

This is a separable issue so I've put it in its own section. It can be considered independently of the above, and it's just to make async and thread::thread consistent.

30.3.1 [thread.thread.class]

Remove the line:

```
template <class F, class ...Args> thread(F&& f, Args&&... args);
```

30.3.1.2 [thread.thread.constr]

Change indicated part of the text as follows (preceding and following text is unchanged):

```
template <class F> explicit thread(F f);
```

~~template <class F, class ...Args> thread(F&& f, Args&&... args);~~

4 *Requires:* F ~~and each type Ti in Args~~ shall be CopyConstructible if an lvalue and otherwise MoveConstructible. INVOKE(f~~, w1, w2, ..., wN~~) (20.7.2) shall be a valid expression ~~for some values w1, w2, ..., wN, where N == sizeof...(Args)~~.

5 *Effects:* Constructs an object of type thread and executes INVOKE(f, t1, t2, ..., tN) in a new thread of execution~~, where t1, t2, ..., tN are the values in args...~~. Any return value from f is ignored. If f terminates with an uncaught exception, std::terminate() shall be called.

Acknowledgments

Thanks to Beman Dawes and Lawrence Crowl for their specific review comments on this paper, and also to Hans Boehm, Peter Dimov, Howard Hinnant, Doug Lea, Arch Robison, Bjarne Stroustrup, Anthony Williams and the many other cpp-threads participants for extensive illuminating discussion and feedback on these proposals. All remaining errors and boneheaded misunderstandings are, as always, my own.