Doc No: WG21 N3451 = .16 12-0141
Date: 2012-09-23
Reply to: Herb Sutter (hsutter@microsoft.com)
Subgroup: SG1 – Concurrency

async and ~future

Herb Sutter

C++11 futures are a hit – their most important benefit is that they provide a wonderfully useful *lingua franca* type for combining asynchronous operations from multiple libraries, and they are gaining widespread adoption and usage experience.

Practical field experience with futures has demonstrated one major problem, which was discussed in May 2012 at the SG1 meeting in Bellevue.

~future/~shared_future Must Never Block

"Blocking is evil." – Various

The good news is that the Standard's specification of *future* and *shared_future* destructors specifies that they never block. This is vital.

The bad news is that the Standard specifies that the associated state of an operation launched by *std::async* (only!) does nevertheless cause future destructors to block. This is very bad for several reasons, three of which are summarized below in rough order from least to most important.

Example 1: Consistency

First, consider these two pieces of code:

```
// Example 1

// (a)                                  // (b)

{                                       {
   async( []{ f(); } );                    auto f1 = async( []{ f(); } );
   async( []{ g(); } );                    auto f2 = async( []{ g(); } );
}                                       }
```

Users are often surprised to discover that (a) and (b) do not have the same behavior, because normally we ignore (and/or don't look at) return values if we end up deciding we're not interested in the value and doing so does not change the meaning of our program.

The two cannot have the same behavior if *~future* joins.

Example 2: Correctness

Consider the following natural code, with surprising(?) semantics:

```
// Example 2: "Async, async everywhere! but…"

{
   async( launch::async, []{ f(); } );
   async( launch::async, []{ g(); } );
}
```

Users are often surprised to discover that *there is no concurrency at all in this "async-rich" code* and the standard requires it to be executed sequentially. They did not care about the return values, only wanted to launch some work (fire-and-forget).

Example 3: Composability (major)

The most important benefit of *std::* futures is that they provide a single common *lingua franca* type for combining asynchronous operations from multiple libraries.

The fact that *std::* future destructors sometimes block strikes at the heart of their most important strength. Consider: What does this code do? In particular, does it block?

```
void func() {

   future<int> f = start_some_work();

   /*... more code that doesn't f.get() or f.wait() */

}
```

The answer is different depending on whether the function (transitively) chose to launch its work via a *std::async* or via anything else (such as a *std::thread*). But it should not matter, because if the caller ends up not caring to know the value, it shouldn't have to block on the operation that produces the value. Note the situation would be worse (in degree only, not in kind) if the above code instead used a *shared_future*, because then whichever thread happens to be unlucky enough to run last will be the one that blocks.

This is not composable – when the primary purpose of *std::* futures is composability – because this potential blocking makes it difficult or impossible to use standard futures in many common situations. We must always be able to tell if code might block. For example, because *~future* might block, this code cannot reliably be called (transitively!) on a thread that must remain responsive, such as a GUI thread – not *func*, not even *start_some_work*. The workaround is to put the future on the heap (strange) or not use *std::* futures (a defeat for futures).

Objections

There were two main arguments that caused the current design where *~future* blocks if the future was produced by *std::async*. We feel these are either not well motivated, or are solving a valid problem in the wrong place.

Objection 1: Detachment

First, it was felt that because the returned future was the only handle to the *async* operation, if it did not block then there would be no way to join with the async operation, leading to a detached task that could run beyond the end of *main* off into static destruction time, which was considered anathema.

However, this argument fails to motivate a blocking *~future* because:

- It's not true that the program has no way to join, because it might use some other method such as signaling at the end of the async operation's body.
- Even if the problem were real, the right place to fix it would be to provide what is actually desired, namely a way to join with *async* tasks, such as to provide a *join_all_asyncs* function that the user could call at the end of *main*, or indeed require the end of *main* to implicitly join with all unjoined *async* tasks.

Objection 2: Reference Capture of Locals

Similarly, it was felt that it was too easy for a lambda spawned by an *async* operation to capture a local variable by reference but then outlive the function. For example:

However, this argument likewise fails to motivate a blocking *~future* because:

- It's not true that the program has no way to join, because it might use some other method such as signaling at the end of the async operation's body.
- The problem is not specific to lambdas passed to *std::async* or *std::thread*, it applies to all lambdas since they could be returned, or copied to a non-stack location, or otherwise outlive the function. The right place to address the problem would more generally be to warn (or prevent where possible) whenever a lambda that captures a local variable by reference *or any pointer or reference to a local variables captured or taken by any other means* outlives the local variable's scope by asynchronous launching *or any other means including by return value, out parameter, or any other write to a non-local location*.

Aside About ~thread

For similar reasons to the two objections above, we are in the peculiar situation where *~thread* calls *terminate* if not joined. That too is a defect; *~thread* should either join implicitly, or do nothing. But that's another paper, and fixing *async* and especially *~future* is much more important because they are the default launcher and the cornerstone of composability, respectively.

Proposed Resolution

Remove the requirement that releasing the shared state of a future spawned by *async* shall block.