Doc No: WG21 N3630
Date: 2013-04-12
Reply to: Herb Sutter (hsutter@microsoft.com)
Subgroup: SG1 – Concurrency
Previous Version: N3451

async, ~future, and ~thread (Revision 1)

Herb Sutter

This paper is an update of paper N3451, "async and ~future." [1]

> *"Fundamentally, functions should return a non-blocking future since they don't know whether the caller needs it to block or not. Whether the lifetime of the task needs to be contained to the current scope is caller's decision."*
>
> *– Niklas Gustafsson*

C++11 futures are a hit. Their most important benefit is **composability**: that they provide a wonderfully useful *lingua franca* type for combining asynchronous operations from multiple libraries (e.g., use two libraries that launch concurrent or parallel work and be able to wait on both), 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 and with a previous version of this paper (N3451) in Portland.

~future/~shared_future Must Never Block

The good news is that the Standard's specification of *future* and *shared_future* destructors specifies that they never block (via the shared state semantics in 30.6.4 which carefully describe only basic reference counting, and do not specify any blocking). This is vital.

The bad news is that 30.6.8/5 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(); } );
}                                   }
```

Example 1(a) has no concurrency; 1(b) does. 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 a blow to the heart of this their most important strength, composability. Consider:

```
// Example 3: What does this code do? In particular, can it block?

void func() {

   future<int> f = start_some_work();

   /*... more code that doesn't f.get() or f.wait(), and performs no other synchronization … */

} // Q: can this code block here?
```

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 and advantage of *std::* futures is composability – because this potential blocking makes it difficult or impossible to use standard futures in many common

situations, notably in nonblocking code. 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, and begging the question of how to clean it up) or not use *std::* futures (a defeat for futures).

Example 3 makes futures nearly unusable as a common composability type. The eventual calling code that handles a future must know whether something blocks, and so cannot use *~future* with the expectation that it won't block, as is common in modern non-blocking code. So the alternative might seem to be that with status quo people must write their code as if *~future* always blocks, but they can't do that because they can't rely on it blocking either. So *~future* really is in this unusable netherworld where we fail to give any guarantee and so don't know how to tell people to write code that uses it if that code may not need to *.get()* or *.wait()* (see Objection 3 below which notes some problematic workarounds). And this damage to futures is only because of the *async* quirk – futures and shared state themselves are otherwise not specified to block.

Potential resolutions include:

- (Preferred change) Remove the requirement that releasing an *async* operation's shared state shall block.
- (Status quo workaround) Teach programmers not to use *launch::async*, including not to use the default launch policy which includes *launch::async*, in order to avoid this problem.

Objections

There were several arguments that caused the current design where *~future* blocks if the future was produced by *std::async* and/or have been raised in the discussion of this question.

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.

Detached fire-and-forget tasks are indeed problematic, but the right place to fix this is not *~future*, but rather would be to provide what is actually desired, namely a way to join with *async* tasks.

Potential resolutions include:

- (Preferred) Require the return from *main* and every call to *exit* to implicitly join with all unjoined *async* tasks.
- Provide a *join_all_asyncs* function that the user could call anytime, including at the return from *main* and before every call to *exit*. However, this would join with all *async* tasks launched by any library, and so would generally be appropriate only at the return from *main* or at a call to *exit* (it could be problematic to call anywhere else) and required in those places as a best practice (so why not automate it so the programmer can't forget).
- Do both and specify the first in terms of the second.
- (Preferred) Also add a *local_future* whose destructor does *wait()*. (See next section.)

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:

```
// Example 4

void func()
{
   int i = 0;

   future<int> f = async( [&]{ i = 1; return i; } );     // crashes, if f's destructor doesn't wait

} // status quo: f's destructor waits, this is okay
```

We will dwell on this argument because it has been raised again and again. (Note: We captured this particular example, but there are slightly longer examples others also gave as objections, that this paper did not capture. Those examples should also be raised again by those who are concerned about them.)

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

- This does not even solve the problem it wants to solve, because it's still broken for *shared_future*s, return values, moving/copying to non-local locations, etc. Consider:

```
// Example 4, cont.: Why the status quo doesn't even solve the problem it targets

// Status quo doesn't address the same code for shared_future
void func()
{
   int i = 0;
   shared_future<int> f = async( [&]{ i = 1; return i; } );
   g( f );                                  // may store a copy of f
} // ~shared_future does not always join – error... sometimes! the very worst kind...
```

  Aside: *shared_future* is problematic for the status quo argument in general. In the email discussion that started with the above case, the comment was made by one expert that: "You could change *~future* to always wait, but this will not help the *shared_future* cases in which someone else holds a *shared_future* (and if someone else did not, you wouldn't have used a shared_future). Or you could make *~shared_future* wait too, but I suspect that this would be rather unhelpful." – This is an excellent point because making *~shared_future* also wait would be exactly in the spirit of the status quo, as well as obviously undesirable. And so it's a helpful way to demonstrate that the status quo is already on the wrong path, and illustrates why, in arguing by "forcing the conclusion": If there are valid reasons why we should block in *~future*, then those reasons also apply to *~shared_future*, and for the same reasons we should indeed block also in *~shared_future*. Since that would be obviously wrong, it helps to demonstrate that the status quo is taking us to the wrong place.

```
// Status quo doesn't address a slight modification of the code
void func()
{
  future<int> f;
  {
    int i = 0;
    f = async( [&]{ i = 1; return i; } );
  }
  f(); /*boom*/                    // still leaks a reference
}

// Status quo doesn't address a returned future
future<int> func()
{
  int i = 0;
  future<int> f = async( [&]{ i = 1; return i; } );
  return f;                        // moves the associated state, still leaks a reference
}

// Status quo doesn't address a moved future
void func()
{
  int i = 0;
  future<int> f = async( [&]{ i = 1; return i; } );
  some_obj.store( move(f) );     // moves the associated state, still leaks a reference
}
```

- The problem is not specific to lambdas passed to *std::async* or *std::thread*, or even to lambdas. It applies to all pointers and references to locals (whether contained in lambdas or not) 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 variable 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*. This really has nothing to do with *std::async* specifically, and *std::async* shouldn't be hacked in a mistaken attempt at solving one small corner of a far broader and longstanding issue.

// Example 4, cont.: Why the status quo is targeting a problem in no way related to async

```
// Status quo doesn't address same code without async
void func()
{
  function<size_t()> f;
  {
    int i = 0;
```

```
    f = [&]{ i = 1; return i; };
  }
  f(); /* boom */                  // functor outlives func (NB: no async required)
}

// Status quo doesn't address same code without async
function<int()> func()
{
  int i = 0;
  auto f = [&]{ i = 1; return i; };
  return f;                        // functor outlives func (NB: no async required)
}

// Status quo doesn't address same code without async
void func()
{
  int i = 0;
  auto f = [&]{ i = 1; return i; };
  some_obj.store( f );             // functor outlives func (NB: no async required)
}

// Status quo doesn't address same code without async
void func()
{
  size_t* f;
  {
    int i = 0;
    f = &i;
  }
  *f; /* boom */                   // pointer outlives func (NB: no async required)
}

// Status quo doesn't address same code without async
int& func()
{
  int i = 0;
  return i;                        // reference outlives func (NB: no async required)
}

// Status quo doesn't address same code without async
void func()
{
  int i = 0;
  some_obj.store( &i );            // reference outlives func (NB: no async required)
}
```

- Fundamentally, functions should return a non-blocking future since they don't know whether the caller needs it to block or not. Whether the lifetime of the task needs to be contained to the current scope is caller's decision, which we propose he be able to make locally by choosing to store the result in a *future* without calling *.wait()*, or by choosing to wait by calling *.wait()* or a ScopeGuard or a new type *local_future* (as proposed by Peter Dimov) as appropriate. Note that all of these are correct for both *future* and *shared_future*, unlike Example 4 above:

```
// Example 4(a): Callers that want to wait can say so

void func()
{
  int i = 0;

  auto f = async( [&]{ i = 1; return i; } );          // ok, because of f.wait later on

  // .. more code that doesn't throw or return early …

  f.wait();                                           // ok, joins before return
}

// Example 4(b): Or just use any vanilla RAII/release-at-end-of-scope style
void func()
{
  int i = 0;

  auto f = async( [&]{ i = 1; return i; } );          // ok, because of f.wait at end of scope
  ScopeGuard finally([&]{ f.wait(); });               // ok, joins before return

  // .. more code that might throw or return early …
}

// Example 4(c)': Or use a new local_future type (proposed by Peter Dimov)
void func()
{
  int i = 0;

  local_future<int> f = async( [&]{ i = 1; return i; } );      // ok, f's destructor will wait

  // .. more code that might throw or return early …
}
```

Potential resolution: Add a *local_future<T>* that is just like *future<T>*except that its destructor always performs a *.wait()*, and is constructible from either a *future<T>* or a *shared_future<T>*. Benefits:

1. It introduces only one new type.
2. It separates the RAII concern from the delayed-value concern.
3. The cases that some have raised and are rightly concerned about are those where local knowledge of the danger is actually present: The issue cannot occur unless you are capturing local variables by reference. We can teach users that this is (in general) dangerous, but if

passing a *shared_ptr<T>* by value instead of relying on by-reference capture isn't feasible, then catch the future in a *local_future*, by golly!

4. It would work for both *future<T>* and *shared_future<T>*, and would not depend on the origin of the future or whether its associated state has been shared. The destructor makes a call to *wait()*. Period.
5. It enables exception-safety for all futures, not just those that come from *async*: Since exceptions can result from promise-based futures, too, this gives us an elegant way of saying, in an RAII fashion, that this scope is where I want to observe any exceptions that may come my way through the future. We are actually much more concerned about ignoring exceptions (postponing them to exit from *main* is generally not a good idea, since so many programs never exit from *main*, at least not in a reasonable time frame) than the local-reference issue, but this proposal addresses both.

Objection 3: "Nearly no valid correct code wouldn't block on ~future."

This can be answered by many counterexamples. The following are two.

One counterexample is speculation:

```
void CalculateTrajectory( Spacecraft s, Location loc, Destination dest )
{
   vector<future<Trajectory>> result;
   for( auto& server : my_servers )
      result.push_back( async( [=]{ return server.CalcTheTrajectoryPlease( s, loc, dest ); } );

   return wait_any( result ).get();          // only care about the first answer
   // optionally cancel the losers if we have a way to do so, and care
   }
```

This function returns as soon as the first result is available. It doesn't "work" as intended if *~future* joins – that would make it hit the worst case of all servers, not the best case. We see no correctness problem in this code, particularly if all asyncs join automatically on return from *main*.

Another counterexample is optional work:

```
void SinkFunctionThatMightUseTheFuture( future<int>&& f )
{
   // f unused here
   if( some_condition() ) {
      DoSomethingElseWith( f.get() );
   }
   // f unused here
}
```

Why should this code be forced to block if it doesn't end up needing the value of f?

Question: In such cases, what's the workaround to write this code with the intended semantics if *~future* does join? Difficult and expensive: Somehow the futures must be put on the heap and their cleanup managed by polling or wasting a cleanup thread.

Objection 4: Code Breakage

As with any intended or unintended behavior in the standard, there is now an unknown not nonzero amount of existing code that relies on the current joining behavior of *~future*. There is a valid concern that removing the requirement that releasing an *async* operation's shared state shall block could break this code.

This section argues that the code that would be broken by this change is likely smaller, and more easily fixed, than the complementary set of existing code that is being written in ignorance of the current behavior quirk and so is already broken today.

(a) Code that relies on the status quo.

We argue that:

- The code that relies on this behavior is likely small, because it can only be relying on this with futures known specifically to be derived from either a call to *async(launch::async, f)* or a call to *async(/*default-policy*/ f)*. Furthermore, this means the caller is typically also the author of the *async* call and therefore both the future and the async call are usually or often in the same local scope.
- There is less such code now than there will be at any time in the future.[1]
- Not breaking such code incurs more serious costs. If we don't fix this now, then in the worst case we may end up eventually having to deprecate (formally or informally) the existing futures and replace them with a properly behaved type which would be a far larger breaking change. At minimum we would feel we will have to teach people to avoid *launch::async* and any default that includes it as "almost great but in practice subtly broken" because of this issue which is also a real cost. So "staying with the status quo means we don't break code that may rely on the behavior" does not mean that staying with the status quo incurs no costs.
- The fix is simple. If any code does rely on *~future* blocking and is broken if we accept the proposed change, just add *.wait()*.

(b) Code that relies on *~future* not blocking.

On the other hand, we believe there is code being written that assumes the opposite, namely that *~future* does not block. We have seen developers write code that uses futures and expects *~future* not block (on various compilers, not just ours because we're nonconforming), because they do not realize this problem and because today their code appears to (and does) work as expected in testing and in release as long as they do not encounter a future attached to a non-ready task launched by *async* with

---

[1] Note and disclaimer: Visual C++ does not yet implement the standard behavior for this case while awaiting for the result of this discussion. Our implementation being not conforming in this place should not be viewed as 'existing practice' or otherwise taken as a reason to take this change, and were are not arguing that not making the change would break customers who rely on our nonconforming implementation. If the standard does not change, we will conform and break any customer code that might rely on the nonstandard behavior as entirely our own responsibility and this should not weigh in the decision. – However, what we are noting is that, if the committee otherwise arrives at a consensus to view the current situation as a design flaw and wants to fix the problem but is concerned about code breakage, then the fact that Visual C++ already has the new behavior mitigates that concern by reducing the amount of code that might be relying on the old behavior.

launch::async. Their code will no longer work correctly (will block in a difficult to debug way, and in extreme cases could deadlock) if eventually the future they're handling is attached to a non-ready task launched by *async* with *launch::async*.

If any code does rely on *~future* not blocking and is broken if we do not accept the proposed change, the fix is more difficult: It involves moving the future to the heap or another non-local location and then managing its cleanup somehow. Note that we cannot tell whether the future is ready and therefore safe to destroy without polling; the only obvious way to destroy the future without polling would be to waste a thread, that is, to create a thread that does nothing but wait for the future.

We have no data about how much code is bring written with the expectation that *~future* blocks vs. the expectation that *~future* does not block. However, we do know that the first is easier to fix.

~thread Should Join

For similar reasons to the notes above, we are in the peculiar situation where *~thread* calls *terminate* if not joined. That too is a problem, as noted for example in [2]:

```
// Example 5

void doSomeWork();

void f1()
{
   std::thread t(doSomeWork);
   ...                                    // no join, no detach
}
```

What happens?

Your program is terminated.

Instead, exactly one of the following should be true:

- (preferred) either *thread* owns the resource as an RAII type and therefore *~thread* should join implicitly;
- or it is not and *~thread* should do nothing.

We propose the former as this is consistent with the rest of the intent of the *thread* type, such as movability.

Proposed Resolutions

1. Require that return-from-*main* and *exit* join with outstanding *async* operations.

2. Remove the requirement that releasing an *async* operation's shared state shall block.

#1 guarantees those *async* operations will join before static destruction begins, while still permitting programs that desire it to launch new async operations after the end of main during static destruction.

#2 solves the problem articulated in the first part of this paper.

Change 30.6.8/5 as follows:

5 *Synchronization:* Regardless of the provided `policy` argument,

— the invocation of `async` synchronizes with (1.10) the invocation of `f`. [*Note*: This statement applies even when the corresponding `future` object is moved to another thread. —*end note*]; and

— the completion of the function `f` is sequenced before (1.10) the shared state is made ready. [*Note*: `f` might not be called at all, so its completion might never happen. —*end note*]

If the implementation chooses the `launch::async` policy,

— a call to a waiting function on an asynchronous return object that shares the shared state created by this `async` call shall block until the associated thread has completed, as if joined (30.3.1.5);

— the associated thread completion synchronizes with (1.10) the return from the first function that successfully detects the ready status of the shared state ~~or with the return from the last function that releases the shared state, whichever happens first.~~; and

— if the invocation of `async` happens-before (1.10) the return from `main` or a call to `exit`, then the associated thread completion synchronizes with (1.10) the return from `main` and every call to `exit`.

3. Require that *~thread* and *thread::operator=* implicitly join.

This has no effect on programs that do not currently terminate. It just replaces the requirement to call *terminate* with the requirement to instead call *join*.

Change 30.3.1.3 as follows:

```
~thread();
```

1 If `joinable()`, calls join() ~~`std::terminate()`~~. Otherwise, has no effects. ~~[*Note*: Either implicitly detaching or joining a `joinable()` thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. —*end note*]~~

Change 30.3.1.4 as follows:

```
thread& operator=(thread&& x) noexcept;
```

1 *Effects:* If `joinable()`, calls join() ~~`std::terminate()`~~. ~~Otherwise,~~Then assigns the state of `x` to `*this` and sets `x` to a default constructed state.

2 *Postconditions:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment.

3 *Returns:* `*this`

4. Add a "scoped" local_future whose destructor always blocks.

In 30.6, add a new subclause to specify a *local_future<T>* as follows.

Clone the text of 30.6.7 *shared_future<T>*, changing all occurrences of *shared_future* to *local_future*.

Add moving and copying from *shared_future*:

- In the class synopsis, add moving and copying from *shared_future*:
  ```
  local_future(const shared_future<R>& rhs);
  local_future(shared_future<R>&& rhs) noexcept;
  local_future& operator=(const shared_future<R>& rhs);
  local_future& operator=(shared_future<R>&&) noexcept;
  ```
- At the description of *local_future(const local_future& rhs);* add also (to share the same description):
  ```
  local_future(const shared_future<R>& rhs);
  ```
- At the description of *local_future(local_future&& rhs);* add also (to share the same description):
  ```
  local_future(shared_future<R>&& rhs) noexcept;
  ```
- At the description of *local_future& operator=(const local_future& rhs);* add also (to share the same description):
  ```
  local_future& operator=(const shared_future<R>& rhs);
  ```
- At the description of *local_future& operator=(local_future&& rhs);* add also (to share the same description):
  ```
  local_future& operator=(shared_future<R>&&) noexcept;
  ```

Replace the destructor behavior with blocking behavior:

- Change the destructor description to:
  ```
  ~local_future();
  ```
  Effects:
  — calls `wait()`;
  — releases any shared state (30.6.4);
  — destroys `*this`.

Acknowledgments

Thanks to Hans Boehm, Peter Dimov, Niklas Gustafsson, Artur Laksberg, and Anthony Williams for their comments on drafts of this paper and contributing examples and discussion. Any errors or mischaracterizations or missing examples are our fault, not theirs.

References

[1] H. Sutter. "async and ~future" (WG21 paper N3451, September 23, 2012).

[2] S. Meyers. "Thread Handle Destruction and Behavioral Consistency" (March 25, 2013). Retrieved on April 10, 2013.