This paper is a followup to paper N3630.

In discussion of N3630, SG1 expressed support for the following direction:

1. Have a distinct ‘future’ type whose destructor never waits. Have a unique and a shared version of this type.
2. Have a distinct ‘future’ type whose destructor always waits if the caller did not already call .get() or .wait(). Have a unique and a shared version of this type.
3. Have compatibility for existing code that uses async and relies on its existing semantics, including deferred work. Ideally, code that is valid C++11 but that changes meaning should not compile.

Summary

Accomplish the above as follows:

1. Have future<T> with unique ownership, and shared_future<T> with shared ownership, be the type whose destructor never waits. This already true except only when the shared state came from async with launch::async.
2. Add waiting_future<T> with unique ownership, and shared_waiting_future<T> with shared ownership, as the type whose unique or last destructor always waits for non-deferred tasks if the caller did not already call .get() or .wait(). A waiting_future<T> is explicitly move-convertible to a future<T> by calling .detach(), modeled after .share().
3. Have async return a waiting_future<T>.

The type conversions are:
Here are the types in action, with existing valid C++11 code shaded:

```cpp
future<int> f1 = async([]{ return 1; }); // error, detach required
// (this is the only breaking change case, incl. shared_future variant below)

auto f2 = async([]{ return 1; }); // ok, preserves C++11 meaning

future<int> f3 = f2.detach(); // ok

waiting_future<int> f4 = async([]{ return 1; }); // ok
waiting_future<int> f5 = f2; // error, detach required

waiting_future<int> f6 = move(f2); // ok
waiting_future<int> f7 = f3; // error, move required
waiting_future<int> f8 = move(f3); // ok

future<int> f9 = f8;
future<int> f10 = f8.detach(); // error, detach required

shared_future<int> f11 = async([]{ return 1; }); // error, detach required

shared_future<int> f12 = f2.detach(); // ok
shared_future<int> f13 = move(f8); // error, detach required
shared_future<int> f14 = f8.detach(); // ok (move/share implicit)

shared_waiting_future<int> f15 = async([]{ return 1; }); // ok
shared_waiting_future<int> f16 = f8; // error, move required
shared_waiting_future<int> f17 = move(f8); // ok
shared_waiting_future<int> f18 = f9; // error, move required
shared_waiting_future<int> f19 = move(f9); // ok

shared_future<int> f20 = f3; // error, move or share required
shared_future<int> f21 = move(f3); // ok
shared_future<int> f22 = f3.share(); // ok
```

Proposed Wording

1. future and shared_future

Change 30.6.6/9-11 as follows:

```cpp
~future();

9 Effects:
— releases any shared state (30.6.4) without blocking until the shared state is ready;
— destroys *this.

future& operator=(future&& rhs) noexcept;

10 Effects:
— releases any shared state (30.6.4) without blocking until the shared state is ready;
— move assigns the contents of rhs to *this.
```
11 Postconditions:
— valid() returns the same value as rhs.valid() prior to the assignment.
— rhs.valid() == false.

Change 30.6.7/11-15 as follows:

~shared_future();

11 Effects:
— releases any shared state (30.6.4), without blocking for the shared state to be ready;
— destroys *this.

shared_future& operator=(shared_future&& rhs) noexcept;

12 Effects:
— releases any shared state (30.6.4), without blocking for the shared state to be ready;
— move assigns the contents of rhs to *this.

13 Postconditions:
— valid() returns the same value as rhs.valid() prior to the assignment.
— rhs.valid() == false.

shared_future& operator=(const shared_future& rhs) noexcept;

14 Effects:
— releases any shared state (30.6.4), without blocking for the shared state to be ready;
— assigns the contents of rhs to *this. [Note: As a result, *this refers to the same shared state as rhs (if any). —end note]

15 Postconditions: valid() == rhs.valid().

Change 30.6.4 as follows:

5 When an asynchronous return object or an asynchronous provider is said to release its shared state, it means that without blocking for the shared state to be ready:
— if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and
— the return object or provider gives up its reference to its shared state.

2. waiting_future and shared_waiting_future
Add a new sections 30.6.X and .X++ as follows to add waiting_future and shared_waiting_future (based on std::future and std::shared_future, with the major differences from the originals highlighted):

30.6.X Class template waiting_future [futures.waiting_future]
The class template waiting_future defines a type for asynchronous return objects which do not share their shared state with other asynchronous return objects and wait() for non-deferred shared state automatically when assigned to or destroyed. A default-constructed future object has no shared state. A waiting_future object with shared state can be created from a future, or from the type returned by std::async (30.6.8), or by moving from another waiting_future, and shares its shared state with the original asynchronous provider. The result (value or exception) of a waiting_future object can be set by calling a function on an object that shares the same shared state.

[Note: Member functions of waiting_future do not synchronize with themselves or with member functions of future, shared_future, or shared_waiting_future. —endnote]

The effect of calling any member function other than the destructor, the move-assignment operator, or valid on a waiting_future object for which valid() == false is undefined. [Note: Implementations are encouraged to detect this case and throw an object of type future_error with an error condition of future_errc::no_state. —endnote]

```cpp
namespace std {
    template <class R>
    class waiting_future {
    public:

        waiting_future() noexcept;
        waiting_future(waiting_future &&) noexcept;
        waiting_future(future &&) noexcept;
        waiting_future(const waiting_future& rhs) = delete;
        ~waiting_future();

        waiting_future& operator=(const waiting_future& rhs) = delete;
        waiting_future& operator=(waiting_future&&) noexcept;

        future<R> detach();
        shared_waiting_future<R> share();

        // retrieving the value
        see below get();

        // functions to check state
        bool valid() const noexcept;

        void wait() const;
        template <class Rep, class Period>
        future_status wait_for(
            const chrono::duration<Rep,Period>& rel_time) const;
        template <class Clock, class Duration>
        future_status wait_until(
            const chrono::time_point<Clock,Duration>& abs_time) const;
    }
}
```
4 The implementation shall provide the template `waiting_future` and two specializations, `waiting_future<R&>` and `waiting_future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

```
waiting_future() noexcept;
```

5 **Effects:** constructs an *empty* `waiting_future` object that does not refer to a shared state.

6 **Postcondition:** `valid() == false`.

```
waiting_future(waiting_future&& rhs) noexcept;
waiting_future(future&& rhs) noexcept;
```

7 **Effects:** move constructs a `waiting_future` object that refers to the shared state that was originally referred to by `rhs` (if any).

8 **Postconditions:**
   — `valid()` returns the same value as `rhs.valid()` prior to the constructor invocation.
   — `rhs.valid() == false`.

```
~waiting_future();
```

9 **Effects:**
   — if `valid()` is true and the shared state does not contain a deferred function, calls `wait()`;
   — releases any shared state (30.6.4);
   — destroys `*this`.

```
waiting_future& operator=(waiting_future&& rhs) noexcept;
```

10 **Effects:**
   — if `valid()` is true and the shared state does not contain a deferred function, calls `wait()`;
   — releases any shared state (30.6.4).
   — move assigns the contents of `rhs` to `*this`.

11 **Postconditions:**
   — `valid()` returns the same value as `rhs.valid()` prior to the assignment.
   — `rhs.valid() == false`.

```
future<R> detach();
```

12 **Effects:** transfers ownership of any shared state (30.6.4) of `*this` to a newly constructed `future<R>` object.

13 **Returns:** a `future<R>` object that refers to the shared state that was originally referred to by `*this` (if any).

14 **Postconditions:** `valid() == false`. 
shared_waiting_future<R> share();

15 **Returns:** shared_waiting_future<R>(std::move(*this)).

16 **Postcondition:** valid() == false.

    R waiting_future::get();
    R& waiting_future<R&>::get();
    void waiting_future<void>::get();

17 **Note:** As described above, the template and its two required specializations differ only in the return type and return value of the member function `get`.

18 **Effects:** `wait()`s until the shared state is ready, then retrieves the value stored in the shared state.

19 **Returns:**
   - `future::get()` returns the value `v` stored in the object's shared state as `std::move(v)`.
   - `future<R&>::get()` returns the reference stored as value in the object’s shared state.
   - `future<void>::get()` returns nothing.

20 **Throws:** the stored exception, if an exception was stored in the shared state.

21 **Postcondition:** valid() == false.

    bool valid() const noexcept;

22 **Returns:** `true` only if `*this` refers to a shared state.

    void wait() const;

23 **Effects:** blocks until the shared state is ready.

    template <class Rep, class Period>
    future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;

24 **Effects:** none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared state is ready or until the relative timeout (30.2.4) specified by `rel_time` has expired.

25 **Returns:**
   - `future_status::deferred` if the shared state contains a deferred function.
   - `future_status::ready` if the shared state is ready.
   - `future_status::timeout` if the function is returning because the relative timeout (30.2.4) specified by `rel_time` has expired.

    template <class Clock, class Duration>
future_status wait_until(
    const chrono::time_point<Clock, Duration>& abs_time) const;

26  Effects: none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared state is ready or until the absolute timeout (30.2.4) specified by abs_time has expired.

27  Returns:
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the absolute timeout (30.2.4) specified by abs_time has expired.

30.6.X++ Class template shared_waiting_future [futures.shared_waiting_future]

1  The class template shared_waiting_future defines a type for asynchronous return objects which may share their shared state with other asynchronous return objects and wait() for non-deferred shared state automatically when the shared_waiting_future that is the last asynchronous return object that references the shared state is assigned to or destroyed. A default-constructed shared_waiting_future object has no shared state. A shared_waiting_future object with shared state can be created by conversion from a future or waiting_future object and shares its shared state with the original asynchronous provider (30.6.4) of the shared state. The result (value or exception) of a shared_waiting_future object can be set by calling a respective function on an object that shares the same shared state.

2  [Note: Member functions of shared_waiting_future do not synchronize with themselves, but they synchronize with the shared shared state. —end note]

3  The effect of calling any member function other than the destructor, the move-assignment operator, or valid() on a shared_waiting_future object for which valid() == false is undefined. [Note: Implementations are encouraged to detect this case and throw an object of type future_error with an error condition of future_errc::no_state. —end note]

namespace std {
    template <class R>
    class shared_waiting_future {
    public:
        shared_waiting_future() noexcept;
        shared_waiting_future(const shared_waiting_future& rhs);
        shared_waiting_future(waiting_future<R>&& rhs) noexcept;
        shared_waiting_future(shared_waiting_future&& rhs) noexcept;
        ~shared_waiting_future();
        shared_waiting_future& operator=(const shared_waiting_future& rhs);
        shared_waiting_future& operator=(shared_waiting_future&& rhs) noexcept;
        // retrieving the value
        see below get() const;
    }
// functions to check state
bool valid() const noexcept;

void wait() const;
template <class Rep, class Period>
future_status wait_for(
    const chrono::duration<Rep, Period>& rel_time) const;
template <class Clock, class Duration>
future_status wait_until(
    const chrono::time_point<Clock, Duration>& abs_time) const;
}

The implementation shall provide the template shared_waiting_future and two
specializations, shared_waiting_future<R&> and shared_waiting_future<void>. These
differ only in the return type and return value of the member function get, as set out in its
description, below.

shared_waiting_future() noexcept;

Effects: constructs an empty shared_waiting_future object that does not refer to an shared
state.

Postcondition: valid() == false.

shared_waiting_future(const shared_waiting_future& rhs);

Effects: constructs a shared_waiting_future object that refers to the same shared state as rhs
(if any).

Postcondition: valid() returns the same value as rhs.valid().

shared_waiting_future(shared_future<R>&& rhs) noexcept;
shared_waiting_future(shared_waiting_future&& rhs) noexcept;

Effects: move constructs a shared_waiting_future object that refers to the shared state that
was originally referred to by rhs (if any).

Postconditions:
— valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
— rhs.valid() == false.

~shared_waiting_future();

Effects:
— if valid() is true, and *this is the last asynchronous return object that references the
  shared state, and the shared state does not contain a deferred function, then calls wait();
— releases any shared state (30.6.4);
— destroys *this.

shared_waiting_future& operator=(shared_waiting_future&& rhs) noexcept;

12 **Effects:**
— if valid() is true, and *this is the last asynchronous return object that references the
  shared state, and the shared state does not contain a deferred function, then calls wait();
— releases any shared state (30.6.4);
— move assigns the contents of rhs to *this.

13 **Postconditions:**
— valid() returns the same value as rhs.valid() returned prior to the assignment.
— rhs.valid() == false.

shared_waiting_future& operator=(const shared_waiting_future& rhs);

14 **Effects:**
— if valid() is true, and *this is the last asynchronous return object that references the
  shared state, and the shared state does not contain a deferred function, then calls wait();
— releases any shared state (30.6.4);
— assigns the contents of rhs to *this. [Note: As a result, *this refers to the same shared state
  as rhs (if any). —end note]

15 **Postconditions:** valid() == rhs.valid().

const R& shared_waiting_future::get() const;
R& shared_waiting_future<R&>::get() const;
void shared_waiting_future<void>::get() const;

16 **Note:** as described above, the template and its two required specializations differ only in the
return type and return value of the member function get.

17 **Note:** access to a value object stored in the shared state is unsynchronized, so programmers
should apply only those operations on R that do not introduce a data race (1.10).

18 **Effects:** wait()s until the shared state is ready, then retrieves the value stored in the shared
state.

19 **Returns:**
— shared_waiting_future::get() returns a const reference to the value stored in the
object’s shared state. [Note: Access through that reference after the shared state has been
destroyed produces undefined behavior; this can be avoided by not storing the reference in any
storage with a greater lifetime than the shared_waiting_future object that returned the
reference. —end note]
— shared_waiting_future<R&>::get() returns the reference stored as value in the object’s
shared state.
— shared_waiting_future<void>::get() returns nothing.

20 **Throws:** the stored exception, if an exception was stored in the shared state.

bool valid() const noexcept;

21 **Returns:** true only if *this refers to a shared state.

void wait() const;

22 **Effects:** blocks until the shared state is ready.

```cpp
template <class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

23 **Effects:** none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared state is ready or until the relative timeout (30.2.4) specified by rel_time has expired.

24 **Returns:**
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the relative timeout (30.2.4) specified by rel_time has expired.

```cpp
template <class Clock, class Duration>
future_status wait_until(
    const chrono::time_point<Clock, Duration>& abs_time) const;
```

25 **Effects:** none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared state is ready or until the absolute timeout (30.2.4) specified by abs_time has expired.

26 **Returns:**
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the absolute timeout (30.2.4) specified by abs_time has expired.

3. **async changes**

In 30.6.1, change the declarations of async as follows:

1 The function template async provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a waiting_future object with which it shares ownership of a shared state.
template <class F, class... Args>
    waiting_future<typename result_of<typename decay<F>::type(typename
decay<Args>::type...)>::type>
    async(F&& f, Args&&... args);

template <class F, class... Args>
    waiting_future<typename result_of<typename decay<F>::type(typename
decay<Args>::type...)>::type>
    async(launch policy, F&& f, Args&&... args);

Change 30.6.8/1 as follows:

    template <class F, class... Args>
        waiting_future<typename result_of<typename decay<F>::type(typename
decay<Args>::type...)>::type>
        async(F&& f, Args&&... args);

    template <class F, class... Args>
        waiting_future<typename result_of<typename decay<F>::type(typename
decay<Args>::type...)>::type>
        async(launch policy, F&& f, Args&&... args);

Change 30.6.8/4 as follows:

4 Returns: An object of type waiting_future<typename result_of<typename
    decay<F>::type(typename decay<Args>::type...)>::type> that refers to the shared state
    created by this call to async.

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.

— the associated thread holds a reference to the associated shared state which is released (30.6.4) when the associated thread exits.