Doc. No.: WG21/P0930
Date: 2018-02-11
Authors: Lee Howes lwh@fb.com,
Andrii Grynenko andrii@fb.com,
Jay Feldblum yfeldblum@fb.com
Reply-to: Lee Howes
E-mail: lwh@fb.com
Audience: SG1

P0930 - Semifying Awaitables

Summary

In **P0904** we discussed how SemiFuture and ContinuableFuture can improve the future API, and how those two classes can plug into a world with executors that have to deal with both bulk operations, and greedily constructed dependence graphs as we would see in GPU work queues.

This paper aims to also consider how we should think about the relationship between Futures and Awaitable types as discussed in **N4680.** We look at how the same SemiFuture/ContinuableFuture model can apply to coroutines, to achieve the same goals, and how futures may safely represent coroutines for API boundaries where interoperation is a requirement.

Existing Coroutine Concepts

We rely on the `Awaitable` concept from the coroutines TS/**N4680**. For these purposes this can be defined as "a type that supports `co_await`" but for clarity we leave it more fleshed out below. We propose no changes to the `Awaitable` types as used in **N4680**. Most awaitables in use will be simple callback-based awaitables that can optimise away to nothing. There is no expectation that such an awaitable will resume on any particular executor or indeed support executors in any meaningful way. Very roughly:

```
concept Awaitable {
      bool await_ready();
      template<class T>
      bool await_suspend(coroutine_handle<T>);
      void await_resume();
};
```

New coroutine concepts

For the purposes of this paper, we define a set of new classes of types.

As noted in **P0904**, for some libraries it is important to be able to separate the caller's execution context from the callee's. In this world, picture an asynchronous library on which we expect to `co_await`, but in a situation where the library wants to guarantee that when it calls `resume()` on the waiter's coroutine handle, it is resumed on some well defined executor rather than executed inline.

That library should return a `SemiAwaitable`. `SemiAwaitable` exists to allow asynchronous APIs to return an awaitable associated with some coroutine, but with a guarantee that the calling coroutine will have to ensure that the returned Awaitable will have some executor to complete on. This guarantees a safe contract between asynchronous coroutine-based APIs and callers.

`SemiAwaitable` is defined as a type that has a `via` operation exposed as a customization point that itself takes an r-value of the `SemiAwaitable` type and an executor and that returns an `AsyncAwaitable` of that executor. The return concept of `AsyncAwaitable` will be described next. A `SemiAwaitable` can be constructed from any `Awaitable` (or indeed `AsyncAwaitable`), acting as a simple execution-context-erasing wrapper.

```
template<T>
concept SemiAwaitable {
      SemiAwaitable(Awaitable<T>);
};

template<OneWayExecutor Ex, SemiAwaitable<T> ConcreteSemiAwaitable>
AsyncAwaitable<Ex> via(ConcreteSemiAwaitable&&, Ex);
```

In this case `Ex` has to be a `OneWayExecutor`, or it must be an executor that can be converted into a `OneWayExecutor` because the coroutine execution model depends on lazy enqueue.

`SemiAwaitable` is not itself `Awaitable`, but it is convertible to `AsyncAwaitable` with a simple implementation of `await_transform` on any given executor-carrying Awaitable by calling `via` with the `SemiAwaitable` and the caller's executor.

NOTE: As an optional strengthening of safety rules, we can require that `Ex` satisfy the `never_blocking` property of the executors, or be convertible to such an executor using require. This removes an executor that executes the function inline with the `execute` method as a valid implementation, but would give a stronger guarantee to the library. In this case the executor associated with the returned `AsyncAwaitable` might not match the type of the one passed to via.

An `AsyncAwaitable` is an `Awaitable` that acts at asynchronous boundaries and executor transitions and is executor aware. When it is resumed and forwards resumption onto the waiter, it will package the call to the waiter's `resume()` method into a task and launch it on its executor. The caller will hence be resumed on some known executor - most likely whatever executor it was running on when it awaited. This acquires the functionality of `SemiAwaitable` and `Awaitable`, and can hence be `co_await`ed and can also be converted to an `AsyncAwaitable` carrying some other executor type.

```
template<class T, Executor Ex>
concept AsyncAwaitable : SemiAwaitable<T>, Awaitable<T> {
      AsyncAwaitable(/* self type */&&);

      // Note that an AsyncAwaitable is not publically constructible
      // from arbitrary SemiAwaitables or Awaitables.

      Ex get_executor() const;
};

template<OneWayExecutor Ex, SemiAwaitable<T> ConcreteSemiAwaitable>
AsyncAwaitable<Ex> via(ConcreteAsyncAwaitable&& aw, Ex);
```

A call to `via(std::move(aa), ex)` is allowed to return `std::move(aw)` if the passed executor instance, `ex`, matches the executor attached to `aw`.

An Awaitable need not have an attached executor. An `AsyncAwaitable` must have one, and will always resume the awaiter's coroutine_handle on the executor attached to the `AsyncAwaitable`.

Interactions between Futures and Awaitables

As discussed in **P0930** we see a separation between `SemiFuture` and `ContinuableFuture` as future concepts in the standard. This separation we can now see aligns closely with the separation we propose for Awaitables, and is proposed for the same reason in both cases.

Furthermore, `SemiFuture` is a `SemiAwaitable` type in that it has a `via` customization point exposed, which returns an instance of the `ContinuableFuture` concept. `ContinuableFuture` is an `AsyncAwaitable`.

```
template<class T>
concept SemiFuture : SemiAwaitable {
      explicit SemiFuture(/* implementation-defined ContinuableFuture */&&);

      // Move constructible
      SemiFuture(/*self type*/&&);

      // get and get_expected are both destructive.
```

```
    // get will throw on exception. get_expected will return either a value
    // or an exception.
    T get() &&;
    expected<T, exception_ptr> get_expected() noexcept &&;

    // Wait is not destructive.
    SemiFuture<T>& wait() noexcept &;
    SemiFuture<T>&& wait() noexcept &&;

    bool is_ready() noexcept;
};
```

The `via` customization point of `SemiFuture` is consistent with that of SemiAwaitable, but will of course return a `ContinuableFuture`, both types being narrowed relative to the more general `SemiAwaitable` definition:

```
template<OneWayExecutor Ex, SemiFuture<T> ConcreteSemiFuture>
ContinuableFuture<Ex> via(ConcreteSemiFuture&&, Ex);
```

We add the ability to enqueue continuations on a future using the `ContinuableFuture` concept. `ContinuableFuture` matches the `AsyncAwaitable` concept and is trivially awaitable because we can always await by using a continuation to trigger the callback, so we can fall back to a default form.

```
template<class T, Executor Ex>
concept ContinuableFuture : SemiFuture, AsyncAwaitable<Ex>  {
    using executor_type = Ex;
    using semi_future_type = /* implementation-defined */

    // Move constructor
    ContinuableFuture(/*self type*/&&);

    template<...>
    ContinuableFuture<ReturnT, Ex> then(F&&);

    template<...>
    ContinuableFuture<ReturnT, Ex2> bulk_then(
        F&& f,
        executor_shape_t<Ex> shape,
        SharedFactory&& s,
        ResultFactory&& r);


    bool await_ready();
    template<class T>
    bool await_suspend(coroutine_handle<T>);
    void await_resume();
```

```
    Ex get_executor() noexcept;
    semi_future_type semi() &&;
};
```

Standardised Future type extended with awaitables

The standard future types specified in **P0904** need to be extended with Awaitable support.

`SemiFuture` may be constructed from any `SemiAwaitable`, which includes other `SemiFuture`s, but also `AsyncAwaitable` types, including `ContinuableFuture`, where the executor is erased from the public interface. This is a generalisation of the support discussed in **P0904**.

SemiFutures are also constructible from arbitrary Awaitables. This makes wrapping a coroutine and returning into older continuation-style code very simple. Note that this is safe because the nature of the `SemiFuture` concept guarantees that coroutine's callback can only satisfy the future, not end up performing the caller's work as well.

Calls to `get()`, `get_expected()` and to `wait()` are blocking and support forward progress delegation as discussed in **P0904**. When a Future is wrapping an Awaitable, the effect of this is that awaitable types delegate forward progress to the caller, as arbitrary functions do, and this continues to be true through an underlying call to `co_await`. Note that this means in particular that `.get()` on such a future is free of synchronization because the coroutine is always running in the caller's context, and there is no need for a stored value or shared state between coroutine and future.

As well as being constructible from values, `StandardSemiFuture` is also constructible from anything that satisfies the `SemiAwaitable` concept, including any other `SemiFuture` types, and `AsyncAwaitable`s. Finally, it may be constructed from any type that satisfies the Awaitable concept, and may wrap `.get()` and `.wait()` efficiently internally with a `sync_await` operation on the awaitable, which implicitly delegates forward progress to the caller of `.get()` as `co_await` itself does. This is a generalisation of the support discussed in **P0904** and makes `StandardSemiFuture` a powerful construct for interfacing between coroutine-based and legacy continuation-based code.

```
template<class T>
class StandardSemiFuture {
    public:
    StandardSemiFuture(T);

    // This allows us to take a SemiAwaitable and defer conversion
    // to an Awaitable until we convert the SemiFuture with via, or do the
    // equivalent of sync_await in .get().
    template<SemiAwaitable AW>
    StandardSemiFuture(AW&&);
```

```
        // This should be safe as the callback will only complete the future,
        // not allow chaining.
        template<Awaitable AW>
        StandardSemiFuture(AW&&);

        // Explicit overload for matched ContinuableFuture as a shared
        // implementation will be more efficient than co_await.
        StandardSemiFuture(StandardContinuableFuture<T>&&);

        template<Callable F, class ReturnT>
        StandardSemiFuture<ReturnT> defer(F&&);

        // get and get_expected are both destructive.
        // get will throw on exception. get_expected will return either a value
        // or an exception.
        T get() &&;
        expected<T, exception_ptr> get_expected() noexcept &&;

        // Wait is not destructive.
        SemiFuture<T>& wait() noexcept &;
        SemiFuture<T>&& wait() noexcept &&;

        bool is_ready() noexcept;
};
```

The standard version of `ContinuableFuture` is similarly extended with construction from `AsyncAwaitable` instances carrying the right executor type.

```
template<class T, Executor Ex>
class StandardContinuableFuture {
        public:
        using executor_type = Ex;
        using semi_future_type = StandardSemiFuture<T>;

        // Move constructor
        StandardContinuableFuture(StandardContinuableFuture&&);

        template<AsyncAwaitable<Ex> AW>
        StandardContinuableFuture (AW&&);

        template<Callable F>
        StandardContinuableFuture <invoke_result_t<F, Args...>, Ex> then(F&& f);

        template<Callable F>
        StandardContinuableFuture <invoke_result_t<F>, Ex> bulk_then(F&&);
```

```
        // get and get_expected are both destructive.
        // get will throw on exception. get_expected will return either a value
        // or an exception.
        T get() &&;
        expected<T, exception_ptr> get_expected() noexcept &&;

        // Wait is not destructive.
        StandardContinuableFuture<T, Ex>& wait() noexcept &;
        StandardContinuableFuture<T, Ex>&& wait() noexcept &&;

        bool is_ready() noexcept;

        Ex get_executor() noexcept;

        semi_future_type semi() &&;
};
```

The ability to construct from an `AsyncAwaitable` means that a `StandardContinuableFuture` is constructible from any other future type that implements the `ContinuableFuture` concept and shares the same executor.

Synchronization

A `SemiFuture`, constructed off of an `Awaitable` need not have separate shared state at all, it just acts as type erasure for a `sync_await` operation - there is no data race here if the Awaitable executes lazily and so limited synchronization is necessary unless that Awaitable works asynchronously anyway. Even when we convert to a `ContinuableFuture` using `via`, we only need to store the executor and can defer use of that to an asynchronous coroutine wait operation. Any cost is then minimally what the coroutine implementation would allow. The only required synchronization should be in the execute method of the executor - the Awaitable callback will trigger a call to execute, and synchronization is as heavy or as light as the Executor supports.

A future/promise pair will need some sort of synchronization in the core between setting the value and retrieving it. This synchronization need not affect the implementation of futures constructed from awaitables.