Document No.	P1171R0
Date	2018-10-07
Reply To	Lewis Baker <lbaker@fb.com>
Audience	SG1, LEWG

Synchronously waiting on asynchronous operations

Overview

The paper P1056R0 introduces a new `std::experimental::task<T>` type. This type represents an asynchronous operation that requires applying operator co_await to the task retrieve the result. The `task<T>` type is an instance of the more general concept of Awaitable types.

The limitation of Awaitable types is that they can only be co_awaited from within a coroutine. For example, from the body of another task<T> coroutine. However, then you still end up with another Awaitable type that must be co_awaited within another coroutine.

This presents a problem of how to start executing the first task and wait for it to complete.

```
task<int> f();

task<int> g() {
  // Call to f() returns a not-yet-started task.
  // Applying operator co_await() to the task starts its execution
  // and suspends the current coroutine until it completes.
  int a = co_await f();
  co_return a + 1;
}

int main() {
  task<int> t = g();

  // But how do we start executing g() and waiting for it to complete
  // when outside of a coroutine context, such as in main()?
  int x = ???;
```

```
  return x;
}
```

This paper proposes a new function, `sync_wait()`, that will allow a caller to pass an arbitrary Awaitable type into the function. The function will `co_await` the passed Awaitable object on the current thread and then block waiting for that `co_await` operation to complete; either synchronously on the current thread, or asynchronously on another thread. When the operation completes, the result is captured on whichever thread the operation completed on. Then, if necessary, the waiting thread is woken up and it moves the captured result into the `sync_wait()` return value.

With the `sync_wait()` function the `main()` function in the above example becomes:

```
int main() {
  // Call sync_wait() to start executing the task and wait until it completes.
  int x = sync_wait(g());
  return x;
}
```

API Synopsis

You can think of the interface as basically the following, with the added ability to customise the behaviour by providing overloads of `sync_wait()` or `sync_wait_r()` for user-defined types that are found using ADL. Technically, the interface is implemented as customization-point objects that overload `operator()`.

```
namespace std
{
  // See paper P1288R0 for definition of Awaitable, AwaitableOf concepts and
  // await_result_t trait.
  template<typename T> concept Awaitable = ...;
  template<typename T, typename Result> concept AwaitableOf = ...;
  template<typename Awaitable> using await_result_t = ...;
}

namespace std::this_thread
{
  // Helper that decays rvalue types to unqualified prvalues
  template<typename T>
  using __rvalue_decay_t =
    conditional_t<is_lvalue_reference_v<T>, T, remove_cvref_t<T>>;

  template<Awaitable A>
    requires MoveConstructible<__rvalue_decay_t<await_result_t<A>>>
```

```
  auto sync_wait(A&& awaitable) -> __rvalue_decay_t<await_result_t<A>>;

  template<typename Result, AwaitableOf<Result> A>
  Result sync_wait_r(A&& awaitable);
}
```

Note that the `sync_wait()` function places the following requirements on `Awaitable` type:

- It must satisfy the Awaitable concept (see P1288R0 for details)
- The `await_result_t<Awaitable>` type must be move-constructible.
- If `await_result_t<Awaitable>` is not an lvalue-reference, then `remove_cvref_t<await_result_t<Awaitable>>` must be implicitly constructible from a value of type `await_result_t<Awaitable>`. This typically means the result must be move-constructible, but may require copy-constructible in the case where the result of the co_await expression is a const-qualified rvalue.

The `sync_wait_r<R>()` function operates similarly to `sync_wait()` except that it allows the caller to override the deduced return-type and instead implicitly casts the result of the `co_await` expression to type `R`. See the section on handling co_await expressions that return r-values below for more context.

If the expression `co_await static_cast<Awaitable&&>(awaitable)` completes with an exception then the exception is caught and rethrown to the caller of `sync_wait()`.

Bikeshedding

Some potential names (and namespaces) for this new function:

- `std::this_thread::wait()`
- `std::this_thread::wait_synchronously()`
- `std::this_thread::get()`
- `std::this_thread::sync_wait()`
- `std::this_thread::sync_get()`
- `std::this_thread::sync_await()`
- `std::this_thread::blocking_wait()`
- `std::this_thread::blocking_get()`
- `std::this_thread::await()`
- `std::this_thread::await_synchronously()`

For the rest of this document I will assume this function is called `sync_wait()`, however feel free to mentally replace this name with any of the alternatives.

If this paper is merged into the Coroutines TS first before merging into the DS then these functions could alternatively be placed inside `std::experimental::this_thread` namespace.

Design Discussion

Handling `co_await` expressions that return rvalue references

Q. Should an awaitable that returns an rvalue reference from `await_resume()` return an rvalue reference from `sync_wait()` or should an rvalue-reference result be decayed to a prvalue result?

It's possible that a given awaitable type could return an rvalue-reference to an object that is stored inside the temporary awaiter object returned from operator co_await(). This temporary object will be placed on the temporary coroutine frame created inside sync_wait() for awaiting the passed awaitable. This temporary coroutine frame will be destroyed before sync_wait() returns and so for this operation to be safe, we need to ensure the result is moved from the returned rvalue-reference into a new object before returning from sync_wait().

For example, a simple async operation that stores its result in the temporary awaiter object:

```
struct big_object
{
  big_object() noexcept;
  big_object(big_object&& other) noexcept; // Expensive.
  ...
};

struct my_operation
{
  class awaiter
  {
    std::optional<big_object> result;
  public:
    bool await_ready();
    void await_suspend(std::experimental::coroutine_handle<> h);
    big_object&& await_resume() { return std::move(*result); }
  };

  awaiter operator co_await();
};
```

Such an awaitable operation would be used as follows:

```
void consume_big_object(big_object&& o);

task<void> usage()
{
```

```
  my_operation op;
  consume_big_object(co_await op);
}

// The above code is equivalent to the following which shows more
// clearly where the storage is placed. Note that big_object's
// move-constructor is not called during this chain.
task<void> usage_expanded()
{
  my_operation op;
  {
    my_operation::awaiter awaiter = op.operator co_await();
    big_object&& result = co_await awaiter;
    consume_big_object(std::move(result));
  }
}
```

If we were to pass an object of type `my_operation` into `sync_wait()` then the temporary awaiter object created on the temporary coroutine frame would be destroyed before `sync_wait()` returns. Thus if we were to return an rvalue reference to this object from `sync_wait()` then we would be returning a dangling reference.

Note that an awaitable type could also do something similar and then return an lvalue reference to an object stored within the temporary awaiter object and the proposed API would not promote that to a prvalue and thus would return a dangling reference. However, returning an lvalue reference to a value stored within a temporary object seems like a bad API design and so perhaps we can just discourage that pattern.

It is still valid to sometimes return an lvalue-reference to some non-temporary value, however, so we still want to avoid promoting lvalue references to prvalues.

For example: An awaitable that returns an lvalue reference

```
struct record_cache
{
public:
  record_cache() = default;

  // This method returns a task that yields an lvalue reference to an
  // entry in the map when co_awaited.
  task<const record&> get(std::string s)
  {
    auto lock = co_await mutex.scoped_lock();
    auto iter = cache.find(s);
    if (iter == cache.end())
```

```
    {
      record r = co_await load(s);
      iter = cache.emplace(std::move(s), std::move(r)).first;
    }
    co_return iter->second;
  }

private:
  task<record> load(std::string s);

  cppcoro::async_mutex mutex;
  std::unordered_map<std::string, record> cache;
};
```

Note that there may be valid cases where the awaitable returns an rvalue reference to an object that is not stored in the temporary awaiter object and so therefore would be safe to pass through as the return-value of `sync_wait()`. eg. if the object was stored in the awaitable object itself rather than the awaiter object returned by `operator co_await`.

We could potentially provide a variant of `sync_wait()` that let the user explicitly specify the return-type of `sync_wait()` via a template argument. This variant would be valid iff the `await_result_t<Awaitable>` was implicitly convertible to the specified return-type. This would allow the user to override the default xvalue → prvalue promotion.

```
namespace std::experimental::this_thread
{
  template<typename _Result, typename _Awaitable>
  _Result sync_wait_r(_Awaitable&& __awaitable)
  {
    return __make_sync_wait_task(
      static_cast<_Awaitable&&>(__awaitable)).__get();
  }
}
```

Usage of this `sync_wait_r()` function would look like this:

```
task<big_object&&> get_big_object();

void consume_big_object(big_object&& x);

void usage()
{
  task<big_object&&> t = get_big_object();
```

```
  // No extra call to big_object move-constructor here.
  consume_big_object(sync_wait_r<big_object&&>(std::move(t)));
}
```

Allowing customisation of blocking behaviour by current Executor

Q. Should the `sync_wait()` function provide some hook for an executor to customise the blocking behaviour?

One of the risks of providing a blocking API for waiting on asynchronous operations is that of introducing deadlock if the blocking call is made from an execution context that is required to execute other work in order to make forward progress on the task being waited-on.

For example, performing a blocking-wait on an operation that needs to execute on a UI thread to be able to complete will deadlock if that blocking-wait call is made from the UI thread.

This deadlocking could be avoided if we allow the blocking-wait call to re-enter the event loop from within the `sync_wait()` call and continue to process events (aka “boost block”) until the operation we are waiting for completes. However, doing so is not straight-forward because the current thread’s executor is not a parameter to the `sync_wait()` call. This means that the current executor would need to install, in a thread-local variable, some type-erased callback that the `sync_wait()` implementation could call to delegate the blocking behaviour to the current thread’s executor.

While this customisation point could be added at a later date, I would like to defer introducing this extra level of complexity until its need is proven. The preferred direction for now is to keep it simple and just say that the operation blocks the current thread (as implied by its location in namespace `std::this_thread`).

Generally, we should try to discourage calls to synchronously wait from within execution contexts that are owned by an executor and instead provide tools for letting the application write as much of the code using the asynchronous paradigm as possible. For example, using coroutines.

Ideally there would only be a handful of calls to `sync_wait()` in an application, typically from top-level calls like `main()` or from unit-tests.

Allowing customisation of blocking behaviour by Awaitable type

Q. Should the `sync_wait()` function provide a customisation point to allow different implementations of blocking for particular Awaitable types?

There may be certain types of Awaitable objects that can provide a more efficient blocking-wait operation than by executing `operator co_await()` on the awaitable and synchronising with a `mutex`/`condition_variable` or `binary_semaphore`.

For example, a hypothetical `cuda_task` awaitable that wrapped a given CUDA operation on a GPU can be more efficiently waited-on by calling the CUDA library's `cudaEventSynchronize()` function instead of attaching a callback to be executed on the CUDA device thread that would then signal a `condition_variable`.

To support customising the behaviour of `sync_wait()` for various awaitable types, the `sync_wait()` function needs to be made a customisation point that allows different awaitable types to overload the behaviour of `sync_wait()`. If an overload of `sync_wait()` is found via argument-dependent lookup then this overload should be called instead of the default `sync_wait()`.

See the appendix for an example implementation of a `cuda_task` that customises `sync_wait()`.

Supporting blocking-wait on other kinds of async operations

Q. Should the `sync_wait()` function provide overloads for waiting on other kinds of async operations?

Do we want this to be a uniform async→sync blocking interface?

For example:

- `T sync_wait(std::future<T>&& f) { return std::move(f).get(); }`
- Senders from the upcoming revised Executors proposal.
  See Code Samples appendix for an example implementation.

The decision about whether we want this function to be a universal blocking-wait on asynchronous operations other than coroutine awaitables will likely influence the name of this function. If so, we may want to avoid names that are coroutine-specific.

Free function vs Member function

Q. Why should this function be a free-function and not a method on `task<T>`?

Making it a free-function allows the implementation to work on arbitrary awaitable types and not be limited to use with `task<T>`. We expect many different awaitable types to be written and `sync_wait()` should be able to work with all of them.

Timed-wait variants of `sync_wait()`

Q. The other waiting functions in std::this_thread provide overloads that take either a time_point or a duration to allow time-limited blocking wait operations. Should we also provide a timed-wait operation for sync_wait()?

It is actually unsafe in general to return early from the `sync_wait()` call if the `co_await` expression has not yet completed. The temporary coroutine that has been created to co_await the passed awaitable still holds a reference to the awaitable object. If we were to return-early from the `sync_wait()` function then calling thread may go on to run the destructor of the awaitable object while the coroutine is still executing, leaving the coroutine with a dangling reference.

We could partly work around this issue by requiring the coroutine to take a copy of the awaitable object that it stored locally within the coroutine frame. However, the awaitable object may in turn reference other objects that are owned by the caller which could also be at risk of being destroyed by the caller before the async operation completes.

If we were to return from `sync_wait()` early, we would need to return some representation of the operation that allowed the caller to subsequently wait for the operation to complete. e.g. a `std::future<T>`-like thing. This object would then also need to provide some kind of timed-wait operation. At this point it seems like it would be better to just simply wrap the awaitable in a `std::future<T>` coroutine and use the existing `std::future<T>::wait_for()` and `std::future<T>::wait_until()` functions to perform the timed-wait.

For example: If we add the following to allow coroutines to return `std::future<T>`

```
namespace std::experimental
{
  template<typename T>
  struct _future_promise
  {
    std::promise<T> promise;
    std::future<T> get_return_object() { return promise.get_future(); }
    void unhandled_exception() {
      promise.set_exception(std::current_exception()); }
    void return_value(T&& value) {
      promise.set_value(static_cast<T&&>(value)); }
    void return_value(const T& value) { promise.set_value(value); }
    std::experimental::suspend_never initial_suspend() { return {}; }
    std::experimental::suspend_never final_suspend() { return {}; }
  };

  // Specialisations for T& and void omitted for brevity.

  template<typename T, typename... Args>
  struct coroutine_traits<std::future<T>, Args...>
  {
    using promise_type = _future_promise<T>;
  };
```

```
  template<typename Awaitable>
  auto make_future_from_awaitable(Awaitable awaitable)
    -> std::future<await_result_t<T>>
  {
    co_return co_await static_cast<Awaitable&&>(awaitable);
  }
}
```

Then we could enable timed wait of an awaitable by wrapping the awaitable in a `std::future<T>` coroutine:

```
void example_usage()
{
  using namespace std;
  using namespace std::experimental;
  using namespace std::literals::chrono_literals;

  task<int> t = some_async_function();
  future<int> f = make_future_from_awaitable(std::move(t));
  if (f.wait_for(500ms) == future_status::timeout)
  {
    // ... do something else for a while

    // Later, do another timed wait
    f.wait_for(500ms);
  }

  // Or eventually a blocking wait.
  int result = f.get();
}
```

The use of a type like `std::future` (or any eagerly-started async operation) can be error-prone, however, since it is possible to exit the scope without waiting for operations to complete. The caller needs to be extra careful to make sure that they wait for the operation on all code-paths before the scope exits.

The `std::future` approach mentioned above allows you to perform a blocking wait and exit early from the blocking wait while still letting the operation continue to run in the background. Another use-case for a timed wait is to cancel the operation after a certain period of time, e.g. because you don't want the result any more once a timeout has elapsed. This can be implemented using `sync_wait()` with a combination of `when_all()`, `cancellation_token` and an asynchronous `sleep_for()` operation (implementations of which can be found in cppcoro).

For example: The following code shows how you can use `when_all()` to start two concurrent operations, one that sleeps for a specified duration of time and another that performs the actual operation. When either of the

tasks completes it requests cancellation of the other and then the `when_all()` waits until both operations complete.

```
task<int> cancellable_work(cancellation_token ct);

template<typename Awaitable>
task<await_result_t<Awaitable>> cancel_on_completion(
  cancellation_source cs, Awaitable a)
{
  scope_guard cancelOnExit = [&]{ cs.request_cancellation(); };
  co_return co_await std::move(a);
}

int main()
{
  static_thread_pool tp;
  cancellation_source cs;
  try {
    auto [_, result] = sync_wait(when_all(
      cancel_on_completion(cs, tp.sleep_for(500ms, cs.get_token())),
      cancel_on_completion(cs, cancellable_work(cs.get_token()))));
    return result;
  } catch (operation_cancelled&) {
    return -1;
  }
}
```

Using `sync_wait()` in combination with event-loops

The `sync_wait()` function can be used in conjunction with `when_all()` to allow starting an async operation and then enter an event loop without needing to introduce an eager coroutine-type like `std::future<T>`.

```
// Assuming this task calls io.stop() when it's done.
task<void> run_service(io_context& io);

int main()
{
  io_context io;
  this_thread::sync_wait(when_all(
    run_service(io),
    [&]() -> task<void> { io.run(); co_return; }()));

  return 0;
```

```
}
```

Semantics / Wording

Modify section **33.3.1 Header <thread> synopsis**

```
namespace std {
  class thread;
  void swap(thread& x, thread& y) noexcept;
  namespace this_thread {
    thread::id get_id() noexcept;
    void yield() noexcept;
    template <class Clock, class Duration>
      void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      void sleep_for(const chrono::duration<Rep, Period>& rel_time);

    inline namespace unspecified {
      inline constexpr unspecified sync_wait = unspecified;
      template<typename Result>
      inline constexpr unspecified sync_wait_r = unspecified;
    }
  }
}
```

Modify section **33.3.3 Namespace `this_thread`**

```
namespace std::this_thread {
  thread::id get_id() noexcept;
  void yield() noexcept;
  template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
  template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);

  inline namespace unspecified {
    inline constexpr unspecified sync_wait = unspecified;
    template<typename Result>
    inline constexpr unspecified sync_wait_r = unspecified;
  }
}
```

```
thread::id this_thread::get_id() noexcept;
```

Returns: An object of type `thread::id` that uniquely identifies the current thread of execution. No other thread of execution shall have this id and this thread of execution shall always have this id. The object returned shall not compare equal to a default constructed `thread::id`.

```
void this_thread::yield() noexcept;
```
Effects: Offers the implementation the opportunity to reschedule.
Synchronization: None.

```
template <class Clock, class Duration>
  void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
```
Effects: Blocks the calling thread for the absolute timeout (33.2.4) specified by abs_time.
Synchronization: None.
Throws: Timeout-related exceptions (33.2.4).

```
template <class Rep, class Period>
  void sleep_for(const chrono::duration<Rep, Period>& rel_time);
```
Effects: Blocks the calling thread for the relative timeout (33.2.4) specified by rel_time.
Synchronization: None.
Throws: Timeout-related exceptions (33.2.4).

`std::this_thread::sync_wait`
The name `std::this_thread::sync_wait` denotes a customization point object ([customization.point.object]). The expression `std::this_thread::sync_wait(E)` for some subexpression `E` is expression-equivalent to:

- `sync_wait(E)`, if that expression is valid, with overload resolution performed in a context that does not include a declaration of `std::this_thread::sync_wait`.
- Otherwise, if the expression `co_await E` is valid inside a coroutine with a promise type that does not define an `await_transform` member, then evaluates `co_await E` on the current thread inside a new coroutine function invocation. If the coroutine suspends without running to completion then the current thread blocks until the operation completes on another thread.
  Returns: If the result of the expression `co_await E` is an lvalue reference then the lvalue reference is returned from `sync_wait()`. Otherwise, if the result of the `co_await` expression is non-void then returns a new unqualified prvalue of the same type as the `co_await E` expression that is move-constructed from the value returned by the `co_await` expression. In this case, the move-constructor is executed on the thread that called `sync_wait()`. Otherwise, if the result of the `co_await` expression has type `void` then `sync_wait()` returns `void`.
  Synchronization: Synchronizes with the completion of the `co_await E` expression. Operations that are sequenced after the `sync_wait()` call *happens-after* the completion of the expression `co_await E`.
  Throws: Rethrows any exception thrown by the expression `co_await E`. If any internal synchronization operations fail with an exception then the `sync_wait()` function is unable to fulfill its requirements and `std::terminate()` is called.
- Otherwise, `std::this_thread::sync_wait(E)` is ill-formed.

`std::this_thread::sync_wait_r<R>`
The name `std::this_thread::sync_wait_r<R>` denotes a customization point object ([customization.point.object]). The expression `std::this_thread::sync_wait_r<R>(E)` for some type `R` and some subexpression `E` is expression-equivalent to:

- `sync_wait_r<R>(E)`, if that expression is valid, with overload resolution performed in a context that does not include a declaration of `std::this_thread::sync_wait_r`.
- Otherwise, if the type of the expression `co_await E` is implicitly convertible to `R`, then evaluates `co_await E` on the current thread inside a new coroutine function invocation. The cast of the result of `co_await E` to type `R` is executed on the thread that called `sync_wait()`. If the coroutine suspends without running to completion then the current thread blocks until the operation completes on another thread.
  Returns: The result of the expression `co_await E` implicitly cast to type `R`.
  Synchronization: Synchronizes with the completion of the `co_await E` expression. Operations that are sequenced after the `sync_wait()` call *happens-after* the completion of the expression `co_await E`.
  Throws: Rethrows any exception thrown by the expression `co_await E`. If any internal synchronization operations fail with an exception then the `sync_wait()` function is unable to fulfill its requirements and `std::terminate()` is called.
- Otherwise, `std::this_thread::sync_wait_r<R>(E)` is ill-formed.

[Note: [customization.point.object] refers to http://eel.is/c++draft/customization.point.object]

Appendix - Code Samples

Reference `sync_wait()` implementation

This implementation is for exposition purposes only.

Standard library vendors are free to pursue other implementations.
For example, an implementation may choose to use a futex for thread-synchronization on platforms that support it instead of using `std::mutex` and `std::condition_variable`.

You can play with the compiled code here: https://godbolt.org/z/DMU-Tn

```
 ////////////////////////////////////////////////////////////////////////////
//
 // Expositional implementation of std::experimental::this_thread::sync_wait()
 //
 // Supporting material for ISO C++ standard proposal P1171.
 //
 // See https://wg21.link/P1171
```

```
#include <experimental/coroutine>
#include <mutex>
#include <condition_variable>
#include <type_traits>
#include <exception>

namespace std::experimental
{
  // NOTE: This section duplicates type-traits and concept-checks that have
  // been proposed in P1288R0.
  template<typename _Tp>
  struct __is_coroutine_handle : false_type {};

  template<typename _Promise>
  struct __is_coroutine_handle<coroutine_handle<_Promise>> : true_type {};

  template<typename _Tp>
  struct __is_valid_await_suspend_result
  : disjunction<is_void<_Tp>,
                is_same<_Tp, bool>,
                __is_coroutine_handle<_Tp>> {};

  template<typename _Tp, typename = void>
  struct __is_awaiter : false_type {};

  template<typename _Tp>
  struct __is_awaiter<_Tp, void_t<
    decltype(std::declval<_Tp&>().await_ready()),
    decltype(std::declval<_Tp&>().await_suspend(coroutine_handle<void>{})),
    decltype(std::declval<_Tp&>().await_resume())>>
  : conjunction<
    is_same<decltype(std::declval<_Tp&>().await_ready()), bool>,
    __is_valid_await_suspend_result<
    decltype(std::declval<_Tp&>().await_suspend(coroutine_handle<void>{}))>>
  {};

  template<typename _Tp, typename = void>
  struct __has_member_operator_co_await : false_type {};

  template<typename _Tp>
  struct __has_member_operator_co_await<_Tp, void_t<
    decltype(std::declval<_Tp>().operator co_await())>>
```

```
    : true_type {};

  template<typename _Tp, typename = void>
  struct __has_free_operator_co_await : false_type {};

  template<typename _Tp>
  struct __has_free_operator_co_await<_Tp, void_t<
    decltype(operator co_await(std::declval<_Tp>()))>>
  : true_type {};

  template<typename _Awaitable>
  decltype(auto) get_awaiter(_Awaitable&& __awaitable)
  {
    if constexpr (__has_member_operator_co_await<_Awaitable>::value)
    {
      return static_cast<_Awaitable&&>(__awaitable).operator co_await();
    }
    else if constexpr (__has_free_operator_co_await<_Awaitable>::value)
    {
      return operator co_await(static_cast<_Awaitable&&>(__awaitable));
    }
    else
    {
      return static_cast<_Awaitable&&>(__awaitable);
    }
  }

  template<typename _Tp, typename = void>
  struct __awaiter_type {};

  template<typename _Tp>
  struct __awaiter_type<_Tp, void_t<
    decltype(std::experimental::get_awaiter(std::declval<_Tp>()))>>
  {
    using type =
decltype(std::experimental::get_awaiter(std::declval<_Tp>()));
  };

  template<typename _Tp>
  using __awaiter_type_t = typename __awaiter_type<_Tp>::type;

  template<typename _Tp, typename = void>
  struct __await_result {};
```

```
template<typename _Tp>
struct __await_result<_Tp, void_t<decltype(
  std::declval<add_lvalue_reference_t<__awaiter_type_t<_Tp>>>()
  .await_resume())>>
{
  using type = decltype(
    std::declval<add_lvalue_reference_t<__awaiter_type_t<_Tp>>>()
    .await_resume());
};

template<typename _Tp>
struct await_result : __await_result<_Tp> {};

template<typename _Tp>
using await_result_t = typename __await_result<_Tp>::type;

class __event
{
public:
  __event() noexcept : __isSet_(false) {}

  void __set() noexcept
  {
    scoped_lock __lock{ __mutex_ };
    __isSet_ = true;
    __cv_.notify_all();
  }

  void __wait() noexcept
  {
    unique_lock __lock{ __mutex_ };
    __cv_.wait(__lock, [this] { return __isSet_; });
  }

private:
  // NOTE: If standardised, the std::binary_semaphore from P0514R4
  // could be used here instead of mutex/condition_variable.
  mutex __mutex_;
  condition_variable __cv_;
  bool __isSet_;
};
```

```
template<typename _Tp>
class __sync_wait_task;

class __sync_wait_promise_base
{
  struct final_awaiter
  {
    bool await_ready() noexcept { return false; }

    template<typename _Promise>
    void await_suspend(coroutine_handle<_Promise> __h) noexcept
    {
      __sync_wait_promise_base& __promise = __h.promise();
      __promise.__event_.__set();
    }

    void await_resume() noexcept {}
  };

public:
  suspend_never initial_suspend() noexcept { return {}; }
  final_awaiter final_suspend() noexcept { return {}; }

  void unhandled_exception() noexcept
  {
    __exception_ = current_exception();
  }

protected:
  void __wait() noexcept { __event_.__wait(); }
  void __throw_if_exception()
  {
    if (__exception_)
      std::rethrow_exception(std::move(__exception_));
  }

  friend struct final_awaiter;
  __event __event_;
  std::exception_ptr __exception_;
};

template<typename _Tp>
struct __sync_wait_promise : __sync_wait_promise_base
```

```
{
  __sync_wait_task<_Tp> get_return_object() noexcept;

  auto yield_value(_Tp&& __value) noexcept
  {
    __value_ = std::addressof(__value);
    return this->final_suspend();
  }

  void return_void() { std::abort(); }

  _Tp&& __get()
  {
    this->__wait();
    this->__throw_if_exception();
    return static_cast<_Tp&&>(*__value_);
  }

  std::add_pointer_t<_Tp> __value_;
};

template<>
struct __sync_wait_promise<void> : __sync_wait_promise_base
{
  __sync_wait_task<void> get_return_object() noexcept;

  void return_void() noexcept {}

  void __get()
  {
    this->__wait();
    this->__throw_if_exception();
  }
};

template<typename _Tp>
struct __sync_wait_task
{
  using promise_type = __sync_wait_promise<_Tp>;
  explicit __sync_wait_task(coroutine_handle<promise_type> __coro) noexcept
  : __coro_(__coro)
  {}
```

```
    __sync_wait_task(__sync_wait_task&& __t)
    : __coro_(exchange(__t.__coro_, {}))
    {}

    ~__sync_wait_task()
    {
      if (__coro_) __coro_.destroy();
    }

    decltype(auto) __get()
    {
      return __coro_.promise().__get();
    }
  private:
    coroutine_handle<promise_type> __coro_;
  };

  template<typename _Tp>
  __sync_wait_task<_Tp> __sync_wait_promise<_Tp>::get_return_object() noexcept
  {
    return __sync_wait_task<_Tp>{
      coroutine_handle<__sync_wait_promise<_Tp>>::from_promise(*this)
    };
  }

  inline
  __sync_wait_task<void> __sync_wait_promise<void>::get_return_object()
noexcept
  {
    return __sync_wait_task<void>{
      coroutine_handle<__sync_wait_promise<void>>::from_promise(*this)
    };
  }

  template<
    typename _Awaitable,
    enable_if_t<!is_void_v<await_result_t<_Awaitable>>, int> = 0>
  auto __make_sync_wait_task(_Awaitable&& __awaitable)
    -> __sync_wait_task<await_result_t<_Awaitable>>
  {
    co_yield co_await static_cast<_Awaitable&&>(__awaitable);
  }
```

```
  template<
    typename _Awaitable,
    enable_if_t<is_void_v<await_result_t<_Awaitable>>, int> = 0>
  auto __make_sync_wait_task(_Awaitable&& __awaitable)
    -> __sync_wait_task<void>
  {
    co_await static_cast<_Awaitable&&>(__awaitable);
  }
}

namespace std::experimental::this_thread
{
  namespace __adl
  {
    template<typename _Awaitable>
    auto sync_wait(_Awaitable&& __awaitable)
      -> conditional_t<
        is_lvalue_reference_v<await_result_t<_Awaitable>>,
        await_result_t<_Awaitable>,
        remove_cv_t<remove_reference_t<await_result_t<_Awaitable>>>>
    {
      return std::experimental::__make_sync_wait_task(
        static_cast<_Awaitable&&>(__awaitable)).__get();
    }

    struct __sync_wait_fn
    {
      template<typename _Awaitable>
      auto operator()(_Awaitable&& __awaitable) const
        noexcept(noexcept(sync_wait(static_cast<_Awaitable&&>(__awaitable))))
      -> decltype(sync_wait(static_cast<_Awaitable&&>(__awaitable)))
      {
        return sync_wait(static_cast<_Awaitable&&>(__awaitable));
      }
    };

    // Optional overload that allows sync_wait_r<ReturnType>(awaitable)
    template<typename _Result, typename _Awaitable>
    enable_if_t<is_convertible_v<await_result_t<_Awaitable>, _Result>,
_Result>
    sync_wait_r(_Awaitable&& __awaitable)
    {
      return std::experimental::__make_sync_wait_task(
```

```
        static_cast<_Awaitable&&>(__awaitable)).__get();
    }

    template<typename _Result>
    struct __sync_wait_r_fn
    {
      template<typename _Awaitable>
      auto operator()(_Awaitable&& __awaitable) const
        noexcept(noexcept(sync_wait_r<_Result>(
          static_cast<_Awaitable&&>(__awaitable))))
        -> decltype(sync_wait_r<_Result>(
          static_cast<_Awaitable&&>(__awaitable)))
      {
        return sync_wait_r<_Result>(static_cast<_Awaitable&&>(__awaitable));
      }
    };
  }

  inline namespace __v1 {
    inline constexpr __adl::__sync_wait_fn sync_wait;

    template<typename _Result>
    inline constexpr __adl::__sync_wait_r_fn<_Result> sync_wait_r;
  }
}
```

Some example usage code:

```
////////////////////////////////////////////////////////////
// Example usage

template<typename _Tp>
struct ready_awaitable
{
  _Tp __value_;
  ready_awaitable(_Tp&& __value) : __value_(std::forward<_Tp>(__value)) {}
  bool await_ready() { return true; }
  void await_suspend(std::experimental::coroutine_handle<>) {}
  _Tp await_resume() { return std::forward<_Tp>(__value_); }
};

template<>
struct ready_awaitable<void>
```

```
{
  bool await_ready() noexcept { return true; }
  void await_suspend(std::experimental::coroutine_handle<>) noexcept {}
  void await_resume() noexcept {}
};

struct move_only
{
  move_only() {}
  move_only(move_only&&) {}
  move_only(const move_only&) = delete;
};

struct unmovable
{
  unmovable();
  unmovable(unmovable&&) = delete;
  unmovable(const unmovable&) = delete;
};

void test()
{
  using std::experimental::this_thread::sync_wait;

  decltype(auto) x = sync_wait(ready_awaitable{ 123 });
  static_assert(std::is_same_v<decltype(x), int>);

  int value = 123;
  decltype(auto) y = sync_wait(ready_awaitable<int&>{value});
  static_assert(std::is_same_v<decltype(y), int&>);

  move_only mo;
  decltype(auto) z = sync_wait(ready_awaitable<move_only&&>{std::move(mo)});
  static_assert(std::is_same_v<decltype(z), move_only>);

  sync_wait(ready_awaitable<void>{});

  using std::experimental::this_thread::sync_wait_r;

  unmovable um;
  decltype(auto) w = sync_wait_r<unmovable&&>(
    ready_awaitable<unmovable&&>(std::move(um)));
  static_assert(std::is_same_v<decltype(w), unmovable&&>);
```

```
}
```

Executor Sender/Receiver customisation

This is an example implementation of `sync_wait()` for the Sender concept from pushmi (an exploratory prototype design for generalised Executors/Futures).

An alternative implementation could be to define a free-function `operator co_await()` for all `Sender` types that makes `Sender` types `Awaitable`.

```
namespace std::experimental::this_thread::__adl
{
  namespace __detail
  {
    template<typename _T>
    class __blocking_receiver
    {
    public:
      __blocking_receiver() noexcept : __done_(false) {}

      template<typename _U>
      void value(_U&& __val)
      {
        __value_.emplace(static_cast<_U&&>(__val));
      }

      void error(exception_ptr __e) noexcept
      {
        __error_ = std::move(__e);
        done();
      }

      void done() noexcept
      {
        scoped_lock __lock{ __mutex_ };
        __done_ = true;
        __cv_.notify_all();
      }

      optional<_T> __get() &&
      {
```

```
        __wait();
        if (__error_) std::rethrow_exception(__error_);
        return std::move(__value_);
      }

    private:
      void __wait() noexcept
      {
        unique_lock __lock{ __mutex_ };
        __cv_.wait(__lock, [this] { return __done_; });
      }

      optional<_T> __value_;
      exception_ptr __error_;
      mutex __mutex_;
      condition_variable __cv_;
      bool __done_;
    };
  }

  template<typename _Sender>
  auto sync_wait(_Sender&& __sender) -> optional<sender_value_t<_Sender>>
  {
    __detail::__blocking_receiver<sender_value_t<_Sender>> __receiver;
    static_cast<_Sender&&>(__sender).submit(std::ref(__receiver));
    return std::move(__receiver).__get();
  }
}
```

CUDA task sync_wait() customisation

An example implementation of a hypothetical CUDA awaitable type, `cuda_task`, and specialisation of `sync_wait()` for that awaitable type.

```
namespace cuda
{
  class cuda_task
  {
    cudaStream_t stream_;
    cudaEvent_t event_;
    cudaError_t status_;
```

```
    std::experimental::coroutine_handle<> continuation_;

  public:
    explicit cuda_task(cudaStream_t stream, cudaEvent_t event) noexcept
    : stream_(stream), event_(event) {}

    ~cuda_task()
    {
      cudaEventDestroy(event_);
      cudaStreamDestroy(stream_);
    }

    bool await_ready() noexcept
    {
      status_ = cudaEventQuery(event_);
      return status_ != cudaErrorNotReady;
    }

    bool await_suspend(std::experimental::coroutine_handle<> continuation)
    {
      continuation_ = continuation;
      cudaError_t result = cudaStreamAddCallback(
        stream_,
        &cuda_task::cuda_callback,
        static_cast<void*>(this),
        0);
      if (result != cudaSuccess) {
        status_ = result;
        return false;
      }
      return true;
    }

    cudaError_t await_resume() noexcept
    {
      return status_;
    }

    // Customisation of sync_wait() for cuda_task
    friend cudaError_t sync_wait(cuda_task&& task) noexcept
    {
      return cudaEventSynchronize(task.event_);
    }
```

```
  private:

    static void CUDART_CB cuda_callback(
      cudaStream_t stream, cudaError_t status, void* userData) noexcept
    {
      cuda_task* t = static_cast<cuda_task*>(userData);

      t->status_ = status;

      // TODO: Schedule coroutine resumption on another thread
      t->continuation_.resume();
    }
  };
}
```

Example usage:

```
// Some task factory that returns a cuda_task
cuda::cuda_task parallel_sort(double values[], size_t count);

std::vector<double> get_values();

std::task<void> async_example()
{
  auto values = get_values();

  // Compiles down to call to cudaStreamAddCallback() and schedules
  // continuation to run in callback when the task completes.
  cudaError_t status = co_await parallel_sort(values.data(), values.size());
}

void sync_example()
{
  auto values = get_values();

  // Compiles down to call to cudaEventSynchronize()
  cudaError_t status = std::this_thread::sync_wait(
    parallel_sort(values.data(), values.size()));
}
```