Document Number: Dxxxx
Date: 2019-02-24
Revises: P0876R5
Reply to: Oliver Kowalke
oliver.kowalke@gmail.com
Nat Goodspeed
nat@lindenlab.com

Fiber TS

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 Scope [scope]

1 This document describes extensions to the C++ Programming Language (Clause **??**) that introduce fibers. <additional description>

2 ISO/IEC 14882 provides important context and specification for this document. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from ISO/IEC 14882 use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

2 Normative references [refs]

1

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2017, *Programming Languages – C++*

ISO/IEC 14882:2017 is hereafter called the *C++ Standard*. The numbering of clauses, subclauses, and paragraphs in this document reflects the numbering in the C++ Standard. References to clauses and subclauses not appearing in this document refer to the original, unmodified text in the C++ Standard.

3 Terms and definitions [defs]

No terms and definitions are listed in this document.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at `https://www.iso.org/obp`

— IEC Electropedia: available at `http://www.electropedia.org`

3.1.1 **[defs.sync.op]**
synchronous operation
operation where control is not returned until the operation completes

3.1.2 **[defs.async.op]**
asynchronous operation
operation where control is returned immediately without waiting for the operation to complete

[*Note:* Multiple asynchronous operations may be executed concurrently. — *end Note*]

4 General [general]

4.1 Implementation compliance [general.compliance]

1 Conformance requirements for this document are those defined in ISO 14882:2017, 4.1 except that references to the C++ Standard therein shall be taken as referring to the document that is the result of applying the editing instructions. Similarly, all references to the C++ Standard in the resulting document shall be taken as referring to the resulting document itself. [*Note:* Conformance is defined in terms of the behavior of programs. — *end Note*]

4.2 Acknowledgments [general.ack]

1 This document is based, in part, on the design and implementation described in the paper P0876R5 "*fiber_-context - fibers without scheduler*" authored by Oliver Kowalke and Nat Goodspeed as well as the boost.context library written by Oliver Kowalke.

5 API [fiber]

5.1 Header `<experimental/fiber>` synopsis [fiber.synop]

```
namespace std {
namespace experimental {
namespace fiber {
inline namespace v1 {

class fiber_context;

class unwind_exception;

void unwind_fiber(fiber_context&& other);

} // namespace v1
} // namespace fiber
} // namespace experimental
} // namespace std
```

5.2 Class `fiber_context` [fiber.ctx]

1 Class `fiber_context` represents a lightweight thread of execution.

```
namespace std {
namespace experimental {
namespace fiber {
inline namespace v1 {

class fiber_context {
public:
    fiber_context() noexcept;

    template<typename Fn>
    explicit fiber_context(Fn&& fn);

    ~fiber_context();

    fiber_context(fiber_context&& other) noexcept;
    fiber_context& operator=(fiber_context&& other) noexcept;
    fiber_context(const fiber_context& other) noexcept = delete;
    fiber_context& operator=(const fiber_context& other) noexcept = delete;

    fiber_context resume() &&;
    template<typename Fn>
    fiber_context resume_with(Fn&& fn) &&;
    fiber_context resume_from_any_thread() &&;
    template<typename Fn>
    fiber_context resume_from_any_thread_with(Fn&& fn) &&;

    bool can_resume() noexcept;
    bool can_resume_from_any_thread() noexcept;
    explicit operator bool() const noexcept;
    void swap(fiber_context& other) noexcept;
};

} // namespace v1
} // namespace fiber
} // namespace experimental
} // namespace std
```

5.2.1 `fiber_context` constructors [fiber.ctx.cons]

`fiber_context() noexcept`

1 *Effects:* Constructs an object of class `fiber_context` which is an invalid `std::fiber_context`.
Postconditions: `!*this`

```
template<typename Fn>
explicit fiber_context(Fn&& fn)
```

2 The constructor takes an invocable (function, lambda, object with `operator()()`) as argument. The invocable must have signature as described in **??**.
This constructor template shall not participate in overload resolution unless `Fn` is *LvalueCallable* (23.14.13.2) for the argument type `std::fiber_context&&` and the return type `std::fiber_context`.
Remark: The entry-function `fn` is *not* immediately entered. The stack and any other necessary resources are created on construction, but `fn` is not entered until `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()` is called.
Remark: The entry-function `fn` passed to `std::fiber_context` will be passed a synthesized `std::fiber_context` instance representing the suspended caller of `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()`.

`fiber_context(fiber_context&& other) noexcept`

3 moves underlying state to new `std::fiber_context`
Postconditions: `!*this` if `!other` before move; otherwise `*this` and `!other`

`fiber_context(const fiber_context& other)=delete`

4 copy constructor deleted

5.2.2 `fiber_context` destructor [fiber.ctx.dtor]

`~fiber_context()`

1 Destroys a `std::fiber_context` instance. If this instance represents a fiber of execution (`*this` returns `true`), then the fiber of execution is destroyed too. Specifically, the stack is unwound by throwing `std::unwind_exception`.
Remark: In a program in which exceptions are thrown, it is prudent to code a fiber's *entry-function* with a last-ditch `catch (...)` clause: in general, exceptions must *not* leak out of the *entry-function* . However, since stack unwinding is implemented by throwing an exception, a correct *entry-function* `try` statement must also `catch (std::unwind_exception const&)` and rethrow it.

5.2.3 `fiber_context` assignment [fiber.ctx.assign]

`fiber_context& operator=(fiber_context&& other) noexcept`

1 assigns the state of `other` to `*this` using move semantics
Returns: `*this`
Postconditions: `!*this` if `!other` before move; otherwise `*this` and `!other`

`fiber_context& operator=(const fiber_context& other)=delete`

2 copy assignment operator deleted

5.2.4 `fiber_context` modifiers [fiber.ctx.modifiers]

`void swap(fiber_context& other) noexcept`

1 *Effects:* Interchanges the targets of `*this` and `other`.

5.2.5 `fiber_context` switch [fiber.ctx.switch]

`fiber_context resume() &&`

1 *Requires:* `*this` an if `can_resume_from_any_thread()` would return `false`, the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run
Effects: suspends the active fiber, resumes fiber `*this`
Returns: the returned instance represents the fiber that has been suspended in order to resume the current fiber
Postcondition: `!*this` and the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run
Throws: `std::unwind_exception` when, while suspended, the `std::fiber_context` instance representing the suspended fiber is destroyed

```
template<typename Fn>
fiber_context resume_with(Fn&& fn) &&
```

2 *Requires:* `*this)` an if `can_resume_from_any_thread()` would return `false`, the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run
Effects: Suspends the active fiber, resumes fiber `*this` but calls `fn()` in the resumed fiber (as if called by the suspended function). `fn` is a invocable injected into resumed fiber.
These member function templates shall not participate in overload resolution unless `Fn` is *LvalueCallable* (23.14.13.2) for the argument type `std::fiber_context&&` and the return type `std::fiber_context`.
Returns: the returned instance represents the fiber that has been suspended in order to resume the current fiber
Postcondition: `!*this`
Throws: `std::unwind_exception` when, while suspended, the `std::fiber_context` instance representing the suspended fiber is destroyed

`fiber_context resume_from_any_thread() &&`

3 *Requires:* `*this` an if `can_resume_from_any_thread()` would return `false`, the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run
Effects: suspends the active fiber, resumes fiber `*this`
Returns: the returned instance represents the fiber that has been suspended in order to resume the current fiber
Postcondition: `!*this` and the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run
Throws: `std::unwind_exception` when, while suspended, the `std::fiber_context` instance representing the suspended fiber is destroyed

```
template<typename Fn>
fiber_context resume_from_any_thread_with(Fn&& fn) &&
```

4 *Requires:* `*this` an if `can_resume_from_any_thread()` would return `false`, the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run

Effects: Suspends the active fiber, resumes fiber `*this` but calls `fn()` in the resumed fiber (as if called by the suspended function). `fn` is a invocable injected into resumed fiber.
These member function templates shall not participate in overload resolution unless `Fn` is *LvalueCallable* (23.14.13.2) for the argument type `std::fiber_context&&` and the return type `std::fiber_context`.
Returns: the returned instance represents the fiber that has been suspended in order to resume the current fiber
Postcondition: `!*this`
Throws: `std::unwind_exception` when, while suspended, the `std::fiber_context` instance representing the suspended fiber is destroyed

`resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()` can throw *any* exception if, while suspended:

(4.1) — some other fiber calls `resume_with()` or `resume_from_any_thread_with()` to resume this suspended fiber

(4.2) — the function `fn` passed to `resume_with()` or `resume_from_any_thread_with()`– or some function called by `fn` – throws an exception

Any exception thrown by the function `fn` passed to `resume_with()` or `resume_from_any_thread_with()`, or any function called by `fn`, is thrown in the fiber referenced by `*this` rather than in the fiber of the caller of `resume_with()` or `resume_from_any_thread_with()`.

The intent of the distinction between `resume()` and `resume_from_any_thread()`, as between `resume_with()` and `resume_from_any_thread_with()`, is both for validation and for code auditing. If an application only ever calls `resume()` and `resume_with()`, no fiber will ever be resumed on a thread other than the one on which it was initially resumed.

The intent of the names `resume_from_any_thread()` and `resume_from_any_thread_with()` is to clarify the direction in which cross-thread resumption occurs. The calling thread always directly resumes a suspended fiber: control is passed into the suspended fiber, and the currently-running fiber suspends. These method names mean that the fiber represented by `*this` will be resumed whether or not it was last resumed on the calling thread.

`resume()`, `resume_with()`, `resume_from_any_thread()` and `resume_from_any_thread_with()` preserve the execution context of the calling fiber. Those data are restored if the calling fiber is resumed.
A suspended `fiber_context` can be destroyed. Its resources will be cleaned up at that time.
The returned `fiber_context` indicates via `*this` whether the previous active fiber has terminated (returned from *entry-function*).
Because `resume()`, `resume_with()`, `resume_from_any_thread()` and `resume_from_any_thread_with()` invalidate the instance on which they are called, *no valid `std::fiber_context` instance ever represents the currently-running fiber.* In order to express the invalidation explicitly, these methods are rvalue-reference qualified.
When calling any of these methods, it is conventional to replace the newly-invalidated instance – the instance on which the method was was called – with the new instance returned by that call. This helps to avoid subsequent inadvertent attempts to resume the old, invalidated instance.
An injected function `fn()` must have signature `std::fiber_context fn(std::fiber_context&&)`. It will be passed a synthesized `std::fiber_context` instance representing the suspended caller of `resume_with()` or `resume_from_any_thread_with()`. The `std::fiber_context` instance returned by `fn()` is, in turn, used as the return value for the suspended function: `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()`.

5.2.6 `fiber_context` operations [fiber.ctx.ops]

`bool can_resume_from_any_thread() noexcept`

1 query whether the calling thread can resume the suspended `std::fiber_context` instance by calling `resume_from_any_thread()` or `resume_from_any_thread_with()`. The implementation must return `false` if the suspended `std::fiber_context` instance represents a fiber with a system-provided stack, and the calling thread is not that thread.
Returns: `false` if `!*this` or if the stack used by the fiber was provided by the operating system, and the calling thread is not that thread; otherwise `true`.
Remark: When `main()`, or the entry-function of a `std::thread`, or any function directly called by these, is suspended, a `std::fiber_context` instance represents that suspended fiber. You may resume that suspended fiber *on the same thread* using any of `resume()`, `resume_with()`, `resume_from_-any_thread()` or `resume_from_any_thread_with()`. Attempting to resume that suspended fiber from any other thread is Undefined Behavior. `can_resume_from_any_thread()` returns `true` if the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run, or if the `std::fiber_context` instance represents a fiber explicitly created by `std::fiber_-context`'s constructor. `can_resume_from_any_thread()` is not marked `const` because in at least one implementation, it requires an internal context switch.

`bool can_resume() noexcept`

2 *Returns:* `true` if the calling thread is the same as the thread on which the fiber represented by `*this` was most recently run, or if `*this` has not yet been resumed. When `can_resume()` returns `true`, the `std::fiber_context` instance may be resumed by `resume()`, `resume_with()`, `resume_from_any_-thread()` or `resume_from_any_thread_with()`.
Remark: `can_resume()` is not marked `const` because in at least one implementation, it requires an internal context switch.

5.2.7 `fiber_context` validity [fiber.ctx.validity]

`explicit operator bool() const noexcept`

1 *Returns:* `true` if `*this` represents a fiber of execution, `false` otherwise.
A `std::fiber_context` instance might not represent a valid fiber for any of a number of reasons.

(1.1) — It might have been default-constructed.

(1.2) — It might have been moved from.

(1.3) — It might already have been resumed – calling `resume()`, `resume_with()`, `resume_from_any_-thread()` or
`resume_from_any_thread_with()` invalidates the instance.

(1.4) — The *entry-function* might have voluntarily terminated the fiber by returning.

The essential points:

(1.5) — Regardless of the number of `std::fiber_context` declarations, exactly one `std::fiber_context` instance represents each suspended fiber.

(1.6) — No `std::fiber_context` instance represents the currently-running fiber.

5.3 `fiber_context` unwinding [fiber.unwind]

5.3.1 Function `unwind_fiber()` [fiber.unwind.func]

terminate the current running fiber, switching to the fiber represented by the passed `std::fiber_context`. This is like returning that `std::fiber_context` from the *entry-function* , but may be called from any function on that fiber.

`[[ noreturn ]] void unwind_fiber(fiber_context&& other)`

1 *Requires:* `*other`
Returns: does not return
Throws: `std::unwind_exception`
Throws `std::unwind_exception`, binding the passed `std::fiber_context`. The running fiber's first stack entry catches `std::unwind_exception`, extracts the bound `std::fiber_context` and terminates the current fiber by returning that `std::fiber_context`. `other` is the `std::fiber_context` to which to switch once the current fiber has terminated

5.3.2 Class `unwind_exception` [fiber.unwind.ex]

`unwind_exception` is the exception used to unwind the stack referenced by a `std::fiber_context` being destroyed. It is thrown by `std::unwind_fiber()`. `std::unwind_exception` binds a `std::fiber_context` referencing the fiber to which control should be passed once the current fiber is unwound and destroyed.

```
namespace std {
namespace experimental {
namespace fiber {
inline namespace v1 {

class unwind_exception {
};

} // namespace v1
} // namespace fiber
} // namespace experimental
} // namespace std
```