What’s in a name?

• fiber_context can be used to build coroutines (e.g. Boost.Coroutine2), userspace threads (e.g. Boost.Fiber)...
• WG21 decided years ago that “coroutine” means stackless (co_await), “fiber” means stackful
• fiber_context is the low-level context switching (term of art), leaving “fiber” for a higher-level userspace thread library
Target API level

- This paper does not propose higher-level libraries, which can be built in portable C++ once we have fiber_context
- fiber_context requires runtime implementation magic, hence important to standardize
- fiber_context API is designed for minimal overhead rather than convenience
  - e.g. avoids requiring underlying thread-locals
Why fiber_context?

- If thread concurrency was enough, would be no async I/O
- Async I/O gets us more concurrency than threads
- Code written in an async I/O environment already avoids any operation that blocks the entire thread
- Fibers let you write async code as if blocking
  - Easier to code
  - More readable and maintainable
  - Therefore more robust
Why fiber_context, given co_await?

- If any function in a library, at any level of abstraction, uses co_await, every caller must also use co_await
- Viral: changing one caller requires changing all its callers, etc.
- Many existing libraries and library algorithms accept caller-specified functors
- To use any such library with a functor that suspends using co_await, the library must be duplicated, modified and rebuilt
- fiber_context permits using existing builds of existing libraries
- More information:
  - Using Boost.Coroutine to untangle a state machine
  - Coroutines, Fibers and Threads, Oh My
  - The Fiber Library
  - Pulling Visitors
  - Elegant Asynchronous Code
Fiber

• “fiber” is a weakly parallel thread of execution
• Implemented as a new, separate function call stack
• Multiple fibers coexist within an operating-system thread
• A fiber may not migrate from one thread to another
• The thread’s OS stack can be regarded as “default fiber”
fiber_context concepts

- Running fiber suspends by calling resume() or resume_with() on some fiber_context instance
- Resuming a fiber_context empties it
  - fiber_context stores SP of suspended stack: dangerously inapplicable once resumed
- Every context switch synthesizes a new fiber_context instance representing newly-suspended fiber, passing it to newly-resumed fiber
  - On initial entry, previous fiber_context is passed into entry function
  - On resumption from suspension (return from resume() or resume_with()), previous fiber_context is returned
- To terminate the fiber, the entry function returns fiber_context of fiber to resume
Header

#include <fiber_context>
define __cpp_lib_fiber_context 202302
Launching a fiber

- template <typename F>
  fiber_context(F&& entry);
- Entry function signature fiber_context(fiber_context&&)
- Sets up new fiber’s stack
- New fiber_context, when resumed, will call entry function
- New fiber’s resources destroyed on return from entry function
fiber_context(F&& entry, span<byte, N> stack)

Constructor accepting explicit stack addresses use cases:

- control over size
- environments avoiding heap storage
- special allocation (e.g. guard page)
- consumer objects sharing same block of memory
- caller is responsible for stack cleanup on fiber exit

Using Allocator doesn’t quite fit:

- consumer of the Allocator specifies the size
- Allocator is intended to allocate multiple objects
fiber_context resume() &

- Must be same thread
- Suspends caller
- Synthesizes fiber_context instance representing caller
- Switches context to designated fiber
- Passes caller fiber_context to designated fiber:
  - First resumption: passes caller fiber_context to entry function
  - Subsequent: returns caller fiber_context from resumed fiber’s resume() or resume_with() call
fiber_context resume_with(Fn&& fn) &&

• Fn signature fiber_context(fiber_context&&)
• Same as resume(), except on switching to newly-resumed fiber:
  – Call fn(caller fiber_context)
  – Pass fiber_context returned by fn to resumed fiber, as for resume()
resume_with() rationale

- Important for communication between fibers
- Example in P0876: wrapper class that continually updates its stored fiber_context to persistently represent same fiber
bool empty() const noexcept

- Default-constructed fiber_context is empty
- Moved-from fiber_context is empty
- Previously-resumed fiber_context is empty
- Exactly one fiber_context represents each suspended fiber
- No fiber_context represents running fiber
explicit operator bool() const noexcept

- Returns (! empty())
bool can_resume() noexcept

- [SG1 request]
- false if fiber_context empty()
- false if referenced fiber previously resumed on other thread
void swap(fiber_context&) noexcept

• As expected
The Checklist

• Examples?
  – Yes, simple examples

• Field experience?
  – Implementation experience?
    • Boost.Context implements a previous revision
  – Usage experience? / Deployment experience?
    • The paper cites ten different existing libraries based on Boost.Context

• Performance considerations?
  – Paper has some timing data
  – Avoiding OS context switching is a win
The Checklist

• Discussion of prior art?
  – ucontext, Pth library

• Changes Library Evolution previously requested?
  – N/A

• Wording?
  – yes

• Breaking changes?
  – N/A

• Feature test macro?
  – yes
The Checklist

• Freestanding?
  – Possible but not sought
• Format and/or iostream support?
  – N/A: not meaningful to stream a fiber_context
• std::hash?
  – N/A: fiber_context values are transient, unsuited for container keys
Questions and Bike-Shedding