Concerns with the proposed addition of fibers to C++ 26

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3620R0

Working Group: Library, Core, Evolution

Date: 2025-02-03

David Chisnall <David.Chisnall@cl.cam.ac.uk>

Matthew Taylor <mjtaylor214@hotmail.com>

Introduction

Several languages have N:M threading abstractions. The languages that have implemented this successfully (Java, ocaml, and so on) all have one thing in common: they have a clear distinction between code in the language virtual machine and foreign code. In C++, this is not the case and, because C++ has rich support for separate compilation and shared libraries, the equivalent of foreign code may be C++ code compiled last year.

Fibres as an operating-system-level abstraction have numerous problems, which have led to most systems that attempted to deploy them abandoning them. As a language-level abstraction in a low-level language, they have even more problems.

What are fibers?

Fibres are a building block for N:M threading, allowing a set of thread-like abstractions to be multiplexed onto a single scheduled entity. In the general case, they may be cooperatively scheduled (with explicit yielding) or preemptively scheduled in response to timer events. The goal for fibers is to be faster for context switching than normal thread switching.

What is the current proposal?

The current proposal is for fibers independent of any kind of scheduler. This is intended to allow cooperative multitasking between fibers scheduled on one or more (preemptively scheduled) OS threads.

Issues with the current proposal

The current proposal has several issues that will be discussed in detail.

What is a context?

Fibres intended for userspace context switches. The root problem here is defining what a context is. In the simplest case, it's the contents of a register file, which includes a stack pointer. This is not sufficient for most operating systems. Other state may include:

- The current signal mask.
- Pending signals or asynchronous callbacks.
- In-flight asynchronous I/O
- And so on.

For example, userspace threading implementations historically used SIGALARM, where a thread registers for the kernel to deliver a signal after a certain period of time. Not switching that state in fibers is problematic in the presence of third-party code. If one function calls `alarm` and then calls a yielding function and the resumed fiber masks SIGALARM, the alarm will not be delivered. Existing software may do both of these things, yet not document that they do because it's an implementation detail. Both of these are POSIX APIs that are not specified by C++ and so ISO C++ cannot mandate their behaviour. The same applies to Win32 asynchronous callbacks.

The IEEE Std 1003.1-2004 ("POSIX.1") revision marked the functions makecontext() and swapcontext() (the ucontext APIs) as obsolete, noting that they were intended for use for safely returning from a signal handler into the signalled frame (which is not possible with setjmp / longjmp) and were not intended to be used for threading. The IEEE Std 1003.1-2008 ("POSIX.1") revision removed the functions from the specification.

Correctly implementing this would require code that is specific to both architectures and operating systems. This would make it unique in the C++ standard library. Some features are architecture-specific but operating-system agnostic, some are operating-system specific but architecture-agnostic. No existing features have this property.

Users of TLS expect it to be private

Thread-local state is expected to be modified only in the same lexical scopes. P0876 explicitly notes this in one context: exceptions. The exception implementation typically maintains a list of in-flight exceptions in thread-local storage. If a fiber switch occurs in a `catch` block, this linked list can be corrupted.

Unfortunately, this is a general problem. If a library creates any kind of linked list in thread-local storage that follows lexical scope, this will be corrupted. Note that this does not rely on a fiber migrating between threads (which would be disallowed by P0876R19), though permitting this would introduce more problems. Variations of this idiom occur in various forms. For example:

- Using a parallel stack for large returns.

- Managing in-flight transactions for software-transactional memory.
- Providing exception-like behaviour with `setjmp` for languages without exceptions.

These and more may exist in libraries called by C++. If any such library calls a C++ function that yields, it may resume with this state corrupted.

Coroutines do not have this problem because the point at which a coroutine is created and the point at which it yields are both visible. A coroutine cannot yield with library code on the stack.

Accidental deadlocks

Systems that provide a scheduler and N:M threading modify their locks to make them fiber-aware. This is essential to avoid accidental deadlocks. For example, consider the simple case of a library that implements thread safety with a global lock. Each function will look something like this:

```
std::mutex giant;

void someFunction()
{
    std::lock_guard g{giant};
    ...
}
```

This trivial case is safe, but consider this small change:

```
struct UserExtensibleType
{
    virtual void some_method() = 0;
};

std::mutex giant;

void someFunction(UserExtensibleType &object)
{
    std::lock_guard g{giant};
    ...
    object.some_method();
}
```

If the user's implementation of `some_method` yields (directly or indirectly) and the new fiber invokes another function from this library, the thread deadlocks.

This example may come from the composition of three different libraries:

- The library that implements the `UserExtensibleType`.
- The library that implements `someFunction` and other implementations.

- The library that the implementation of `UserExtensibleType::some_method` calls, which uses fibers and yields.

Layering issues

The current proposal references a patch that pivots the TLS for an Itanium ABI threading implementation such as libsupc++, libcxxrt, or libc++abi. This is necessary because the ABI library sits at a lower level in the stack than the standard library.

In existing implementations of C++, threads are a feature of the lowest level of the implementation. The low-level C++ ABI or runtime library depends on such features. The C++ standard library then exposes them to users via C++-friendly abstractions.

Fibers are unusual in taking a feature that is not universally exposed by the platform libraries (having been implemented on some platforms and removed on most of those that tried) and then implementing it in the C++ standard library.

The deadlock from the previous section can be avoided in fiber implementations that include a scheduler by modifying the locking primitives to yield if they fail to acquire a lock. This is easy to do when fibers are introduced at the lowest level of the stack, but not when they are added at the highest.

For example, the C++ ABI / runtime library for Itanium ABI implementations provides features for thread-safe static initialisation. These define the lock that is used to protect calls to the constructor for function-local `static` variables with non-trivial constructors. This cannot be made fiber-safe with the current proposal because fibers are implemented in the standard library, which sits at a higher level in the stack.

One of the follow-on proposals suggests addressing the TLS-related issues discussed earlier by requiring all TLS to be replaced with fiber-local storage, an approach taken by Windows UWP applications. This would further complicate the layering issues because, on most platforms, TLS is provided one or two layers below the C++ stack and so is out of the control of a C++ implementation. C11 introduced `_Thread_local`, which is currently equivalent to C++11's `thread_local`, but would not be if `thread_local` became fiber-local and the underling platform did not support fibers.

Summary

Fibres introduce a lot of problems for composition with existing code. They are hard to implement correctly and are not the same shape as anything else in the standard library. Rather than wrapping existing operating system features and exposing them to C++, they require that the standard library implement low-level feature that are then not shared with low-level libraries in other languages, such as C.