P0866: Response to “Fibers under the magnifying glass”

Document number:	P0866R0
Date:	2019-01-06

Reply-to:	Nat Goodspeed <nat@lindenlab.com>, Oliver Kowalke <oliver.kowalke@gmail.com>

Audience:	EWG, SG1

Abstract

Microsoft has submitted P1364R0: Fibers under the magnifying glass.[1] Unfortunately that glass is not entirely clean. Not only is that paper misleading by implication, but it contains several outright misstatements. Since P1364 questions whether fiber technology belongs in the C++ Standard at all,[1] it is important to be aware of these unfortunate errors.

TL;DR

- Do not conflate the concept of a *fiber* with the Windows Fibers implementation. Its limitations are not inherent to fibers in general.
- Most fibers do not require a megabyte of stack space.
- There is existence proof that context-switching can be more performant than the Windows Fibers implementation.
- A suite of stackless coroutines that outlives its initial invoker has memory characteristics more extreme than a fiber based on segmented stacks: *every* stackless function call requires an allocation; *every* return from such a function requires a release.
- Stackless coroutines are not immune to problems accessing `thread_local`.
- Stackless coroutines multiplexed within a `std::thread` must refrain from performing any operation that blocks the entire thread. The same is true of simple callback functions. That problem is not unique to fibers.
- Stackless coroutines *cannot* be adopted incrementally. The transitive closure of every caller of every such function must be modified.

1 P1364R0 §4

- In an application already interleaving asynchronous operations within a kernel thread, fibers can be adopted without having to weed out blocking operations or thread synchronization primitives. Such applications already avoid those.

Background

User-mode context switching has a long history.[2] For present purposes, it is useful to distinguish two kinds of use cases.

Let us say that a *coroutine* has a lifespan related to that of its invoker. For example, a function might invoke a generator to produce values lazily: each value is computed only when requested. Once the consuming function is satisfied, the generator is discarded.

Contrast this with a *user-mode cooperative thread*, which has a lifespan beyond that of its invoker. In this phrase, "user mode" means that the kernel does not mediate context switching, while "cooperative" means that context is switched explicitly rather than preemptively. This use of the term "thread" alludes to the way such a thread of execution outlives its invoker.

For example, a user gesture prompts a program to perform an operation requiring a sequence of different network requests. If the user gesture handler were to wait for each network result, the mouse cursor would stop responding until the last such request had completed and the results were displayed. Instead, the user gesture handler conceptually launches a separate thread of execution to process the required operations, and then returns to the main event-processing loop.

A more succinct term for a user-mode cooperative thread is a *fiber.*

Nature of a Fiber

Since 1996, Microsoft has offered a user-mode cooperative thread facility called Windows Fibers.[3] It is plausible to assert that use of the general term *fiber* for user-mode cooperative threads originated with this facility. Nonetheless, it is important to recognize that the Windows Fibers facility is a specific *implementation* of the more general concept of user-mode cooperative threads.

P1364 regrettably conflates the concept of a *fiber* with the specific Windows Fibers *implementation.* The concept of user-mode cooperative threads should not be judged solely on the characteristics of the elderly Windows Fibers implementation.

P0876 presents *fibers without scheduler,*[2] a foundational facility on which libraries such as Boost.Coroutine2[7] and Boost.Fiber[8] may be built in portable C++. At the request of WG21, that facility is intentionally lower-level than the platonic ideal of a fiber. As the name suggests, much of the delta is the lack of a scheduler: when switching context, a P0876 fiber must explicitly designate its successor. In general, a fiber *suspends* (passes control to a scheduler) rather than explicitly *resuming* a specific other fiber.

2 P1364R0 §1
3 P1364R0 §3.1

That said, in responding to P1364, this paper is more about the concept of fibers than about the specific facility proposed in P0876.

Points from P1364

Memory Footprint

P1364 asserts[4] that a fiber stack requires a megabyte of address space. That may be true of Windows Fibers; it is not true in general.

Fixed size very small stack

It is true that using a fixed-size stack smaller than a 4 kilobyte memory page must be attempted only with specific knowledge of the stack consumption of that thread of execution.[5] However, P1364 skips lightly over the three orders of magnitude difference between 1 megabyte and 4 kilobytes. It works to use a fixed-size stack considerably smaller than a megabyte. Focusing on very small fixed-sized stacks is misleading.

A fixed-size stack:

- must be allocated large enough for the maximum stack consumption for that thread of execution
- has no "safety net," therefore its pre-specified size should include some cushion
- optimizes for speed, since each function activation frame is allocated with an increment and freed with a decrement.

Stack with guard page

Engaging the operating system to manage stack memory:[6]

- still reserves the address range for that stack, whatever its pre-specified size might be
- need not commit physical memory until required
- can be specified smaller, since the operating system can detect overrun
- incurs operating-system overhead, at initial stack creation to construct the guard page and every time stack use exceeds the physical memory committed thus far.

It's worth noting that a guard page does not map to physical memory: it exists only as a page-sized address range.

In fact there is another useful point on this spectrum – a fixed-size stack with a guard page, which:

- reserves the address range

4 P1364R0 §2.1
5 P1364R0 §2.1.1
6 P1364R0 §2.1.2

- commits the whole size at allocation
- can be specified smaller, since the operating system can detect overrun
- incurs operating-system overhead only on initial allocation.

Optimize for speed		*Optimize for space*
Fixed-size stack	Fixed-size stack with guard page	Dynamic stack with guard page

Split/segmented stacks

A further refinement is an approach in which user-mode library code, rather than the operating system, is responsible for growing and shrinking the execution stack as needed.

Avoiding operating system involvement requires a linked list of stack segments, rather than a continuous but only partially committed address range. It also rules out use of a guard page to detect segment overrun. With this technique, each function prologue is instrumented to check whether the current segment can hold its new activation frame. If not, a new stack segment is allocated and linked onto the list to become the new current segment.

The function epilogue is similarly instrumented to detect destruction of the only remaining activation frame in the current segment. At that point, the previous current segment is made current again.

The size of the stack segments is a possible tuning knob in the time/space tradeoff. Larger segments mean fewer allocations and segment hops, but typically more unused memory in the current segment. Smaller segments are more conservative of memory, at the cost of more allocations and segment hops.

Whether to release a newly-empty segment is another tuning knob in the time/space tradeoff.[7]

P1364 points out[8] that a stackless coroutine allocates only the space required for its own activation frame. Moreover, much has been made of the potential for the compiler to optimize away even that allocation.[9]

A stackless coroutine *used as a coroutine*, with a lifespan bounded by that of its invoker, is potentially subject to that optimization – though there are questions[10] about the extent to which consumer code can rely on such optimization.

However, a stackless coroutine *simulating a user-mode cooperative thread* must *always* allocate its activation frame. Heap Allocation eLision Optimization (HALO) places the coroutine's local variables in its invoker's stack frame. If the lifespan of the coroutine can possibly exceed the lifespan of its

7 P0876R5 does not present stack allocation tuning knobs. At the committee's request, that feature is being proposed in a separate paper.
8 P1364R0 §2.1.2
9 P0981R0
10 P1063R1

invoker, correctness *requires* that those local variables be owned by the coroutine itself rather than by its invoker.

A notional thread of execution composed of stackless coroutines is, in effect, using a segmented stack – in which each segment can fit only *one* activation frame. The time/space slider is unconditionally set all the way to "optimize for space." Every entry to any such stackless coroutine performs a heap allocation; every return necessarily performs a release.

One might reasonably ask: what about a memory pool? That is a good idea. A memory pool of function activation frames is called a "stack."

A thread of execution based on a segmented stack will incur *less* "hot split" overhead than a notional thread of execution based on stackless coroutines.

Context Switching Overhead

P1364 presents a table[11] reporting Windows Fibers context-switching overhead. This may be a quality-of-implementation issue. Comparable Boost.Context[12] metrics and Boost.Fiber[13] are markedly smaller.

Context switching overhead is important, and P0876's fiber_context facility can switch context in 6ns. That said, it's worth remembering that actual business logic does not spend most of its time switching context: it spends most of its time calling functions and returning from them.

Every call to a nontrivial stackless coroutine function performs a heap allocation. Every return from such a function releases that heap block. By contrast, on a fiber, even with segmented stacks, most function calls only increment and decrement the processor's stack pointer.

And that's considering only pure overhead. With either approach, typical business logic will swamp that overhead.

Dangers of N : M model

Whenever a process runs more than one `std::thread`, any libraries or standard facilities that internally may use `static` storage could result in undefined behavior such as corrupting memory, reading garbage or both.

Despite that danger, `std::thread` was introduced anyway because it's *useful.*

This is exactly analogous to the situation with fibers and `thread_local` storage. We introduce a kind of execution agent finer-grained than before. A storage class sufficient for older kinds of execution agents may no longer be appropriate for the newer one.

Note that P0772 introduces the notion of "execution agent local storage."

11 P1364R0 §2.2
12 https://www.boost.org/doc/libs/release/libs/context/doc/html/context/performance.html
13 https://www.boost.org/doc/libs/release/libs/fiber/doc/html/fiber/performance.html

P1364 asserts[14] that stackless coroutines have no problems using `thread_local` storage. This is not entirely true.

For a given `thread_local` variable, every invocation of a stackless coroutine within the same thread shares the same instance. With fibers, this problem could be solved by introducing a "fiber local" or "execution agent local" storage class. But with stackless coroutines, it's not clear that such a solution is possible at all, because the mapping of stackless coroutine to execution agent is not well-defined.

Since a stackless coroutine can be resumed on a thread other than the one on which it suspended, it may unexpectedly find itself sharing a `thread_local` instance it has never before encountered. This is true of any function that might migrate to another thread during suspension. It is a problem for stackless coroutines as well as functions running on a fiber.

Hazards of 1 : N model

P1364 states:[15] "any blocking call completely stops progress of all N fibers." More generally, any call that blocks the running thread blocks every activity on that thread.

This is true of fibers. It's true of callbacks. *It is true of stackless coroutines as well.* It is *not* a differentiator between stackless coroutines and fibers.

Any application attempting to multiplex different activities within a `std::thread`, regardless of the mechanism, must scrupulously avoid making any system call or library call that blocks the current thread. That specifically includes applications written to use stackless coroutines for concurrency.

P1364 further states[16] that a user mode scheduler is required for fibers. This is a good thing. Scheduling of stackless coroutines is accidental and prone to starvation.

P1364 notes[17] that ".NET's garbage collector captures the user mode stack location of a thread and uses [it] to look for roots. If a [Windows] fiber switches the user mode stack, roots will not be scanned, and the memory may be reclaimed, resulting in use after free." This is a quality-of-implementation issue.

Case Studies

P1364 admonishes,[18] in bold all-caps: **"DO NOT USE FIBERS!"**

We respectfully submit that this injunction should be interpreted to mean Windows Fibers specifically, rather than fibers as framework and conceptual tool.

14 P1364R0 §2.3.1
15 P1364R0 §2.3.2
16 *Ibid.*
17 *Ibid.*
18 P1364R0 §3

Fiber use on Windows

P1364 quotes[19] an earlier Microsoft blog post:[9] “In fact, the recommendation is that instead of spending your time rewriting your app to use fibers (and it IS a rewrite), instead it's better to rearchitect your app to use a ‘minimal context’ model - instead of maintaining the state of your server on the stack, maintain it in a small data structure, and have that structure drive a small one-thread-per-cpu state machine. You'll still have the issue of unexpected blocking points (you call malloc and malloc blocks accessing the heap critical section), but that issue exists regardless of how your app's architected.”

With all due respect, it is less work to adapt an application to use fibers than it would be to rearchitect it entirely.

It is less work to adapt an application to use fibers than to sprinkle requisite stackless coroutine markup throughout the code base.

Linux

P1364 quotes[20] a Red Hat document[10] that states: “Huge numbers of threads are no issue since the scheduler and all the other core routines have constant execution time (O(1)) as opposed to linear time with respect to the number of active processes and threads.”

That may or may not be true with respect to time. Empirically, however, kernel threads consume more of other resources than fibers. On a recent Linux system, an attempt[21] to run the skynet microbenchmark[11] – which spawns one million threads of execution to evaluate an implementation of threads of execution – utterly failed with both pthread and `std::thread` due to resource exhaustion. Those test runs had to be trimmed back to ten thousand threads of execution instead.

That test ran with the full million Boost.Fiber fibers, with execution times from one to three orders of magnitude less than the pthread or `std::thread` threads of execution.

POSIX

P1364 notes[22] that POSIX.1-2008 deprecated its ucontext facility, without mentioning the reason. The reason is simply that they could no longer express the historical API with a modern C function signature. It is erroneous to conclude that the feature isn’t useful.

Facebook experience

P1364 states[23] that Facebook wants to move away from their internal fiber library because:

- a sparingly-allocated fiber stack can overflow when calling a stack-hungry known-synchronous API. This can be avoided by explicitly switching back to the main thread stack to perform such

19 P1364R0 §3.1
20 P1364R0 §3.3
21 https://www.boost.org/doc/libs/release/libs/fiber/doc/html/fiber/performance.html
22 P1364R0 §3.4
23 P1364R0 §3.5

calls, but any such call not wrapped that way is at risk. P0876 does not (yet) suggest or require compiler involvement: it is a library proposal. However, the compiler could be taught to implicitly switch to the main thread stack for deep synchronous calls.

- the need to use fiber-aware synchronization primitives rather than standard primitives that block the whole thread. This stated reason is perplexing, since stackless coroutines must similarly avoid blocking the current thread.
- unintentional sharing of a given `thread_local` instance between fibers. But stackless coroutines multiplexed onto a single thread face the same issue.

That said, in some use cases fibers are a better fit, in others stackless coroutines may be.

“Data” is not the plural of “anecdote.”

Dubious conclusions

P1364 claims[24] that fibers are not an appropriate solution for writing scalable concurrent software. This is a matter of opinion, arguably biased opinion. We counter that fibers are the best available way to organize asynchronous code.

We have already noted the viral cost of retrofitting an application to use stackless coroutines to manage asynchronous operations. Less obvious, but even more important, is the danger of *forgetting* to mark up a stackless coroutine call as required.[25,[3]] The compiler doesn’t recognize that as a problem. It will silently emit incorrect code.

Even use of `[[nodiscard]]` isn’t foolproof, as the caller might capture the returned value in a variable rather than using `co_await`.

P1364 describes fibers as a highly platform-dependent facility. True! That is the best reason to introduce them into the C++ Standard: so that each vendor can provide an appropriate implementation for their platform. Fibers are inherently platform-dependent in the same way that the code emitted by the C++ compiler, and its library intrinsics, are platform-dependent.

P0876 presents an API very deliberately cast at a low level, to minimize the complexity of each platform-dependent implementation. Richer layers can be built on that API in portable C++.

24 P1364R0 §4
25 P0114R0[4] §12.7

Stackless vs Stackful

A few more corrections to P1364:[26]

Creation cost: stackful coroutines require a system call for guard page and expandable stack?	**False.** That's an option, not a requirement.
Thread local: use by stackful coroutines is undefined behavior?	**False.** One compiler's current implementation has a problem if a suspended fiber that uses thread-local storage migrates to another thread. Use of thread-local within a consistent thread has the same characteristics as for multiplexed stackless coroutines.
Platform dependence: needs OS support for dynamically-growing stack?	**False.** A viable fiber implementation does not require that particular stack implementation.
Platform dependence: stack switching is highly dependent on OS/CPU architecture?	**True.** So are the parts of the standard library that interface with the OS and CPU in other ways. Each vendor provides a suitable implementation, which is the reason to standardize those parts.

26 P1364R0 §7

Fiber Motivation

Asynchronous Code

If operating-system kernel threads provided sufficient concurrency, we would need no asynchronous system calls. All system and library calls would be synchronous (blocking). A synchronous API is always simpler than the corresponding asynchronous API. All concurrency would be effected by spawning kernel threads.

Yet the world continues to trend towards *more* asynchronous APIs, not fewer.

It is therefore evident that applications require more concurrency than is provided by kernel threads. We must be able to interleave operations within a single thread. The question is, how should we organize the code that initiates and responds to such asynchronous operations?

Traditionally, classic C and C++ provided little help. This led to such patterns as chains of callbacks (whether free functions or virtual methods), functions containing voluminous switch statements or more complex state-machine implementations.

Two recent technologies are under consideration for the C++ Standard: Coroutines TS stackless coroutines,[5] and fibers-without-scheduler `fiber_context`.[2] Either can be used to organize the code in an asynchronous application. Each allows coding a function that performs some setup and suspends to wait for a result – *without* blocking the thread on which that function was called.

P1364 states[27] that “In 2018, with the further improvement to the NT kernel, even with the very good user mode scheduler, there are no significant performance improvements when using UMS...”

But the major benefit to using either of the technologies cited above is code organization, clarity and maintainability. Performance may be similar to chains of classic callback functions, but there is real cost, in both money and time, to maintaining badly-organized code.

Scalability

Empirically, even with modern operating systems, hardware and compiler technology, kernel threads cannot support the same volume of concurrency as fibers. In one documented case,[28] the Boost.Fiber implementation ran two orders of magnitude more fibers than the operating system could run threads.

Scalability is one of the strengths of stackless coroutines as well. Given that entry to each stackless coroutine allocates exactly the space required for its own activation frame, it seems safe to say that an application based on stackless coroutines will generally consume less memory than an equivalent application based on fibers. For the same reason, it seems safe to say that the fiber-based application will generally run faster.

27 P1364R0 §3.1
28 https://www.boost.org/doc/libs/release/libs/fiber/doc/html/fiber/performance.html

Coexistence

We stipulate that there are use cases better suited to stackless coroutines than to fiber technology. For example, a case often cited is a heavily-loaded 32-bit Windows server process. In that environment, segmented stacks are not an option because Windows doesn't support them: Windows stacks must be contiguous. Every such stack must reserve a range of the available address space, even if less than the full range is backed by real memory. In a 32-bit process, address space becomes the scarce resource.

We assert, however, that there are use cases in which fiber technology is the better fit.

Consider a large application already written to interleave concurrent asynchronous operations within a single kernel thread. Such an application necessarily avoids any operation that blocks the entire thread. Moreover, within any completion handler, regardless of structure – for instance, a callback function – `thread_local` is no better than `static` for storing data that must persist between calls.

These things are *already* true, even before the introduction of fibers.

P1364 asserts[29] that in the async use case, stackless coroutines can be adopted incrementally, one function at a time. That is simply untrue. If maintenance to some function requires that it suspend – when it did not previously suspend – every one of its callers must be modified; every one of *their* callers must be modified. The transitive closure of every function that calls any function that might call the modified function must itself be modified. Stackless coroutine markup is viral.

P1364 further asserts[30] that in the async use case, adopting fibers requires wholesale changes to all synchronization primitives and blocking calls. This is also untrue, because – as noted above – an application already interleaving asynchronous operations within a thread already avoids blocking operations and synchronization.

Summary

P1364 questions whether fibers should become part of the C++ Standard. But given P1364's faulty premises, that conclusion must itself be challenged.

29 P1364R0 §7
30 *Ibid.*

References

[1] P1364R0: Fibers under the magnifying glass
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1364r0.pdf

[2] P0876R3: fiber_context – fibers without scheduler
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0876r3.pdf

[3] P0099R1: A low-level API for stackful context switching
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0099r1.pdf

[4] P0114R0: Resumable Expressions
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0114r0.pdf

[5] Working Draft, C++ Extensions for Coroutines
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4775.pdf

[6] Boost.Context
https://www.boost.org/doc/libs/release/libs/context/doc/html/index.html

[7] Boost.Coroutine2
https://www.boost.org/doc/libs/release/libs/coroutine2/doc/html/index.html

[8] Boost.Fiber
https://www.boost.org/doc/libs/release/libs/fiber/doc/html/index.html

[9] Why does Win32 even have Fibers?
https://blogs.msdn.microsoft.com/larryosterman/2005/01/05/why-does-win32-even-have-fibers/

[10] The Native POSIX Thread Library for Linux
https://akkadia.org/drepper/nptl-design.pdf

[11] Skynet 1M threads microbenchmark
https://github.com/atemerev/skynet