Response to response to "Fibers under the magnifying glass"

Document Number: **P1520 R0** Date: 2019-03-08
Authors: Gor Nishanov
Audience: SG1

1 Introduction

Fibers (sometimes called stackful coroutines or user mode cooperatively scheduled threads) and stackless coroutines (compiler synthesized state machines) represent two distinct programming facilities with vast performance and functionality differences.

Standardizing stackless coroutines (Coroutines TS) does not gets in the way of acquiring stackful coroutines (fibers) in the future. Developers can choose the right tool for the job as long as they understand the trade-offs.

We agree with the authors of p0866r0 that no one tool (threads, fibers and stackless coroutines) can handle all of the use cases with the same efficiency or convenience. p1364r0 highlighted a particular use case (scalable asynchronous code), that fibers do not look like a good match. The authors of p0866r0 assert that there are use cases for which fibers are a better fit (than stackless coroutines or threads) and **we** wholeheartedly **agree** with them)[1].

In the rest of the paper, we will go over some details that were not sufficiently clear in p1364r0. In part, some misunderstanding was the result of sometimes unclear text of p1364r0 and sometimes, we believe, less than careful reading by the authors of p0866r0.

2 Polishing the magifying glass

We would like to revisit several areas of concerns about fibers hoping to highlight the areas of agreement and clear some misunderstanding that resulted in p0866r0, specifically in these three areas:

- Memory footprint
- Switching cost
- Compatibility / Incremental adoption

[1] Note that p1364r0 is not in conflict with that assertion. p1364r0 only questions whether there are enough strong motivating use cases to justify adding fiber facility to the general-purpose language such as C++

2.1 Memory Footprint

Section 2.1 of p1364r0 makes a point that the memory footprint of fibers is comparable to threads.

	Thread	Fiber
Kernel context	2k	0
Kernel stack	8k	0
User stack[2]	**1 meg**	**1 meg**
Fiber context	0	64 – 352 bytes

Regarding the table above, the authors of p0866r0 note that most fibers do not have to consume 1 megabyte of user stack and we **completely agree** with them. Both fibers and threads (pthread, win32 threads and std::thread with [P0484R0]) allow the user to control the stack size of the user stack of a thread. If users are fully aware of what are the requirements of all the functions that might be called from a fibers / thread, they can reduce the stack size when creating a thread or a fiber and therefore it could consume less than 1 megabyte in both cases.

2.2 Context switching overhead

P0866r0 states: "[in section 2.2] P1364r0 presents a table reporting Windows Fibers context-switching overhead. This may be a quality-of-implementation issue. Comparable Boost.Context metrics and Boost.Fiber are markedly smaller".

We urge the authors of P0866 to look carefully at section 2.2, reproduced here in its entirety for convenience:

While Fibers do not offer significant savings in terms of the memory footprint compared to threads, they do have a capability to switch from fiber to fiber without involving kernel transition and the cost of the fiber switch is cheaper that the cost of a thread switch. However, the fiber switch has still significant cost compared to a normal function call and return or (stackless) coroutine suspend and resume [Wandbox][3].

The following table samples fiber switching costs on several popular platforms:

	Instructions	Data to move (bytes)
System V x86_x64	23	64
MachO_arm64	28	176
Win_x86_x64	69	352

The table above specifically lists switching overhead of Boost.Context and the instruction and data movement count are obtained from the hyperlinked assembly routines. None of the entries in the table refer to Windows Fibers, neither Win_x86_x64, nor System V nor MachO entries. Also, Wandbox minibenchmark runs on linux.

Moreover, the cost of switching a fiber is not so much related to the skill of the assembly writer, but, to what calling conventions are supported on the platform that mandates the set of registers that have to be preserved on the context switch and what platform specific data stored in the thread context need to be adjusted.

[2] Typically, only 1 megabyte of virtual address space is consumed with physical memory allocated by the operating system dynamically as the stack grows.

[3] [Wandbox] link contains a simple benchmark highlighting the difference in switching cost between coroutines implemented on top of fibers vs compiler based coroutines. In that example, fibers have 20 times larger context switch overhead (with inlining disabled for stackless coroutines), otherwise, the difference grows to 2000+ times.

2.3 Compatibility / incremental adoption

P0866r0 makes a claim “Stackless coroutines cannot be adopted incrementally. The transitive closure of every caller of every such function must be modified”. Based on our face-to-face discussion in Kona 2019, we believe that authors of p0866 no longer hold that position, however, for those reading at home, here is a summary.

P1364r0 focuses on a particular problem domain of development of scalable asynchronous code that is typically deployed in N : M configuration where a small number of threads (say N = 2x number of cores) drive large number of asynchronous state machines (M > 100,000).

We assume that state machines are written either as a handcrafted-classes with callbacks [AsioServer] or as a scary looking .then chaining with future like types. In such codebases, stackless coroutines are introduced incrementally and provide an easier way of authoring a state machine (either a new one or to upgrade an implementation of an existing one) without having to touch anything else. Such incremental adoption has already happened in codebases that took a dependency on Coroutines TS.

P0866r0 explains that there are applications where fibers can be adopted incrementally without having to change the rest of the code and we have no reason to doubt that.

3 Conclusion

Hopefully this paper helped to bridge a bit of misunderstanding that resulted in P0866. In conclusion we would like to reiterate that:

- There is no conflict between stackful and stackless.
- They are very different facilities, different user interface, different performance characteristics.
- Developers can choose the right tool for the job as long as they understand trade-offs.

4 Bibliography

[N2325] Lawrence Crowl. “Dynamic Initialization and Destruction with Concurrency” (WG21 paper, 2007-01-13).

[N4775] “Working Draft, C++ Extensions for Coroutines” (WG21 paper, 2018-10-07).

[N3985] Oliver Kowalke, Nat Goodspeed. “A proposal to add coroutines to the C++ standard library” (WG21 paper, 2014-05-22).

[P0981R0] Richard Smith, Gor Nishanov. “Halo: coroutine Heap Allocation eLision Optimization” (WG21 paper, 2018-03-18).

[p0534r3] Oliver Kowalke, Nat Goodspeed. “A low-level API for stackful context switching” (WG21 paper, 2017-10-15).

[P1241R0] Lee Howes, Eric Niebler, Lewis Baker. “In support of merging coroutines into C++20” (WG21 paper, 2018-10-08).

[GoLang1.3] “Go 1.3 is released” (release notes, 2014-06-18).

[RustNoSeg] “Abandoning segmented stacks in Rust” (rust-dev relfector, 2013-11-04).

[Rust1.0alpha] “Announcing Rust 1.0 Alpha” (The Rust Programming Language Blog, 2013-11-04).

[RustNoGreen] “RFC: Remove runtime system, and move libgreen into an external library” (github pull request, 2013-11-04).

[SysV_x86_x64] “jump_fcontext implementation for System V x86_x64” (github/boostorg/, 2017-04-14)

[Win_x86_x64] “jump_fcontext implementation for Windows x86_x64” (github/boostorg/, 2017-04-25)

[MachO_arm64] “jump_fcontext implementation for Mach-O arm64” (github/boostorg/, 2016-12-04)

[Wandbox] “Stackful vs stackless context switch overhead” (https://wandbox.org/permlink/gycsul-WQyE8GVinB, 2018-11-22)

[GSoC2006] “Interaction between [stackful] coroutines and threads” (Documentation for boost:Coroutine library developed during GSoC, Summer of 2006)

[Function call x64] An example of code generation for a function call (https://godbolt.org/z/mnCrwi)

[FiberPerils] Ken Henderson. “The perils of the fiber mode” (technet article, 2005-02-01).

[TiobeIndex] “The TIOBE Programming Community index” (retrieved on , 2018-11-25).

[BoostFiber] Oliver Kowalke. “Boost Fiber Documentation” (Boost 1.68, 2018-08-09)

[SqlFibers] Ken Henderson. “Inside the SQL Server 2000 User Mode Scheduler” (technet article, 2004-02-24).

[P0484R0] “Enhancing Thread Constructor Attributes” (WG21 paper, 2017-06-18).

[WhyFibers] Larry Osterman. “Why does Win32 even have Fibers?” (msdn blogs article, 2005-01-05).

[SunOsMt] “Multithreading in the Solaris(tm) Operating Environment” (whitepaper, 2002).

[UMS] “User-Mode scheduling” (MSDN documentation, retrieved on 2018-11-23).

[nptl-design] Ulrich Drepper, Ingo Molnar. "The Native POSIX Thread Library for Linux" (whitepaper, 2005-02-21).

[Stroustrup 1994] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[nptl-design] Ulrich Drepper, Ingo Molnar. "The Native POSIX Thread Library for Linux" (whitepaper, 2005-02-21).

[AsioServer] “asio/asio/src/tests/performance/server.cpp” (github repo).

[p1364r0] Gor Nishanov “Fibers under the magnifying glass” (WG21 paper, 2018-11-20).

[p0866r0] Nat Goodspeed, Oliver Kowalke “Response to “Fibers under the magnifying glass”” (WG21 paper, 2019-01-06).