Fibers under the magnifying glass

Document Number: **P1364 R0**
Reply-to: Gor Nishanov (gorn@microsoft.com)
Date: 2018-11-20
Audience: WG21, EWG, SG1

Abstract

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.

This paper highlights efficiency, scalability and usability problems of fibers and reaches the conclusion that they are not an appropriate solution for writing scalable concurrent software

Contents

1 Introduction

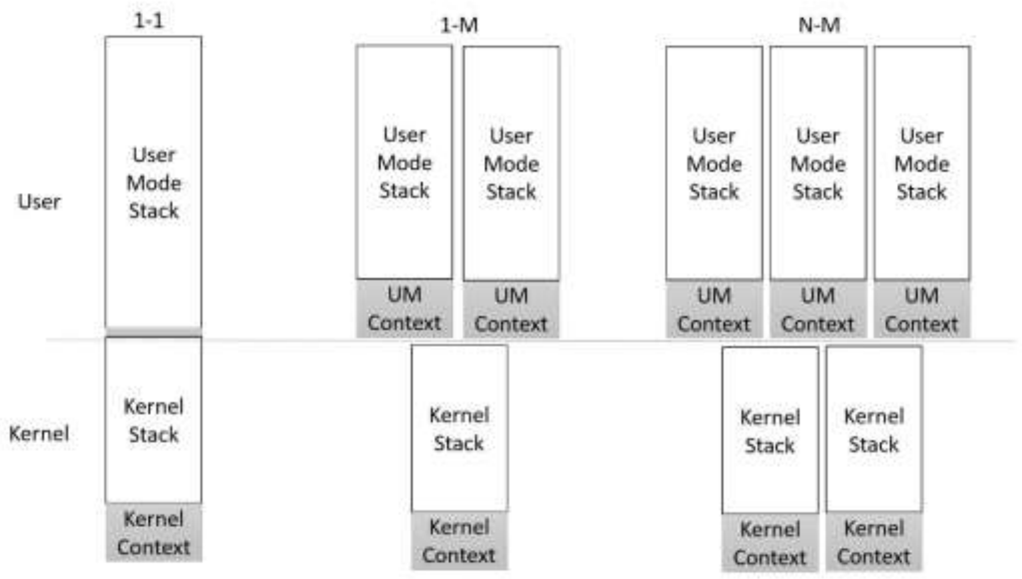
Coroutines is a programming concept that have been known and used since 1958. Knuth defined it as a generalization of a subroutine: regular subroutines always start at the beginning and exit at the end, whereas coroutines can also suspend the execution to be resumed later at the point where they were left off.

There are two common implementation strategies for coroutines: compiler-based transformation into a state machine or implementation as an interface adapter on top of threads (regular or user-mode cooperatively schedules ones). To disambiguate between two implementation strategies, adjective “stackless” is typically used to indicate that a coroutine is implemented by the compiler as a state machine and “stackful” to indicate that a coroutine is implemented on top of a user-mode cooperatively scheduled thread.

To understand the drawbacks of the stackful approach, we need to understand the properties of threads and fibers[1] (shorthand for user-mode cooperatively scheduled threads) and how they compare to state-machine approaches (whether hand-crafted or compiler synthesized ones).

2 Threads and Fibers

Threads typically are the smallest unit of scheduling supported by the operating system. In addition to threads, some operating systems and/or libraries offer a facility to multiplex multiple user mode stacks (fibers) on top of existing operating system thread. Such multiplexing can be done in 1:N configuration where each thread can be associated with some number of fibers and once associated fibers never migrate to other threads or N:M configuration where M fibers are multiplexed over N operating system threads.



We will examine the reasons why fibers, once thought of as a way to improve scalability of the applications with ostensibly more flexible N-M programming model fell out of favor to be replaced with 1-1 programming model.

2.1 Memory Footprint

Fibers have comparable memory footprint to that of the operating system threads saving about 1% due to not needing to save kernel context and kernel stack.

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

[1] Fiber as a term for user mode cooperatively scheduled threads (fibers) appeared in 1996 with the release of service pack 5 of NT3.51 operating system that provided them as a part of the OS API.

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

Because of the high memory footprint of fibers, several mitigation techniques are used:

2.1.1 Fixed size very small stack

This is approach allows fibers to use very small stacks (less than a typical memory page size of 4K), but, at the cost that the developer must have complete knowledge of how big activation frames are for every function that can be called directly/transitively from the fiber. This is a very precarious approach to be used with extreme caution as the smallest mistake can lead to memory corruption and security exploits.

2.1.2 Dynamic stack with guard page

This is a safe approach that reserves virtual memory for the stack in page size increments and sets up a guard page to either grow the stack further (if enough memory was reserved) or to terminate the program with stack exhausted error if the size is exceeded. The smallest fiber would consume 8k of virtual memory with only 4k needed to be committed. This is the same approach that is normally used with user stack memory of the regular OS thread. Some operating systems have specific optimizations for this case, for example, Windows NT combines fiber creation, reservation of the virtual memory and setting up a guard page in one system call. NT will also take care of automatically growing the fiber stack until its maximum size that is provided by the user during fiber creation.

Note that stackless coroutines only need to store locals that are live across suspend point which and frequently consume less than 64 bytes, even if they happen to call a function that requires 500k of stack, since a stackless coroutine uses the stack of the thread executing the coroutine. Fibers, on the other hand, have to have sufficient stack to accommodate any function they may call and therefore would require at least 500k of stack in that case.

2.1.3 Split stacks/segmented stacks

To avoid using virtual memory and memory protection to grow the stack, some systems choose to abandon contiguous stacks and proceed with segmented stacks, where the fiber stack is allocated in chunks and every function prologue is instrumented to check whether there is enough room in the current segment and allocate a new segment if needed. Each prologue is instrumented to check whether it needs to jump to a previous stack segment and if so, it deallocates memory for the current segment.

This approach was used initially in Go programming language. It used 8 kilobytes segments and function calls allocated, and freed segments as needed.

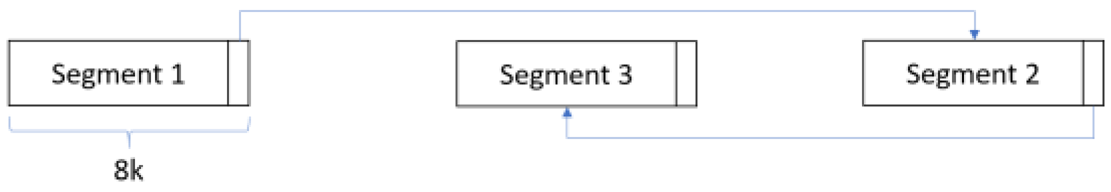


Figure 1: Go segmented stacks: 2009 - 2013

After five years of experience with segmented stacks, Go has abandoned this approach due to hot-split problem [GoLang1.3]. Rust that used to have segmented stack abandoned segmented stack approach as well [Rust-NoSeg].

As a replacement for segmented stacks, Go language chose to proceed with reallocate and copy approach. In this approach the stack is always contiguous, when it needs to grow, a bigger contiguous stack is allocated, and all the content is copied to a new memory location and all pointers referencing the old stack are appropriately adjusted to point at a new stack.

Pointer adjustment is possible to do in a programming language with precise garbage collector, such as Go, but unfeasible in more traditional languages where it is not possible to freely move objects in memory and adjust all of the pointers pointing to them. Recognizing that limitation Rust developers went with the virtual memory and guard page approach described in the previous section. Few years later, Rust removed fiber support from the language altogether [RustNoGreen].

2.2 Context Switching overhead

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

2.3 Compatibility & Scalability

Ability to perform a context switch in the user mode is both the key feature and the key liability of fibers.

2.3.1 Dangers of N : M model

If fibers are deployed in N : M model where M fibers are multiplexed over N operating system threads, using libraries such as TcMalloc, boost::asio or standard facilities, such as std::error_code, function static variables [N2325] and others that internally may user thread local storage will result in undefined behavior such as corrupting memory, reading garbage or both.

[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.

Function f does not use TLS directly	Writes to arbitrary memory location
`void f() {`	`push    {r4, lr}`
`  may_use_thread_local();`	`ldr     r0, .L4`
`  may_incur_fiber_switch();`	`bl      __emutls_get_address`
`  may_use_thread_local();`	`mov     r4, r0                       ; (1) caches TLS address`
`}`	`bl      writes_to_thread_local(int&) ; (2) OK`
	`bl      may_incur_suspend()          ; (3) migrated`
	`mov     r0, r4`
	`bl      writes_to_thread_local(int&) ; (4) writes to cached address`
	`pop     {r4, lr}`
	`bx      lr`

In this example, even though the author of function **f** was not using the thread local storage directly, the function ended up with writes to a thread local after an optimizer inlined some functions into **f**. A compiler also has chosen to cache TLS address in a callee saved register r4 (1) after the fiber was potentially migrated to a different thread (3), r4 is still referring to the cached address. **If we are lucky**, the original thread is still alive, and **we simply corrupt the value of the thread local of a different thread** (4) with an unexpected value, **if we are unlucky**, the thread could have quit, and its memory could have been reused resulting in **use after free** (4).

This problem does not occur in stackless coroutines since all suspend points are statically known to the compiler and thread local access is well defined to refer to the thread local variable of the current thread.

This particular hazard can be avoided if deployment of fibers is limited to 1 : N model. However, it brings its own set of restrictions.

2.3.2 Hazards of 1 : N model

While 1 : N model resolves the problems related to the thread identity (such as thread local access), it creates even bigger problem[4]: any blocking call completely stops progress of all N fibers. To mitigate that, projects using fibers resort to **rewriting all of the APIs and libraries that may have blocking APIs** and providing fiber aware versions of those APIs, for example [BoostFiber]. In addition, a runtime for user mode scheduler will be required. The fiber aware facilities would yield control to the user-mode scheduler instead of blocking so that the fiber runtime can schedule a different fiber instead.

Even with a fully functioning fiber runtime, if any of the code running on a fiber accidentally calls a blocking API or takes a page fault, all progress on all N fibers will be completely stopped.

Moreover, 1 : N does not completely eliminate fiber incompatibility problems with existing software. The first one, as we mentioned, a requirement of not calling into anything that might block, the second, is that software components may rely on other parts of the thread context than just the thread local and will not behave properly if fiber switch happens.

C++ is used together with scripting and managed languages by hosting their runtime in the C++ process. Python, LUA, C# are among those languages used together with C++. Most of these languages have runtimes that include a garbage collector. For example, .NET runtime's garbage collector captures the user mode stack location of a thread and uses to look for roots. If a fiber switches the user mode stack, roots will not be scanned, and the memory may be reclaimed, resulting in use after free.

[4] While this is also a problem with N : M, it is especially harmful in the 1 : N model

This problem does not occur with stackless coroutines as they use always use the stack of the thread that executes them.

3 Case Studies

We have accumulated more than a quarter century of experience with fibers across variety of programming languages and platforms. In the 90s fibers and N : M scheduling looked promising, now, with improvements in hardware, operating system kernel and painful experience of trying to make the fibers work has resulted in a recommendation: **DO NOT USE FIBERS!** Use threads instead and/or write your code using asynchronous APIs with hand-crafted state machines[5].

3.1 Fiber use on Windows

Microsoft spent significant resources trying to make the fibers work.

- In 1996, NT3.51 SP5 added fibers in Win32 API. At request of the SQL team and others [WhyFibers].
- In 1998, SQL 7.0 added an optional fiber scheduling mode [SqlFibers].
- In 2005 or earlier, a recommendation was issued **not to turn it on** [FiberPerils]. Fiber mode was rarely enhancing performance and popular components either did not work or behaved erratically due to fiber caused compatibility problems.
- At some point, Visual C++ compiler added a compiler switch /GT to generate fibers safe TLS access code, by making thread local access slower for all functions whether they were running on a fiber or not. The switch was off by default and was rarely turned on.
- In 2009, Windows 7 introduced user mode scheduling [UMS], that used 1-1 model with a possibility of switching user mode part of a thread without transition to the kernel, with the matching kernel part being switched on any subsequent user mode to kernel transition. (UMS threads reduce compatibility issues compared to fibers).
- 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 and the feature may be deprecated in near future.

Current recommendation is to avoid using fibers and UMS. This advice from 2005 remains unchanged: *"… [I]nstead 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."* [WhyFibers].

3.2 Solaris

After exploring N:M programming model for 7 years, Sun Microsystem abandoned it in favor of simpler 1:1 threading.

- In 1993, Solaris 2.2 included support for 1:1 and N:M threading model, where M lightweight application threads could be multiplexed on top of N kernel threads
- In 2000, Solaris 8 discarded N:M in favor of simple 1:1 threading

[5] or with compiler-synthesized ones if your programming language supports stackless coroutines

“At one time, it was thought that thread stacks were the natural place to store application state. But it became apparent that threads delivered greater scalability when they were either running on a CPU, or blocked in the kernel waiting for some event to occur. In such cases, application state could be kept in well-designed data structures, and a pool of worker threads deployed to process this data. The improved scalability was due to better locality of reference caused by allowing both the user and kernel thread stacks to be used more intensively.” [SunOsMt]

3.3 Linux

Similarly, after exploring N:M model for a bit, Linux kernel adopted 1:1 threading since kernel 2.6

- < 2003, Experimentation with N:M threading. NGPT in Linux Kernel 2.4
- 2003, N:M Support is dropped in favor of NPTL starting from Linux 2.6

“many problems the user-level scheduling helps to prevent are no real problems for the Linux kernel. 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.” [nptl-design]

3.4 POSIX

After providing facilities for user-mode stack switching (ucontext_t) for 8 years, POSIX deprecates and removes it.

- 1995: POSIX.1c: Threads extensions ucontext_t/makecontext/etc
- 2003: POSIX.1-2003 ucontext_t is deprecated
- 2008: POSIX.1-2008 ucontext_t removed

“POSIX.1-2008 removes the specification of getcontext(), citing portability issues, and recommending that applications be rewritten to use POSIX threads instead“

3.5 Facebook experience

Facebook has deployed fibers internally for 5 years and now “... looking to migrate away from it as soon as we are confident of the direction [stackless] Coroutines will go in in the standard. We already try to dissuade non-experts from using them as much as possible, but in the absence of anything better they occasionally do.”.

Reasons cited for desire to migrate away from fibers are:

- pervasive stack overflows that are confusing for developers to debug and that are only partially solved by guard pages and by explicitly switching back to the main thread stack to make stack-hungry known-synchronous calls.
- invasive nature of fiber use that requires modification of synchronization primitive internals to correctly switch fibers, or to replace them with fiber-aware primitives.
- hidden fiber switches in apparently synchronous code due that cause thread-local access interleaving with fiber execution; reuse of code between fiber and non-fiber contexts has done more harm than good in practice.

3.6 Programming Language survey

The only programming language among top 10 most popular languages according to TIOBE index [TiobeIndex] that provides fibers is Go. Go programming language uses fibers successfully, but incurs a very high cost (**~160ns**) whenever it has to interoperate with C libraries, due to the need to swap in normal stack [GoOverhead]. That is not acceptable for C++ where the ability to seamlessly and efficiently interoperate with existing code is very important.

Index	Language	Fibers	Stackless coroutines (await/yield)
1	Java	n/a	n/a
2	C	n/a	n/a
3	C++	n/a	n/a
4	Python	n/a	await and yield
5	Visual Basic .NET	n/a	await and yield
6	C#	n/a	await and yield
7	Java Script	n/a	await and yield
8	PHP	n/a	n/a (await is available in Hack/PHP)
9	SQL	n/a	n/a
10	Go	Goroutine abstraction	n/a
11	Objective-C	n/a	n/a
12	Swift	n/a	(await implementation in progress)

4 Conclusion

While fibers may have looked like an attractive approach to write scalable concurrent code in the 90s, the experience of using fibers, the advances in operating systems, hardware and compiler technology (stackless coroutines), made them no longer a recommended facility.

Given that fibers are not an appropriate solution for writing scalable concurrent software, we are not sure that there are enough motivating reasons for C++ language to adopt and maintain a highly-platform dependent facility when platforms are unwilling to provide it and recommend against using it.

5 Acknowledgements

Many thanks to those who reviewed earlier drafts of this paper and provided valuable feedback, among them: Chandler Carruth, Geoffrey Romer, J. Daniel Garcia, Lee Howes, Lewis Baker, Anton Polukhin.

6 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/gycsulWQyE8GVinB, 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).

[GoOverhead] "What is the overhead of calling a C function from Go?" (golang-nuts discussion, 2009-11-30).

[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).

7 Appendix: Stackless vs Stackful

The following table summarizes differences between stackless and stackful coroutines.

	Stackless Coroutine	**Stackful Coroutine**
What	A function that can suspend and resume. A suspend point is lexically marked in the body of the coroutine.	A user mode cooperatively scheduled thread. Any function running within that thread can request a context switch.
State Size	~16 bytes + local variables live across suspend point. Can call any function without restriction	8k – 2megabytes. Need to have enough stack to support all functions that can be called directly or indirectly from a stackful coroutine
Switch cost	2 instructions	23 – 69 instructions
Interoperability (async use case)	Can be adopted incrementally, one function at a time.	All or nothing proposition. Requires changes to all synchronization and blocking APIs used from within a stackful coroutine
Exceptions	Supports coroutine cancellation that does not depend on exceptions	Requires throwing an exception to be able to cancel a stackful coroutine
Creation cost	State is heap allocated or stored as a local variable	Requires a system call to set up virtual memory and guard page for the stackful coroutine stack.
Thread Local	Defined behavior	Undefined Behavior
Platform Dependence	None. Implemented by the compiler	May need an OS support to support dynamically growing stack. Portability concerns as stack switching is highly dependent on the OS / CPU architecture