P0444: Unifying suspend-by-call and suspend-by-return

Document number:	P0444R0
Date:	2016-10-14
Reply-to:	Nat Goodspeed <nat@lindenlab.com>
Audience:	SG1, EWG

Abstract

This paper is a response to P0073R2 "On unifying the coroutines and resumable functions proposals," with particular reference to P0099R1 "A low-level API for stackful context switching" and P0057R5 "Wording for Coroutines."

This paper proposes an API that could be used for both technologies. Like P0073, it suggests compiler inferencing to eliminate pervasive source-code markup when suspend-by-return technology is selected.

Context for this paper

This paper should be taken as a companion paper to P0099.

The author considers P0099 to be important foundational technology. It is real; it is in use. It should be standardized even before we finish nailing down the admittedly speculative proposal in this paper.

This paper continues the discussion requested by WG21 in Rapperswil in June 2014, extending through N4232 to P0073, on how P0099 and P0057 can fit into a common Big Picture.

Apology

It has been stated that the role of WG21 is not to invent technology, but rather to standardize existing practice. Both P0057 and P0099 describe shipping technology in widespread use. It would be defensible to argue that each should be standardized as-is.

In this case, however, WG21 has taken the strong position that these two mechanisms should be integrated in some way. This necessarily forces us into the realm of invention: the authors of P0073 and P0444 lack the resources to prototype compiler modifications.

That is why this paper is speculative, rather than descriptive of existing practice.

Terminology

Context

Torvald Riegel persuasively argues in P0073 that a coroutine runs on what the standard describes as a thread of execution ("TOE"). In both P0072 and P0073 he describes TOEs lighter than operating-system threads, TOEs with weakly parallel forward progress guarantees, TOEs with a property of mutual exclusivity.

This paper uses the word *context* to allude to a lightweight TOE which, by default, will not run at literally the same time as any other such context.

Within a given process, we presume a hierarchy of concurrency:

- A process contains one or more operating-system threads (std::threads), each referencing the same process address space. Since different threads within a process may run on different processors, or since the operating system might use transparent preemptive time-slicing, code running on distinct threads can be assumed to run simultaneously.
- A thread contains one or more contexts. On that thread, at most one context is executing at any given moment (though of course the running context on that thread executes simultaneously with the running context on every other thread in the process). Within a single thread, control is explicitly passed between different contexts.

The operation of passing control from one context to another is *context switching.*

The running context *suspends* when it passes control to some other context.

That other context *resumes* when it becomes the current thread's running context.

Suspend-by-call, suspend-by-return

The word "coroutine" describes a facility available to the consuming application. Both P0057 and P0099 support "coroutines" and "generators" and other such use cases.

We are trying to arrive at a common API that can be implemented by either P0057 technology or P0099 technology, depending on the constraints of the application environment. For purposes of clarity, this paper prefers to avoid the term "coroutines," which can be built on either technology.

The terms "stackful" and "stackless" have been used to discuss the distinction. However, as it is possible to implement suspend-by-call semantics without using the contiguous i386 processor stack (cf. N4232), even those terms can be somewhat ambiguous.

This paper prefers the following terminology:

suspend-by-call (e.g. N3985, P0099)	Context switching is effected by calling a special function. The caller need not be aware that this function may suspend the calling context. By implication, the caller's caller need not be aware either, and so on. N3985 and P0099 both implement suspend-by-call using *side stacks.*
suspend-by-return (e.g. P0057)	Context switching is effected by returning from a function with some discriminator indicating whether the return means suspension or completion. The caller must distinguish the cases. In particular, when a function suspends, its caller must also suspend, and so must its caller's caller, and so forth. The goal is to remove stack frames from the main stack, back to the point at which the context was launched. This permits reuse of the main stack for non-suspending functions.

Suspendable

This paper borrows from P0073 the term "suspendable" to mean a function that might suspend, either directly or by calling another suspendable function. We import the term for discussion purposes without proposing it as a keyword.

Resumable

When a suspendable function is compiled for suspend-by-return, its generated body is significantly different than for an ordinary C++ function. For the sake of brevity, we will refer to that form as the function's "resumable body," as it uses the technology formerly known as "resumable functions."

When a suspendable function is compiled for suspend-by-call, its generated body is the same as that of an ordinary C++ function. When necessary to distinguish, we will call that its "normal body."

We refrain from proposing either "resumable" or "normal" as keywords.

The gulf

Each of the proposals P0057 and P0099 has a number of refinements. We will not discuss those in this paper. We will focus solely on the most important distinction.

Consider an application which will run function f() on a new context: a context explicitly created by the application. f() will interleave its execution with the original application context, perhaps the default context on which main() is run.

f() calls g().

g() calls h().

h() calls i().

i() calls library functions, such as operator<<(std::ostream&, const std::string&).

h() will perform some asynchronous I/O: the reason for the application to construct a new context for f(). h() is written to suspend during the async operation. It will resume once the result becomes available. Calls to i() happen before or after suspension.

None of f(), g() or i() otherwise need to suspend, and of course current standard library functions know nothing about context switching.

The fundamental gap between suspend-by-call and suspend-by-return is this.

When h() suspends using suspend-by-call, f() and g() are unaffected. As far as each of them is concerned, it is making a perfectly ordinary C++ call to an ordinary function.

When h() suspends using suspend-by-return, g() must recognize that return event as suspension, and must itself suspend. P0057 proposes the keyword co_await for this purpose. g() must co_await h() rather than simply calling it. Similarly, f() must co_await g().

In a large code base that relies heavily on async I/O (or has other reasons to interleave contexts), suspend-by-return markup is pervasive. It has been described as "viral" because, over time, it will tend to propagate through a code base. Every caller of every function that might suspend must itself be prepared to suspend.

From the point of view of maintaining a large existing code base, the critical distinction between suspend-by-call and suspend-by-return is *pervasive source code markup.*

To connect these models, we will require a suspension bridge.

A new hope

P0073 suggests that compiler inferencing can be used to eliminate much (most?) of the markup required by P0057. In other words, although the underlying implementation technologies remain just as divergent, it proposes that the distinction between them can be blurred at source code level.

Put differently, you could write code that *looks and behaves* as though it uses suspend-by-call semantics, even if it really uses suspend-by-return technology under the hood.

That would solve the problem! We could write code using a single context-switching API, but compile it differently to prefer different tradeoffs for different runtime environments.

The key to this solution is to make the compiler infer the special nature of suspend-by-return functions, rather than requiring the coder to mark each of them explicitly.

- The compiler will implicitly pass a discriminator to every such function. The function's declared return type does not need to express the distinction.
- On return from every such function, generated code will implicitly check for suspension and, if the called function suspended, suspend the caller as well. No special keyword marks any such call.

A new API

P0073 is intentionally somewhat tentative about the specific API to use for the two key operations: launching a new context and suspending the running context. It stipulates, without specifying, a "scheduler" to know when resumption is permissible.

P0099 discusses the power of a symmetric context-switching API versus an asymmetric API. These terms reflect the relationship between a pair of contexts (as expressed by the API). An asymmetric API bakes in the distinction between invoker and invokee: the latter can only "yield" back to the invoker, without identifying it. In contrast, a symmetric API treats every context as equivalent: "I am now suspending by resuming *you."*

A symmetric API, though less convenient for direct use by an application, is a more powerful and efficient foundation for higher-level constructs.

The distinguishing characteristic of a symmetric API is that every context-switch operation explicitly designates the context to be resumed. Therefore, it must be possible to represent a context as a C++ object.

I propose P0099's std::execution_context as the common API for suspend-by-call and suspend-by-return context management.

As shown by Boost.Fiber, this API can be used to build userland threads. As shown by Boost.Coroutine2, this API can be used to build coroutines and generators.

Reifying a suspend-by-return context in an execution_context object gives us someplace definite to put the resumption entry point. It need not be embedded in a std::future or future-like object.

The execution_context could even provide a storage pool for some of the suspend-by-return functions' activation frames.[1]

Under the hood

When the user desires a suspend-by-call representation of each context, little compiler magic is needed.

It is when the user desires a suspend-by-return representation of each context that the magic is engaged, perhaps by a compiler switch.

As in P0057, the compiler can readily infer that a function might suspend (is "suspendable") by observing that the function body contains a call to std::execution_context::operator()().

This inference can be propagated through call chains: a function is suspendable if it contains a call to any suspendable function.

P0073 suggests that every suspendable function should be code-generated twice: a normal body for normal callers, a resumable body for suspendable callers. This behavior could be controlled by another compiler switch. One mode would compile two bodies for each suspendable function, with different mangled names as in P0073. The other mode would issue a diagnostic if a function which can never suspend (such as main()) directly called a suspendable function.[2]

A complementary diagnostic would prohibit launching a new context with a normal function.

With a suspend-by-call context representation, neither diagnostic need be issued.

Crossing translation units

The difficulty with making the compiler infer characteristics important to a function's caller based on inspection of that function's body is, of course, that very much software is intentionally written to place a function's definition in a different translation unit than that of its callers. This is considered industry best practice.

Two possible approaches to this issue are:

- As suggested in P0073, an extern function's declaration can be explicitly annotated, perhaps with an attribute such as [[suspends]]. With this direction of inference propagation, the compiler could actually issue a diagnostic if a function declared without [[suspends]] is inferred as suspendable.
    - Also as suggested in P0073, the suspendable compiled body of a function can have a distinctive mangled name – whether or not the function also has a normal compiled body. That should produce a linker error if some caller sees a declaration for the suspendable function that isn't visible during compilation of the suspendable function's definition.
    - This attribute markup is intended only as transitional. In a large code base, having the compiler infer suspendable functions only within a given translation unit may eliminate only a fraction of the required markup.
- Modules will save us all! The inferred suspendable characteristic can be published as part of the compiled module representation.

1 A pool of activation frames is sometimes called a "stack."

2 It is, of course, permissible for main() to launch a new context with a suspendable function.

- Once modules become widely available, we can deprecate the [[suspends]] attribute.

Callbacks

One of the difficulties with suspend-by-return technology, as discussed in N4453 and P0099, is that since you cannot transparently suspend from a nested call, you cannot suspend from a library callback.

And much of the STL is based on passing application function objects. These are callbacks. A suspend-by-return function object cannot suspend from within a (current) STL algorithm. The STL would itself need to be rebuilt with both normal and resumable bodies for every such algorithm.

Suspend-by-call technology (e.g. using side stacks) does support suspending from within a callback. But as has been mentioned, the trick is to ensure that your side stack is big enough to accommodate arbitrary library and/or OS calls by any function on the call chain. Even though such calls do not themselves suspend, they still require stack space, which may be hard to predict.

With suspend-by-return technology, library and/or OS calls are accommodated on the thread's original stack, which is stipulated to be "big enough" for such purposes.

I would like to point out that functionality introduced with P0099R1 permits us to pursue a hybrid approach. The new std::execution_context::operator()(std::invoke_ontop_arg, function, args...) feature executes function(args...) on the stack designated by the execution_context instance.

And an execution_context instance is capable of representing a thread's original stack.

Suppose suspend-by-return function a() calls suspend-by-return function b(). b() calls a library function some_algorithm(), passing it callback c().

c() makes heavy use of library and/or OS functions. Good thing we're using suspend-by-return technology so we can reuse the main stack.

But now c() needs to suspend. Oh no! c() can't use suspend-by-return!

If we launch (a(), b(), some_algorithm() and c()) on a side stack, then c() can suspend as needed, even though it's being called by some_algorithm(), which is definitely consuming one or more stack frames. But that means we potentially need a side stack of significant size to accommodate c()'s use of library and/or OS functions.

However – with the std::invoke_ontop_arg feature, we can instead refactor c() to execute its heavy-stack-use logic *on the original stack!* These particular library and/or OS functions do not themselves suspend, which is why it was okay to call them on the main stack when the (a(), b(), some_algorithm(), c()) context used only suspend-by-return functions.

We can launch a() on a *small* side stack, just big enough for a(), b(), some_algorithm() and c() itself.

It's possible to generate a function's resumable body with a prolog that fully cleans up the processor stack on entry, resetting the stack pointer to the value it had before the resumable body was called. Suppose our code generator does just that. Now we can get even finer-grained.

We can continue to launch a() as a suspend-by-return function, and have it still call b() as a suspend-by-return function. Within b()'s resumable body, the processor stack pointer should have the value it had before a() was called.

Then b() can launch some_algorithm() on an even smaller side stack, just big enough for some_algorithm() and c() itself. As above, c() delegates its heavy-stack-use logic back onto the main stack by using std::invoke_ontop_arg.

It is not yet clear how to express the above in a seamless API. It is exhilarating, however, to realize that a hybrid approach is possible, allowing us to work around the drawbacks of both side stacks and suspend-by-return technology.

Generally speaking, if suspend-by-return functions are perfectly clean as postulated above, a context consisting of an unbroken chain of such functions can detour onto a side stack at any point. The main stack is left untouched from the point at which the context was launched.

Similarly, code running on a side stack can call a chain of suspend-by-return functions. That boundary would require a shim to recognize suspend-by-return and suspend the side stack by call.

However, only non-suspendable functions can be shunted onto the main stack as described above.

Alternate implementation

Ideally we will arrive at an API sufficiently expressive to permit not only the two principal implementations we've been discussing, but additional implementations as well.

Cactus stack

N4232 describes a possible implementation of a logical "call stack" consisting of a singly-linked list of heap activation frames. As in P0057, each activation frame is allocated on entry and freed on return. As in P0057, context switching requires that the entire context down to the suspension point consist solely of similarly-compiled functions.

Unlike P0057, this implementation supports suspend-by-call semantics: switching context requires only changing which linked list of activation frames is current.

One trouble with that mechanism was the potential problem of interleaving normal stack frames into the linked list: the callback problem.

This is no worse than for suspend-by-return functions, and in fact the hybrid approach described in a previous section should work equally well for this.

Summary

- As suggested in P0073, to the extent possible, we should make the compiler infer which functions are suspendable. Eliminating the markup problem would eliminate the major pain point with suspend-by-return technology.
- Unlike P0073, we propose inferring a function's suspendable characteristic from leaf functions outward.
- Module support will make this even better.
- A symmetric context-switching API is a better foundation for higher-level abstractions than an asymmetric API.
- A symmetric API requires reifying a context as a C++ object. We propose P0099's std::execution_context.

References

N3985	A proposal to add coroutines to the C++ standard library http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3985.pdf
N4232	Stackful Coroutines and Stackless Resumable Functions http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4232.pdf
N4453	Resumable Expressions http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4453.pdf
P0057R5	Wording for Coroutines http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0057r5.pdf
P0072R1	Light-Weight Execution Agents http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0072r1.pdf
P0073R2	On unifying the coroutines and resumable functions proposals http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0073r2.pdf
P0099R1	A low-level API for stackful context switching http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0099r1.pdf
Boost.Coroutine2	http://www.boost.org/doc/libs/release/libs/coroutine2/doc/html/index.html
Boost.Fiber	http://www.boost.org/doc/libs/release/libs/fiber/doc/html/index.html