Document number: N4232
Date: 2014-10-13
Project: WG21, SG1
Reply-to: Nat Goodspeed (nat at lindenlab dot com)

Stackful Coroutines and Stackless Resumable Functions

Background

At Rapperswil in June 2014, Nat Goodspeed and Oliver Kowalke brought to SG1 a proposal to add coroutines to C++[i]. Niklas Gustafsson brought a proposal to add resumable functions to C++[ii]. We were directed by SG1 to place these proposals within a single conceptual space so as to unify them to the extent possible.

Nat, Oliver, Niklas and Torvald Riegel discussed this conceptual space at length at Rapperswil. Nat and Gor Nishanov pursued the discussion further in Redmond. This paper is an attempt to respond to the committee's request.

Overview

Coroutines

N3985 proposes a C++ coroutine library. The coder instantiates an asymmetric_coroutine<T>::pull_type, passing a callable. This callable accepts a non-const reference to a complementary library-synthesized asymmetric_coroutine<T>::push_type; methods on this push_type object switch context and transfer data. (There are variations on this theme, but for present purposes this case will suffice.)

An ordinary function can return a value to its caller from the middle of the function body.

The additional feature of these coroutines is that the caller can pass control back to the middle of the function body at the place where it last suspended its execution.

The coroutines proposed in N3985 are intentionally stackful. The authors assert that much of the value of this kind of coroutines is in their stackful nature.

Resumables

N3858, N3977 and D4134 propose new language to support resumable functions. Regardless of whether the compiler needs a keyword to recognize such a function, a resumable function is characterized by the presence of at least one 'await' keyword in its body. (There is also discussion of a 'yield' keyword, but at present we will consider only 'await.')

A resumable function returns a *potential* value – a future – to its caller. As always with futures, it may or may not hold a value at the moment the caller receives it. Call this the 'returned' future. (The proposal also supports other return types, but for simplicity we will discuss futures.)

A 'return' statement in the function's body populates the returned future with a value, and exits the

function.

An exception in the function's body populates the returned future with an exception.

An 'await' expression in the function's body introduces a dependency on some other explicitly-specified future. Call this the 'awaited' future. If the awaited future already holds a value, 'await' delivers the awaited future's value and continues execution of the function body. But if the awaited future is still pending, 'await' arranges to regain control whenever the awaited future is populated with a value. 'await' then returns a pending future to its caller. At some later point the awaited future will be populated; this causes the resumable function to be reentered at the 'await' that suspended it. It will proceed from that point, eventually executing a 'return' statement or another 'await' – or throwing an exception.

Although N3858 considers both a stackful and a stackless implementation of resumable functions, the authors of D4134 assert that the design goal of scalability necessitates a stackless implementation.

Suspension

As Torvald points out, the feature common to both N3985 coroutines and D4134 resumables is the operation of suspending: control can pass out of a function body in such a way that it can later be resumed at the same point.

Semantics

Most discussions so far about the idea of unifying coroutines with resumables have quickly dived into the realm of implementation. For much of this paper, we strive to remain in the realm of the required semantics. There are many cases in the standard in which diverging implementations can nonetheless produce the same semantic behaviors; here we must consider how the semantics themselves diverge.

```
typedef asymmetric_coroutine<int> coro_t;

void coro(coro_t::pull_type& source)
{
    before_suspend();
    source();
    int value = source.get();
    after_suspend();

}
```

N3985 coroutine

```
future<int> somefunc();

future<int> resumable()
{
    before_suspend();

    int value = await somefunc();
    after_suspend();
    return value;
}
```

D4134 resumable function

It's clear that on entry, both `coro()` and `resumable()` will call `before_suspend()`. The call to `source()` unconditionally suspends `coro()`; we presume that at least under some circumstances, `somefunc()` requires waiting on some external resource and will therefore suspend `resumable()`. Finally, on being resumed, both `coro()` and `resumable()` will call `after_suspend()`. These are the similarities.

The key difference is in *how* each function suspends.

For purposes of discussion, let us visualize a chain of function calls as growing downwards. That is, a program's `main()` stack frame is at the top. The stack frame for a function `foo()` called by `main()` is below `main`'s; that of a function `bar()` called by `foo()` is below them both.

Stackless Suspension

Caller Responsibility

A stackless resumable function must necessarily suspend by passing control *up-and-out.* That is, fundamentally, the function must somehow return control to its caller. This is not an implementation detail; it is required by the very nature of "stackless." Call it a constraint on the implementation.

That implies an obligation on the part of the caller to distinguish between ordinary return (value available) and suspension (no value yet). Consider a chain of resumable functions A → B → C. B must behave very differently in the two cases. If C is suspending, B must likewise suspend; when C returns a value, B can resume executing its own body.

There are several ways in which a resumable could communicate that distinction to its caller. For example:

1. The resumable function could suspend by throwing an exception. The caller could catch it, update its own resumption information and rethrow the exception to its caller in turn. Normal return from the resumable function would mean that a value is available. This has a certain straightforward appeal, though perhaps unfortunate performance implications.
2. The resumable function could return a type that might or might not contain a value, such as `std::future`. Prior art has used this tactic, leveraging `std::future::then()` to pass control back into a suspended resumable function once an awaited future is populated.
3. The resumable function could accept a non-const reference parameter through which to convey the distinction to its caller. This is similar enough to the preceding bullet that we will conflate the two.
4. The resumable function could set some "global" state – say rather, state in some per-execution-agent data area.

The essential point is this: regardless of implementation details, the code that calls a stackless resumable function *must* be aware that the function may or may not return a value. (However, a function that already returns (e.g.) `std::future` can be transparently converted to or from a resumable function: the caller need not care how the `future` is produced.)

Maintenance Implications

D4134 assures us that stackless resumable functions can seamlessly call into existing code, libraries and OS APIs without restrictions.

Consider a large existing code base. Parts of it use resumable functions to manage asynchronous I/O; other parts, which need not suspend, are straightforward C++14, written to synchronously return values rather than `std::future`s.

The unfortunate scenario is when we discover that, deep in a call tree in code which never before needed to suspend, we introduce a requirement to obtain a value asynchronously. At that time we are calling A → B → C → D → E → F; function F must obtain the new value.

We must therefore change F's signature, making it resumable.

Not only E, but *every* function that calls F, must be modified to become resumable itself, and to use the appropriate invocation for F. The same is true of every function that calls E, D, C, B or A.

This need to make large, pervasive changes for what should be local, incremental maintenance is the weakness of any stackless coroutine implementation. It's bad enough for a developer – but a nightmare for QA.

Stackful Suspension

A stackful coroutine suspends by passing control *down.* That is, it performs what it considers to be a perfectly ordinary function call. It delegates to the called function the actual mechanics of suspending and resuming. This is what permits a library implementation of stackful coroutines: the caller is compiled like any ordinary C++ function, unaware that magic may happen during a call to a particular other function.

That blissful ignorance is transitive. With stackful suspension, the caller of a function that may suspend is equally unaware. It too is compiled in the ordinary way.

It's true that N3985 coroutines accept a special coroutine parameter whose methods perform the actual suspension and data passing, and that this parameter is conventionally passed through every level of function call involved in any such suspension.

However, it's a short step to store a pointer to a "current" coroutine object in thread-local storage, eliminating the need to pass it explicitly. Another short step introduces a userland scheduler, likewise found via thread-local pointer. Such a scheduler can own a collection of coroutine objects, identify the "current" one and dispatch among ready ones.

In either scenario, a function running on a stackful coroutine need not be specially designated in any way. The compiler does not need to produce special code, either for the function's body or its caller's. Its signature need not reflect the possible need to suspend.

In our hypothetical large code base, if function F needs to introduce a new asynchronous call, F should be the only affected function.

Disjoint?

This difference between stackless suspension (up and out) versus stackful suspension (down) is the central distinction between them. We have not yet either imagined, or heard suggested, a way to unify that semantic disconnect between stackful and stackless.

Stacks, Processor or Simulated

But is there any possible middle ground between D4134 and N3985?

Here we wander from pure semantics into implementation concerns.

For clarity, let us speak of a function's "activation frame" as the place to find its return address, parameter values, local variables and miscellany pertaining to the currently-active call. Normally the activation frame is found on the processor stack. More on this later.

N3985 Coroutines

N3985 proposes a library implementation of stackful coroutines. Such a library necessarily relies on the runtime stack implementation of the program as a whole.

A typical runtime stack implementation requires a single contiguous memory area big enough to hold activation frames for the deepest possible call chain on that stack. Given data-driven recursion, this size cannot, in general, be statically predicted. Once that stack space is exceeded, we enter the realm of undefined behavior. The best outcome is for the program to crash; this can be coerced by appending a guard page with permissions set to forbid any access.

The severe penalty for exceeding reserved stack space leads to reserving large default stack areas. Of course a modern operating system can lazily commit physical memory to the stack on demand. However, as the stack must occupy a contiguous address range, that entire address range must be reserved even if relatively little of it is physically committed.

D4134 discusses the constraints this places on a highly-concurrent program (large number of distinct concurrent tasks) running in a 32-bit address space. It doesn't take very many 1MB stacks to exhaust available addresses.

D4134 further points out that overriding the default stack size to specify small contiguous stacks requires the coder to make promises that, in general, she cannot keep.

Recent versions of the Gnu C++ compiler introduce the notion of split stacks[iii]. This can mitigate the problem, but may still be too coarse: D4134 mentions that concept only to reject it. Moreover, Niklas states that Windows internal validation expressly forbids a noncontiguous processor stack.

In fact Gor asserts that the problem of virtual address consumption is the major obstacle to stackful coroutines. While acknowledging that in some respects they are more powerful and flexible than stackless coroutines, he laments the scalability demands that make them impractical for use on, say, a 32-bit server.

D4134 Resumable Functions

With D4134 resumable functions, a nested chain of suspended function calls conceptually resembles a linked list of individual activation frames, allocated on the heap or in a particular memory pool. Each such frame is minimal in size. Instead of 4KB pages, each resumable function, on entry, allocates exactly the memory it needs for its own activation frame. This frame is the resumable function's only overhead between the time it is suspended and the time it is resumed – which is where the high scalability comes from.

A stackless resumable function consumes no processor stack while suspended because suspension involves returning to its caller. This satisfies the Windows constraint about contiguous processor stack: the linked list of stack frames is distinct from the processor stack.

When a D4134 resumable function is resumed (e.g. via an awaited future's `then()` method), it is

reentered at the top. This creates a new processor stack entry. Generated logic interrogates the function's state in its small activation frame and dispatches to the appropriate resumption point within the function body. From there, every code path causes the function to exit, one way or another – once again removing its processor stack entry.

When the function eventually returns a result, this result is used to populate its returned future. Populating its returned future may pass control to another suspended resumable function – its original caller – which is, in turn, resumed.

This tactic simulates the behavior of the processor stack, while avoiding persistent entries on the processor stack.

Thought Experiment

In this realm, the compiler is fair game: we are not constrained to a library implementation. What if the compiler were directed to implement ordinary function call and return by constructing a linked list of activation frames, minimizing use of the processor stack?

Each function, on entry, would allocate its own activation frame on the heap. (D4134 discusses tactics for customizing the compiler's choices in such matters, such as the specific allocator used for the stack frame.) From the processor stack entry already constructed by its caller, the function prolog would pop key items into the new activation frame, notably its return address and parameters that must persist across calls to opaque functions. By the end of the function prolog, the processor stack pointer would be restored to its caller's value.

Return from such a function would fetch the return address from the heap activation frame, free the frame and jump back to the caller.

Moreover, such a function could call a suspend operation that would resume some *other* chain of activation frames merely by swapping the frame pointer before returning.

Per D4134, a function compiled this way could readily call existing library code or OS APIs; they would consume processor stack as usual – removed, as usual, on return to the calling function.

The compiler could be directed to compile an entire translation unit this way, or could accept #pragma directives, following existing precedent.

Ignorant imagination suggests that a function compiled this way could be less complex – and possibly even more efficient – than the dispatch tactics proposed in D4134. It could be suspended using the "stackful" paradigm, that is, by making an ordinary function call to a not-so-ordinary function – instead of arranging two distinct ways to return. Resumption would not require dispatching to the proper place in the function body; the called function would simply return to the proper place.

This "stackful" functionality would nonetheless use heap-allocated, minimum-size activation frames for scalability as in D4134. Participating functions would not require special signatures, special calling code or special source annotation, easing the maintenance burden.

One might ask: what would be the effect of calling A → B compiled as now, then B calling C → D compiled for linked activation frames, then D calling E → F compiled as now, then F calling G → H compiled for linked activation frames? What would we say about suspending function H?

Obviously A, B, E and F consume processor stack as usual. We *require* that the entire call chain A → B

→ C → D → E → F → G → H be semantically indistinguishable from the case in which all are compiled as now – suspension aside.

We would expect that the lowest-level activation frame resident on the processor stack (in this case, F's) defines a boundary below which independent linked-activation-frame coroutines can cooperatively share the same kernel thread. In this example, we would disregard the fact that C and D have been compiled for linked activation frames: the processor stack frames for E and F render that moot.

It seems entirely plausible to introduce debug-mode diagnostics to catch inadvertent "anchoring" of the processor stack this way.

Summary

D4134 engages compiler support to produce stackless coroutines. This promises high scalability, at the cost of pervasive markup: significant overhead to port code into that environment.

N3985 proposes a library implementation for stackful coroutines. This promises earlier availability (published implementations available now!) plus an easier maintenance burden – at the cost of large reserved address ranges per task, limiting scalability.

It seems possible that compiler support for "stackful" coroutines could provide the benefits of both approaches, leveraging technology prototyped for D4134. This is not itself a proposal, merely a recommendation to explore that design space before committing to either existing proposal.

References

i N3985: A proposal to add coroutines to the C++ standard library
ii N3977, N3858: Resumable Functions; D4134: Resumable Functions v2
iii Split Stacks in GCC