On unifying the coroutines and resumable functions proposals

Revision 1

Document number:	P0073R2
Date:	2016-05-30
Author:	Torvald Riegel
Reply-to:	Torvald Riegel <triegel@redhat.com>
Audience:	EWG

1 The goals of this paper

I have argued in the past that there are commonalities between the different coroutine proposals and that there are opportunities for a unified proposal. This paper is meant to provide more detail about this.

So far, the proposals have been presented as separate vertical solutions, and it has been claimed that there is no substantial common ground between them. However, I believe we should be looking for commonalities, or we will both make future extensions of the features presented in these proposals harder and will make all of them harder to learn for programmers.

To me, the design of coroutines and resumable functions is not just in the space of programming interfaces. A large part of it is also at the level of the abstract machine: In particular, how we deal with concurrency beyond `std::thread`? While it's easy for programmers to use `std::thread`, they often don't need a full-featured operating system thread nor would want one because of additional, unnecessary overhead that might be attached to it; a thread of execution from a thread pool, or something like a coroutine would often be sufficient and better. I believe there is consensus in SG1 that we need more kinds of threads of execution beyond those spawned by `std::thread` (see, for example, the parallelism TS or P0072R0).

The main points I'm arguing for in this paper are:

- Executing coroutines, generators, etc. should be understood as representing threads of execution, with certain execution properties. Note that I did *not* say that they should be OS threads or similar to what `std::thread` spawns.

- Resumable functions enable a certain implementation of threads of execution, in particular how the call stack is implemented. The so-called stackful coroutines use a more traditional stack implementation.

- The compiler can bridge the gap between those two stack implementations under certain conditions by providing the impression of a stackful implementation (e.g., calling through normal non-resumable functions) but generating code that is using resumable functions internally. The Transactional Memory TS is a precedent for very similar compiler support, showing that it can be both specified and implemented.

In this paper I will not discuss interfaces of programming abstractions, except for design examples in Section 4. Personally, I am more concerned about the design at the conceptual level than about details of a particular interface; my focus is on the internals, both design-wise and how it affects implementations or enables certain ones.

2 Coroutines represent threads of execution

Let us first revisit what a thread of execution is according to the standard, and abbreviate it with *TOE* instead of "thread" to avoid the confusion over what a "thread" is (e.g., an operation system thread?). A TOE is defined in the standard as "a single flow of control within a program" (see §1.10p1). *sequenced-before* is defined exactly for a TOE (see §1.9p13). The standard seems not quite clear regarding whether a TOE is a static (e.g., a function) or dynamic entity (i.e., an instance of an execution of a single flow of control); in this paper, I will assume the latter. See N4231 for a more detailed explanation of this terminology (and of the existing terminology in the standard). N4231 and P0072R0 use the term "execution agent" as name for TOEs that have specific execution properties attached to them. The standard uses execution agents to describe lock ownership of TOEs. P0072 argues for lighter-weight modes of execution provided through different kinds of execution agents. "Execution context" was suggested as a different name instead of execution agent, which might make it clearer that the intent is to model execution properties and not primarily to describe the hardware or software resources used for execution. Nonetheless, for simplicity, I will just use TOE in this paper, and associate execution properties with TOEs.

I think that we should embrace parallelism and concurrency—not just by providing programming abstractions in this area but also by being honest when and where parallelism or concurrency exist in a computation. We should not shy away from having multiple TOEs exist at the same time in a running program. We do not need to be afraid of this because we can still have simple forms of interleavings between TOEs; not all forms of concurrency are of the hard-to-handle, low-level kind. Disjoint-access parallelism (e.g., partitioning an array and processing the disjoint partitions in parallel) is a good example of a case that has plenty of TOEs but is still easy to understand for programmers. Another example is that for all of us, having a dialogue with another person is a completely natural thing to do.

Applied to coroutines, this means in my opinion that we need to understand coroutines as TOEs. On the level of program logic, there is concurrency in the sense of that a coroutine is an *individual* flow of control in the program, so matching the standard's definition of a TOE. I do not think that it is helpful to try to hide this fact by taking the mutual exclusion property of a coroutine and its callers and use it as a reason to merge the coroutine and the caller into one TOE.

Thus, when creating a coroutine and calling it for the first time, we conceptually spawn a TOE that will execute the separate flow of control that the coroutine is meant to represent (or that even is the coroutine). Constructing a `std::thread` will also spawn a TOE, but a TOE of a different kind with different execution properties (e.g., it is guaranteed to execute steps eventually irrespective of what other TOEs do, which is not the case for coroutines).

Understanding coroutines as TOEs is important for three reasons. First, SG1 has discussed many parallelism and concurrency abstractions that create TOEs: Parallel algorithms from the Parallelism TS, executors, task regions, etc. Those TOEs have execution properties that are often lighter-weight than threads, for example regarding aspects such as forward progress guarantees, thread-specific state, or how call stacks are implemented (see below for more details on these). Those potentially lighter-weight properties can be common across TOEs spawned by very different high-level abstractions (e.g., same weaker-than-`std::thread` forward progress guarantees for parallel-vector loops and generators, see below for details).

Second, there often is no single right choice for what kind of TOE to spawn from a higher-level abstraction. For example, which stack implementation does one really need for a coroutine? Or which support for thread-local storage is required? This applies to coroutines and generators as well.

Third, TOEs can be considered the basic entity of computation, when looking at the implementation level and control over where something executes. The executor proposals try to provide programmers with such control, and while it is not always obvious from the programming interfaces, often this control happens by tuning or managing TOEs (e.g., through thread pools). Thus, if coroutines are TOEs, it becomes cleaner to apply executor features to them as well. Note that while often it may be most natural to resume a coroutine using the same compute resource as the caller, this isn't necessarily always best: Imagine a generator that needs to access lots of data to provide a result; this generator is most efficient to run on a CPU close to the accessed data in terms of the memory hierarchy. A similar example can be made in the space of accelerators. Also note that the basic scheme of interaction with the coroutine (e.g., the interleaving) is still the same in this case, so where the coroutine's TOE is executed is really an orthogonal property.

Defining different kinds of TOEs thus gives us a foundation upon which higher-level constructs can be specified and built, including constructs such as coroutines. Having this common base makes it easier for us and easier to grasp for users.

Of course, coroutines are not just TOEs in the sense of the TOE being independent of anything else. Like other abstractions such as parallel loops that spawn TOEs, coroutines give additional guarantees such as mutual exclusion between the coroutine's TOE and

the TOE represented by the caller of the coroutine.

2.1 Examples for specific properties of execution

Before looking at the stack implementation aspect in detail next, I want to give examples using some of the other properties.

Forward progress guarantees of TOEs (or of execution agents) are discussed in P0072R0. It is interesting to see that a lock-step execution of a vector-parallel loop and a typical coroutine execution can both be characterized as non-preemptive, collaborative scheduling. In terms of forward progress, the TOEs spawned by the loop and the TOE representing the coroutine all are weakly parallel TOEs as defined by P0072R0, with boost blocking by either the TOE calling the loop or the coroutine. Thus, one set of properties can describe the TOEs spawned from different abstractions.

Thread-specific state such as thread-local storage (TLS) or lock ownership are another example: SG1 has been discussing how (and whether at all) TOEs used for parallelism should support TLS (see P0072R0 for more background). There is no single right answer because while TLS might not be easily supportable on an accelerator, it sometimes is just required by code; yet seldomly does it seem necessary to run all TLS constructors and destructors for such TOEs, as would be required by `std::thread`. Right now, the coroutine proposals and the resumable function proposal make one specific choice, and they state this rather implicitly (e.g., through stating that a thread used to execute code could change, or just through leaving open which thread executes the code). It would be better in my opinion to provide choice to programmers, so that they can pick the TLS semantics they really need.

Lock ownership is a similar example to TLS. Should a generator inherit and modify the lock ownership of the TOE that called it most recently, or should it have it's own set of lock ownerships? There does not seem to be one right answer to this because it really depends on what the program tries to accomplish.

3 Call stack implementations

The major points that characterize a TOE with non-preemptive suspension are:

1. What happens before and after suspension,
2. How hand-off or hand-shake happen between this TOE and other TOEs on creation, suspension, and termination of this TOE, and
3. Whether there are any constraints on the code that is running in this TOE.

Such a TOE is then used by, say, a coroutine with a specific interface. In the discussions I had so far on this topic, I got the impression that for some people, the coroutine interface is tied to a specific choice regarding the three points above and vice versa—which seems to match with the separate vertical coroutine proposals that we currently have on the table.

I do not think this has to be the case. While I can agree that some combinations might be easier to implement and allow one to optimize this or that slightly, all a coroutine basically needs is a non-preemptively suspendable TOE.

Furthermore, and as the discussions in the committee show, there are different use cases for coroutines that evaluate the choices made regarding the three points above differently. For example, in some use cases it is more important to not rely on inter-procedural compiler transforms whereas in others it could be vital to call functions without having to duplicate and modify them at the source code level.

Thus, I think it would be helpful to users if they would have choice regarding how the call stack and suspension implementation used for a coroutine looks like; the more flexibility we have in the coroutine building blocks, the better.

This would allow for *unifying* the coroutine proposals. To back up my claim that this is possible, I will next outline how the call stack implementation can provide the properties that the programmer is asking for and allow a programmer to combine them.

Stackful coroutines (e.g., N4232) allow the coroutine TOE to call arbitrary functions and support suspension points in functions called by arbitrary functions. Thus, they give the impression of a "normal" TOE that can just call any code. Suspension points are transparent as far as the calleable code is concerned.[1] Side stacks of various forms or normal OS threads are valid implementations.

Resumable functions (e.g., N4499) are a lower-level mechanism because they require suspension points to be treated specially in the calling code, including functions that call other functions that may suspend. Thus, suspension is not transparent but requires functions to be annotated and use the `await` keyword on *all* function calls that may result in a suspension. The goal of this is to enable a specific call stack implementation, namely one that compresses the space required for the stack by (1) keeping only the essential live-variable information for each stack frame on the side, (2) rolling back all stack frames before suspension (i.e., returning from those functions), and (3) reconstructing stack frames after suspension using the live-variable information. Compiler support is required, but only of the intra-procedural kind.

So, simplified, resumable functions have the advantage of a potential performance improvement (depending upon use, however—they are not always faster), while stackful coroutines have the advantage of minimizing impact on existing code and improving the likelihood of code reuse.

4 Bridging the gap between stackful and resumable

Given that both options have their uses and advantages, it would be good if programmers do not have to choose between those two extremes. We can make this happen through relying on a bit more compiler support. In particular, we want the compiler to generate code that uses the approach of resumable functions internally while putting as little requirements on programmers as possible to actually change the source code.

[1] But they are not transparent regarding forward progress guarantees and thread-specific state, depending on which choices the specific coroutine feature makes regarding these execution properties.

The first step towards this is allowing the programmer to express the same intent behind a resumable function, while keeping the annotations or changes required on code small. When using a resumable function, the programmer is essentially asserting that:

- All blocking synchronization is compatible with the forward progress scheme of the program (e.g., no deadlock due to blocking on a TOE that is not guaranteed to make progress). This is very much program-specific; whereas one application may care about millisecond latencies, another may just need eventual progress and thus can allow blocking for I/O. Thus, what this entails is not specified by resumable functions.

- Any attempt to suspend a coroutine in a stackless manner is from a call stack consisting of just resumable functions; for a coroutine TOE, this means that all suspension points must be reachable only through resumable functions, starting at the initial function for the coroutine.

The latter point is partially enforced in resumable functions because (1) such functions are entities isolated from their callers[2] and (2) `await` is only allowed in a resumable function. This isolates these functions from the context their are called in, translating any suspend operation into a non-ready state of the futures that must be returned from resumable functions. This is only a partial enforcement because nothing is stopping a resumable function from blocking on a future returned from a call to another resumable function (i.e., not using `await` but, for example, calling another non-resumable function that blocks until another resumable function produces it's final result; at the least this prevents stack frame rollback).

Thus, having to return a future is not inherent to a stackless TOE implementation nor required by what the programmer intends or has to assert. Instead, it is an artefact of a specific implementation—which is just one among other possible implementations. This also applies to how the `await` keyword is designed, which is a combination of the suspend action and this future-based implementation. The general combination of suspending and telling the scheduler what one is waiting for is of course useful—but this does not mean that it has to be applied at the function call level within a coroutine.

These implementation artefacts are what is primarily preventing code reuse: There is no way to just call through an existing function without having to write a resumable version of it that applies the future-return-type approach and uses await at each call that may lead to a suspend action.

4.1 Suspendable functions

We know we need different code for resumable than for normal code that doesn't suspend[3], but we can let the compiler take care of that if the programmer annotates the

[2] Resumable functions have to use a future-based interface that makes each function call within a coroutine look similar to blocking on another coroutine.

[3] We need different code that rolls back and restores stack frames to be able to suspend stackless; remember that resumable functions are an enabler for a certain optimization.

affected functions with a `suspendable`[4] keyword. The programmer adds this annotation on each function that she or he would write as a resumable function when using the resumable function proposal. No change to the function body or return type is required, and the annotation just replaces the `resumable` annotation required otherwise[5]. To avoid confusion with resumable functions, let us call such functions suspendable (and likewise for suspendable coroutines and suspend operations)—but note that the goal is still to be very similar in functionality to resumable functions. The programmer-supplied annotations must ensure that each suspendable suspension point is only reachable from the initial function of a suspendable coroutine through functions annotated as `suspendable`; otherwise, the program is incorrect.

Functions annotated this way can execute both normally and in a suspendable mode. One possible implementation of this is to, for each suspendable function, generate normal code for such a function as well as a clone of the function that is compiled differently and, for example, has a different mangled name.[6] The suspendable mode is only accessible when using a suspendable coroutine (see below). This is very similar to the approach used in the Transactional Memory TS, where we have normal code and clones of such code instrumented for use in a transaction, and where only transactions will ever cause execution of the transactional clones. This is just one precedent currently but we will likely have to use this approach more often if we ever want to support hardware heterogeneity fully (e.g., different ISAs for both host and accelerator CPUs) without relying on JIT compilation. Some of the vectorization proposals discussed in SG1 also propose support for requesting different clones of functions that only differ in which hardware vector extensions are targeted.

Because it matters that only the suspendable clones are executed when in suspendable mode and because we want to take care of as much as possible of this at compile time, we need to cover indirect calls too. Fortunately, that is almost the same problem as for transaction-safe functions in the Transactional Memory TS; thus, we can just copy this TS's approach, which has already been approved by the committee and implemented in GCC. In a nutshell, the `suspendable` annotation becomes part of the type system, and a call to a suspendable function in suspendable mode (either through a pointer or to a function known at compile time) will call the target function in suspendable mode too (e.g., call the suspendable clone of the function). See the Transactional Memory TS for details. Different to transaction-safe functions, suspendable functions are allowed to call non-suspendable ones, same as resumable functions are allowed to call non-resumable ones.

[4] This is a placeholder name; `resumabler` was also considered.

[5] P0057R0 does not yet require it, but several people in the committee seemed to want to require that all resumable functions are explicitly annotated as that when declared.

[6] Unused clones (or normal functions) can of course be removed by the compiler/linker in the same way as other dead code can be removed.

4.2 Suspending and executing suspendable functions

The implementation can generate any code it sees fit for suspendable functions, but we still need to expose two features to programmers: Executing a suspendable coroutine, and suspending a coroutine.

First, starting to execute a suspendable function basically involves setting up whatever the implementation needs to communicate about the state of the coroutine (e.g., constructing a future) and then calling the code created for suspendable mode (e.g., the suspendable clone of the top-level function of the coroutine). Resuming a suspendable coroutine is similar but will use the coroutine state to restore from the proper point. The details of all of this are internal to the implementation (e.g., in a combination of the generated code and potentially special support in futures or a scheduler).

All we really need to minimally expose to the user is a spawn-like call that let's the program request that a new coroutine TOE is started that uses suspendable mode. For example (a keyword or other forms would be possible too):

```
template <typename F> std::future<void> run_suspendable(F&& f);
```

This expects that the passed function is suspendable, and it will execute it in suspendable mode. Higher-level abstractions can make common use cases more convenient.

Second, suspending is minimally just a suspendable function that is called to suspend:

```
void suspend() suspendable;
```

This is possible because the stackless implementation is hidden behind the `suspendable` annotation, so the implementation can just do whatever is required in the suspendable implementation of this function.

It can of course be useful to communicate to a scheduler why a coroutine would want to suspend (e.g., because a generator has just produced a value, or in an `await`-like blocking operation). But this is orthogonal to the stackless aspect and can be implemented in higher-level abstractions that use this `suspend` function internally. Also see the definition of `suspend-resume-point` in P0057R0, which is similar.

4.3 Unification with stackful coroutines

The constructs laid out so far can be implemented so that they use resumable functions internally: suspendable functions can be transformed into resumable functions by the compiler, `run_suspendable` calls such a resumable function, and `suspend` can do parts of `await`. There are details in the implementation that depend on how the higher-level abstractions are designed (e.g., how much can be communicated to a scheduler), but I think it should be clear that the implementation possibilities are comparable to what the resumable function proposal offers. Most importantly, the primary goals of a stackless implementation and low space overhead are achievable, while requiring much less invasive changes to existing code or specialization of new code.

Furthermore, a correct implementation of suspendable functions is a stackful implementation—any potential[7] space overhead differences are a quality-of-implementation aspect. The

[7] Note that the space overhead for the stack pages of a stackful implementation might not exist in less

normal code (i.e., not the suspendable execution mode) is sufficient because stack frames do not need to be rolled back in a stackful implementation. `run_suspendable` could also be used to spawn stackful coroutines, which also allows the implementation to make a choice (e.g., depending on what's most efficient or possible on the host or accelerator CPU).

The semantics of the `suspend` function are the same irrespective of whether it is a suspendable or stackful coroutine. Because it is simply a suspendable function, its implementation can do the right thing depending on whether it is called in suspendable execution mode or not. If it is called from a TOE that is not a coroutine, it could either simply block this TOE or result in a runtime error. The former is most useful when combined with information from a higher-level construct regarding why the TOE is suspended (e.g., an `await`-like construct could use a traditional blocking synchronization operation automatically (i.e., without non-preemptive suspension), as has been suggested by Chris Kohlhoff). This possibility is not a new case to be considered for resumable functions, because nothing prevents those from using a, say, condition variable to block on some condition.

4.4 Allowing for more code reuse

Only needing to use the `suspendable` annotation on function declarations is already a big improvement compared to having to write or rewrite a function as a resumable function specifically. It is not ideal though because (1) it still requires changes in the code and (2) it does not allow reuse of a function in a suspendable context whose declaration cannot be changed by the programmer.

At the beginning of Section 4, I listed the two assertions that a programmer is making when using resumable functions. The first one is about blocking synchronization generally, and while the compiler could help with that, it is hard because program-specific constraints will vary a lot. However, the compiler can help with the second assertion: ensuring that any attempt to suspend in a stackless manner is done in the suspendable execution mode (i.e., on a call stack consisting of just suspendable functions).

To achieve the latter, we can introduce an `autosuspendable` annotation that a programmer can put on a function. An auto-suspendable function is always a suspendable function, but it additionally requires the compiler to treat all callees of this function that are not annotated as `suspendable` functions as if they were marked by the programmer as auto-suspendable. This ensures that all callees, transitively, will either have been explicitly taken care of by the programmer (i.e., if the programmer asserts they are suspendable), or will be conservatively considered to be suspendable (i.e., treated as if marked as auto-suspendable).

Consider the example in Figure 1, which runs a suspendable coroutine with minimal annotations required, and reuses the code for a second coroutine that cannot be made suspendable: If using the resumable functions proposal, `produce` and `coroutine1` would have to be written differently, and `external_for_each` would have to accept

typical hardware implementations, for example on accelerators; in these cases, a stackless implementation might not yield any gains or might even be a little slower if call stacks are deep.

```
void produce() {
  // Do something, produce a result that can be consumed:
  // [...]
  // And suspend:
  suspend();
}

void compute() {
  // Just do a bit of math here ...
}

void coroutine1() autosuspendable {
  compute();
  produce();
}

void coroutine2() {
  // Third-party code that will call the callback (produce)
  external_for_each(data, produce);
}

void runner() {
  auto f1 = run_suspendable(coroutine1);
  auto f2 = run_stackful(coroutine2);
}
```

Figure 1: Example of code reuse between suspendable and stackful coroutines.

callbacks that have the interface of resumable functions. In contrast, with the unified approach outlined in this paper, this example just works because the compiler will make `coroutine1` and `produce` suspendable functions, and `suspend` will work in suspendable mode and also in stackful mode when called through third-party code that's just available as binary (`external_for_each` via `coroutine2`).

This does not mean that the compiler has to create special code for callees that will never suspend (e.g., `compute` in Figure 1); if the compiler has checked that there is no way to call `suspend`, it can just use normal code.[8] However, it requires a compiler to be able to analyze the code, which it might not be able to do across compilation units; in such a case, a compiler is allowed to produce a compile-time error. Programmers can then use the `autosuspendable` or `suspendable` annotation to make this work across translation units.

Note that, again, `suspendable` can also be used to assert that a function will never attempt to suspend in the suspendable execution mode. However, this could result in more suspendable clones being created than necessary, so we might want to consider adding another annotation to cover this use case specifically (e.g., `nosuspendable`).

For the compiler implementation, this requires analysis of the call graph of each `autosuspendable` function. This extends the intra-procedural analysis that is already required for resumable functions[9] to an inter-procedural analysis. Such compiler support needs to be implemented, but is not magic either. The Transactional Memory TS is a precedent for requiring such support, so if a compiler implements or wants to implement this TS, it does or will have to do something very similar already.

There are difficulties with inter-procedural analyses, such as when crossing into different translation units. Nonetheless, these are easier to solve for an implementation than in the case of the analysis required by the Transactional Memory TS because strictly speaking, using the suspendable execution mode is an optimization—the implementation can always use a stackful implementation if it cannot analyze a part of the call graph (and it can warn about this at compile time). This highlights why it can be useful to not expose the implementation details of coroutines to a program.

5 Making executors aware of blocking

Previously, I have shown that both stackful and stackless coroutines can use the same code, including the places where they get suspended. This is important but in order to run them efficiently, their scheduler, for example the executor whose resources they run on, needs to know when it is allowed and necessary to resume execution after suspension. Therefore, we need to make the executor aware of blocking relationships and conditions of coroutines.[10]

[8] Of course this assumes a correct program, including a correct use of the `suspendable` annotation.

[9] Note that this depends on whether functions that contain `await` need to be annotated or not; in the discussion of resumable functions in Lenexa, some people wanted the annotations whereas others thought they were unnecessary.

[10] A `std::thread` TOE's scheduler doesn't need to be made aware explicitly to the same extent, because of the stronger forward progress guarantees and because blocking is visible to it through OS-level

Next, I will outline possible low-level support for this, constrained to what is necessary to implement a generator. Other variants of such support are possible too – the focus of what follows is on things the executor and implementation need to know or need for performance, and not on what is the most convenient and polished design for users. In other words, the target audience is library implementers. Higher-level interfaces for this would be what we would expose to the average user. Note that what follows has similarities to await and synchronous coroutines, which is futher indication that those are compatible with suspendable functions and can be unified.

One requirement for the low-level support is that it is usable from different TOEs (i.e., `std::thread` as well as stackful and stackless coroutines). This is why it is based on blocking, which they all can do. Note that the blocking I'm referring to here is of the program-logic kind: A TOE cannot continue to do the things it wants to do until some condition is met (see P0296R1 for a clarified definition of blocking in the standard).

Blocking is important for an implementation because when blocked, resources of the TOE can often be used elsewhere:

- a spin-waiting TOE can slow down or pause for a while and thus let adjacent CPU cores run more cycles,
- an OS thread / `std::thread` can be descheduled (e.g., what might happen on a FUTEX_WAIT system call on Linux, when a condition variable implementation blocks),
- a stackful coroutine can suspend and allow for its OS thread to be used to execute another stackful coroutine, or
- a stackless coroutine can suspend and thus let both its OS thread and stack space be used for another coroutine.

The other aspect of blocking that is important is what condition a TOE is blocked on and which TOE(s) can satisfy this condition. If an implementation is aware of this, it can schedule accordingly. Therefore, the support outlined next follows a pull-style model that makes the TOE that another TOE is blocked on known a priori to the executor.

This allows for convenient handling of forward progress by specifying that blocking (i.e., the pull operation) always has blocking-with-forward-progress-delegation semantics (see P0229R0). Other models would be possible too, but then we need to handle forward progress differently (e.g., execute all coroutines on an executor until none can run anymore or some coroutine satisfies the blocking condition).

We can use a simple `event` class to model an event that a TOE can wait for. For simplicity, and to avoid synchronization overheads, let us assume that this is a single-sender single-waiter event, and that the waiter is responsible for destruction:

```
class event
{
  void signal();
}
```

blocking mechanisms such as futexes on Linux.

An event is always associated with the TOE that constructed it, and the `signal` member function must only be called by this TOE. This makes executors aware of which TOE can satisfy an event, and thus must execute to unblock another TOE blocked on this event.

Next, we provide two blocking functions:

```
void block(event until);
void block_and_unblock(event& until, event& signal);
```

`block` blocks the calling TOE until the event has been signaled. Coroutines (stackful and stackless) may always suspend during this call, even when the condition is satisfied (this gives the executor more leeway to schedule—remember that we schedule cooperatively). Also, when we block, the calling TOE delegates its forward progress guarantees and thus potentially strengthens the forward progress of the TOE it depends on. We assume that the scheduler takes care of synchronization, as explained below; we could provide events that allow for more concurrent execution but this isn't necessary for what I want to show here.

`block_and_unblock` is a specialized version of `block` in that it both blocks and signals another event in one indivisible step. This captures what a "call" operation would do in a model of synchronous coroutines. For simplicity, we require that the TOE we block on (i.e., `until`) is the same that we will signal. This also tells the executor that it can simply stop executing the calling coroutine and resume the coroutine this will block on.

To spawn a coroutine, we provide this function:

```
template<class F> event spawn_coroutine(F func, event& block_on);
```

This will spawn a coroutine that will execute `func`, and will create a stackful or stackless one depending on whether `func` is a suspendable function. Before starting execution, the coroutine will block on `block_on`. This function returns an event that is associated with the coroutine, and can be used by `func` to send its first sign of life.[11] The spawned coroutine has weakly parallel forward progress requirements (see P0299R0), which is sufficient because we use blocking-with-forward-progress-guarantee-delegation to make sure it gets executed when needed.

Finally, for convenience, we add another function that requests the executor to signal an event right after a coroutine has terminated. This just reduces the number of suspensions that are necessary, but is a good example of things the executor can do to improve performance by being aware of blocking relationships.

```
void unblock_on_toe_exit(event& signal);
```

With these functions available, we can build and use a generator as shown in Figure 2. The `generator` class simply models the lifecycle of a generator together with a crude way to return data until no more data can be generated. The constructor only sets up the coroutine but does not yet execute it; note that we set up the events such that one is associated with the generator and one with the caller of the generator, thus building a pattern similar to a synchronous coroutine.

[11] In the generator example below, this is all we need, but the coroutine could also set it up to signal that it is done initializing more events.

```
template <typename T, class F>
class generator
{
  event run;
  event data_ready;
  bool hd;
  T data;

  // This associates run with the TOE that created the generator, and
  // data_ready will be associated with the spawned coroutine.  The
  // coroutine will block on run before starting to execute.
  generator(F func) : run(), data_ready(spawn_coroutine(func, run), hd(true)
  {}

  produce(T data, bool has_more_data)
  {
    generator.data = data;
    if (has_more_data)
      block_and_unblock(run, data_ready);
    else
      {
        unblock_on_toe_exit(data_ready);
        hmd = false;
      }
  }

  T consume()
  {
    block_and_unblock(data_ready, run);
    return generator.data;
  }

  bool has_data() { return hd; }
};

int count_to_10() autosuspendable
{
  for (int i = 1; i < 10; i++) generator.produce(i, (i < 10));
}

// Note that autosuspendable is not necessary but just there to make user()
// also callable from a suspendable function
void user() autosuspendable
{
  generator<int> gen(count_to_10);
  while (gen.has_data())
    gen.consume();
}
```

Figure 2: Example for how to build and use a generator.

`consume` is called from the "caller" of the generator and just blocks the calling TOE and unblocks the generator coroutine (as one indivisible step). This will start or resume the coroutine (the event `data_ready` is used for both).

`produce` is called from within the generator and publishes the generated data. If there is more data, it blocks until the consumer requests more data and unblocks the consumer so it can consume the data. If no more data will be generated, it will set a flag accordingly and unblock the consumer when the coroutine TOE has exited, which avoids another suspension and switch between consumer and generator just to let the generator exit.

The `user` function then makes uses of this by running `count_to_10` as a generator. Note that this can be run as a stackless coroutine because `count_to_10` is a suspendable function. Likewise, `user` can be called from either a `std::thread`, a stackful, or a stackless coroutine because blocking is well-defined for all of these. The executor of the TOE calling `user` would execute the coroutine as well (e.g., if this is a `std::thread`, the generator would run on this OS thread).

In the case of `std::thread`, forward progress is guaranteed because of the concurrent forward progress guarantee that `std::thread` gives (see P0296R0); in the case of coroutines, forward progress is as strong as forward progress of the coroutine (which in turn may be strengthened by another TOE delegating its forward progress guarantees).

Also note that because the implementation (e.g., the executor) is aware of the blocking relationships and the `block_and_unblock` pattern, no use of atomic operations is necessary—the implementation simply knows that there is no concurrency not under its own tight control, so it can do synchronization via scheduling.

To conclude, I'd like to reiterate that this section was not supposed to present features that you have never seen before in another proposal—instead, it is meant to show that by focusing on the common base of the coroutine proposals and TOE in general (i.e., blocking, forward progress, suspension when blocking, knowledge that is helpful to the scheduler and the implementation), we can unify the coroutine proposals and also avoid making them unneccesarily different from the TOE we already have (i.e., `std::thread` and the thread that runs `main`).

6 Revision history

Changes between R1 and R2:

- Added Section 5.

Changes between R0 and R1:

- Added proposal for an alternative way to write resumable functions (split out as Section 4).