Document No: D4134
Supersedes: N3977
Date: 2014/09/05
Reply to: Gor Nishanov (gorn@microsoft.com), Jim Radigan (jradigan@microsoft.com)

Resumable Functions v.2

Contents

Revisions and History

This document supersedes N3858 and N3977 and elaborates on extensibility and scalability of resumable functions.

Terms and Definitions

Coroutine

A generalized routine that in addition to traditional routine operations such as invoke and return supports suspend and resume operations.

Coroutine State / Coroutine Frame

A state that is created when coroutine is first invoked and destroyed once coroutine execution completes. Coroutine state includes a coroutine promise, formal parameters, variables and temporaries with automatic storage duration declared in the coroutine body and an implementation defined platform context. A platform context may contain room to save and restore platform specific data as needed to implement suspend and resume operations.

Coroutine Promise

A coroutine promise contains library specific data required for implementation of a higher-level abstraction exposed by a coroutine. For example, a coroutine implementing a task-like semantics providing an eventual value as std::future<T> is likely to have a coroutine promise that contains std::promise<T>. A coroutine implementing a generator may have a promise that stores an endpoint of a channel that coroutine can use to yield values to the consumer.

Coroutine Object / Coroutine Handle / Return Object of the Coroutine

An object returned from an initial invocation of a coroutine. A library developer defines the higher-level semantics exposed by the coroutine object. For example, generator coroutines can provide an iterator interface that allows to consume values produced by the generator. For task-like coroutines, coroutine object can be used to obtain an eventual value (future<T>, for example).

Generator

A coroutine that implements a high-level abstraction of an input iterator or an input channel. The body of the generator coroutine uses a **yield** expression to specify a value to be passed to the consumer. Emitting a value suspends the coroutine, invoking a pull operation on a channel resumes the coroutine.

Stackless Coroutine

A stackless coroutine is a coroutine which state includes variables and temporaries with automatic storage duration in the body of the coroutine and **does not** include the call stack.

Stackful Coroutine

A stackful coroutine state **does** include the full call stack associated with the execution of the coroutine enabling suspension from nested stack frames.

Split Stack / Linked Stack / Segmented Stack

A compiler / linker technology that enables non-contiguous stacks.

Resumable Function

Proposed C++ language mechanism to implement stackless coroutines.

Discussion

Motivation for extending C++ language and libraries to support coroutines was covered by papers N3858 (resumable functions) and N3985 (A proposal to add coroutines to C++ standard library) and will not be repeated here.

Design goals for this revision of resumable functions were to extend C++ language and standard library to support coroutines with the following characteristics:

- Highly scalable (to millions of concurrent coroutines)
- Highly efficient resume and suspend operations comparable in cost to a function call overhead
- Seamless interaction with existing facilities with no overhead
- Open ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics, such as generators, goroutines, tasks and more.
- Usable in environments where exception are forbidden or not available

Unlike N3985 (A proposal to add coroutine to the C++ standard library), which proposes certain high-level abstractions (coroutine-based input / output iterators), this paper focuses only on providing efficient language supported mechanism to suspend and resume a coroutine and leaves high-level semantics of what coroutines are to the discretion of a library developer and thus is comparable to Boost.Context rather than Boost.Coroutine / N3985.

Stackless vs Stackful

Design goals of scalability and seamless interaction with existing facilities without overhead (namely calling into existing libraries and OS APIs without restrictions) necessitates stackless coroutines.

General purpose stackful coroutines that reserve default stack for every coroutine (1MB on Windows, 2MB on Linux) will exhaust all available virtual memory in 32-bit address space with only a few thousand coroutines. Besides consuming virtual memory, stackful coroutines lead to memory fragmentation, since with common stack implementations, besides reserving virtual memory, the platform also commits first two pages of the stack (one as a read/write access to be used as a stack, another to act as a guard page to implement automatic stack growth), even though the actual state required by a coroutine could be as small as a few bytes.

A mitigation approach such as using split-stacks requires the entire program (including all the libraries and OS facilities it calls) to be either compiled with split-stacks or to incur run-time penalties when invoking code that is not compiled with split-stack support.

A mitigation approach such as using a small fixed sized stack limits what can be called from such coroutines as it must be guaranteed that none of the functions called shall ever consume more memory than allotted in a small fixed sized stack.

Implementation Experience

We implemented language changes described in this paper in Microsoft C++ compiler to gain experience and validate coroutine customization machinery. The following are illustrations of what library designers can achieve using coroutine mechanism described in this paper.

Asynchronous I/O

The following code implements zero-overhead abstractions over asynchronous socket API and windows threadpool.

```
std::future<void> tcp_reader(int total)
{
	char buf[64 * 1024];
	auto conn = await Tcp::Dial("127.0.0.1", 1337);
	do
	{
		auto bytesRead = await conn.read(buf, sizeof(buf));
		total -= bytesRead;
	}
	while (total > 0);
}

int main() { tcp_reader(1000 * 1000 * 1000).get(); }
```

Execution of this program incurs only one memory allocation[1][2] and no virtual function calls. The generated code is as good as or better than what could be written in C over raw OS facilities. The better part is due to the fact that OVERLAPPED structures (used in the implementation of Tcp::Dial and

[1] Allocation of the frame of a resumable function

[2] not counting memory allocations incurred by OS facilities to perform an I/O

conn.read) are temporary objects on the frame of the coroutine whereas in traditional asynchronous C programs OVERLAPPED structures are dynamically allocated for every I/O operation (or for every distinct kind of I/O operation) on the heap.

Allocation of a future shared state (N3936/[futures.state]) associated with the future is combined with coroutine frame allocation and does not incur an extra allocation.

Generator

Another coroutine type was implemented to validate the generator pattern and a coroutine cancellation mechanics:

```
generator<int> fib(int n)
{
	int a = 0;
	int b = 1;
	while (n-- > 0)
	{
		yield a;
		auto next = a + b;
		a = b;
		b = next;
	}
}

int main()
{
	for (auto v : fib(35)) {
		std::cout << v << std::endl;
		if (v > 10)
			break;
	}
}
```

Recursive application of generators allows to mitigate stackless coroutine inability to suspend from nested stack frames. This example is probably the most convoluted way to print number in range [1..100).

```
recursive_generator<int> range(int a, int b)
{
	auto n = b - a;

	if (n <= 0)
		return;

	if (n == 1)
	{
		yield a;
		return;
	}

	auto mid = a + n / 2;

	yield range(a, mid);
	yield range(mid, b);
}
```

```
int main() {
	for (auto v : range(1, 100))
		std::cout << v << std::endl;
}
```

Parent-stealing parallel computations

It is possible to adopt coroutine mechanics to support parallel scheduling techniques such as parent stealing [N3872].

```
spawnable<int> fib(int n) {
	if (n < 2) return n;
	return await(fib(n - 1) + fib(n - 2));
}

int main() { std::cout << fib(5).get() << std::endl; }
```

In this example **operator**+ is overloaded for spawnable<T> type. Operator + schedules fib(n – 2) to a work queue whereas the execution continues into fib(n-1). When eventual values for both fib(n-1) and fib(n-2) are ready, fib(n) is resumed and result of await expression is computed as the sum of the eventual value of the left and right operand to + operator.

Utilizing parent-stealing scheduling allows to compute fib(42) in less than 12k of space, whereas attempting to use more traditional scheduling will cause state explosion that will consume more than 2gig of memory around fib(32).

Note, there are much better[3] ways to compute Fibonacci numbers.

Go-like channels and goroutines

The following example (inspired by programming language go [GoLang]) creates one million goroutines, connects them to each other using channels and passes a value that will travel through all the coroutines.

```
goroutine pusher(channel<int>& left, channel<int>& right)
{
	for (;;) {
		auto val = await left.pull();
		await right.push(val + 1);
	}
}

int main()
{
	static const int N = 1000 * 1000;
	std::vector<channel<int>> c(N + 1);

	for (int i = 0; i < N; ++i)
		goroutine::go(pusher(c[i], c[i + 1]));

	c.front().sync_push(0);

	std::cout << c.back().sync_pull() << std::endl;
```

[3] In constant space with only log n iterations

```
}
```

Measured on Lenovo W540 laptop, creation of a million of goroutine takes 64ms (using a default allocator) and passing a value through a million of goroutines takes 32ms (32ns per coroutine).

Conceptual Model

Resumable Function

A function or a lambda is called **resumable** function or resumable lambda if a body of the function or lambda contains a suspendable expression. Suspendable expressions are expressions with await or yield operators. From this point on, we will use the term resumable function to refer to either resumable lambda or resumable function.

Suspend expression marks the point where execution of the resumable function can be suspended and control returned to the current caller with an ability to resume execution at the point of suspend expression later.

N3936/[intro.execution]/7 defines that suspension of a block preserves the automatic variables in a case of a function call or receipt of a signal. We propose to extend that to coroutine suspension as well.

Design Note: Original design relied on resumable keyword to annotate resumable functions or lambdas. This proposal does away with resumable keyword relying on the presence of await or yield expressions. There were several motivations for this change.

1. It eliminates questions such as: Is resumable a part of signature or not? Does it change a calling conventions? Should it be specified only on a function definition?

2. It eliminates compilation errors due to the absence of resumable keyword that were in the category: "A compiler knows exactly what you mean, but won't accept the code until you type 'resumable'."

3. Usability experience with the resumable functions implemented in MSVC++ compiler by the authors. Initial implementation had resumable keyword and writing code felt unnecessarily verbose with having to type resumable in the declarations and definitions of functions or lambda expressions.

Resumable traits

Resumable traits are specialized by resumable functions to select an allocator and a coroutine promise to use in a particular resumable function.

If the signature of a resumable function is

R func(T1, T2, … Tn)

then, a traits specialization std::resumable_traits<R,T1,T2,…,Tn> will indicate what allocator and what coroutine promise to use.

For example, for coroutines returning future<R>, the following trait specialization can be provided.

```
template <typename R, typename... Ts>
struct resumable_traits<std::future<R>, Ts...>
{
	using allocator_type = std::allocator<char>;
	using promise_type = some-type-satisfying-coroutine-promise-requirements;
```

```
	};
```

allocator_type should satisfy allocator requirements **N3936/[allocator.requirements]**. If allocator_type is not specified, a resumable function will defaults to using std::allocator<char>.

promise_type should satisfy requirements specified in "Coroutine Promise Requirements". If promise_type is not specified it is assumed to be as if defined as follows:

```
using promise_type = typename R::promise_type;
```

C++ standard library defines the resumable traits as follows:

```
	template <typename R, typename... Ts>
	struct resumable_traits
	{
		using allocator_type = std::allocator<char>;
		using promise_type = typename R::promise_type;
	};
```

Design note: Another design option is to use only the return type in specializing the resumable traits. The intention for including parameter types is to enable using parameters to alter allocation strategies or other implementation details while retaining the same coroutine type.

Allocation and parameter copy optimizations

An invocation of a coroutine may incur an extra copy or move operation for the formal parameters if they need to be transferred from an ABI prescribed location into a memory allocated for the coroutine frame.

A parameter copy is not required if a coroutine never suspends or if it suspends but its parameters will not be accessed after the coroutine is resumed.

If a parameter copy/move is required, class object moves are performed according to the rules described in Copying and moving class objects section of the working draft standard 3936/[class.copy].

An implementation is allowed to elide calls to the allocator's allocate and deallocate functions and use stack memory of the caller instead if the meaning of the program will be unchanged except for the execution of the allocate and deallocate functions.

auto / decltype(auto) return type

If a function return type is auto or declspec(auto) and no trailing return type is specified, the resumable function is assumed to be of type default-standard-task-type<T> if an await expression is present in a function, where type T is deduced from return statements as described in N3936/[dcl.spec.auto]**.** If a yield expression is present in a function the return value is assumed to be of type default-generator-type<T>, where T is deduced from the yield statements according to the rules of return type deduction described in N3936/[dcl.spec.auto]**.**

If both await and yield expressions are present in the function, return type cannot be deduced and diagnostics should be given to the user.

At the moment we do not have a proposal for what default-standard-task-type or default-generator-types should be. We envision that once resumable functions are available as a language feature, C++

community will come up with ingenious libraries utilizing that feature and some of them will get standardized and become default-generator-type and default-task-type.

Until that time, an attempt to define resumable functions with auto / decltype(auto) and no trailing return type should result in a diagnostic message.

Coroutine promise Requirements

A library developer supplies the definition of the coroutine promise to implement desired high-level semantics associated with a coroutine type. The following tables describe the requirements on coroutine promise types.

Table 1: Descriptive Variable definitions

Variable	**Definition**
P	Coroutine promise type
p	A value of type P
e	A value of std::exception_ptr type
rh	A value of type std::resumable_handle<P>
T	An arbitrary type T
v	A value of type T

Table 2: Coroutine Promise Requirements

Expression	**Note**
P{}	Constructs a promise type.
p.get_return_object(rh)	get_return_object is invoked by the coroutine to construct the return object prior to the first suspend operation. get_return_object receives a value std::resumable_handle<Promise> as the first parameter. An object of std::resumable_handle<Promise> type can be used to resume the coroutine or get access to its promise.
p.set_value(v)	Sets the value associated with the promise. set_value is invoked by a resumable function when return <expr> ; statement is encountered in a resumable function. If p.set_value(v) member function is not present, coroutine does not support eventual return value and presence of return <expr> statement in the body is a compile time error.
p.set_value()	set_value() is invoked by the resumable function when return; statement is encountered or the control reaches the end of the resumable function.

	If set_value() is not present, it is assumed that the function supports eventual value and diagnostic should be given if return <expr> is not present in the body of the resumable function If both set_value() and set_value(v) are present in a promise type, the type does not satisfy coroutine promise requirement and a diagnostic message should be given to the user.
p.set_exception(e)	set_exception is invoked by a resumable function when an unhandled exception occurs within a body of the resumable function
p.yield_value(v)	returns: awaitable expression yield_value is invoked when yield expression is evaluated in the resumable function. If yield_value member function is not present, using yield operator in the body of the resumable function results in a compile time error. `yield` <expr> is equivalent to `await <Promise>.yield_value`(<expr>) Where a <Promise> refers to the coroutine promise of the enclosing resumable function.
initial_suspend()	Returns: awaitable expression A resumable function awaits on a value returned by the initial_suspend() member function immediately before user provided body of the resumable function is entered. This member function gives a library designer an option to suspend a coroutine after the coroutine frame is allocated, parameters are copied and return object is obtained, but before entering the user-provided body of the coroutine. For example, in a generator scenario, a library designer can choose to suspend a generator prior to invoking user provided body of the coroutine and to resume it once the user of the generator tries to pull the first value.
final_suspend()	Returns: nothrow awaitable expression Throws: nothing Resumable function awaits on a value returned by final_suspend() immediately after the user provided body of the resumable function

	i.e. point prior to the destruction and deallocation of a coroutine frame. This allows library designer to store the eventual value of the task, or the current value of the generator within the coroutine promise. Once the eventual value or last value is consumed, coroutine can be resumed to free up resources associated with it.
cancellation_requested()	Returns: bool to indicate whether coroutine is being cancelled cancellation_requested() is evaluated on resume code path. If it evaluates to true, control is transferred to the point immediately prior to compiler synthesized await promise-expr.final_suspend(), otherwise control is transferred to the current resume point. All of the objects with non-trivial destructors, will be destroyed in the same manner as if “goto end-label” statement was executed immediately after the resume point. (Assuming that “goto” was allowed to be used within an expression)

Resumption function object

A resumable function has the ability to suspend evaluation by means of await or yield expressions in its body. Evaluation may later be resumed at the point of the suspending of await or yield expression by invoking a resumption function object.

A resumption function object is defined by C++ standard library as follows:

```
template <typename Promise = void>
struct resumable_handle;

template <> struct resumable_handle<void>
{
        void operator() ();
        static resumable_handle<void> from_address(void*);
        void * to_address() ;

        explicit operator bool() const;

        resumable_handle() = default;
        explicit resumable_handle(std::nullptr_t);
        resumable_handle& operator = (nullptr_t);
};

template <typename Promise>
struct resumable_handle: public resumable_handle<>
{
        Promise & promise();
        Promise const & promise() const;
        static resumable_handle<Promise> from_promise(Promise*);
```

```
        using resumable_handle<>::resumable_handle;
        resumable_handle& operator = (nullptr_t);
    };


    template <typename Promise>
    bool operator == (resumable_handle<Promise> const& a,
                      resumable_handle<Promise> const& b);

    template <typename Promise>
    bool operator != (resumable_handle<Promise> const& a,
                      resumable_handle<Promise> const& b)
```

Note, that by design, a resumption function object can be “round tripped” to void * and back. This property allows seamless interactions of resumable functions with existing C APIs[4].

Resumption function object has two forms. One that provides an ability to resume evaluation of a resumable function and another, which additionally allows access to the coroutine promise of a particular resumable function.

Bikeshed: resumption_handle, resumption_object, resumable_ptr, basic_resumable_handle instead of resumable_handle<void>, from_raw_address, to_raw_address, from_pvoid, to_pvoid.

await operator

is a unary operator expression of the form: **await** *cast-expression*

1. The await operator shall not be invoked in a catch block of a try-statement[5]
2. The result of await is of type T, where T is the return type of the await_resume function invoked as described in the evaluation of await expression section. If T is void, then the await expression cannot be the operand of another expression.

Evaluation of await expression

An await expression in a form **await** *cast-expression* is equivalent to (if it were possible to write an expression in terms of a block, where return from the block becomes the result of the expression)

```
{
    auto && __expr = cast-expression;
    if ( !await_ready-expr ) {
       await_suspend-expr;
       suspend-resume-point
    }
    cancel-check;
    return await_resume-expr;
}
```

[4] Usually C APIs take a callback and void* context. When the library/OS calls back into the user code, it invokes the callback passing the context back. from_address() static member function allows to reconstitute resumable_handle<> from void* context and resume the coroutine

[5] The motivation for this is to avoid interfering with existing exception propagation mechanisms, as they may be significantly (and negatively so) impacted should await be allowed to occur within exception handlers.

Where __expr is a variable defined for exposition only, and _ExprT is the type of the *cast-expression*, and _PromiseT is a type of the coroutine promise associated with current resumable function and the rest defined as follows:

await_ready-expr await_suspend-expr await_resume-expr	— if _PromiseT is a class type, the unqualified-ids **await_ready**, **await_suspend** and **await_resume** are looked up in the scope of class _PromiseT as if by class member access lookup (N3936/3.4.5 [basic.lookup.classref]), and if it finds at least one declaration, then await_ready-expr, await_suspend-expr and await_resume-expr are __expr.await_ready(), __expr.await_suspend(resumption-function-object) and __expr.await_resume(), respectively; — otherwise, await_ready-expr, await_suspend-expr and await_resume-expr are await_ready(__expr), await_suspend(__expr,resumption-function-object) and await_resume(__expr), respectively, where await_ready, await_suspend and await_resume are looked up in the associated namespaces (N3936/3.4.2). [Note: Ordinary unqualified lookup (3.4.1) is not performed. —end note] A type for which await_ready, await_suspend and await_resume function can be looked up by the rules described above is called an **awaitable type**. If none of await_xxx functions can throw an exception, the awaitable type is called a **nothrow awaitable type** and expression of this type a **nothrow awaitable expressions**.
resumption-function-object	A function object of type std::resumable_handle<_PromiseT>. When function object is invoked it will resume execution of the resumable function at the point marked by suspend-resume-point.
suspend-resume-point	When this point is reached, the coroutine is suspended. Once resumed, execution continues immediately after the suspend-resume-point
cancel-check	For all await expressions except for the one synthesized by a compiler for using await <promise.expr>.final_suspend() it is `if ( <promise-expr>.cancellation_requested() ) goto <end-label>;` where <promise-expr> is a reference to a coroutine promise associated with the current resumable function and an end-label is a label at the end of the user provided body of the resumable function, just prior to the await <promise-expr>.final_suspend().

Design Note: rules for lookup of await_xxx identifiers mirror the look up rules for range-based for statement. We also considered two other alternatives (we implemented all three approaches to test their usability, but found the other two less convenient than the one described above):

1. To have only ADL based lookup and not check for member functions. This approach was rejected as it disallowed one of the convenient patterns that was developed utilizing await. Namely to have compact declaration for asynchronous functions in a form: auto Soket::AsyncRead(int count) { struct awaiter {…}; return awaiter{this, count}) };

2. Another approach considered and rejected was to have an **operator await** function found via ADL and having it to return an awaitable_type that should have await_xxx member functions defined. It was found more verbose than the proposed alternative.

yield operator

is a unary operator expression of the form **yield** *cast-expression*.

`yield` <expr> is equivalent to

  `await <Promise>.yield_value`(<expr>)

Where a <Promise> refers to the coroutine promise of the enclosing resumable function.

Design note: yield is a popular identifier, it is used in the standard library, e.g. this_thread::yield(). Introducing a yield keyword will break existing code. Having a two word keyword, such as **yield return** could be another choice.

Trivial awaitable types

Standard library provides three awaitable types defined as follows:

```
namespace std {
	struct suspend_always {
		bool await_ready() const { return false;  }
		void await_suspend(std::resumable_handle<>) {}
		void await_resume() {}
	};
	struct suspend_never {
		bool await_ready() const { return true;  }
		void await_suspend(std::resumable_handle<>) {}
		void await_resume() {}
	};
	struct suspend_if
	{
		bool ready;

		suspend_if(bool condition): ready(!condition){}

		bool await_ready() const { return ready; }
		void await_suspend(std::resumable_handle<>) {}
		void await_resume() {}
	};
}
```

These types are used in implementations of coroutine promises. Though they are trivial to implement, including them in the standard library eliminates the need for every library designer from doing their own implementation.

For example, generator<T> coroutine listed in the Appendix A, defines yield_value member function as follows:

```
std::suspend_always promise_type::yield_value(T const& v) {
   this->current_value = &v;
   return{};
}
```

Return statement

A return statement in a resumable function in a form **return** *expression*; is equivalent to:

{ promise-expr.set_value(*expression*); goto end-label; }

A return statement in a resumable function in a form **return** *braced-init-list*; is equivalent to:

{ promise-expr.set_value(*braced-init-list*); goto end-label; }

A return statement in a resumable function in a form **return;** is equivalent to:

{ promise-expr.set_value(); goto end-label; }

Where end-label is a label at the end of the user provided body of the resumable function, just prior to the await <promise-expr>.final_suspend().

If resumable function does not have return statements in the form **return** *expression*; or **return** *braced-init-list*; then the function acts as if there is an implicit **return;** statement at the end of the function.

An expository Resumable Function Implementation

Note: The following section is for illustration purposes only. It does not prescribe how resumable functions must be implemented.

Given a user authored function:

```
R foo(T1 a, T2 b) { body-containing-await-or-yield-expressions }
```

Compiler can constructs a function that behaves as if the following code was generated:

```
R foo(T1 a, T2 b) {
   using __traits = std::resumable_traits<R, T1, T2>;
   struct __Context {
      __traits::promise_type _Promise;
      T1 a;
      T2 b;

      template <typename U1, typename U2>
      __Context(U1&& a, U2&& b) : a(forward<U1>(a)), b(forward<U2>(b)) {}

      void operator()() noexcept {
          await _Promise.initial_suspend();
          try { body-containing-await-or-yield-with-some-changes }
          catch (...) { _Promise.set_exception(std::current_exception()); }
      __return_label:
          await _Promise.final_suspend();
          <deallocate-frame> (this, sizeof(__Context) + <X>);
      }
   };

   auto mem = <allocate-frame>(sizeof(__Context) + <X>);
```

```
   __Context * coro = nullptr;
   try {
      coro = new (mem) __Context(a, b);
      auto result = __traits::get_return_object(
         std::resumable_handle<__traits::promise_type>::from_promise(&coro->__Promise);
      (*coro)();
      return result;
   }
   catch (...) {
      if (coro) coro->~__Context();
      <deallocate-frame> (mem, sizeof(__Context) + <X>);
      throw;
   }
}
```

Where, <X> is a constant representing the number of bytes that needs to be allocated to accommodate variables with automatic storage duration in the body of the resumable function and platform specific data that is needed to support resume and suspend.

Access to variables and temporaries with automatic storage duration in the body of operator() should be relative to "this" pointer at the offset equal to sizeof(*this).

<body-containing-await-or-yield-with-some-changes> is identical to <body-containing-await-or-yield> with the exception that await and yield operators and return statement are transformed as described in earlier sections, __return_label is the <end-label> that return and await refer to and accesses to formal parameters a, b are replaced with accesses to member variables a and b of __Context class.

no-except operations

C++ exceptions represent a barrier to adoption of full power of C++. While this is unfortunate and may be rectified in the future, the current experience shows that kernel mode software, embedded software for devices and airplanes [JSF] forgo the use of C++ exceptions for various reasons.

Making coroutine dependent on C++ exceptions will limit their usefulness in contexts where asynchronous programming help is especially valuable (kernel mode drivers, embedded software, etc).

The following sections described how exceptions can be avoided in implementation and applications of resumable functions.

Allocation failure

To enable handling of allocation failures without relying on exception mechanism, resumable_traits specialization can declare an optional allocation_check_needed type that should be identical to or derived from std::true_type or std::false_type. If allocation_check_needed type is not defined, it is assumed to be std::false_type.

If allocation_check_needed() evaluates to true, it is assumed that an allocator's allocate function will violate the standard requirements and will return nullptr in case of an allocation failure.

If an allocation has failed, a resumable function will use static member function `get_return_object_on_allocation_failure()` of resumable_traits specialization to construct the return value.

The following is an example of such specialization

```
namespace std {
	template <typename T, typename... Ts>
	struct resumable_traits<kernel_mode_future<T>, Ts...> {
		using allocator_type = std::kernel_allocator<char>;
		using promise_type = kernel_mode_resumable_promise<T>;

		using check_frame_allocation = std::true_type;
		static auto get_return_object_on_allocation_failure() { … }
	};
}
```

Generalizing coroutine's promise set_exception

In exception-less environment, a requirement on set_exception member of coroutine promise needs to be relaxed to be able to take a value of an arbitrary error type E and not be limited to just the values of type std::exception_ptr. In exception-less environment, not only std::exception_ptr type may not be supported, but even if it were supported it is impossible to extract an error from it without relying on throw and catch mechanics.

Await expression: Unwrapping of an eventual value

As described earlier **await** *cast-expression* expands into a code equivalent to:

```
{
    auto && __expr = cast-expression;
    if ( !await_ready-expr ) {
       await_suspend-expr;
       suspend-resume-point
    }
    cancel-check;
    return await_resume-expr;
}
```

Where cancel-check is expanded as

```
	if ( promise-expr.cancellation_requested() ) goto end-label;
```

A straightforward implementation of await_resume() for getting an eventual value from the future<T> will call .get() that will either return the stored value or throw an exception. If unhandled, an exception will be caught by a catch(…) handler of the resumable function and stored as an eventual result in a coroutine return object.

In the environments where exceptions are not allowed, implementation can probe for success or failure of the operation prior to resuming of the coroutine and use <promise>.set_exception to convey the failure to the promise. Coroutine promise, in this case, need to have cancellation_requested() to return true if an error is stored in the promise.

Here is how await_suspend may be defined for our hypothetical kernel_future<T> as follows:

```
template <typename Promise>
void kernel_future::await_suspend(std::resumable_handle<Promise> p) {
   this->then([p](kernel_future<T> const& result) {
      if (result.has_error())
```

```
      {
         p.promise().set_exception(result.error());
      }
      p(); // resume the coroutine
   });
}
```

Await expression: Failure to launch an asynchronous operation

If an await_suspend function failed to launch an asynchronous operation, it needs to prevent suspension of a resumable function at the await point. Normally, it would have thrown an exception and would have avoided suspend-resume-point. In the absence of exceptions, we can require that await_suspend must return false, if it failed to launch an operation and true otherwise. If false is returned from await_suspend, then coroutine will not be suspended and will continue execution. Failure can be indicate via set_exception mechanism as described in the previous section.

With all of the changes described in this section, await expr will be expanded into equivalent of:

```
{
    auto && __expr = cast-expression;
    if ( ! await_ready-expr && await_suspend-expr)
       suspend-resume-point
    }
    cancel-expression;
    return __expr.await_resume();
}
```

With the extensions described above it is possible to utilize await and resumable functions in the environment where exceptions are banned or not supported.

Asynchronous cancellation

An attempt to cancel a coroutine that is currently suspended awaiting completion of an asynchronous I/O, can race with the resumption of a coroutine due to I/O completion. The coroutine model described in this paper can be extended to tackle asynchronous cancellation. Here is a sketch.

A coroutine promise can expose set_cancel_routine(Fn) function, where Fn is a function or a function object returning a value convertible to bool. A set_cancel_routine function should return true if cancel_routine is set and there is no cancellation in progress and false otherwise.

await_suspend(std::resumable_handle<Promise> rh), in addition to subscribing to get a completion of an asynchronous operation can use rh.promise().set_cancel_routine(Fn) to provide a callback that can attempt to cancel an asynchronous operation.

If a coroutine needs to be cancelled, it invokes a cancel_routine if one is currently associated with the coroutine promise. If cancel_routine returns true, it indicates that the operation in progress was successfully cancelled and the coroutine will not be resumed by the asynchronous operation. Thus, the execution of the coroutine is under full control of the caller. If cancel_routine returns false, it means that an asynchronous operation cannot be cancelled and coroutine may have already been resumed or will be resumed at some point in the future. Thus, coroutine resources cannot be released until pending asynchronous operation resumes the coroutine.

We do not propose this mechanism yet as we would like to gain more experience with developing libraries utilizing resumable functions described in this paper.

Proposed Standard Wording

No wording is provided at the moment.

Acknowledgments

Great thanks to Artur Laksberg, Gabriel Dos Reis, Deon Brewis, James McNellis, Stephan T. Lavavej, Herb Sutter, Robert Schumacher, Niklas Gustafsson, Nick Maliwacki, Vladimir Petter, Oliver Kowalke, Nat Goodspeed, Christopher Kohlhoff for your review and comments and Herb, Artur, Deon and Niklas for trailblazing, proposing and implementing resumable functions v1.

References

[N3936] Working Draft, Standard for Programming Language C++

[Revisiting Coroutines] Moura, Ana Lúcia De and Ierusalimschy, Roberto. "Revisiting coroutines". ACM Trans. Program. Lang. Syst., Volume 31 Issue 2, February 2009, Article No. 6

[N3328] Resumable Functions

[N3977] Resumable Functions: wording

[N3985] A proposal to add coroutines to the C++ standard library (Revision 1)

[Boost.Context] Boost.Context Overview

[Boost.Coroutine] Boost.Coroutine Overview

[SplitStacks] Split Stacks in GCC

[JSF] Join Strike Fighter C++ Coding Standards

[GoLang] http://golang.org/doc/

[N3872] A Primer on Scheduling Fork-Join Parallelism with Work Stealing

Appendix A: An example of generator coroutine implementation

```
#include <resumable>
#include <iterator>

template <typename T> class generator
{
   struct promise_type {
       T const * value;
       std::exception_ptr error;
       enum state_t { active, cancelling, closed};
       state_t state = state_t::active;

       generator<T> get_return_object() { return{ *this }; }
       std::suspend_always initial_suspend() { return{}; }
       std::suspend_always final_suspend() { state = closed; return{}; }

       bool cancel_requested() { return state == cancelling; }

       void set_value(){}
       void set_exception(std::exception_ptr e) { error = std::move(e); }

       std::suspend_always yield_value(T const& v) { this->value = &v; return{}; }
   };

    struct iterator : public std::iterator<std::input_iterator_tag, T>
    {
        std::resumable_handle<promise_type> rh;

        iterator(std::nullptr_t) : rh(nullptr) {}
        iterator(std::resumable_handle<promise_type> rh) : rh(rh) {}
        iterator & operator ++() {
            rh();
            if (rh.promise().state == promise_type::closed)
               rh = nullptr;
            return *this;
        }
        bool operator == (iterator const& rhs) { return rh == rhs.rh; }
        bool operator != (iterator const& rhs) { return rh != rhs.rh; }

        T const& operator*() {
           auto & p = rh.promise();
           if (p.error) std::rethrow_exception(p.error);
           return *p.value;
        }
    };

    iterator begin() {
       impl();
       if (impl.promise().state == promise_type::closed)
          return{ nullptr };
       return{ impl };
    }

    iterator end() { return{ nullptr }; }

   ~generator() {
       auto & p = impl.promise();
```

```
        if (p.state == promise_type::active) {
           p.state = promise_type::cancelling;
           impl();
        }
        impl();
     }
private:
   generator(promise_type& p):
      impl(std::resumable_handle<promise_type>::from_promise(&p)) {}

   std::resumable_handle<promise_type> impl;
};
```

Appendix B: PPL task adapters

```
#include <resumable>
#include <ppltasks.h>

namespace std {
   template <class T>
   struct resumable_traits<concurrency::task<T>> {
      struct promise_type {
         concurrency::task_completion_event<T> tce;

         concurrency::task<T> get_return_object(){return concurrency::create_task(tce); }

         suspend_never initial_suspend() { return{}; }
         suspend_never final_suspend() { return{}; }

         template <class U> enable_if_t<is_same<U,T>::value && is_same<U,void>::value>
         set_value() { tce.set(); }

         template <class U> enable_if_t<is_same<U,T>::value && !is_same<U,void>::value>
         set_value(U value) { tce.set(std::move(value)); }

         void set_exception(std::exception_ptr e) { tce.set_exception(std::move(e)); }
         bool cancel_requested() { return false; }
     };
};

namespace concurrency {
       template <class T>
       bool await_ready(concurrency::task<T> & t) { return t.is_done(); }

       template <class T>
       void await_suspend(concurrency::task<T> & t, std::resumable_handle<> cb) {
             t.then([cb](concurrency::task<T>){cb();});
       }

       template <class T>
       auto await_resume(concurrency::task<T> & t) { return t.get(); }
}
```

Appendix C: Awaitable adapter over OS async facilities

```
#include <resumable>
#include <threadpoolapiset.h>

// usage:  await sleep_for(100ms);
auto sleep_for(std::chrono::system_clock::duration duration)  {
   class awaiter {
      static void TimerCallback(PTP_CALLBACK_INSTANCE, void* Context, PTP_TIMER) {
         std::resumable_handle<>::from_address(Context)();
      }
      PTP_TIMER timer = nullptr;
      std::chrono::system_clock::duration duration;
   public:
      awaiter(std::chrono::system_clock::duration d) : duration(d){}
      bool await_ready() const { return duration.count() <= 0; }
      void await_suspend(std::resumable_handle<> resume_cb)  {
         int64_t relative_count = -duration.count();
         timer = CreateThreadpoolTimer(TimerCallback, resume_cb.to_address(), nullptr);
         if (timer == 0) throw std::system_error(GetLastError(), std::system_category());
         SetThreadpoolTimer(timer, (PFILETIME)&relative_count, 0, 0);
      }
      void await_resume() {}
      ~awaiter() {
         if (timer) CloseThreadpoolTimer(timer);
      }
   };
   return awaiter{ duration };
}
```