Document number: P0099R0
Supersedes: N4397
Date: 2015-09-27
Project: WG21, SG1
Author: Oliver Kowalke (oliver.kowalke@gmail.com), Nat Goodspeed (nat@lindenlab.com)

A low-level API for stackful context switching

Revision History This document supersedes N4397. It elaborates a low-level API for stackful context switching.

Changes since N4397:

- `std::execution_context` presented as pure library facility.
- Motivation section added.
- `std::execution_context` no longer tracks parent relationships.
- Removed `operator bool()` and `operator!()`.

Abstract

This paper proposes a *low-level* API for a stackful execution context, suitable to act as **building-block** for **high-level** constructs such as stackful coroutines as well as cooperative multitasking (aka fibers/user-land threads/green threads).

The most important features are:

- first-class object that can be stored in variables or containers
- symmetric transfer of execution control, i.e. suspend-by-call - enables a richer set of control flows than asymmetric transfer of control (i.e. suspend-by-return as described in N4402[7] et al.)
- benefits of traditional stack management retained
- ordinary function calls and returns not affected
- working implementation in Boost.Context[13]

Motivation

Content This paper proposes `std::execution_context`. `execution_context` is the crucial foundation on which a number of valuable higher-level facilities can be built using pure C++.

`execution_context` itself is, however, impossible to code in portable C++. Having it provided with the runtime library will be enabling technology for both third-party libraries and user applications.

As a library-only facility, `boost::context::execution_context`[13] is existing practice.
`execution_context` is used to implement Boost.Coroutine2[14] and the candidate Boost.Fiber library.[15]

This facility adds significant semantics beyond resumable functions. For instance, you cannot build cooperative user-mode threads (such as those supported by Boost.Fiber) on resumable functions.

These libraries based on Boost.Context can seamlessly interoperate with Boost.Asio:[12] Asio's `async_result` mechanism (proposed for standardization in N4045[3]) supports adapters that can, using a natural syntax, suspend the caller for the duration of an asynchronous network call. For instance, using Boost.Coroutine:

```
void read_msg(yield_context yield) {
    try {
        array< char, 64 > bf1;
        async_read(socket, buffer(b1), yield);
        header hdr(bf1);
        std::size_t n = hdr.payload_size();
        std::vector< char > b2(n, '\0');
        async_read(socket, buffer(bf2), yield);
        payload pld(bf2);
        // process message ...
    } catch (exception const&) {
        // ...
    }
}
```

The same example recast using Boost.Fiber:

```
void read_msg() {
    try {
        array< char, 64 > bf1;
        async_read(socket, buffer(b1), boost::fibers::asio::yield);
        header hdr(bf1);
        std::size_t n = hdr.payload_size();
        std::vector< char > b2(n, '\0');
        async_read(socket, buffer(bf2), boost::fibers::asio::yield);
        payload pld(bf2);
        // process message ...
    } catch (exception const&) {
        // ...
    }
}
```

Libraries based on `std::execution_context` will similarly interoperate cleanly with the proposed standard Networking Library.[9]

The authors intend to bring future papers to propose a cooperative fiber library and "stackful" coroutines. `execution_context` is foundational for all such evolutionary work.

Why Bother? Given N4402 resumable functions, why should we even consider a completely different mechanism for suspending and resuming a function? Isn't this completely redundant with that?

The answer is "no," for several different reasons.[*]

- With resumable functions, the markup to support suspension propagates virally through a code base. Resumable functions suspend by returning. Every caller must therefore distinguish between "callee returned value" and "callee suspended, you must also suspend." The new `await` keyword makes that distinction, suspending the containing function until the callee produces a value. Every call to a resumable function must itself use `await` – which then imposes the same requirement on *its* caller, and so on.
- Forgetting to annotate a call to a resumable function with `await` (perhaps because the caller is unaware of its nature, perhaps because the callee changed its nature) can result in subtle timing bugs. Suppose function `f()` returns a `std::future<void>`. The statement `await f();` suspends the containing function until the `future` returned by `f()` is fulfilled. Coding simply `f();` merely *discards* that future. Both forms are perfectly legal; both are likely to survive desk-checking and code review. It's even worse if `f()` *usually* fulfills its `future` by the time it returns, only *occasionally* taking longer. That way the error will survive most QA as well, and escape into production.
- The need to `await` called functions means that we must evolve a whole new family of `await`-aware STL algorithms.[†]
- Even if one is willing to accept the viral `await` markup of resumable functions, using normal encapsulation to manage layers of abstraction could become expensive in runtime. Every entry to a resumable function requires a heap allocation, freed on return. By contrast, once you have allocated a side stack, function call and return on that side stack is just as efficient as function call and return on the original application stack. It's a classic time/space tradeoff. (If you determine to address that issue by drastically pruning local variables from your resumable functions, note that the same tactic could allow you to use a far smaller side stack – which would still be faster for nontrivial call chains.)

[*]The authors are indebted to Christopher Kohlhoff for his excellent summary in N4453,[8] sections 4.1 and 4.2.
[†]N4453[8] section 4.2 explains this more fully.

Background

At the November 2014 meeting in Urbana the committee discussed proposals for *stackless* (N4134[4] and N4244[5]) and *stackful* (N3985[1]) coroutines. The members of the committee concluded that stackless and stackful coroutines have distinct use cases and decided to pursue both technologies.

The authors proposed N4397[6] at the May 2015 meeting in Lenexa, recognizing that the lower-level `execution_context` API is a more powerful foundational abstraction than N3985 coroutines. For instance, with a fiber library implemented on `execution_context`, the `main()` context (as well as each new thread) can access the same library functionality as an explicitly-launched fiber.

This paper is an updated version of N4397.

Definitions

This section gives an overview about the wording used in this proposal.

Execution context environment in which program logic is evaluated (CPU registers, stack).

Activation record also known as *activation frame* or *stack frame* (a special activation record).
An *activation record* is a data structure containing the state of a particular function call. The following data are part of an activation record:

- a set of registers (defined by the calling convention of the ABI,[*] e.g. the return address)
- local variables
- parameters to the function
- eventual return value

Stack frame an activation record allocated on the processor stack; stack frames are allocated in strict last-in-first-out order - placed by incrementing/decrementing the stack pointer register on function call/return.

Heap-allocated activation record an activation record allocated in heap storage; every heap allocated activation record holds a pointer to its parent activation record (parent pointer tree).

Processor stack is a chunk of memory into which the processor's stack pointer register is pointing. The processor stack might belong to:

- an application's initial (or only) thread
- an explicitly-launched thread
- a sub-thread execution context (e.g. a coroutine or fiber)

Application stack The processor stack assigned to function `main()` is generally a *linear stack*. Often, the memory management system is used to detect stack overflow and to allocate more memory. This stack is ideally placed well away from any other data in the process's address space so it can be extended as needed.

Thread stack The processor stack assigned to an explicitly-launched thread is generally a *linear stack*. In a 32-bit (or smaller) address space, the operating system cannot, in general, guarantee that such a stack can be arbitrarily extended on overflow: the range of available addresses is constrained by the next adjacent allocated memory block. Instead, stacks with a fixed size are used - usually 1MB (Windows) up to 2MB (Linux), but some platforms use smaller stacks (64KB on HP-UX/PA and 256KB on HP-UX/Itanium).

[*]Application Binary Interface

Side stack is a processor stack that is used for *stackful* context switching. Each execution context gets its own stack. The stack belonging to an inactive context remains unchanged, while function calls and returns push and pop stack frames on the currently-active stack.
A side stack is generally allocated by the running process (library code) rather than by the operating system. It might either be a *linear stack* or a *linked stack*.

Linear stack is a contiguous memory area into which the processor's stack pointer register points. On a processor with a stack pointer register, this is the traditional stack model for C++. Allocation and deallocation of activation records (stack frames in this context) is performed by incrementing/decrementing the stack pointer.

Linked stack also known as *split stack*[10] or *segmented stack*.[11] This is a linked list of contiguous memory blocks of intermediate size, each of which might hold several function stack frames. The processor's stack pointer points into the head block on this list; normally a new stack frame is allocated by adjusting the stack pointer, as usual. When compiler-generated code detects near overflow, it links on a new head block. Thus, the effective stack space can grow arbitrarily without requiring a single contiguous address range.
This mechanism has a low performance penalty - typically adding around 5-10% overhead to the instruction sequence implementing a function call. Only an application consisting solely of empty functions would see that impact overall. Small functions would see the most overhead – but modern C++ compilers work really hard to inline small functions, and of course the overhead vanishes entirely for inline functions.
Applications compiled with support for linked stacks can use (link against) libraries not supporting linked stacks. (See GCC's documentation,[10] chapter 'Backward compatibility'.)

Non-contiguous stack is a *parent pointer tree*-structure used to store heap-allocated activation records. It is a stack with branches; an activation record can outlive the context in which it was created. With this mechanism, the processor's stack pointer register does not point to (or into) any of these activation records. Compare to *Linked stack*.
A heap activation record is allocated on entry to a function and freed on return, as with N4402.[7]
This structure is also called *cactus stack*[16] or *spaghetti stack*.

Parent pointer tree data structure in which each node has a pointer to its parent, but no pointer to its children. Traversal from any node to its ancestors is possible but not to any other node.

Coroutine function that enables explicit suspend and resume of its progress by preserving execution state (*activation record*), thus providing an enhanced control flow. Coroutines have the following characteristics:[1]

- values of local data persist across context switches
- execution is suspended as control leaves coroutine and resumed at certain time later
- symmetric or asymmetric control-transfer mechanism
- first-class object or compiler-internal structure
- stackless or stackful

Toplevel context function is a function executed as a coroutine (stackless or stackful). This term is particularly useful when discussing library-supported context switching, as the execution context must be constructed explicitly and passed a function.

Asymmetric coroutine provides two distinct operations for the context switch - one operation to resume and one operation to suspend the coroutine.
An asymmetric coroutine is tightly coupled with its caller, i.e. suspending the coroutine transfers execution control back to the point in the code from which the coroutine was last resumed. The asymmetric control-transfer mechanism is usually used in the context of generators.

Symmetric coroutine only one operation to resume/suspend the context is needed.
A symmetric coroutine does not know its caller; control can be transferred to any other symmetric coroutine (which must be explicitly specified). The symmetric control-transfer mechanism is usually used to implement cooperative multitasking.

Fiber/user-mode thread execute tasks in a cooperative multitasking environment involving a scheduler. Coroutines and fibers are distinct (N4024[2]).
Code running in a user-mode thread (or "fiber") suspends by passing control to the scheduler, which selects the next fiber to resume. Such code is not inherently coupled to the code that launched the fiber (as with an asymmetric coroutine); nor must it explicitly select the next fiber (as with a symmetric coroutine).

Resumable function N4402[7] describes resumable functions as an efficient language-supported mechanism for stackless context switching introducing two new keywords - `await` and `yield`. A resumable function is a kind of asymmetric coroutine: suspending passes control back to its caller.
Characteristics of resumable functions:

- stackless
- uses heap allocated activation records (referred as activation frames in N4402)
- tight coupling between caller and resumable function (asymmetric control-transfer mechanism)
- implicit *return*-statement[7] (code transformation)

Suspend by return The terms *stackless* and *stackful* may become confusing as different implementations are discussed. An N4402[7] "stackless" resumable function might consume a bit of the current processor stack on entry, cleaning it off on suspension. A *cactus stack* coroutine implementation might provide "stackful" semantics while completely avoiding the processor stack.
We use the phrase *suspend by return* to discuss a mechanism such as described in N4402[7] and N4244,[5] in which suspending is implemented by returning from the function body.

Suspend by call describes a mechanism such as described in N3985,[1] in which suspending is implemented by calling some other (library) function.

Introduction

Traditionally C++ code is run on a linear stack, i.e. the activation records are allocated in strict *last-in-first-out* order. This stack model allocates activation records on function call/return by incrementing/decrementing the stack pointer.
But in the context of coroutines, that is, switching between different execution contexts, a linear stack introduces problems. Calling a function creates an activation record on the stack which is removed if the function returns. But for a suspended coroutine the activation record **must not** be **removed**!
Consider the following scenario:

- Assume the processor stack is built in descending order: that is, a PUSH instruction decrements the stack pointer register. Call the stack pointer's initial value SP0.
- Function `main()` enters coroutine `C()`.
- `C()`'s prolog allocates a stack frame of size `sizeof(C::frame)` by decrementing SP. SP is now at SP1 = (SP0 - `sizeof(C::frame)`).
- `C()` suspends, returning control to `main()`. `main()` must find its stack frame at SP = SP0.
- `main()` now calls function `F()`.
- `F()`'s prolog allocates a stack frame of size `sizeof(F::frame)` by decrementing SP. SP is now at SP2 = (SP0 - `sizeof(F::frame)`).
- Unless either `(sizeof(C::frame)== 0)` or `(sizeof(F::frame)== 0)`, any data written by `F()` to its own stack frame will necessarily overwrite any data saved by `C()` in its stack frame.

- `F()` returns. SP is back to SP0.
- `main()` resumes `C()`. SP is set to SP1.
- `C()` attempts to access data in its stack frame – which has been overwritten by `F()`. We are now in the realm of Undefined Behavior.

In order to prevent stack corruption, a stackless coroutine uses a heap-allocated activation record (N4402[7]), while stackful coroutines use a side stack (N3985[1]).
Since an N4402 stackless resumable function uses *suspend by return*, when it suspends, the stack pointer is restored to its value before the resumable function was called. While executing, a resumable function can consume additional space in the linear processor stack; it can call traditional functions. But they must all return before the resumable function suspends. If resumable function `A()` calls function `B()`, and `B()` wishes to suspend, `B()` must also be a resumable function. Thus the term *stackless*: a suspended resumable function leaves no activation record on the linear processor stack. A single linear stack can be reused by an arbitrary number of suspended resumable functions.
Using a side stack permits *stackful* coroutines to use *suspend by call*. An arbitrary number of ordinary stack frames can be left on the side stack for a suspended coroutine context; arbitrary stack frames can be pushed or popped on the currently-active stack, independently of any suspended stack.
This is the fundamental difference between stackless and stackful coroutines.

Traditional stack management – a single linear stack per thread – is inadequate for coroutines because coroutines must outlive the context in which they were created.

Discussion

Calling subroutines

The advantage of stackless coroutines is that they reuse the same linear processor stack for stack frames for called subroutines. The advantage of stackful context switching is that it permits **suspending from nested calls**.

If a resumable function calls a traditional function (rather than another resumable function), then the activation record belonging to the traditional function is allocated on the processor stack (so it is called a stack frame). As a consequence, stack frames of called functions must be removed from the processor stack before the resumable function yields back to its caller.
In other words: the calling convention of the **ABI dictates** that, when the resumable function returns (suspends), the stack pointer must contain the same address as before the resumable function was entered.
Hence a yield from nested call is **not permitted** – unless every called function down to the yield point is also a resumable function.
The benefit of stackless coroutines consists in reusing the processor stack for called subroutines: no separate stack memory need be allocated.

Of course even a stackless resumable function might fail if its called functions exhaust the available stack.

In stackful context switching, each execution context owns a distinct side stack which is assigned to the stack pointer (thus the stack pointer must be exchanged during each context switch).
All activation records (stack frames) of subroutines are placed on the side stack. Hence each stackful execution context requires enough memory to hold the stack frames of the longest call chain of subroutines. Therefore, to support calling subroutines, stackful context switching has a higher memory footprint than resumable functions.
On the other hand, it is beneficial to use side stacks because the stack frames of active subroutines remain intact while the execution context is suspended. This is the reason why stackful context switching permits **yielding from nested calls**.

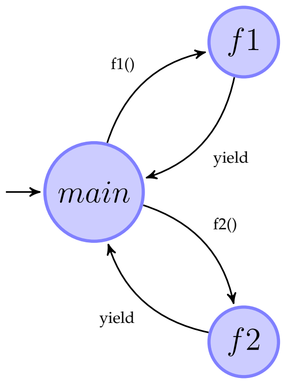
```
// P0099: stackful execution context
// access current execution context l1
auto l1=std::execution_context::current();
// create stackful execution context l2
std::execution_context l2(
    [&l1](){
        std::printf("inside l2\n");
        // suspend l2 and resume l1
        l1();
    });
// resume l2
l2();
```

Variable *l1* is passed to stackful execution context *l2* via the capture list of *l2*. The stack frames of `std::printf()` are allocated on the side stack owned by *l2*.

Asymmetric vs. symmetric

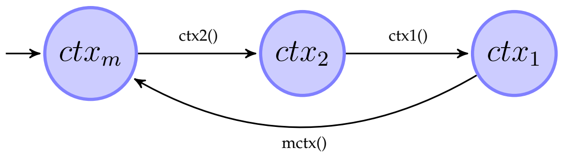
As a building block for user-mode threads, symmetric control transfer is more efficient than the asymmetric mechanism.

Asymmetric coroutines provide two operations for context switching. The caller and the coroutine are coupled, that is, such a coroutine can only jump back to the code that most recently resumed it.



For **N** asymmetric coroutines, **2N** context switches are required. This is sufficient in the case of generators, but in the context of cooperative multitasking it is inefficient.

The proposed stackful execution context (`std::execution_context`) provides only one operation to resume/suspend the context (`operator()()`). Control is directly transferred from one execution context to another (symmetric control transfer) - no jump back to the caller. In addition to supporting generators, this enables an efficient implementation of cooperative multitasking: **no additional context switch** back to caller, direct context switch to next task.
The next execution context must be explicitly specified.



Resuming **N** instances of `std::execution_context` takes **N+1** context switches.

Passing data Because of the asymmetric resume/suspend operations of N4402, the proposal works well for generator examples, returning data from the resumable function.

Passing data into the body of resumable functions (N4402) requires helper classes like `channel<>`.

```
// N4402: stackless resumable function
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;
}
```

In this case, the way data are passed into the body is not intuitive and introduces some problems.

Experience with `execution_context` from Boost.Context[13] turned up a common pattern: to pass a lambda to the `execution_context`, using its capture list to transfer data into and to return data from the body. Parameters (input/output) are accessed via captured references or pointers.

```
// P0099: stackful execution context
class X{
private:
    int* inp_;
    std::string outp_;
    std::exception_ptr excp_;
    std::execution_context caller_;
    std::execution_context callee_;

public:
    X():
        inp_(nullptr),outp_(),
        caller_(std::execution_context::current()),
        callee_([=](){
                    try {
                        outp_=lexical_cast<std::string>(*inp_);
                    } catch (...) {
                        excp_ = std::current_exception();
                    }
                    caller_(); // context switch to main()
                })
    {}

    std::string operator()(int i){
        inp_=&i;
        callee_(); // context switch to coroutine (lambda)
        if (excp_) {
            std::rethrow_exception(excp_);
        }
        return outp_;
    }
};

int main(){
    X x;
    try{
        std::cout<<x(7)<<std::endl;
    }catch(const std::exception& e){
        std::cout<<"exception: "<<e.what()<<std::endl;
    }
}
```

Class `X` (rudimentary coroutine) demonstrates how input and output parameters are transferred between contexts. Member variable *callee_* represents a new execution context and captures the *this*-pointer of `X`. The body converts an integer variable (input) into a string (output). Any exception thrown by the conversion is transported and re-thrown in main().

Stack strategies For stackful coroutines two strategies are typical: a contiguous, fixed-size stack (as used by threads), or a linked stack (grows on demand).
The advantage of a fixed-size stack is the fast allocation/deallocation of activation records. A disadvantage is that the required stacksize must be guessed.
The benefit of using a linked stack is that only the initial size of the stack is required. The stack itself grows on demand, by means of an overflow handler. The performance penalty is low. The disadvantage is that code

executed inside a stackful coroutine must be compiled for this stack model. In the case of GCC's split stacks, special compiler/linker flags must be specified - no changes to source code are required.
When calling a library function not compiled for linked stacks (expecting a traditional contiguous stack), GCC's implementation uses link-time code generation to change the instructions in the caller. The effect is that a reasonably large contiguous stack chunk is temporarily linked in to handle the deepest expected chain of traditional function stack frames (see GCC's documentation[10]).

Design

Class `std::execution_context` is derived from Boost.Context.[13] It provides a **small, basic API** on which to build **higher-level APIs** such as stackful coroutines (N3985[1]) and user-mode threads (cooperative multi-tasking).
In effect, `execution_context` is a copyable handle to an underlying noncopyable *capture record*.
`execution_context::operator()` preserves the CPU register set[*] + stack pointer: the content of those registers is stored in the capture record associated with `execution_context::current()`. Then `operator()` loads the CPU register set and stack pointer from the capture record associated with `*this`.
Finally, `execution_context::operator()` makes `*this` become `current()`.

Class `std::execution_context`

is like a handle to a *capture record* of an execution context.

```
// P0099: stackful execution context
std::execution_context l1=std::execution_context::current();
std::execution_context l2(
    std::allocator_arg,
    fixedsize(4048),
    [&l1](){
        ...
    });
std::execution_context l3(
    std::allocator_arg,
    protected_fixedsize(4048),
    [&l2](){
        ...
    });
```

Because of the symmetric context switching (only one operation transfers control), the target execution context must be explicitly specified.
Exchanging data between different execution contexts requires the use of lambda captures.

First-class object As first-class object the execution context can be stored in a variable or container.

Capture record Each instance of `std::execution_context` owns a toplevel activation record, the capture record. The capture record is a special activation record that stores additional data such as stack pointer and instruction pointer. That means that during a context switch, the execution state of the running context is captured and stored in the capture record while the content of the resumed execution context is loaded (into CPU registers etc.).

Active context The static member function `current()` returns a `std::execution_context` pointing to the current capture record. The current active capture record is stored in an internal, thread local pointer.

Toplevel capture records On entering `main()` as well as the *entry-function* of a thread, an execution context (capture record) is created and stored in the internal pointer underlying `current()`. (In practice, this could be lazily instantiated.)

[*]defined by ABI's calling convention

Suspend-on-call `std::execution_context` is designed to suspend on call of `operator()()`. The *prologue* of `operator()()` captures (preserves) the active execution context and loads the data from the capture record of the resumed context (**this*) into the CPU.

Termination If the toplevel context function returns, `std::exit(0)` is called.

Exceptions If an uncaught exception escapes from the toplevel context function, `std::terminate` is called.

Stack allocators are used to create stacks.

protected_fixedsize Constructs a linear stack of specified size, appending a guard page at the end of each stack to protect against overflow. If the guard page is accessed (read or write operation) a segmentation fault/access violation is generated by the operating system.

member functions

(constructor) constructs new stack allocator

`protected_fixedsize(std::size_t size=default_stacksize)`	(1)

1) fixed size stack, `size` determines the stack size (`default_stacksize` is platform dependent)

Notes
At the end of the protected stack a guard page is appended.

fixedsize Constructs a linear stack of specified size. In contrast to `protected_fixedsize`, it does not append a guard page at the end of each stack. The memory is simply managed by `std::malloc()` and `std::free()`.

member functions

(constructor) constructs new stack allocator

`fixedsize(std::size_t size=default_stacksize)`	(1)

1) fixed size stack, `size` determines the stack size (`default_stacksize` is platform dependent)

segmented Creates a segmented stack with the specified initial size, which grows on demand.

member functions

(constructor) constructs new stack allocator

`segmented(std::size_t size=default_initial_stacksize)`	(1)

1) stack grows on demand, `size` determines the initial stack size (`default_initial_stacksize` is platform dependent)

synopsis declaration of class `std::execution_context`

```
class execution_context {
public:
    static execution_context current() noexcept;

    template<typename StackAlloc, typename Fn, typename ... Args>
    execution_context(std::allocator_arg_t, StackAlloc salloc,
                      Fn&& fn, Args&& ... args);

    template<typename Fn, typename ... Args>
    explicit execution_context(Fn&& fn, Args&& ... args);

    execution_context( execution_context const&)=default;
    execution_context( execution_context &&)=default;

    execution_context& operator=( execution_context const&)=default;
    execution_context& operator=( execution_context &)=default;

    void operator()() noexcept;
};
```

member functions

(constructor) constructs new execution context

`template<typename StackAlloc, typename Fn, typename ... Args>`
`execution_context(std::allocator_arg_t, StackAlloc salloc,`
`Fn&& fn, Args&& ... args)`
	(1)

`template<typename Fn, typename ... Args>`
`explicit execution_context(Fn&& fn, Args&& ... args)`
	(2)

`execution_context(const execution_context& other)=default`	(3)
`execution_context(execution_context&& other)=default`	(4)

1) this constructor takes (e.g.) a lambda as argument, stack is constructed using *salloc*

2) takes (e.g.) lambda as argument, stack is constructed using either `fixedsize` or `segmented`. An implementation may infer which of these best suits the code in `fn`. If it cannot infer, `fixedsize` will be used.

3) copies `std::execution_context`, i.e. underlying capture record is shared

4) moves underlying capture record to new `std::execution_context`

Notes
When an `execution_context` is constructed using either of the constructors accepting `fn`, control is *not* immediately passed to `fn`. Resuming (entering) `fn` is performed by calling `operator()()` on the new `execution_context` instance.

If an instance of `std::execution_context` is copied, both instances share the same underlying capture record.

(destructor) destroys an execution context

`~execution_context()`	(1)

1) destroys a `std::execution_context`. If associated with a context of execution and holds the last reference to the internal capture record, then the context of execution is destroyed too. Specifically, the stack is unwound.

operator= copies/moves the context object

`execution_context& operator=(execution_context&& other)`	(1)
`execution_context& operator=(const execution_context& other)`	(2)

1) assigns the state of *other* to **this* using move semantics

2) copies the state of *other* to **this*, state (capture record) is shared

Parameters

other another execution context to assign to this object

Return value

***this**

operator() jump context of execution

`execution_context& operator()()`	(1)

1) suspends the active context, resumes the execution context

Exceptions

1) calls `std::terminate` if an exception escapes toplevel `fn`

Notes

The *prologue* preserves the execution context of the calling context as well as stack parts like *parameter list* and *return address*.[*] Those data are restored by the *epilogue* if the calling context is resumed.
If an uncaught exception escapes from the execution context, `std::terminate` is called.
If the toplevel context function returns (reaches end), `std::exit(0)` is called.
The only way to leave an `execution_context` without terminating the calling process is to call another `execution_context`'s `operator()()`.
A suspended `execution_context` can be destroyed. Its resources will be cleaned up at that time.
The behaviour is undefined if `operator()` is called while `current()` returns **this* (attempting to resume an already running context).

current accesses the current active execution context

`static execution_context current()`	(1)

1) construct an instance of `std::execution_context` pointing to the capture record of the current, active execution context

Notes

The current active execution context is thread-specific, e.g. for each thread (including `main()`) an execution context is created at start-up.

[*]required only by some x86 ABIs

Acknowledgements

The authors would especially like to thank Christopher Kohlhoff and Ville Voutilainen for encouragement and valuable suggestions.

References

[1] N3985: A proposal to add coroutines to the C++ standard library, Revision 1

[2] N4024: Distinguishing coroutines and fibers

[3] N4045: Library Foundations for Asynchronous Operations, Revision 2

[4] N4134: Resumable Functions v.2

[5] N4244: Resumable Lambdas

[6] N4397: A low-level API for stackful coroutines

[7] N4402: Resumable Functions v.4

[8] N4453: Resumable Expressions

[9] P0112: Networking Library Proposal (Revision 6) (supersedes N4478)

[10] Split Stacks / GCC

[11] Segmented Stacks / LLVM

[12] Library *Boost.Asio*: documentation

[13] Library *Boost.Context*: git repo, documentation

[14] Library *Boost.Coroutine2*: git repo, documentation

[15] Library *Boost.Fiber*: git repo, documentation

[16] *cactus stack*: cactus stack, parent pointer tree

A. recursive descent parser using `std::execution_context`

This example (taken from Boost.Context[13]) uses a recursive descent parser for processing a stream of characters. The parser (a SAX parser or other event-dispatching framework would be equivalent) uses callbacks to push parsed data to the user code. With `std::execution_context` the user code inverts the control-flow. In this case, the user code *pulls* data from the parser.

```
// P0099: stackful execution context
// grammar:
//   P ---> E '\0'
//   E ---> T {('+'|'-') T}
//   T ---> S {('*'|'/') S}
//   S ---> digit | '(' E ')'
class Parser{
   int level;
   char next;
   istream& is;
   function<void(char)> cb;

   char pull(){
        return char_traits<char>::to_char_type(is.get());
   }

   void scan(){
       do{
           next=pull();
       }
       while(isspace(next));
   }

public:
   Parser(istream& is_,function<void(char)> cb_) :
      level(0),next(),is(is_),cb(cb_)
    {}

   void run() {
      scan();
      E();
   }

private:
   void E(){
      T();
      while (next=='+'||next=='-'){
         cb(next); // callback; signal new symbol
         scan();
         T();
      }
   }

   void T(){
      S();
      while (next=='*'||next=='/'){
         cb(next); // callback; signal new symbol
         scan();
         S();
      }
   }

   void S(){
      if (isdigit(next)){
         cb(next); // callback; signal new symbol
         scan();
      }
      else if(next=='('){
         cb(next); // callback; signal new symbol
         scan();
         E();
         if (next==')'){
             cb(next); // callback; signal new symbol
             scan();
         }else{
             throw std::runtime_error("parsing failed");
         }
      }
      else{
          throw std::runtime_error("parsing failed");
      }
   }
};

int main(){
    std::istringstream is("1+1");
    char c;
    std::exception_ptr excp;
    // access current execution context
    auto m=std::execution_context::current();
    std::execution_context l(
        std::allocator_arg,
        segmented(1024),
        [&is,&m,&c,&excp](){
            Parser p(is,
                     // callback, used to signal new symbol
                     [&m,&c](char ch){
                        c=ch;
                        m(); // resume main-context
                     });
            try {
                p.run(); // start parsing
            } catch (...) {
                excp = std::current_exception();
            }
        });
    try {
        // inversion of control: user-code pulls parsed symbols from parser
        while(l()){
            if (excp) {
                std::rethrow_exception(excp);
            }
            std::cout<<"Parsed: "<<c<<std::endl;
        }
    } catch(const std::exception& e){
        std::cerr<<"exception: "<<e.what()<<std::endl;
        }
}
```

You cannot use resumable functions for this example without transforming all functions in the call chain into resumable functions - which is impossible for closed source code, impractical at best for a large, complex code base.