Document number: N4398
Date: 2015-04-09
Project: WG21, SG1
Author: Oliver Kowalke (oliver.kowalke@gmail.com)

A unified syntax for stackless and stackful coroutines

Abstract

This paper proposes a *unified syntax* for stackless and stackful coroutines. The syntax is based on N4397.[3]

The most important features are:

- first-class object that can be stored in variables or containers
- introduction of new keyword `resumable` together with a lambda-like expression
- symmetric transfer of execution control, e.g. suspend-by-call - enables a richer set of control flows than asymmetric transfer of control
- ordinary function calls and returns not affected

Background

At the November 2014 meeting in Urbana the committee decided to pursue both kinds of coroutines and encouraged the community to propose a unified syntax.
This paper proposes a syntax suitable for stackless and stackful coroutines based on the ideas of proposal N4397.[3]

Introduction

Traditionally C++ code is compiled for a linear stack. That means that 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.
Coroutines enable a more advanced control flow. That requires that a coroutine outlives the context in which it was created (the activation record of a suspended coroutine **must not** be **removed**)! Introducing fixed activation records into the middle of a linear stack essentially makes normal stack operations unworkable. Traditional stack management is inadequate for coroutines.

N4397[3] describes the *first-class* construct `std::execution_context`, representing an execution state. A program implicitly contains at least one execution context. As explained in N4397, `std::execution_context`

can be used to implement stackful coroutines as proposed in N3985[1] or might be the building block of *oneshot delimited continuations* operators*.
In the remaining chapters the proposal describes under which constraints `std::execution_context` can be used for stackless and stackful context switching.

Definitions

This proposal uses the wording (definitions) of N4397.[3]

Discussion

N4397[3] describes `std::execution_context` as a mechanism to implement stackful context switching (for instance coroutines). Each context owns its own side stack.
How can this formalism be used to express *stackless* as well as *stackful* execution contexts? The answer is the concept of *suspend-by-calling*.

Calling convention A calling convention is a scheme, part of the ABI†, that describes how a subroutine must be called. This includes parameter list, return address, *stack layout* and cleanup.
Some calling conventions (for instance x86 architecture) require that data like *parameter list* as well as *return address* are stored on the caller's stack before the subroutine is invoked. Other calling conventions of other architectures (for instance ARM's AAPCS) do not have this constraint ‡, e.g. stack consumption is minimized§
The proposed syntax for stackless and stackful context switching requires that the stack is clean: no parameter list and no return address may remain on the caller's stack when a context switch happens. This is required since a coroutine can outlive the context in which it was created.
In other words, those parts that would remain on the caller's stack must be preserved and restored by a context switch (member function `std::execution_context::operator()()`). This supports the *suspend-by-calling* concept. Previous proposals like N4134[2] describe a *suspend-by-returning* mechanism, i.e. the coroutine is suspended by calling `yield` etc. (for resumable functions a return value transformation happens – the return value is substituted by a future-like object).

Suspend-by-calling *suspend-by-calling* requires a symmetric transfer of execution control as well as first-class objects in order to specify the next context to be resumed.
This enables an arbitrary flow of context switches providing a broad range of control flows (for instance *delimited continuations*).
As a consequence the part belonging to the called function on the caller's stack must be popped and preserved in the caller's *capture record*. The caller context suspends by calling
`std::execution_context::operator()()` of another execution context. Other execution contexts are able to use the stack in the meanwhile. If the suspended context is resumed, the preserved data are pushed to the stack and execution returns from `std::execution_context::operator()()`.

Prologue The prologue of `std::execution_context::operator()()` updates the capture record (CPU registers) and pops some parts (part of callee's stack frame) from caller's stack and preserves the data into caller's capture record too.

Epilogue If the context (caller context from above) is resumed, the *epilogue* of
`std::execution_context::operator()()` loads the capture record and pushes callee's partial stack frame on caller's stack. The calling convention remains intact, and the code which called

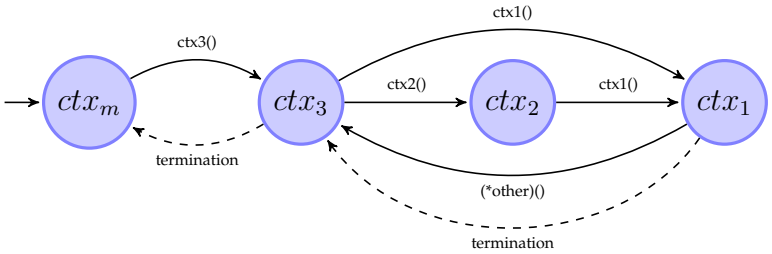
*Shift, Reset, and All That
A delimited continuation marks a part of the continuation frame so that the compiler can reify it in into a functor (closure). Delimited continuations returns a value and might be composed.
†Application Binary Interfaces; an executable must be conform to, in order to be executable in the specified execution environment
‡AAPCS64: parameters in registers R0-R7/V0-V7, return address in link register LR
§of course a long parameter list requires the stack; but this is negligible as shown in the text

`std::execution_context::operator()()` can not distinguish between an ordinary function and a context switch. It's completely transparent to the caller.

Flow of control Because `std::execution_context` uses a symmetric execution control transfer mechanism, the flow of control can be arbitrary.

```
// N4398: stackless execution context
int main(){
    std::execution_context* other=nullptr;
    auto ctx1=[other]()resumable{
        (* other)(); // suspend ctx1, resume other
    };
    auto ctx2=[&ctx1]()resumable{
        ctx1(); // suspend ctx2; resume ctx1
    };
    auto ctx3=[&ctx1,&ctx2]()resumable{
        ctx2(); // suspend ctx3, resume ctx2
        ctx1(); // suspend ctx3, resume ctx1
    };
    other=&ctx3;
    ctx3(); // suspend main context, resume ctx3
}
```

As shown in the example, the contexts of *ctx1*, *ctx2* and *ctx3* form a cycle of flow of control.



The cycle is started by calling `ctx3()` from the main context (ctx_m, created on start-up). Context ctx_3 starts ctx_2 while ctx_2 resumes ctx_1. Context ctx_1 is suspended by resuming ctx_3 with `(*other)()`. Function `ctx2()` returns in ctx_3 and `ctx1()` next resumes ctx_1. Context ctx_1 terminates after returning from `(*other)()`. After termination of ctx_1, ctx_3 is resumed because it has become the parent context of ctx_1 (by calling `ctx1()`). As ctx_3 terminates, the main context ctx_m is resumed (return from `ctx3()`).

Stackless and stackful The compiler allocates for a *stackless context* only one, suitable sized, capture record and for a *stackful context* a side stack (linked stack, e.g. non-contiguous, growing on demand).
In order to decide what kind of context has to be generated, the compiler has to analyse the toplevel context function.The following use cases must be distinguished:

- no context switch: generate an ordinary function
- context switch at toplevel: create a *stackless* context
- in all other cases: generate a *stackful* context

```
// N4398: stackless execution context
// suspend in body of toplevel context function
#define yield(x) p=x; mctx();
int main(){
    int n=35;
    int p=0;
    auto mctx(std::execution_context::current());
```

```
    auto ctx([n,&p,mctx]()resumable{
            int a=0,b=1;
            while(n-->0){
                yield(a);
                auto next=a+b;
                a=b;
                b=next;
            }
        });
    for(int i=0;i<10;++i){
        ctx();
        std::cout<<p<<std::endl;
    }
}
```

In other words, if `std::execution_context::operator()()` is called inside the body of the toplevel context function (as shown in Fibonacci example above) a *stackless* context is sufficient.

```
// N4398: stackful execution context
// example taken from proposal N4397
int main(){
    std::istringstream is("1+1");
    char c;
    // access current execution context
    auto m=std::execution_context::current();
    // use of linked stack (grows on demand) with initial size of 1KB
    std::execution_context l(
    auto l=[&m,&is,&c]()resumable{
            Parser p(is,
                    // callback, used to signal new symbol
                    [&m,&c](char ch){
                       c=ch;
                       m(); // resume main-context from nested call stack
                    });
            p.run(); // start parsing
        });
    try{
        // inversion of control: user-code pulls parsed data from parser
        while(l){
            l(); // resume parser-context
            std::cout<<"Parsed: "<<c<<std::endl;
        }
    }catch(const std::exception& e){
        std::cerr<<"exception: "<<e.what()<<std::endl;
    }
}
```

Calling a context switch from a nested call stack requires a *stackful* context.

Design

The design of `std::execution_context` is based on N4397[3] with one modification - the *lambda-like expression* does not have the *hint* attribute.
Instead the compiler desides if one activation record is sufficient (*stackless*) or a side stack is required (*stackful*). The compiler makes the decision based on the analysis of the toplevel context function. If the compiler can prove that context switches are only done at the toplevel function and not from nested call stack (from subroutines), than a *stackless* execution context is created. Otherwise the compiler constructs a

`std::execution_context` with a non-contiguous, linked side stack. The initial default stacksize depends on the platform.

First-class object As a 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 like stack pointer, instruction pointer and a link to its parent execution context. That means that during an execution 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.).

Parent context The pointer to its *parent* execution context allows traversing the chain of ancestor contexts, i.e. the execution context which has resumed (called `std::execution_context::operator()()` on) the running context.

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

Toplevel capture records On entering `main()` as well as the *thread-function* of a thread, an execution context (capture record) is created and stored in the internal thread-local pointer underlying `std::execution_context::current()`.

Termination If the body of the toplevel context function reaches its end, the parent execution context (pointer in the capture record) is resumed. That means that in the parent context the function `std::execution_context::operator()()` returns. For this purpose the *epilogue* loads the capture record (instruction pointer, stack pointer etc.) of the parent context, so that it is resumed.

Exceptions An exception thrown inside the execution context is caught, the parent execution context is resumed and the exception is re-thrown inside the parent context: the exception is emitted by `std::execution_context::operator()()`.

member functions

(constructor) constructs new execution context

`[captures](params) mutable resumable attrs -> ret {body}`	(1)
`execution_context(const execution_context& other)=default`	(2)
`execution_context(execution_context&& other)=default`	(3)

1) the constructor does not take a lambda as argument, instead the compiler evaluates the lambda-like syntax and constructs a `std::execution_context` directly

captures list of captures

params only empty parameter-list allowed

mutable parameters captured by copy can be modified by *body*

resumable identify resumable context

attrs attributes for `operator()`

ret only `void` allowed (use capture list instead)

body function body

2) copies `std::execution_context`, e.g. underlying capture record is shared

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

(destructor) destroys a execution context

`~execution_context()`	(1)

1) destroys an `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 coroutine 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) re-throws the exception of the resumed context in the parent context

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.
An exception thrown inside execution context is caught, the parent execution context is resumed and the exception is re-thrown in the parent context (out of `std::execution_context::operator()()`).
If the toplevel context function terminates (reaches end), the parent context is resumed (return of `std::execution_context::operator()()` in the parent execution context).
The behaviour is undefined if `operator()` is called while `current()` returns **this* (e.g. resuming an already running context).

explicit bool operator test if context has not reached its end

```
explicit bool operator() noexcept   (1)
```

1) returns *true* if context is not terminated

Exceptions

1) noexcept specification: `noexcept`

operator! test if context has reached its end

```
bool operator!() noexcept   (1)
```

1) returns *true* if context is terminated

Exceptions

1) noexcept specification: `noexcept`

current accesses the current active execution context

```
static execution_context current()   (1)
```

1) construct a 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: for each thread (including `main()`) an execution context is effectively created at start-up.

* required only by some x86 ABIs

Acknowledgement

I'd like to thank Nat Goodspeed for reviewing earlier drafts of this proposal.

References

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

[2] N4134: Resumable Functions v.2

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

[4] Library *boost.context*: git repo, documentation