Document number: N4397
Date: 2015-03-23
Project: WG21, SG1
Author: Oliver Kowalke (oliver dot kowalke at gmail dot com)

A low-level API for stackful coroutines

Revision History This document supersedes N3985 and elaborates a low-level API for stackful context switching.

Background

At the meeting in Urbana the committee accepted that stackless and stackful coroutines are distinct and decided to pursue both kinds of coroutines.
This paper proposes a low-level API for context switching suitable to act as building-block for high-level constructs like stackful coroutines from N3985[1] (boost.coroutine2[7]) or cooperative multitasking (boost.fiber[8]).
This proposal evaluates an alternative, 'resumable lambda'-like syntax for stackful context switching.

Definitions

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

Control block holds a set of registers (callee-saved registers, instruction pointer, stack pointer) describing the execution context.

Coroutine enables explicit suspend and resume of its progress via additional operations by preserving execution state and thus provides an enhanced control flow. Coroutines have following characteristics:[1]

- values of local data persist between successive 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

Asymmetric coroutine provides two distingt 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, e.g. suspending the coroutine transferes the execution control back to the point in the code were it was called. The asymmetric control transfer-mechanism is usually used in the context of generators.

Symmetric coroutine only one operation to resume/suspend the context is available.
A symmetric coroutine does not know its caller, e.g. the execution control can be transfered to any other symmetric coroutine (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]).

Resumable function N4134[3] describes resumable functions as an efficient language supported mechanism for stackless context switching introducing two new keywords - `await` and `yield`. Resumable functions are equivalent to asymmetric coroutines.
Characteristics of resumable functions:

- stackless
- allocates memory (activation frame) for the body of the resumable function to store local data and control block
- thight coupling between caller and resumable function (asymmetric control transfer-mechanism)
- implicit *return*-statement[3] (code transformation)

Processor stack also known as call stack - is a chunk of memory into which the processor's stack pointer is pointing. The processor stack might belong to:

- an application's inital (or only) thread
- an explicitly-launched thread
- a sub-thread execution context (coroutine)

The stack is used to store data like local variables, content of CPU registers (because of subroutine calls) etc. The processor stack assigned to function `main()` grows on demand while the stack assigned to a thread has a fixed size - usually 1MB (Windows) till 2MB (Linux), but some platforms use smaller stacks (64KB on HP-UX/PA and 256KB on HP-UX/Itanium).

Side stack is a call stack that is used in the case of stackful context switching, e.g. each execution context gets its own stack (assigned to stack pointer). Thus stack frames of subroutines are allocated on the side stack, the application stack remains unchanged.

Activation frame is a chunk of memory used by resumable functions to store local (stack) variables and the control block. Each resumable function is bound to its own activation frame. The stack pointer remains unchanged, e.g. it still points to the processor stack. Thus stack frames of subroutines are allocated on the processor stack.

Linked stack also known as *split stack*[4] or *segmented stack*,[5] represents a on demand growing stack with a non-contigous address range.
Applications compiled with support for linked stacks can use (link against) libraries not supporting linked stacks (see GCC's documentation,[4] chapter 'Backward compatibility').

Parent pointer tree data structure in which each node has a pointer to its parent, but no pointer to its children. Traversion from any node to its ancestors is possible but not to any other node.
Used as call stack, the structure is called *cactus stack*.

Discussion

Memory requirements For simple function bodies, stackless and stackful context switching have similar memory requirements.

Resumable functions require storage for local (stack) variables and the control block (preserved CPU register). The compiler analyzes the function body, determines the required size and allocates the activation frame[*]

```
// N4134: stackless resumable function
generator<int> fib(int n){
    int a=0,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;
    }
}
```

In the example of calculating Fibonacci numbers using a resumable function (simple function body, no subroutine is called), the compiler reserves space (activation frame) for local variables *n*, *a*, *b* and *next*. Those variables are not accessed via the stack pointer. The processor stack is not used and the stack pointer is not changed[†].

Each stackful execution context owns a stack, e.g. the side stack is assigned to the stack pointer. The control block (CPU register) is pushed to/ poped from the side stack, thus the applciation stack remains unchanged.

```
// N4397: stackful execution context
#define yield(x) p=x; mctx();
int main(){
    int n=35;
    int p=0;
    auto mctx(std::execution_context::current());
```

[*]as N4134 describes, the allocation of the activation frame can be optimized out if the resumable function does not yield
[†]depending on the calling convetion, *x86* requires to store parameters and return address on the stack - other architectures/ABIs do not have this requirement

```
    auto ctx([n,&p,mctx]()mutable resumable(fixedsize(1024)){
            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 the case of calculating the Fibonacci numbers using a stackful execution context, the compiler stores the local variables *n*, *a*, *b* and *next* on the side stack. The local variables are accessed via the stack pointer.

Calling subroutines Stackless coroutines do not need to allocate memory for stack frames for called subroutines while (in contrast) stackful context switching permitts suspention from nested call stack.

Resumable functions share the same processor stack, e.g. the stack pointer is unchanged and holds an address from the processor stack (`main()` or thread).
If a subroutine is invoked inside the body of a resumable function, then a stack frame is allocated on the processor stack (not on the activation frame)! As a consequence stack frames of subroutines have to be removed from the application stack before the resumable function yields back to its caller (suspend to its resumption point). Otherwise the stack frames would be corrupted (overwritten) by other code using the processor stack.
In other words: the calling convention of the ABI* dictates that, after the resumable function returns (suspended), the stack pointer contains the same address as before the resumable function was resumed.
Hence a yield from nested call stack is not permitted.
The benefit of stackless resumable context switching consists in sharing the processor stack because a resumable function does not need to allocate memory for the stack frames of subroutines, invoked inside the body of that resumable function.

Of course even stackless resumable functions might fail if the available stack space is exceeded by subroutines (for instance recursive invocation).

In the context of stackful context switching each execution context owns a distinct side stack. In contrast to resumable functions the side stack is assigned to the stack pointer (e.g. the stack pointer has to be exchanged during each context switch).
As a consequence, all local variables and all stack frames of subroutines are allocated 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 stackful context switching has a higher memory footprint than resumable functions (but this applies only in the context of calling subroutines).
On the other side, it is benefital to use side stacks because the stack frames of subroutines remain intact while the execution context gets suspended. This is the reason why stackful context switching permits yielding from nested call stack.

```
// N4397: stackful execution context
// access current execution context l1
auto l1=std::execution_context::current();
// create stackful execution context l2
auto l2=[&l1]()resumable(segmented()){
    std::printf("inside l2\n");
    // suspend l2 and resume l1
```

*Application Binary Interface

```
    l1();
}
// resume l2
l2();
```

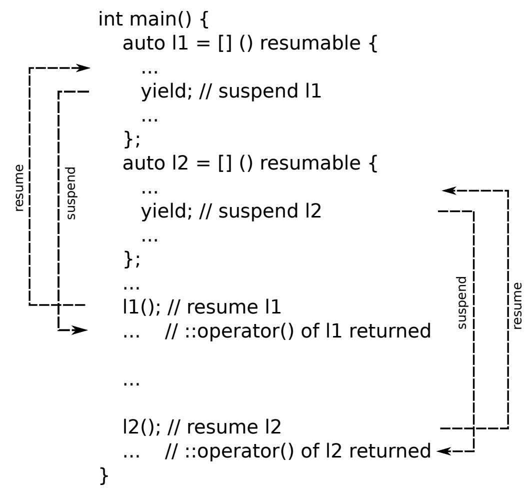
Variable *l1* is passed to stackful execution context *l2* via the capture list of *l2*. The stack frames of `std::printf()` are allocated on the side stack that *l2* owns.

Control block The number of registers preserved (in the control block) by a context switch is insignificant for the overall performance.

The number of CPU registers which have to be preserved (e.g. to preserve the execution state) are described by the calling convention (part of the ABI).
A load/store operation of CPU register takes usually one CPU cycle and some architectures provide instructions to load/store multiple registers at once.
Of course a smart compiler might optimize the context switch by reducing the register set due code analyzation. But compared with ordinary code like memory allocation, which takes multiple hundreds of CPU cycles, the load/store of CPU registers is negligible.

Asymmetric vs. symmetric Symmetric transfer of execution control is more efficient than the asymmetric mechanism.

Resumable functions (as proposed in N4134) provide two operations for context switching (asymmetric transfer of execution control). The caller and the resumable function are coupled, e.g. a resumable function can only jump back to the point in the code where it was resumed.



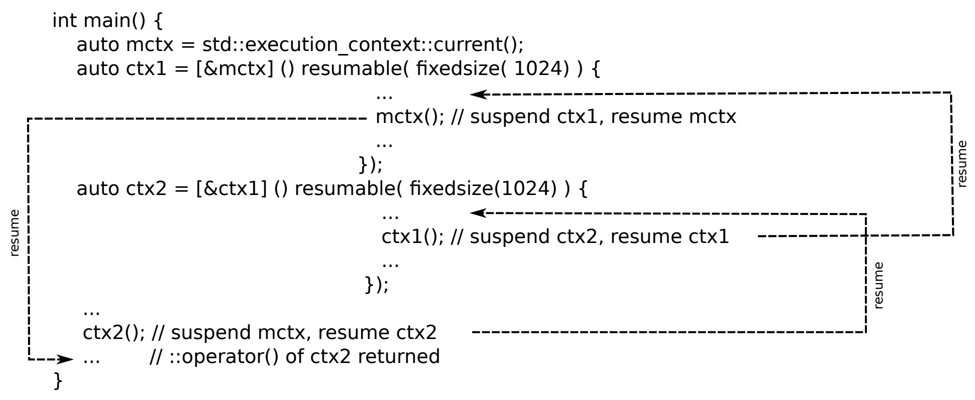
```
int main() {
    auto l1 = [] () resumable {
        ...
        yield; // suspend l1
        ...
    };
    auto l2 = [] () resumable {
        ...
        yield; // suspend l2
        ...
    };
    ...
    l1(); // resume l1
    ...    // ::operator() of l1 returned

    ...

    l2(); // resume l2
    ...    // ::operator() of l2 returned
}
```

For N resumable functionss *2N* context switches are required.

This is sufficient in the case of generators but in the context of cooperative multitasking it is inefficient. In contrast to resumable functions the proposed stackful execution context (`std::execution_context`) provides only one operation to resume/suspend the context (`std::execution_context::operator()()`). The execution control is directly transfered from one execution context to another (symmetric transfer-mechanism) - no jump back to the caller. This enables a efficient implementation of cooperative multitasking (no additional context switch back to caller, e.g. direct context switch to next task) as well as the implementation of generators. As an consequence, the next execution context has to be explicitly specified.



```
int main() {
    auto mctx = std::execution_context::current();
    auto ctx1 = [&mctx] () resumable( fixedsize( 1024) ) {
                                        ...
                                        mctx(); // suspend ctx1, resume mctx
                                        ...
                                    });
    auto ctx2 = [&ctx1] () resumable( fixedsize(1024) ) {
                                        ...
                                        ctx1(); // suspend ctx2, resume ctx1
                                        ...
                                    });
     ...
     ctx2(); // suspend mctx, resume ctx2
     ...       // ::operator() of ctx2 returned
}
```

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

Passing data Because of the asymmetric resume/suspend operations of N4134, the proposal applies well to generator-examples, e.g. returning data from the resumable function.

Passing data into the body of resumable functions (N4123) requires helper classes like `channel<>`.

```
// N4134: 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 how data are passed into the body is not intuitive and provides some problems.

During the use of `execution_context` from boost.context[6] a common pattern was to pass lambdas to the execution context and use the capture list to transfer data into and to return data from the body. Exceptions are transfered via `std::exception_ptr` and parameters (input/output) are accessed via captured addresses (reference or pointer).

```
// N4397: stackful execution context
class X{
private:
    int* inp_;
    std::string outp_;
    std::exception_ptr excptr_;
    std::execution_context caller_;
    std::execution_context callee_;

public:
    X():
        inp_(nullptr),outp_(),excptr_(),
        caller_(std::execution_context::current()),
        callee_([=]()resumable(segmented(1024)){
                     try{
                         outp_=lexical_cast<std::string>(*inp_);
                         caller_(); // context switch to main()
                     }catch (...){
                         excptr_=std::current_exception();
                     }
                 })
    {}

    std::string operator()(int i){
        inp_=&i;
        callee_(); // context switch for conversion
        if(excptr_){
            std::rethrow_exception(excptr_);
        }
        return outp_;
    }
};

int main(){
    X x;
    std::cout<<x(7)<<std::endl;
}
```

Class `X` (rudimentary coroutine) demonstrates how input and output parameter are transfered between a context switch. Member variable *callee_* represents a new execution context and captures the *this*-pointer of `X`. The body converts a integer variable (input) into a string (output). Exceptions, thrown by the conversion, are transported via member variable *exptr_*.

Summary *stackful* execution context

- each stackful execution context owns a side stack (fixed-size or linked stack)
- activation frame and stack are the same, not distinct
- stack is not shared
- suspend operation from nested call stack is permitted
- symmetric context switching is preferred because of efficiency
- passing data via lambda capture list (addresses of lvalues)

Higher level frameworks based on `std::execution_context` can realize more sophisticated APIs (like range based for etc.) - as an example the API of N3985[1] is implemented in boost.coroutine2[7] based on `execution_context`.

```
// N3985: stackful coroutine (boost.coroutine2)
// implemented with execution_context
typedef coroutine<int> coro_t;
int main(){
    int n=35;
    coro_t::pull_type fib([n](coro_t::push_type& yield){
            int a=0,b=1;
            while(n-->0){
                yield(a);
                auto next=a+b;
                a=b;
                b=next;
            }
```

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

Design

Class `std::execution_context` is derived from the work on boost.context[6] - it provides a small, basic API, suitable to implement high-level APIs for stackful coroutines (N3985,[1] boost.coroutine2[7]) and user-mode threads (executing tasks in a cooperative multitasking environment, boost.fiber[8]).

Class `std::execution_context` Based on the implementation experience with `execution_context` in boost.coroutine2[7] and boost.fiber,[8] the author noticed that `execution_context` is almost always used together with lambdas (passed as argument to the constructor of `execution_context`). Especially the lambda captures are suitable to transport data between different execution context's (address of lvalue, reside on stack/heap).
Why not construct `std::execution_context` with an 'resumable lambda'-like syntax, instead of passing a lambda as argument? `std::execution_context` is constructed like a lambda but actually it is not of this type (because it is constructible from `std::execution_context::current()` too).

```
// N4397: stackful execution context
std::execution_context l1=[]()resumable(fixedsize(1024)){
    ...
};
auto l2=[&l1]()resumable(segmented(1024)){
    ...
};
```

The keyword `resumable` together with an hint (attribute) about the type and size of the side stack tells the compiler to generate a stackful execution context.
Because of the symmetric context switching (only one operation transfers the control of execution) the target context must be explicitly specified.
Exchanging data between different execution context's requires the use of lambda captures.

Start-up A main execution context is created at startup (entering function `main()`; same applies for threads). The current (active) execution context can be accessed through the static member function `std::execution_context::current()`.

Control block Each instance of `std::execution_context` owns one control block and each control block contains a pointer to its *parent* control block, e.g. it for a *parent pointer tree*. With this data structure the parent context, who has resumed the execution context (`std::execution_context::operator()()` called), is known.
Static member function `std::execution_context::current()` returns a `std::execution_context` pointing to the current control block. The current active control block is stored in an internal thread local pointer.

Termination If the body (lambda) of the coroutine reaches its end, the parent execution context is resumed, e.g. function `std::execution_context::operator()()` returns in the parent context.

Exceptions A exception thrown inside the coroutine body (lambda) is catched, the parent execution context is resumed and the exception is re-thrown inside the parent context (e.g. the exception is re-throw in `std::execution_context::operator()()`).

member functions

(constructor) constructs new execution context

`[captures](params) mutable resumable(hint) exceptions 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 allows to modify parameters captured by copy

resumable identify resumable context

hint stack type hint:

- `fixedsize(x=default_stacksize)`: fixed size stack (`default_stacksize` is platform depended)
- `segmented(x=default_stacksize)`: stack grows on demand (`default_stacksize` is platform depended)

exceptions only `noexcept` allowed; emitted exceptions trigger `std::terminate()`

attrs attributes for `operator()`

ret only `void` allowed; resumable lambda returns nothing (use of capture list instead)

body function body

2) copies `std::execution_context`, e.g. underlying control block is shared

3) moves underlying control block to new `std::execution_context`

Notes
If an instance of `std::execution_context` is copied, both instances share the same underlying control block. Resuming one instance modifies the control block of the other `std::execution_context` too.

(destructor) destroys a 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 control block, 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 (control block) is shared

Parameters

other another execution context to assign to this object

Return value

***this**

Notes
If an instance of `std::execution_context` is copied, both instances share the same underlying control block. Resuming one instance modifies the control block of the other `std::execution_context` too.

operator() jump context of execution

`void operator()() noexcept`	(1)

1) resumes the execution context

Exceptions

1) noexcept specification: `noexcept`

Notes
A exception thrown inside the coroutine body (lambda) is catched, the parent execution context is resumed and the exception is re-throw in the parent context (out of `std::execution_context::operator()()`). If the coroutine 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).

explict bool operator test if context has not reached end of coroutine function

`explicit bool operator() noexcept`	(1)

1) returns *true* if context is not terminated

Exceptions

1) noexcept specification: `noexcept`

operator! test if context has reached end of coroutine function

`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 control block of the current, active execution context

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

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] N4024: Distinguishing coroutines and fibers

[3] N4134: Resumable Functions v.2

[4] Split Stacks / GCC

[5] Segmented Stacks / LLVM

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

[7] Library *boost.coroutine2*: git repo, documentation

[8] Library *boost.fiber*: git repo, documentation

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

This example (taken from boost.context[6]) 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.

```
// N4397: 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;
    // 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=[&is,&m,&c]()resumable(segmented(1024)){
            Parser p(is,
                     // callback, used to signal new symbol
                     [&m,&c](char ch){
                        c=ch;
                        m(); // resume main-context
                     });
            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;
    }
}
```

Code like this example is not possible without transforming all functions in the call chain into resumable functions - which is impossible for closed source code or for a too large/complex code base.