Document number: P0876R0
Date: 2018-02-11
Author: Oliver Kowalke (oliver.kowalke@gmail.com)
Audience: SG1

fibers without scheduler

abstract

At the November 2017 meeting in Albuquerque the members of the committee reaffirmed that stackful context switching is important and that we should continue pursuing the technology.

This paper addresses concerns, questions and suggestions from the past meetings. The proposed API supersedes the former proposals N3985,[5] P0099R1[7] and P0534R3.[8]
Because of name clashes with *coroutine* from coroutine TS, *execution context* from executor proposals and *continuation* used in the context of `future::then()`, *fiber*, as suggested by some committee members, has been chosen - *filament*, *tasklet*, *lightfiber* or *coop_task* are possible alternative names.

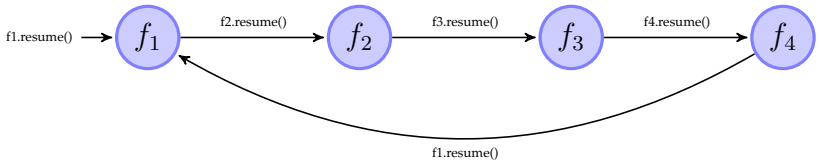
The minimal API enables stackful context switching **without** the need for a **scheduler**. The API is suitable to act as building-block for high-level constructs such as stackful coroutines as well as cooperative multitasking (aka user-land/green threads that incorporate a **scheduling facility**).

A fiber is represented by the first-class object `std::fiber`.

control transfer mechanism

According to the literature,[4] coroutine-like control-transfer operations can be distinguished into the concepts of *symmetric* and *asymmetric* operations.

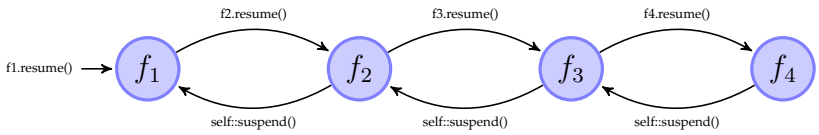
symmetric fiber A symmetric fiber provides a single control-transfer operation. This single operation requires that the control is passed explicitly between the fibers.




```
1  fiber* pf1;
2  fiber f4{[&pf1]{
3      pf1->resume();
4  };
5  fiber f3{[&f4]{
6      f4.resume();
7  }};
8  fiber f2{[&f3]{
9      f3.resume();
10 }};
11 fiber f1{[&f2]{
12     f2.resume();
13 }};
14 pf1=&f1;
15 f1.resume();
```

In the pseudo-code example above, a chain of fibers is created.
Control is transferred to fiber `f1` at line 15 and the lambda passed to constructor of `f1` is entered. Control is transferred from fiber `f1` to `f2` at line 12 and from `f2` to `f3` (line 9) and so on. Fiber `f4` itself transfers control directly back to fiber `f1` at line 3.

asymmetric fiber Two control-transfer operations are part of asymmetric fiber's interface: one operation for resuming (`resume()`) and one for suspending (`suspend()`) the fiber. The suspending operation returns control back to the calling fiber.




```
1  fiber f4{[]{
2      self::suspend();
3  };
4  fiber f3{[&f4]{
5      f4.resume();
6      self::suspend();
7  }};
8  fiber f2{[&f3]{
9      f3.resume();
10     self::suspend();
11 }};
12 fiber f1{[&f2]{
13     f2.resume();
14     self::suspend();
15 }};
16 f1.resume();
```

In the pseudo code above execution control is transferred to fiber `f1` at line 16. Fiber `f1` resumes fiber `f2` at line 13 and so on. At line 2 fiber `f4` calls its suspend operation `self::suspend()`. Fiber `f4` is suspended and `f3` resumed. Inside the lambda, `f3` returns from `f4.resume()` and calls `self::suspend()` (line 6). Fiber `f3` gets suspended while `f2` will be resumed and so on ...
The asymmetric version needs **N-1 more** fiber switches than the variant using symmetric fibers.

While asymmetric fibers establish a caller-callee relationship (strongly coupled), symmetric fibers operate as siblings (loosely coupled).

Symmetric fibers represent independent units of execution, making symmetric fibers a suitable mechanism for concurrent programming. Additionally, constructs that produce sequences of values (*generators*) are easily constructed out of two symmetric fibers (one represents the caller, the other the callee).

Asymmetric fibers incorporate additional fiber switches as shown in the pseudo code above. It is obvious that for a broad range of use cases, asymmetric fibers are less efficient than their symmetric counterparts.
Additionally, the calling fiber must be kept alive until the called fiber terminates. Otherwise the call of `suspend()` will be undefined behaviour (where to transfer execution control to?).

Symmetric fibers are more efficient, have fewer restrictions (no caller-callee relationship) and can be used to create a wider set of applications (generators, cooperative multitasking, backtracking ...).

fiber as a first-class object

Because the symmetric control-transfer operation requires explicitly passing control between fibers, fibers must be expressed as *first-class objects*.

Fibers exposed as first-class objects can be passed to and returned from functions, assigned to variables or stored into containers. With fibers as first-class objects, a program can **explicitly control the flow of execution** by suspending and resuming fibers, enabling control to pass into a function at exactly the point where it previously suspended.

Symmetric control-transfer operations require fibers to be first-class objects. First-class objects can be returned from functions, assigned to variables or stored into containers.

encapsulating the stack

Each fiber is associated with a stack and is responsible for managing the lifespan of its stack (allocation at construction, deallocation when fiber terminates). The RAII-pattern* should apply.

Copying a fiber must not be permitted!
If a fiber were copyable, then its stack with all the objects allocated on it must be copied too. That presents two implementation choices.

- One approach would be to capture sufficient metadata to permit object-by-object copying of stack contents. That would require dramatically more runtime information than is presently available – and would take considerably more overhead than a coder might expect. Naturally, any one move-only object on the stack would prohibit copying the entire stack.
- The other approach would be a bytewise copy of the memory occupied by the stack. That would force undefined behaviour if any stack objects were RAII-classes (managing a resource via RAII pattern). When the first of the fiber copies terminates (unwinds its stack), the RAII class destructors will release their managed resources. When the second copy terminates, the same destructors will try to doubly-release the same resources, leading to undefined behavior.

A fiber API must:

- **encapsulate the stack (hiding for user)**
- **manage lifespan of an explicitly-allocated stack: the stack gets deallocated when fiber goes out of scope**
- **prevent accidentally copying the stack**

Class `std::fiber` must be *moveable-only*.

*resource acquisition is initialisation

invalidation at resumption

The framework must prevent the resumption of an already running or terminated (computation has finished) fiber.
Resuming an already running fiber will cause overwriting and corrupting the stack frames (note, the stack is not copyable).
Resuming a terminated fiber will cause undefined behaviour because the stack might already be unwound (objects allocated on the stack were destroyed or the memory used as stack was already deallocated).
As a consequence each call of `resume()` will invalidate the `std::fiber` instance, i.e. no valid instance of `std::fiber` represents the currently-running fiber.
Whether a `std::fiber` is valid or not can be tested with member function `operator bool()`.
To make this more explicit, functions `resume()` and `resume_with()` are rvalue-reference qualified (binding only on rvalues).
The essential points:

- regardless of the number of `std::fiber` declarations, exactly one `std::fiber` instance represents each suspended fiber
- no `std::fiber` instance represents the currently-running fiber

Section solution: avoiding non-const global variables and undefined behaviour describes how an instance of `std::fiber` is synthesized from the active fiber that suspends.

A fiber API must:

- **prevent accidentally resuming a running fiber**
- **prevent accidentally resuming a terminated (computation has finished) fiber**
- **`resume()` and `resume_with()` are rvalue-reference qualified to bind on rvalues only**

problem: avoiding non-const global variables and undefined behaviour

According to C++ *core guidelines*,[9] non-const global variables should be avoided: they hide dependencies and make the dependencies subject to unpredictable changes.
Global variables can be changed by assigning them indirectly using a pointer or by a function call. As a consequence, the compiler can't cache the value of a global variable in a register, degrading performance (unnecessary loads and stores to global memory especially in performance critical loops).
Accessing a register is one to three orders of magnitude faster than accessing memory (depending on whether the cache line is in cache and not invalidated by another core; and depending on whether the page is in the TLB).
The order of initialisation (and thus destruction) of static global variables is not defined, introducing additional problems with static global variables.

A library designed to be used as building block by other higher-level frameworks should avoid introducing global variables. If this API were specified in terms of internal global variables, no higher level layer could undo that: it would be stuck with the global variables.

switch back to *main()* by returning Switching back to `main()` by returning from the fiber function has two drawbacks: it requires an internal global variable pointing to the suspended `main()` and restricts the valid use cases.

```
int main() {
    fiber f{[]{
        ...
        // switch to 'main()' only by returning
    }};
    f.resume(); // resume 'f'
    return 0;
}
```

For instance the generator pattern is impossible because the only way for a fiber to transfer execution control back to `main()` is to terminate. But this means that no way exists to transfer data (sequence of values) back and forth between a fiber and `main()`.

Switching to `main()` only by returning is impractical because it limits the applicability of fibers and requires an internal global variable pointing to `main()`.

static member function returns active fiber P0099R0[6] introduced a static member function (`execution_context::current()`) that returned an instance of the active fiber. This allows passing the active fiber `m` (for instance representing `main()`) into the fiber `l` via lambda capture. This mechanism enables switching back and forth between the fiber and `main()`, enabling a rich set of applications (for instance generators).

```
int main(){
    int a;
    fiber m=fiber::current(); // get active fiber
    fiber f{[&]{
        a=0;
        int b=1;
        for(;;){
            m=m.resume(); // switch to `main()`
            int next=a+b;
            a=b;
            b=next;
        }
    }};
    for(int j=0; j<10; ++j) {
        f=f.resume(); // resume `f`
        std::cout << a << " ";
    }
    return 0;
}
```

But this solution requires an internal global variable pointing to the active fiber and some kind of reference counting. Reference counting is needed because `fiber::current()` necessarily requires multiple instances of `std::fiber` for the active fiber. Only when the last reference goes out of scope can the fiber be destroyed and its stack deallocated.

```
fiber f1=fiber::current();
fiber f2=fiber::current();
assert(f1==f2); // f1 and f2 point to the same (active) fiber
```

Additionally a static member function returning an instance of the active fiber would violate the protection requirements of sections encapsulating the stack and invalidation at resumption. For instance you could accidentally attempt to resume the active fiber by invoking `resume()`.

```
fiber m=fiber::current();
m.resume(); // tries to resume active fiber == UB
```

A static member function returning the active fiber requires a reference counted global variable and does not prevent accidentally attempting to resume the active fiber.

solution: avoiding non-const global variables and undefined behaviour

The *avoid non-const global variables* guideline has an important impact on the design of the fiber API!

synthesizing the suspended fiber The problem of global variables or the need for a static member function returning the active fiber can be avoided by **synthesizing** the **suspended fiber** and passing it into the resumed fiber (as parameter when the fiber is started the first time or returned from `resume()`).

```
1 void foo(){
2     fiber f{[](fiber&& m){
3         m=std::move(m).resume(); // switch to `foo()`
4         m=std::move(m).resume(); // switch to `foo()`
5         ...
6     }};
7     f=std::move(f).resume(); // start `f`
8     f=std::move(f).resume(); // resume `f`
9 }
```

In the pseudo-code above the fiber `f` is started by invoking its member function `resume()` at line 7. This operation suspends `foo`, invalidates instance `f` and synthesizes a new fiber `m` that is passed as parameter to the lambda of `f` (line 2).

Invoking `m.resume()` (line 3) suspends the lambda, invalidates `m` and synthesizes a fiber that is returned by `f.resume()` at line 7. The synthesized fiber is assigned to `f`. Instance `f` now represents the suspended fiber running the lambda (suspended at line 3). Control is transferred from line 3 (lambda) to line 7 (`foo()`).
Call `f.resume()` at line 8 invalidates `f` and suspends `foo()` again. A fiber representing the suspended `foo()` is synthesized, returned from `m.resume()` and assigned to `m` at line 3. Control is transferred back to the lambda and instance `m` represents the suspended `foo()`.
Function `foo()` is resumed at line 4 by executing `m.resume()` so that control returns at line 8 and so on ...

Class `symmetric_coroutine<>::yield_type` from N3985[5] is **not** equivalent to the synthesized fiber.
`symmetric_coroutine<>::yield_type` does not represent the suspended context, instead it is a special representation of the same coroutine. Thus `main()` or the current thread's *entry-function* can **not** be represented by `yield_type` (see next section representing *main()* and thread's *entry-function* as fiber).
Because `symmetric_coroutine<>::yield_type()` yields back to the starting point, e.g. invocation of `symmetric_coroutine<>::call_type::operator()()`, both instances (`call_type` as well as `yield_type`) must be preserved. Additionally the caller must be kept alive until the called coroutine terminates or UB happens at resumption.

This API is specified in terms of passing the suspended fiber. A higher level layer can hide that by using private variables.

representing *main()* and thread's *entry-function* as fiber As shown in the previous section a synthesized fiber is created and passed into the resumed fiber as an instance of `std::fiber`.

```
int main(){
    fiber f{[](fiber&& m){
        m=std::move(m).resume(); // switch to `main()`
        ...
    }};
    f=std::move(f).resume(); // resume `f`
    return 0;
}
```

The mechanism presented in this proposal describes switching between stacks: each fiber has its own stack. The stacks of `main()` and explicitly-launched threads are not excluded; these can be used as targets too.
Thus every program can be considered to consist of fibers – some created by the OS (`main()` stack; each thread's initial stack) and some created explicitly by the code.
This is a nice feature because it allows (the stacks of) `main()` and each thread's *entry-function* to be represented as fibers. A fiber representing `main()` or a thread's *entry-function* can be handled like an explicitly created fiber: it can passed to and returned from functions or stored in a container.

In the code snippet above the suspended `main()` is represented by instance `m` and could be stored in containers or managed just like `f` by a scheduling algorithm.

The proposed fiber API allows representing and handling `main()` and the current thread's *entry-function* by an instance of `std::fiber` in the same way as explicitly created fibers.

fiber returns (terminates) When a fiber returns (terminates), what should happen next? Which fiber should be resumed next? The only way to avoid internal global variables that point to `main()` is to explicitly return a valid fiber instance that will be resumed after the active fiber terminates.

```
1 int main(){
2     fiber f{[](fiber&& m){
3         return std::move(m); // resume `main()` by returning `m`
4     }};
5     std::move(f).resume(); // resume `f`
6     return 0;
7 }
```

In line 5 the fiber is started by invoking `resume()` on instance `f`. `main()` is suspended and an instance of type `fiber` is synthesized and passed as parameter `m` to the lambda at line 2. The fiber terminates by returning `m`. Control is transferred

to `main()` (returning from `f.resume()` at line 5) while fiber `f` is destroyed.

In a more advanced example another fiber is used as return value instead of the passed in synthesized fiber.

```
1  int main(){
2      fiber m;
3      fiber f1{[&](fiber&& f){
4          std::cout << "f1: entered first time" << std::endl;
5          assert(!f);
6          return std::move(m); // resume (main-)fiber that has started `f2`
7      }};
8      fiber f2{[&](fiber&& f){
9          std::cout << "f2: entered first time" << std::endl;
10         m=std::move(f); // preserve `f` (== suspended main())
11         return std::move(f1);
12     }};
13     std::move(f2).resume();
14     std::cout << "main: done" << std::endl;
15     return 0;
16 }
17
18 output:
19 f2: entered first time
20 f1: entered first time
21 main: done
```

At line 13 fiber `f2` is resumed and the lambda is entered at line 8. The synthesized fiber `f` (representing suspended `main()`) is passed as a parameter `f` and stored in `m` (captured by the lambda) at line 10. This is necessary in order to prevent destructing `f` when the lambda returns. Fiber `f2` uses `f1`, that was also captured by the lambda, as return value. Fiber `f2` terminates while fiber `f1` is resumed (entered the first time). The synthesized fiber `f` passed into the lambda at line 3 represents the terminated fiber `f2` (e.g. the calling fiber). Thus instance `f` is invalid as the assert statement verifies at line 5. Fiber `f1` uses the captured fiber `m` as return value (line 6). Control is returned to `main()`, returning from `f2.resume()` at line 13.

The function passed to `std::fiber`'s constructor must have signature '`fiber(fiber&&)`'. Using `std::fiber` as the return value from such a function avoids global variables.

returning synthesized fiber instance from `resume()` An instance of `std::fiber` remains invalid after return from `resume()` or `resume_with()` – the synthesized fiber is returned, instead of implicitly updating the `std::fiber` instance on which `resume()` was called.
If the `std::fiber` object were implicitly udpated, the fiber would change its identity because each fiber is associated with a stack. Each stack contains a chain of function calls (call stack). If this association were implicitly modified, unexpected behaviour happens.
The example below demonstrates the problem:

```
1  int main(){
2      fiber f1,f2,f3;
3      f3=fiber{[&](fiber&& f)->fiber{
4          f2=std::move(f);
5          for(;;){
6              std::cout << "f3 ";
7              std::move(f1).resume();
8          }
9          return {};
10     }};
11     f2=fiber{[&](fiber&& f)->fiber{
12         f1=std::move(f);
13         for(;;){
14             std::cout << "f2 ";
15             std::move(f3).resume();
16         }
17         return {};
18     }};
```

```
19      f1=fiber{[&](fiber&& /*main*/)->fiber{
20          for(;;){
21              std::cout << "f1 ";
22              std::move(f2).resume();
23          }
24          return {};
25      }};
26      std::move(f1).resume();
27      return 0;
28  }
29
30  output:
31  1 2 3 1 3 1 3 1 3 ...
```

In this pseudo-code the `std::fiber` object is implicitly updated.
The example creates a circle of fibers: each fiber prints its name and resumes the next fiber (f1 -> f2 -> f3 -> f1 -> ...). Fiber `f1` is started at line 26. The synthesized fiber `main` passed to the resumed fiber is not used (control flow cycles through the three fibers).* The for-loop prints the name *f1* and resumes fiber `f2`. Inside `f2`'s for-loop the name is printed and `f3` is resumed. Fiber `f3` resumes fiber `f1` at line 7. Inside `f1` control returns from `f2.resume()`. `f1` loops, prints out the name and invokes `f2.resume()`. But this time fiber `f3` instead of `f2` is resumed. This is caused by the fact the instance `f2` gets the synthesized fiber of `f3` implicitly assigned. Remember that at line 7 fiber `f3` gets suspended while `f1` is resumed through `f1.resume()`.

This problem can be solved by returning the synthesized fiber from `resume()` or `resume_with()`.

```
int main(){
    fiber f1,f2,f3;
    f3=fiber{[&](fiber&& f)->fiber{
        f2=std::move(f);
        for(;;){
            std::cout << "f3 ";
            f2=std::move(f1).resume();
        }
        return {};
    }};
    f2=fiber{[&](fiber&& f)->fiber{
        f1=std::move(f);
        for(;;){
            std::cout << "f2 ";
            f1=std::move(f3).resume();
        }
        return {};
    }};
    f1=fiber{[&](fiber&& /*main*/)->fiber{
        for(;;){
            std::cout << "f1 ";
            f3=std::move(f2).resume();
        }
        return {};
    }};
    std::move(f1).resume();
    return 0;
}

output:
1 2 3 1 2 3 1 2 3 ...
```

In the example above the synthesized fiber returned by `resume()` is move-assigned to the invoking fiber instance (that has resumed the current fiber).

*The operating-system stack provided for `main()` or the current thread's *entry-function* is not destroyed when the corresponding `std::fiber` instance is destroyed.

The synthesized fiber must be returned from `resume()` and `resume_with()` in order to prevent changing the identity of the fiber.

If the overall control flow isn't known, member function `resume_with()` (see section inject function into suspended fiber) can be used to assign the synthesized `std::fiber` to the correct `std::fiber` instance (the caller).

```
class filament{
private:
    fiber       f_;

public:
    ...
    void resume_next( filament& fila){
        std::move(fila.f_).resume_with([this](fiber&& f)->fiber{
            f_=std::move(f);
            return {};
        }
    }
};
```

Member function `resume_next()` resumes the next `filament` passed as parameter. The active fiber invokes `resume_with()` on the fiber aggregated by `fila`. The lambda captures `this`, the current fiber is suspended, a new `std::fiber` is synthesized and passed as parameter `f` to the function (that's the lambda) injected to the resumed fiber of `fila`.
Inside the lambda `f` is moved into the instance `f_` aggregated by the `filament` that was suspended.*

It is not necessary to know the overall control flow. It is sufficient to pass a reference/pointer of the *caller* (fiber that gets suspended) to the resumed fiber that move-assigns the synthesized fiber to *caller* (updating the instance).

inject function into suspended fiber

Sometimes it is useful to inject a new function (for instance, to throw an exception or assign the synthesized fiber to the caller as described in returning synthesized fiber instance from `resume()`) into a suspended fiber. For this purpose `resume_with(Fn&& fn)` may be called, passing the function `fn()` to execute.
If the fiber represented by the `std::fiber` instance `f` has called a function `suspender()`, which has called `resume()` and is now suspended, the call `f.resume_with(fn)` injects function `fn()` into fiber `f` as if `suspender()`'s `resume()` call had directly called `fn()`.

Like an *entry-function* passed to `std::fiber`, `fn()` must accept `std::fiber&&` and return `std::fiber`. The `std::fiber` instance returned by `fn()` will, in turn, be returned to `suspender()` by `resume()`.

Suppose that code running on the program's main fiber calls `resume()` (line 12), thereby entering the first lambda shown below. This is the point at which `m` is synthesized and passed into the lambda at line 2.
Suppose further that after doing some work (line 4), the lambda calls `m.resume()`, thereby switching back to the main fiber. The lambda remains suspended in the call to `m.resume()` at line 5.
At line 18 the main fiber calls `f.resume_with()` where the passed lambda accepts `fiber &&`. That new lambda is called on the fiber of the suspended lambda. It is as if the `m.resume()` call at line 8 directly called the second lambda.

The function passed to `resume_with()` has almost the same range of possibilities as any function called on the fiber represented by `f`. Its special invocation matters when control leaves it in either of two ways:

1. If it throws an exception, that exception unwinds all previous stack entries in that fiber (such as the first lambda's) as well, back to a matching `catch` clause.†
2. If the function returns, the returned `std::fiber` instance is returned by the suspended `m.resume()` (or `resume_with()`) call.

```
1  int data = 0;
2  fiber f{[&data](fiber&& m){
```

* *Boost.Fiber*[13] uses this pattern for resuming user-land threads. The injected lambda assigns the synthesized fiber to the caller and unlocks a spinlock if provided or re-schedules the suspended user-land thread.
†As stated in exceptions, if there is no matching `catch` clause in that fiber, `std::terminate()` is called.

```
3       std::cout << "f1: entered first time: " << data  << std::endl;
4       data+=1;
5       m=std::move(m).resume();
6       std::cout << "f1: entered second time: " << data  << std::endl;
7       data+=1;
8       m=std::move(m).resume();
9       std::cout << "f1: entered third time: " << data << std::endl;
10      return std::move(m);
11  }};
12  f=std::move(f).resume();
13  std::cout << "f1: returned first time: " << data << std::endl;
14  data+=1;
15  f=std::move(f).resume();
16  std::cout << "f1: returned second time: " << data << std::endl;
17  data+=1;
18  f=std::move(f).resume_with([&data](fiber&& m){
19      std::cout << "f2: entered: " << data << std::endl;
20      data=-1;
21      return std::move(m);
22  });
23  std::cout << "f1: returned third time" << std::endl;
24
25  output:
26      f1: entered first time: 0
27      f1: returned first time: 1
28      f1: entered second time: 2
29      f1: returned second time: 3
30      f2: entered: 4
31      f1: entered third time: -1
32      f1: returned third time
```

The `f.resume_with(<lambda>)` call at line 18 passes control to the second lambda on the fiber of the first lambda.

As usual, `resume_with()` synthesizes a `std::fiber` instance representing the calling fiber, passed into the lambda as `m`. This particular lambda returns `m` unchanged at line 21; thus that `m` instance is returned by the `resume()` call at line 8.

Finally, the first lambda returns at line 10 the `m` variable updated at line 8, switching back to the main fiber.

Member function `resume_with()` allows you to inject a function into suspended fiber.

passing data between fibers

Data can be transferred between two fibers via global pointer, calling wrapper (like `std::bind`) or lambda capture.

```
1   int i=1;
2   std::fiber lambda{[&i](fiber&& caller){
3       std::cout << "inside lambda,i==" << i << std::endl;
4       i+=1;
5       caller=std::move(caller).resume();
6       return std::move(caller);
7   }};
8   lambda=std::move(lambda).resume();
9   std::cout << "i==" << i << std::endl;
10  lambda=std::move(lambda).resume();
11
12  output:
13      inside lambda,i==1
14      i==2
```

The `resume()` call at line 8 enters the lambda and passes 1 into the new fiber. The value is incremented by one, as shown at line 4. The expression `caller.resume()` at line 5 resumes the original context (represented within the lambda by `caller`).

The call to `lambda.resume()` at line 10 resumes the lambda, returning from the `caller.resume()` call at line 5. The `std::fiber` instance `caller` invalidated by the `resume()` call at line 5 is replaced with the new instance returned by that same `resume()` call.
Finally the lambda returns (the updated) `caller` at line 6, terminating its context.

Since the updated `caller` represents the fiber suspended by the call at line 10, control returns to `main()`.

However, since context `lambda` has now terminated, the updated `lambda` is invalid. Its `operator bool()` returns `false`; its `operator!()` returns `true`.

Using lambda capture is the preferred way to transfer data between two fibers; global pointer and calling wrapper (like `std::bind`) are alternatives.

termination

There are a few different ways to terminate a given fiber without terminating the whole process, or engaging undefined behavior.

Any of the following will gracefully terminate a fiber:

- Cause its *entry-function*to return a valid fiber.
- From within the fiber you wish to terminate, call `std::unwind_fiber()` with a valid fiber. `std::unwind_fiber()` will trigger a specific action that unwinds the stack (walking the stack and destroying automatic variables in reverse order of construction) *.
- Call `fiber::resume_with(unwind_fiber)`. This is what `~fiber()` does. Since `std::unwind_fiber()` accepts a `std::fiber`, and since `resume_with()` synthesizes a `std::fiber` representing its caller and passes it to the subject function, this terminates the fiber referenced by the original `std::fiber` instance and switches back to the caller.
- Engage `~fiber()`: switch to some other fiber, which will receive a `std::fiber` instance representing the current fiber. Make that other fiber destroy the received `std::fiber` instance.

(However, since the operating system allocates the stack for `main()` and for a thread's *entry-function*, of course there is no implicit top-level stack frame, no implicit `catch (std::unwind_exception)`. In a conforming implementation, returning from a thread's *entry-function* may terminate all fibers on that thread. Returning from `main()` may terminate the whole process.)
When an explicitly-launched fiber's *entry-function* returns a valid `std::fiber` instance, that fiber is terminated. Control switches to the fiber indicated by the returned `std::fiber` instance.

Returning an invalid `std::fiber` instance (`operator bool()` returns `false`) invokes undefined behavior.

If the *entry-function* returns the same `std::fiber` instance it was originally passed (or rather, the `std::fiber` instance most recently returned by `resume()` or `resume_with()`), control returns to the fiber that most recently resumed the running fiber. However, the *entry-function* may return (switch to) any reachable valid `std::fiber` instance.

Calling `resume()` means: "Please switch to the indicated fiber; I am suspending; please resume me later."

Returning a particular `std::fiber` means: "Please switch to the indicated fiber; and by the way, I am done."

exceptions

In general, if an uncaught exception escapes from the *entry-function*, `std::terminate` is called.
Of course, no `try/catch` block is needed if neither *entry-function* nor anything it calls throws exceptions.

stack destruction

On construction of a fiber a stack is allocated. If the *entry-function* returns, the stack will be destroyed. If the function has not yet returned and the (destructor) of the `std::fiber` instance representing that context is called, the stack will be unwound and destroyed.

*for instance the *__Unwind*-API* part of the *SYSV ABI* could be utilized

Consider a running fiber `f2` that destroys the `std::fiber` instance representing `f1`.

`f1`'s destructor, running on `f2`, implicitly calls member-function `resume_with()`, passing `std::unwind_fiber()` as argument. Fiber `f1` will be temporarily resumed and `std::unwind_fiber()` is invoked. Function `std::unwind_fiber()` binds an instance of `std::fiber` that represents `f2`, then triggers a specific action that unwinds `f1`'s stack (walking the stack and destroying automatic variables in reverse order of construction)*. The first frame on `f1`'s stack, the one created by `std::fiber`'s constructor, extracts the bound `std::fiber` representing `f2` and terminates `f1` by returning `f2`. Control is returned to `f2` and `f1`'s stack gets deallocated.

The StackAllocator's deallocate operation runs on the fiber that invoked `~fiber()`, in this case `f2`.

The stack on which `main()` is executed, as well as the stack implicitly created by `std::thread`'s constructor, is allocated by the operating system. Such stacks are recognized by `std::fiber`, and are not deallocated by its destructor.

stack allocators

Stack allocators are used to create stacks and might implement arbitrary stack strategies. For instance, a stack allocator might append a guard page at the end of the stack, or cache stacks for reuse, or create stacks that grow on demand.

Because stack allocators are provided by the implementation, and are only used as parameters of the constructor, the StackAllocator concept is an implementation detail, used only by the internal mechanisms of the implementation. Different implementations might use different StackAllocator concepts.

However, when an implementation provides a stack allocator matching one of the descriptions below, it should use the specified name.

Possible types of stack allocators:

- `protected_fixedsize`: The constructor accepts a `size_t` parameter. This stack allocator constructs a contiguous stack of specified size, appending a guard page at the end 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.
- `fixedsize`: The constructor accepts a `size_t` parameter. This stack allocator constructs a contiguous stack of specified size. In contrast to `protected_fixedsize`, it does not append a guard page. The memory is simply managed by `std::malloc()` and `std::free()`, avoiding kernel involvement.
- `segmented`: The constructor accepts a `size_t` parameter. This stack allocator creates a segmented stack[18] with the specified initial size, which **grows on demand**.†

It is expected that the StackAllocator's allocation operation will run in the context of the `std::fiber` constructor, and that the StackAllocator's deallocation operation will run on the fiber calling `~fiber()` (after control returns from the destroyed fiber). No special constraints need apply to either operation.

fiber as building block for higher-level frameworks

A low-level API enables a rich set of higher-level frameworks that provide specific syntaxes/semantics suitable for specific domains. As an example, the following four frameworks are based on the low-level fiber switching API of *Boost.Context*[11] (implements the API of this proposal).

Boost.Coroutine2[12] implements **asymmetric coroutines** `coroutine<>::push_type` and `coroutine<>::pull_type`, providing a unidirectional transfer of data. These stackful coroutines are only used in pairs. When `coroutine<>::push_type` is explicitly instantiated, `coroutine<>::pull_type` is synthesized and passed as parameter into the coroutine function. In the example below, `coroutine<>::push_type` (variable `writer`) provides the resume operation, while `coroutine<>::pull_type` (variable `in`) represents the suspend operation. Inside the lambda,`in.get()` pulls strings provided by `coroutine<>::push_type`'s output iterator support.

```
struct FinalEOL{ ~FinalEOL(){ std::cout << std::endl; } };
std::vector<std::string> words{
    "peas", "porridge", "hot", "peas",
    "porridge", "cold", "peas", "porridge",
```

*for instance the *__Unwind*-API* part of the *SYSV ABI* could be utilized

†An implementation of the `segmented` StackAllocator necessarily interacts with the C++ runtime. For instance, with gcc, the Boost.Context[11] library invokes the `__splitstack_makecontext()` and `__splitstack_releasecontext()` intrinsic functions.[19]

```
    "in", "the", "pot", "nine",
    "days", "old" };
int num=5,width=15;
boost::coroutines2::coroutine<std::string>::push_type writer{
    [&](boost::coroutines2::coroutine<std::string>::pull_type& in){
        FinalEOL eol;
        for (;;){
            for (int i=0; i<num; ++i){
                if (!in){
                    return;
                }
                std::cout << std::setw(width) << in.get();
                in();
            }
            std::cout << std::endl;
        }
    }};
std::copy(std::begin(words), std::end(words), std::begin(writer));
```

Synca[17] (by Grigory Demchenko) is a small, efficient library to perform asynchronous operations in synchronous manner. The main features are a **GO-like** syntax, support for transferring execution context explicitly between different thread pools or schedulers (portals/teleports) and asynchronous network support.

```
int fibo(int v){
    if (v<2) return v;
    int v1,v2;
    Waiter()
        .go([v,&v1]{ v1=fibo(v-1); })
        .go([v,&v2]{ v2=fibo(v-2); })
        .wait();
    return v1+v2;
}
```

The code itself looks like synchronous invocations while internally it uses asynchronous scheduling.

Boost.Fiber[13] implements **user-land threads** and combines fibers with schedulers (scheduler-algorithms are customization points). The API is modelled after the `std::thread`-API and contains objects like `future`, `mutex`, `condition_variable` ...

```
boost::fibers::unbuffered_channel<unsigned int> chan;
boost::fibers::fiber f1{[&chan]{
    chan.push(1);
    chan.push(1);
    chan.push(2);
    chan.push(3);
    chan.push(5);
    chan.push(8);
    chan.push(12);
    chan.close();
}};
boost::fibers::fiber f2{[&chan]{
    for (unsigned int value: chan) {
        std::cout << value << " ";
    }
    std::cout << std::endl;
}};
f1.join();
f2.join();
```

Facebook's ***folly::fibers***[16] is an asynchronous C++ framework using **user-land threads** for parallelism. In contrast to *Boost.Fiber*, *folly::fibers* exposes the scheduler and permits integration with various event dispatching libraries.

```
folly::EventBase ev_base;
auto& fiber_manager=folly::fibers::getFiberManager(ev_base);
folly::fibers::Baton baton;
fiber_manager.addTask([&]{
        std::cout << "task 1: start" << std::endl;
        baton.wait();
        std::cout << "task 1: after baton.wait()" << std::endl;
});
fiber_manager.addTask([&]{
        std::cout << "task 2: start" << std::endl;
        baton.post();
        std::cout << "task 2: after baton.post()" << std::endl;
});
ev_base.loop();
```

folly::fibers is used in many critical applications at Facebook for instance in *mcrouter*[14] and some other Facebook services/libraries like ServiceRouter (routing framework for *Thrift*[15]), Node API (graph ORM API for graph databases) ...

As shown in this section a low-level API can act as building block for a rich set of high-level frameworks designed for specific application domains that require different aspects of design, semantics and syntax.

interaction with STL algorithms

In the following example STL algorithm `std::generator` and fiber `g` generate a sequence of Fibonacci numbers and store them into `std::vector v`.

```
int a;
std::fiber g{[&a](std::fiber&& m){
    a=0;
    int b=1;
    for(;;){
        m=std::move(m).resume();
        int next=a+b;
        a=b;
        b=next;
    }
    return std::move(m);
}};
std::vector<int> v(10);
std::generate(v.begin(), v.end(), [&a,&g]() mutable {
    g=std::move(g).resume();
    return a;
});
std::cout << "v: ";
for (auto i: v) {
    std::cout << i << " ";
}
std::cout << "\n";

output: v: 0 1 1 2 3 5 8 13 21 34
```

The proposed fiber API does not require modifications of the STL and can be used together with existing STL algorithms.

possible implementation strategies

This proposal does N O T seek to standardize any particular implementation or impose any specific calling convention!

Modern **micro-processors** are **register machines**; the content of processor registers represent the execution context of the program at a given point in time.
Operating systems maintain for each process all relevant data (execution context, other hardware registers etc.) in the process table. The operating system's **CPU scheduler** periodically suspends and resumes processes in order to share CPU time between multiple processes. When a process is suspended, its execution context (processor registers, instruction pointer, stack pointer, ...) is stored in the associated process table entry. On resumption, the CPU scheduler loads the execution context into the CPU and the process continues execution.
The CPU scheduler does a **full context switch**. Besides preserving the execution context (complete CPU state), the cache must be invalidated and the memory map modified.
A kernel-level context switch is several orders of magnitude slower than a context switch at user-level.[3]

hypothetical fiber preserving complete CPU state This strategy tries to preserve the complete CPU state, e.g. all CPU registers. This requires that the implementation identifies the concrete micro-processor type and supported processor features. For instance the x86-architecture has several flavours of extensions such as MMX, SSE1-4, AVX1-2, AVX-512.
Depending on the detected processor features, implementations of certain functionality must be switched on or off. The CPU scheduler in the operating system uses such information for context switching between processes.
A fiber implementation using this strategy requires such a detection mechanism too (equivalent to swapper/`system_32()` in the Linux kernel).
Aside from the complexity of such detection mechanisms, preserving the complete CPU state for each fiber switch is expensive.

A context switch facility that preserves the complete CPU state like an operating system is possible but impractical for user-land.

fiber switch using the calling convention For `std::fiber`, not all registers need be preserved because the context switch is effected by a visible function call. It need not be completely transparent like an operating-system context switch; it only needs to be as transparent as a call to any other function. The calling convention – the part of the ABI that specifies how a function's arguments and return values are passed – determines which subset of micro-processor registers must be preserved by the called subroutine.

The **calling convention**[10] of **SYSV ABI** for **x86_64** architecture determines that general purpose registers R12, R13, R14, R15, RBX and RBP must be preserved by the sub-routine - the first arguments are passed to functions via RDI, RSI, RDX, RCX, R8 and R9 and return values are stored in RAX, RDX.
So on that platform, the `resume()` implementation preserves the **general purpose registers** (R12-R15, RBX and RBP) specified by the calling convention. In addition, the **stack pointer** and **instruction pointer** are preserved and exchanged too – thus, from the point of view of calling code, `resume()` behaves like an ordinary function call.
In other words, `resume()` acts on the level of a simple function invocation – with the same performance characteristics (in terms of CPU cycles).

This technique is used in *Boost.Context*[11] which acts as building block for *folly::fibers*. The *folly::fibers* framework itself is the basis of many critical applications at Facebook, such as *mcrouter*[14] and some other Facebook services/libraries like ServiceRouter (routing framework for *Thrift*[15]), Node API (graph ORM API for graph databases) ...

in-place substitution at compile time During code generation, a compiler-based implementation could inject the assembler code responsible for the fiber switch directly into each function that calls `resume()`. That would save an extra indirection (JMP + PUSH/MOV of certain registers used to invoke `resume()`).

CPU state at the stack Because each fiber must preserve CPU registers at suspension and load those registers at resumption, some storage is required.
Instead of allocating extra memory for each fiber, an implementation can use the stack by simply advancing the stack pointer at suspension and pushing the CPU registers (CPU state) onto the stack owned by the suspending fiber. When the fiber is resumed, the values are popped from the stack and loaded into the appropriate registers.

This strategy works because only a running fiber creates new stack frames (moving the stack pointer). While a fiber is suspended, it is safe to keep the CPU state on its stack.

Using the stack as storage for the CPU state has the additional advantage that `std::fiber` must only contain a pointer to the stack location: its memory footprint can be that of a pointer.
Section synthesizing the suspended fiber describes how global variables are avoided by synthesizing a `std::fiber` from the active fiber (execution context) and passing this synthesized fiber (representing the now-suspended fiber) into the

resumed fiber. Using the stack as storage makes this mechanism very easy to implement.* Inside `resume()` the code pushes the relevant CPU registers onto the stack, and from the resulting stack address creates a new `std::fiber`. This instance is then passed (or returned) into the resumed fiber (see synthesizing the suspended fiber).

Using the active fiber's stack as storage for the CPU state is efficient because no additional allocations or deallocations are required.

fiber switch on architectures with register window

The implementation of fiber switch is possible – many libc implementations still provide the ucontext-API (`swapcontext()` and related functions)† for architectures using a register window (such as SPARC). The implementation of `swapcontext()` could be used as blueprint for a fiber implementation.

how fast is a fiber switch

A fiber switch takes 11 CPU cycles on a *x86_64-Linux* system‡ using an implementation based on the strategy described in fiber switch using the calling convention (implemented in *Boost.Context*,[11] branch *fiber*).

interaction with accelerators

For many core devices several programming models, such as OpenACC, CUDA, OpenCL etc., have been developed targeting **host-directed** execution using an attached or integrated accelerator. The CPU executes the main program while controlling the activity of the accelerator. Accelerator devices typically provide capabilities for efficient vector processing§. Usually the host-directed execution uses **computation offloading** that permits executing computationally intensive work on a separate device (accelerator).[1]
For instance CUDA devices use a **command buffer** to establish communication between host and device. The host puts commands (op-codes) into the command buffer and the device process them **asynchronously**.[2]
It is obvious that a fiber switch does **not** interact with **host-directed device-offloading**. A fiber switch works like a function call (see fiber switch using the calling convention).

multi-threading environment

`std::fiber` is TLS-agnostic - best practice related to TLS applies to fibers too (see P0772R0.)
Migration between threads is forbidden. A `std::fiber` may only be resumed on the `std::thread` on which it was launched.

acknowledgment

I'd like to thank Andrii Grynenko, Detlef Vollmann, Geoffrey Romer, Grigory Demchenko, Lee Howes, Nat Goodspeed and Yedidya Feldblum.

*The implementation of *Boost.Context*[11] utilizes this technique.
†ucontext was removed from POSIX standard by POSIX.1-2008
‡Intel XEON E5 2620v4 2.2GHz
§warp on CUDA devices, wavefront on AMD GPUs, 512-bit SIMD on Intel Xeon Phi

appendix: API

```
class fiber {
public:
    fiber() noexcept;

    template<typename Fn>
    fiber(Fn&& fn);

    template<typename StackAlloc, typename Fn>
    fiber(std::allocator_arg_t, StackAlloc&& salloc, Fn&& fn);

    ~fiber();

    fiber(fiber&& other) noexcept;
    fiber& operator=(fiber&& other) noexcept;
    fiber(const fiber& other) noexcept = delete;
    fiber& operator=(const fiber& other) noexcept = delete;

    fiber resume() &&;
    template<typename Fn>
    fiber resume_with(Fn&& fn) &&;

    explicit operator bool() const noexcept;
    bool operator!() const noexcept;
    bool operator==(const fiber& other) const noexcept;
    bool operator!=(const fibert& other) const noexcept;
    bool operator<(const fiber& other) const noexcept;
    bool operator>(const fiber& other) const noexcept;
    bool operator<=(const fiber& other) const noexcept;
    bool operator>=(const fiber& other) const noexcept;
    void swap(fiber& other) noexcept;
};
```

member functions

(constructor) constructs new fiber

`fiber()noexcept`	(1)
`template<typename Fn>` `fiber(Fn&& fn)`	(2)
`template<typename StackAlloc, typename Fn>` `fiber(std::allocator_arg_t, StackAlloc&& salloc, Fn&& fn)`	(3)
`fiber(fiber&& other)noexcept`	(4)
`fiber(const fiber& other)=delete`	(5)

1) this constructor instantiates an invalid `std::fiber`. Its `operator bool()` returns `false`; its `operator!()` returns `true`.

2) takes a callable (function, lambda, object with `operator()()`) as argument. The callable must have signature as described in solution: avoiding non-const global variables and undefined behaviour. The stack is constructed using either `fixedsize` or `segmented` (see Stack allocators). An implementation may infer which of these best suits the code in `fn`. If it cannot infer, `fixedsize` will be used.

3) takes a callable as argument, requirements as for (2). The stack is constructed using *salloc* (see Stack allocators).*

4) moves underlying state to new `std::fiber`

*This constructor, along with the Stack allocators section, is an optional part of the proposal. It might be that implementations can reliably infer the optimal stack representation.

5) copy constructor deleted

Notes

The entry-function `fn` is *not* immediately entered. The stack and any other necessary resources are created on construction, but `fn` is not entered until `resume()` or `resume_with()` is called.

The entry-function `fn` passed to `std::fiber` will be passed a synthesized `std::fiber` instance representing the suspended caller of `resume()`.

The function `fn` passed to `resume_with()` will be passed a synthesized `std::fiber` instance representing the suspended caller of `resume_with()`.

(destructor) destroys a fiber

```
~fiber()    (1)
```

1) destroys a `std::fiber` instance. If this instance represents a fiber of execution (`operator bool()` returns `true`), then the fiber of execution is destroyed too.

operator= moves the fiber object

```
fiber& operator=(fiber&& other)noexcept        (1)
fiber& operator=(const fiber& other)=delete    (2)
```

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

2) copy assignment operator deleted

Parameters

other another fiber to assign to this object

Return value

***this**

resume() resumes a fiber

```
fiber resume()&&                    (1)
template<typename Fn>
fiber resume_with(Fn&& fn)&&        (2)
```

1) suspends the active fiber, resumes fiber `*this`

2) suspends the active fiber, resumes fiber `*this` but calls `fn()` in the resumed fiber (as if called by the suspended function)

Parameters

fn function injected into resumed fiber

Return value

fiber the returned instance represents the fiber that has been suspended in order to resume the current fiber

Exceptions

1) `resume()` or `resume_with()` might throw *any* exception if, while suspended:

- some other fiber calls `resume_with()` to resume this suspended fiber
- the function `fn` passed to `resume_with()` – or some function called by `fn` – throws an exception

2) Any exception thrown by the function `fn` passed to `resume_with()`, or any function called by `fn`, is thrown in the fiber referenced by `*this` rather than in the fiber of the caller of `resume_with()`.

Preconditions

1) `*this` represents a valid fiber (`operator bool()` returns `true`)

2) the current `std::thread` is the same as the thread on which `*this` was originally launched

Postcondition

1) `*this` is invalidated (`operator bool()` returns `false`)

Notes

`resume()` preserves the execution context of the calling fiber. Those data are restored if the calling fiber is resumed.
A suspended `fiber` can be destroyed. Its resources will be cleaned up at that time.
The returned `fiber` indicates via `operator bool()` whether the previous active fiber has terminated (returned from *entry-function*).
Because `resume()` invalidates the instance on which it is called, *no valid `std::fiber` instance ever represents the currently-running fiber.* In order to express the invalidation explicitly, `resume()` and `resume_with()` are rvalue-reference qualified. This means that both functions can only be invoked on rvalues.
When calling `resume()`, it is conventional to replace the newly-invalidated instance – the instance on which `resume()` was called – with the new instance returned by that `resume()` call. This helps to avoid inadvertent calls to `resume()` on the old, invalidated instance.
An injected function `fn()` must accept `std::fiber&&` and return `std::fiber`. The fiber instance returned by `fn()` is, in turn, used as the return value for the suspended function: `resume()` or `resume_with()`.

operator bool test whether fiber is valid

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

1) returns `true` if `*this` represents a fiber of execution, `false` otherwise.

Notes

A `std::fiber` instance might not represent a valid fiber for any of a number of reasons.

- It might have been default-constructed.
- It might have been assigned to another instance, or passed into a function. `std::fiber` instances are move-only.
- It might already have been resumed – calling `resume()` invalidates the instance.
- The *entry-function* might have voluntarily terminated the fiber by returning.

The essential points:

- Regardless of the number of `std::fiber` declarations, exactly one `std::fiber` instance represents each suspended fiber.
- No `std::fiber` instance represents the currently-running fiber.

operator! test whether fiber is invalid

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

1) returns `false` if `*this` represents a valid fiber, `true` otherwise.

Notes

See **Notes** for `operator bool()`.

(comparisons) establish an arbitrary total ordering for `std::fiber` instances

```
bool operator==(const fiber& other)const noexcept   (1)
bool operator!=(const fiber& other)const noexcept   (1)
bool operator<(const fiber& other)const noexcept    (2)
bool operator>(const fiber& other)const noexcept    (2)
bool operator<=(const fiber& other)const noexcept   (2)
bool operator>=(const fiber& other)const noexcept   (2)
```

1) Every invalid `std::fiber` instance compares equal to every other invalid instance. But because the running fiber is never represented by a valid `std::fiber` instance, and because every suspended fiber is represented by exactly one valid instance, *no valid instance can ever compare equal to any other valid instance.*

2) These comparisons establish an arbitrary total ordering of `std::fiber` instances, for example to store in ordered containers. (However, key lookup is meaningless, since you cannot construct a search key that would compare equal to any entry.) There is no significance to the relative order of two instances.

swap swaps two `std::fiber` instances

```
void swap(fiber& other)noexcept   (1)
```

1) Exchanges the state of `*this` with `other`.

std::unwind_fiber() terminate the current running fiber, switching to the fiber represented by the passed `std::fiber`. This is like returning that `std::fiber` from the *entry-function*, but may be called from any function on that fiber.

```
void unwind_fiber(fiber&& other)   (1)
```

1) Binds the passed `std::fiber` and triggers a specific action that unwinds the stack (walking the stack and destroying automatic variables in reverse order of construction) *. The running fiber's first stack entry extracts the bound `std::fiber` and terminates the current fiber by returning that `std::fiber`.

Parameters

other the `std::fiber` to which to switch once the current fiber has terminated

Preconditions

1) `other` must be valid (`operator bool()` returns `true`)

Return value

1) None: `std::unwind_fiber()` does not return

Stack allocators are the means by which stacks with non-default properties may be requested by the caller of `std::fiber`'s constructor. The stack allocator concept is implementation-dependent; the means by which an implementation's stack allocators communicate with `std::fiber`'s constructor is unspecified.

An implementation may provide zero or more stack allocators. However, a stack allocator with semantics matching any of the following must use the corresponding name.

protected_fixedsize The constructor accepts a `size_t` parameter. This stack allocator constructs a contiguous stack of specified size, appending a guard page at the end 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.

fixedsize The constructor accepts a `size_t` parameter. This stack allocator constructs a contiguous stack of specified size. In contrast to `protected_fixedsize`, it does not append a guard page. The memory is simply managed by `std::malloc()` and `std::free()`, avoiding kernel involvement.

segmented The constructor accepts a `size_t` parameter. This stack allocator creates a segmented stack[18] with the specified initial size, which grows on demand.

References

[1] Chandrasekaran, Sunita and Juckeland, Guido (2018). "OpenACC for Programmers: Concepts and Strategies", (1st ed.). Pearson Education, Inc

[2] Wilt, Nicolas (2013). "The CUDA Handbook: A Comprehensive Guide to GPU Programming", (1st ed.). Addison Wesley

[3] Tannenbaum, Andrew S. (2009). "Operating Systems. Design and Implementation", (3rd ed.). Pearson Education, Inc

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

[5] N3985: A proposal to add coroutines to the C++ standard library

[6] P0099R0: A low-level API for stackful context switching

[7] P0099R1: A low-level API for stackful context switching

[8] P0534R3: call/cc (call-with-current-continuation): A low-level API for stackful context switching

[9] C++ Core Guidelines

[10] System V Application Binary Interface AMD64 Architecture Processor Supplement

*for instance the *__Unwind*-API* part of the *SYSV ABI* could be utilized

[11] Library *Boost.Context*
[12] Library *Boost.Coroutine2*
[13] Library *Boost.Fiber*
[14] Facebook's *mcrouter*
[15] Facebook's *Thrift*
[16] Facebook's *folly::fibers*
[17] Library *Synca*
[18] Split Stacks / GCC
[19] Re: using split stacks