fiber_context - fibers without scheduler

Revision History .......................................................... 1
abstract ................................................................. 3
control transfer mechanism ........................................... 3
std::fiber_context as a first-class object ......................... 4
encapsulating the stack ................................................ 5
invalidation at resumption ............................................. 5
problem: avoiding non-const global variables and undefined behaviour .................................................. 6
solution: avoiding non-const global variables and undefined behaviour .................................................... 7
inject function into suspended fiber ................................. 11
passing data between fibers .......................................... 13
termination ............................................................... 14
exceptions ............................................................... 15
std::fiber_context as building block for higher-level frameworks .................................................. 16
interaction with STL algorithms ..................................... 18
possible implementation strategies ................................ 18
fiber switch on architectures with register window ............. 20
how fast is a fiber switch ............................................. 20
interaction with accelerators ........................................ 20
multi-threading environment ........................................ 20
acknowledgments ....................................................... 21
API ........................................................................... 22
33.7 Cooperative User-Mode Threads ................................. 22
33.7.1 General .......................................................... 22
33.7.2 Empty vs. Non-Empty ......................................... 22
33.7.3 Explicit Fiber vs. Implicit Fiber ......................... 22
33.7.4 Implicit Top-Level Function ................................. 22
33.7.5 Header <experimental/fiber_context> synopsis ...... 23
33.7.6 Class fiber_context ........................................... 23
33.7.7 Function unwind_fiber() ..................................... 27
references .................................................................. 28

Revision History

This document supersedes D0876R7.

Changes since D0876R7

- Cancellation function removed from std::fiber_context constructor.
- std::unwind_fiber() re-added, with implementation-defined behavior.
- Added elaboration of filament example to bind cancellation function.

P0876R8 diverges from the recommendations of the second SG1 round in Cologne 2019. It does not introduce cancel() or cancel_from_any_thread() member functions. In fact it removes the cancellation-function constructor argument.

std::fiber_context is intended as the lowest-level stackful context-switching API. Binding a cancellation-function on the fiber stack is a flourish rather than a necessity. It adds overhead in both space (on the fiber stack) and time (to traverse the stack to retrieve the cancellation-function). For this API, it suffices to pass the desired cancellation-function to resume_with() or resume_from_any_thread_with(). If it is important to associate a cancellation-function with a particular fiber earlier in the lifespan of the fiber, a struct serves. We have added an example to the front matter.
A more compelling reason to avoid constructing an explicit fiber with a cancellation-function is that no implicit fiber has any such cancellation-function – and the consuming application cannot tell, a priori, whether a given `std::fiber_context` instance represents an explicit or an implicit fiber. If `*this` represents an implicit fiber, what should the proposed `cancel()` member function do?

Passing a specific cancellation-function to `resume_with()` or `resume_from_any_thread_with()` avoids that problem. P0876R8 follows SG1 recommendation in making it Undefined Behavior to destroy (or assign to) a non-empty `std::fiber_context` instance.

`std::unwind_fiber()` is reintroduced with implementation-defined behavior to allow fiber cleanup leveraging implementation internals. Its use is entirely optional (and auditable).

**Changes since P0876R6**

- Implicit stack unwinding (by non-C++ exception) removed.
- `std::unwind_fiber()` removed.
- Cancellation function added to `std::fiber_context` constructor.

In Cologne 2019, SG1 took the position that:

- The `fiber_context` facility is not the only C++ feature that requires “special” unwinding (special function exit path).
- Such functionality should be decoupled from `std::fiber_context`. It requires its own proposal that follows its own course through WG21 process.
- Depending on this (yet to be written) proposal would unduly delay the `fiber_context` facility.
- For now, the `fiber_context` facility should adopt a “less is more” approach, removing promises about implicit unwinding, placing the burden on the consumer of the facility instead.
- This leaves the way open for `fiber_context` to integrate with a new, improved unwind facility when such becomes available.

The idea of making `std::fiber_context`’s constructor accept a cancellation function was suggested to permit consumer opt-in to P0876R5 functionality where permissible, or convey to the fiber in question by any suitable means the need to clean up and terminate.

Requiring the cancellation function is partly because it remains unclear what the default should be. This could be one of the questions to be answered by a TS. Moreover, the absence of a default permits specifying later that the default engages the new, improved unwind facility.

**Changes since P0876R5**

- `std::unwind_exception` removed.
- `fiber_context::can_resume_from_any_thread()` renamed to `can_resume_from_this_thread()`.
- `fiber_context::valid()` renamed to `empty()` with inverted sense.
- Material has been added concerning the top-level wrapper logic governing each fiber.

`std::unwind_exception` was removed in response to deep discussions in Kona 2019 of the surprisingly numerous problems surfaced by using an ordinary C++ exception for that purpose.

Problems resolved by discarding `std::unwind_exception`:

- When unwinding a fiber stack, it is essential to know the subsequent fiber to resume. `std::unwind_exception` therefore bound a `std::fiber_context`. `std::fiber_context` is move-only. But C++ exceptions must be copyable.
- It was possible to catch and discard `std::unwind_exception`, with problematic consequences for its bound `std::fiber_context`.
- Similarly, it was possible to catch `std::unwind_exception` but not rethrow it.
If we attempted to address the problem above by introducing a `std::unwind_exception` operation to extract the bound `std::fiber_context`, it became possible to rethrow the exception with an empty (moved-from) `std::fiber_context` instance.

- Throwing a C++ exception during C++ exception unwinding terminates the program. It was possible for an exception implementation based on `thread_local` to become confused by exceptions on different fibers on the same thread.

- It was possible to capture `std::unwind_exception` with `std::exception_ptr` and migrate it to a different fiber – or a different thread.

**abstract**

This paper addresses concerns, questions and suggestions from the past meetings. The proposed API supersedes the former proposals N3985, P0099R1, P0534R3 and D0876R7.

Because of name clashes with `coroutine` from coroutine TS, `execution context` from executor proposals and `continuation` used in the context of `future::then()`, the committee has indicated that `fiber` is preferable. However, given the foundational, low-level nature of this proposal, we choose `fiber_context`, leaving the term `fiber` for a higher-level facility built on top of this one.

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).

Informally within this proposal, the term `fiber` is used to denote the lightweight thread of execution launched and represented by the first-class object `std::fiber_context`.

**control transfer mechanism**

According to the literature, 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.

```cpp
fiber_context* pf1;
fiber_context f4{[&pf1]{
    pf1->resume();
}};
fiber_context f3{[&f4]{
    f4.resume();
}};
fiber_context f2{[&f3]{
    f3.resume();
}};
fiber_context f1{[&f2]{
    f2.resume();
}};
pf1=&f1;
f1.resume();
```

In the pseudo-code example above, a chain of fibers is created.
Control is transferred to fiber \( f_1 \) at line 15 and the lambda passed to constructor of \( f_1 \) is entered. Control is transferred from fiber \( f_1 \) to \( f_2 \) at line 12 and from \( f_2 \) to \( f_3 \) (line 9) and so on. Fiber \( f_4 \) itself transfers control directly back to fiber \( f_1 \) 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 // hypothetical API
2 fiber_context f4([]{
3       self::suspend();
4   });
5 fiber_context f3([&f4]{
6       f4.resume();
7       self::suspend();
8   });
9 fiber_context f2([&f3]{
10       f3.resume();
11       self::suspend();
12   });
13 fiber_context f1([&f2]{
14       f2.resume();
15       self::suspend();
16   });
17 f1.resume();
```

In the pseudo code above execution control is transferred to fiber \( f_1 \) at line 16. Fiber \( f_1 \) resumes fiber \( f_2 \) at line 13 and so on. At line 2 fiber \( f_4 \) calls its suspend operation \( self::suspend() \). Fiber \( f_4 \) is suspended and \( f_3 \) resumed. Inside the lambda, \( f_3 \) returns from \( f_4.resume() \) and calls \( self::suspend() \) (line 6). Fiber \( f_3 \) gets suspended while \( f_2 \) 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 ...).**

**std::fiber_context 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 `std::fiber_context` must not be permitted!

If a `std::fiber_context` 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
- manage lifespan of an explicitly-allocated stack: the stack gets deallocated when `std::fiber_context` goes out of scope
- prevent accidentally copying the stack

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

### 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 empty the `std::fiber_context` instance, therefore no instance of `std::fiber_context` represents the currently-running fiber.

Whether or not a `std::fiber_context` is empty can be tested with member function `operator bool()`.

To make this more explicit, functions `resume()`, `resume_with()`, `resume_from_any_thread()` and `resume_from_any_thread_with()` are rvalue-reference qualified.

The essential points:

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

∗resource acquisition is initialisation
Section solution: avoiding non-const global variables and undefined behaviour describes how an instance of std::fiber_context 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(), resume_with(), resume_from_any_thread() and resume_from_any_thread_with() are rvalue-reference qualified to bind on rvalues only

problem: avoiding non-const global variables and undefined behaviour

According to C++ core guidelines, 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.

```cpp
int main() {
    fiber_context 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 std::fiber_context  P0099R0 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).

```cpp
int main(){
    int a;
    fiber_context m=fiber_context::current(); // get active fiber
    fiber_context 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_context::current() necessarily requires multiple instances of std::fiber_context for the active fiber. Only when the last reference goes out of scope can the fiber be destroyed and its stack deallocated.

fiber_context f1=fiber_context::current();
fiber_context f2=fiber_context::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_context m=fiber_context::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 std::fiber_context 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 first started, or returned from resume()).

void foo(){
    fiber_context f{[](fiber_context&& m){
        m=std::move(m).resume(); // switch to `foo`
        m=std::move(m).resume(); // switch to `foo`
        ...
    };
    f=std::move(f).resume(); // start `f`
    f=std::move(f).resume(); // resume `f`
}

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

Invoking m.resume() (line 3) suspends the lambda, empties m and synthesizes a std::fiber_context that is returned by f.resume() at line 7. The synthesized std::fiber_context 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 \texttt{f.resume()} at line 8 empties \texttt{f} and suspends \texttt{foo()} again. A \texttt{std::fiber_context} representing the suspended \texttt{foo()} is synthesized, returned from \texttt{m.resume()} and assigned to \texttt{m} at line 3. Control is transferred back to the lambda and instance \texttt{m} represents the suspended \texttt{foo()}.

Function \texttt{foo()} is resumed at line 4 by executing \texttt{m.resume()} so that control returns at line 8 and so on ...

Class \texttt{symmetricCoroutine<>::yield_type} from N3985 is \textbf{not} equivalent to the synthesized \texttt{std::fiber_context}. \texttt{symmetricCoroutine<>::yield_type} does not represent the suspended context, instead it is a special representation of the same coroutine. Thus \texttt{main()} or the current thread’s \texttt{entry-function} \textbf{can not} be represented by \texttt{yield_type} (see next section \texttt{representing main()} and thread’s \texttt{entry-function} as fiber).

Because \texttt{symmetricCoroutine<>::yield_type()} yields back to the starting point, i.e. invocation of \texttt{symmetricCoroutine<>::call_type::operator()}(), both instances (\texttt{call_type} as well as \texttt{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 \texttt{std::fiber_context}. A higher level layer can hide that by using private variables.

representing \texttt{main()} and thread’s \texttt{entry-function} as fiber \hspace{0.5cm} As shown in the previous section a synthesized fiber is created and passed into the resumed fiber as an instance of \texttt{std::fiber_context}.

```cpp
int main(){
    fiber_context f([]{fiber_context&& 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 \texttt{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 (\texttt{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) \texttt{main()} and each thread’s \texttt{entry-function} to be represented as fibers. A \texttt{std::fiber_context} representing \texttt{main()} or a thread’s \texttt{entry-function} can be handled like an explicitly created \texttt{std::fiber_context}: it can passed to and returned from functions or stored in a container. When called on such an instance, \texttt{can_resume_from_this_thread()} indicates whether it is valid to call \texttt{resume_from_any_thread()} or \texttt{resume_from_any_thread_with()} on that instance.

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

The proposed fiber API allows representing and handling \texttt{main()} and the current thread’s \texttt{entry-function} by an instance of \texttt{std::fiber_context} in the same way as explicitly created fibers.

fiber returns (terminates) \hspace{0.5cm} 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 \texttt{main()} is to explicitly return a non-empty \texttt{std::fiber_context} instance that will be resumed after the active fiber terminates.

```cpp
int main(){
    fiber_context f([]{fiber_context&& m){
        return std::move(m); // resume 'main' by returning 'm'
    }});
    std::move(f).resume(); // resume 'f'
    return 0;
}
```
In line 5 the fiber is started by invoking `resume()` on instance `f`. `main()` is suspended and an instance of type `fiber_context` 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 `std::fiber_context` is used as return value instead of the passed in synthesized fiber.

```cpp
int main(){
  fiber_context m;
  fiber_context f1{|&|fiber_context&& f|{|}
    std::cout << "f1: entered first time" << std::endl;
    assert(!f);
    return std::move(m); // resume (main-)fiber that has started 'f2'
  };
  fiber_context f2{|&|fiber_context&& f|{|}
    std::cout << "f2: entered first time" << std::endl;
    m=std::move(f); // preserve 'f' (== suspended main())
    return std::move(f1);
  };
  std::move(f2).resume();
  std::cout << "main: done" << std::endl;
  return 0;
}
```

output:
```
f2: entered first time
f1: entered first time
main: done
```

At line 13 fiber `f2` is resumed and the lambda is entered at line 8. The synthesized `std::fiber_context 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 `std::fiber_context f` passed into the lambda at line 3 represents the terminated fiber `f2` (e.g. the calling fiber). Thus instance `f` is empty as the assert statement verifies at line 5. Fiber `f1` uses the captured `std::fiber_context m` as return value (line 6). Control is returned to `main()`, returning from `f2.resume()` at line 13.

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

**returning synthesized std::fiber_context instance from resume()** An instance of `std::fiber_context` remains empty after return from `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()`—the synthesized fiber is returned, instead of implicitly updating the `std::fiber_context` instance on which `resume()` was called.

If the `std::fiber_context` object were implicitly updated, 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:

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

In this pseudo-code the std::fiber_context 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 std::fiber_context 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 std::fiber_context 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 std::fiber_context from resume(), resume_with(), resume_from_any_thread() or resume_from_any_thread_with().

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

∗The operating-system stack provided for main() or the current thread’s entry-function is not destroyed when the corresponding std::fiber_context instance is destroyed.
return 0;
}

output:
f1 f2 f3 f1 f2 f3 f1 f2 f3 ...

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

The synthesized std::fiber_context must be returned from resume(), resume_with(), resume_from_any_thread() and resume_from_any_thread_with() in order to prevent changing the identity of the fiber.

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

class filament{
private:
  fiber_context f_;}

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

Picture a higher-level framework in which every fiber can find its associated filament instance, as well as others. Every context switch must be mediated by passing the target filament instance to the running fiber’s resume_next().

Running fiber A has an associated filament instance filamentA, whose std::fiber_context f_ is empty – because fiber A is running.

Desiring to switch to suspended fiber B (with associated filament filamentB), running fiber A calls filamentA.resume_next(filamentB).

resume_next() calls filamentB.f_.resume_with(\lambda). This empties filamentB.f_ – because fiber B is now running.

The lambda binds &filamentA as this. Running on fiber B, it receives a std::fiber_context instance representing the newly-suspended fiber A as its parameter f. It moves that std::fiber_context instance to filamentA.f_.

The lambda then returns a default-constructed (therefore empty) std::fiber_context instance. That empty instance is returned by the previously-suspended resume_with() call in filamentB.resume_next() – which is fine because resume_next() drops it on the floor anyway.

Thus, the running fiber’s associated filament::f_ is always empty, whereas the filament associated with each suspended fiber is continually updated with the std::fiber_context instance representing that fiber.*

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 std::fiber_context to caller (updating the instance).

* Boost.Fiber uses this pattern for resuming user-land threads.
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 std::fiber_context instance from resume()) into a suspended fiber. For this purpose resume_with(Fn&& fn) (or resume_from_any_thread_with()) may be called, passing the function fn() to execute.

fiber_context f{[](fiber_context&& caller){
  // ...
  std::move(caller).resume();
  // ...
});
fiber_context fn(fiber_context&&);
f = std::move(f).resume();
// ...
std::move(f).resume_with(fn);

The resume_with() call at line 11 injects function fn() into fiber f as if the resume() call at line 3 had directly called fn().

Like an entry-function passed to std::fiber_context, fn() must accept std::fiber_context&& and return std::fiber_context. The std::fiber_context instance returned by fn() will, in turn, be returned to f’s lambda by the resume() at line 3.

Suppose that code running on the program’s main fiber calls resume() (line 12 below), 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_context&&. 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_context instance is returned by the suspended m.resume() (or resume_with(), or resume_from_any_thread(), or resume_from_any_thread_with()) call.

int data = 0;
fiber_context f{[](fiber_context&& m){
  std::cout << "f1: entered first time: " << data << std::endl;
  data+=1;
  m=std::move(m).resume();
  std::cout << "f1: entered second time: " << data << std::endl;
  data+=1;
  m=std::move(m).resume();
  std::cout << "f1: entered third time: " << data << std::endl;
  return std::move(m);
}};
f=std::move(f).resume();
std::cout << "f1: returned first time: " << data << std::endl;
data+=1;
f=std::move(f).resume();
std::cout << "f1: returned second time: " << data << std::endl;
data+=1;
f=std::move(f).resume_with{[](fiber_context&& m){
  std::cout << "f2: entered: " << data << std::endl;
}};

*As stated in exceptions, if there is no matching catch clause in that fiber, std::terminate() is called.
data=-1;
    return std::move(m);
});
std::cout << "f1: returned third time" << std::endl;
output:
f1: entered first time: 0
f1: returned first time: 1
f1: entered second time: 2
f1: returned second time: 3
f2: entered: 4
f1: entered third time: -1
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_context 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 21, switching back to the main fiber.

One case worth pointing out is when you call resume_with() (or resume_from_any_thread_with()) on a std::fiber_context that has not yet been resumed for the first time:

fiber_context topfunc(fiber_context&& prev);
fiber_context injected(fiber_context&& prev);
fiber_context f(topfunc);
// topfunc() has not yet been entered
std::move(f).resume_with(injected);

In this situation, injected() is called with a std::fiber_context instance representing the caller of resume_with(). When injected() eventually returns that (or some other) std::fiber_context instance, the returned std::fiber_context instance is passed into topfunc() as its prev parameter.

Member functions resume_with() and resume_from_any_thread_with() allow you to inject a function into a suspended fiber.

passing data between fibers

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

int i=1;
std::fiber_context lambda{[&i](fiber_context&& caller){
    std::cout << "inside lambda,i==" << i << std::endl;
    i+=1;
    caller=std::move(caller).resume();
    return std::move(caller);
}};
lambda=std::move(lambda).resume();
std::cout << "i==" << i << std::endl;
lambda=std::move(lambda).resume();

output:
inside lambda,i==1
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_context` instance `caller` emptied 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 empty. Its `operator bool()` returns `false`.

Using lambda capture is the preferred way to transfer data between two fibers; global pointers or a calling wrapper (such as `std::bind`) are alternatives.

**termination**

Every `std::fiber_context` you launch must terminate gracefully by returning from its top-level function.

Alternatively, a running fiber may call `std::unwind_fiber()`, passing a non-empty `std::fiber_context` instance. This is conceptually equivalent to returning that `std::fiber_context` instance from the fiber’s `entry-function`.

When an explicitly-launched fiber’s `entry-function` returns a non-empty `std::fiber_context` instance, the running fiber is terminated. Control switches to the fiber indicated by the returned `std::fiber_context` instance. The `entry-function` may return (switch to) any reachable non-empty `std::fiber_context` instance—it need not be the instance originally passed in, or an instance returned from any of the `resume()` family of methods.

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

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

The code below illustrates a possible way to aggregate an arbitrary `cancellation-function` with a fiber of interest, along with a `factory function` providing an example `cancellation-function`.

```cpp
// cancellable is a wrapper class that binds a cancellation-function along // with an associated fiber_context. It uses the tactic seen earlier in the // example ‘filament’ class to continually update the fiber_context // representing the fiber of interest.

class cancellable {
private:
  std::fiber_context f_;  
  std::function<std::fiber_context(std::fiber_context&&)> cancel_;
public:
  template <typename CANCEL>
  cancellable(std::fiber_context&& f, CANCEL&& cancel): f_(std::move(f)), cancel_(std::forward<CANCEL>(cancel)) { }

  ~cancellable() { 
    if (f_) { 
      std::move(f_).resume_with([this](std::fiber_context&& f) -> std::fiber_context{
        return cancel_(std::move(f));
      });
    }
  }

  cancellable(cancellable const&) = delete;
  cancellable & operator=(cancellable const&) = delete;

  cancellable(cancellable&&);  
  cancellable & operator=(cancellable&&);
};
```
```cpp
void resume_next(cancellable& cable) {
    std::move(cable.f_).resume_with([this] (std::fiber_context& f) {
        f_ = std::move(f);
        return {};
    });
}

// unwind_exception is an exception used internally by launch() to unwind the
// stack of a suspended fiber when the referencing fiber_context instance is
// destroyed.
class unwind_exception: public std::runtime_error {
public:
    unwind_exception(std::fiber_context&& previous):
        std::runtime_error("unwinding std::fiber_context"),
        previous_(std::make_shared<std::fiber_context>(std::move(previous)))
    {}
    std::shared_ptr<std::fiber_context> get_previous() const {
        return previous_;
    }
private:
    // Directly storing fiber_context would make unwind_exception move-only.
    // Instead, store a shared_ptr so unwind_exception remains copyable.
    std::shared_ptr<std::fiber_context> previous_;
};

// launch() is a factory function that returns a cancellable representing a
// new fiber that will run the passed entry_function. It implicitly provides a
// top-level wrapper and a cancellation-function. If the cancellable
// representing this new fiber is eventually destroyed, ~cancellable() will
// call this cancellation-function, which will throw unwind_exception. The
// top-level wrapper will catch it and return the bound fiber_context, thereby
// resuming the fiber that called ~cancellable().
template <typename Fn>
auto launch(Fn&& entry_function) {
    return cancellable{std::fiber_context{
        // entry-function passed to std::fiber_context constructor binds
        // entry_function, calls it within try/catch, catches
        // unwind_exception, extracts its shared_ptr<fiber_context>,
        // dereferences it and returns that fiber_context.
        {entry=std::move(entry_function)}
    }(
        // cancellation-function passed to cancellable constructor
        // throws unwind_exception, binding passed fiber_context instance.
        [] (std::fiber_context&& previous) -> std::fiber_context {
            throw unwind_exception(std::move(previous));
            return {};
        })
    };
}
exceptions

If an uncaught exception escapes from a fiber’s entry-function, std::terminate is called.

std::fiber_context 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 frameworks are based on the low-level fiber switching API of Boost.Context (which implements the API proposed here).

BoostCoroutine2 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<>::pull_type (variable writer) provides the resume operation, while coroutine<>::push_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", "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 (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.

BoostFiber implements user-land threads and combines fibers with schedulers (the scheduler algorithm is a customization point). The API is modelled after the std:thread API and contains objects such as 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\(^{25}\) 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\(^{23}\) and some other Facebook services/libraries like ServiceRouter (routing framework for Thrift\(^{24}\)), Node API (graph ORM API for graph databases) ...

Bloomberg's quantum\(^{26}\) is a full-featured and powerful C++ framework that allows users to dispatch units of work (a.k.a. tasks) as coroutines and execute them concurrently using the 'reactor' pattern. Its main features are support for streaming futures which allows faster processing of large data sets, task prioritization, fast pre-allocated memory pools and parallel forEach and mapReduce functions.

    // Define a coroutine
    int getDummyValue(Bloomberg::quantum::CoroContext<int>::Ptr ctx) {
        int value;
        ...          //do some work
        ctx->yield(); //be nice and let other coroutines run (optional cooperation)
        ...          //do more work and calculate 'value'
        return ctx->set(value);
    }
    // Create a dispatcher
    Bloomberg::quantum::Dispatcher dispatcher;
    // Dispatch a work item to do some work and return a value
    int result = dispatcher.post(getDummyValue)->get();

quantum is used in large projects at Bloomberg.
Habanero Extreme Scale Software Research Project\textsuperscript{27} provides a task-based parallel programming model via its HClib.\textsuperscript{28} The runtime provides work-stealing, async-finish,\textsuperscript{*} parallel-for and future-promise parallel programming patterns. The library is not an exascale programming system itself, but it manages intra-node resources and schedules components within an exascale programming system.

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 \texttt{std::generator} and fiber \texttt{g} generate a sequence of Fibonacci numbers and store them into \texttt{std::vector v}.

```cpp
int a;
std::fiber_context g{[&a](std::fiber_context&& 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](){
    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 NOT seek to standardize any particular implementation or impose any specific calling convention!

Modern micro-processors are register machines; the content of processor registers represents 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.

\textsuperscript{*}async-finish is a variant of the fork-join model. While a task might fork a group of child tasks, the child tasks might fork even more tasks. All tasks can potentially run in parallel with each other. The model allows a parent task to selectively join a subset of child tasks.
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.

---

**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_context`, 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\(^2\) 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*\(^20\) which acts as building block for (e.g.) `folly::fibers` and `quantum`; see section `std::fiber_context` as building block for higher-level frameworks.

---

**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_context` 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_context` from the active fiber (execution context) and passing this synthesized `std::fiber_context` (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

---

\(^*\)The implementation of *Boost.Context*\(^20\) utilizes this technique.
std::fiber_context. 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,‡ 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).†

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 processes them asynchronously.§

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**

Member function can_resume_from_this_thread() returns false if the stack represented by the std::fiber_context instance was created by the operating system (main application or thread stack), and the caller is running on a different thread. When the stack represented by the std::fiber_context instance was created by std::fiber_context’s constructor, can_resume_from_this_thread() returns true.§ You must not attempt to resume an instance representing a stack provided by the operating system on some other thread!

can_resume() can be called to determine whether a given std::fiber_context instance might safely be resumed on a particular thread by calling resume() or resume_with().

std::fiber_context is TLS-agnostic - best practices related to TLS apply to fibers too (see P0772R0.)

There could potentially be Undefined Behavior if:

- code running on a fiber references thread_local variables
- the compiler/runtime implementation caches a pointer to thread_local storage on the stack
- that fiber is suspended, and
- the suspended fiber is resumed on a different thread.

∗ 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
§ A possible implementation could mark the first stack frame by creating a special marker (for instance 0xBADCAFE etc.) at a specific offset in the first stack frame or use a special function name for the first function and walk the stack searching for these markers.
The cached TLS pointer is now pointing to storage belonging to some other thread. If the original thread terminates before the new thread, the cached TLS pointer is now dangling.

For a runtime that caches TLS pointers in such fashion, an implementation of `resume_from_any_thread()` or `resume_from_any_thread_with()` could conceivably walk the suspended stack, patching cached pointers.

**acknowledgments**

The authors would like to thank Andrii Grynenko, Detlef Vollmann, Geoffrey Romer, Grigory Demchenko, Lee Howes, David Hollman, Eric Fiselier and Yedidya Feldblum.
API

33.7 Cooperative User-Mode Threads

33.7.1 General

The extensions proposed here support creation and activation of cooperative user-mode threads, here called fibers.

The term “user-mode” means that a given fiber can be activated without entering the operating-system kernel.

The term “cooperative” means that typically multiple fibers share an underlying execution agent, for example a std::thread. On a given std::thread, only one fiber is running at any given time. Sharing that agent is explicit rather than pre-emptive. The running fiber suspends (or yields) to another fiber. This action launches a new fiber, or resumes a previously-suspended fiber.

Suspending the running fiber in order to resume (or launch) another is called context switching.

The term “thread” in “cooperative user-mode thread” means that even though a given fiber may suspend and later be resumed, it is logically a thread of execution as defined in [intro.multithread].

Context switching can be effected by designating some other fiber’s stack as current, in a manner appropriate to the existing implementation of function stacks.

33.7.2 Empty vs. Non-Empty

A std::fiber_context instance may be empty or non-empty. A default-constructed std::fiber_context is empty.

A moved-from std::fiber_context is empty. A std::fiber_context representing a suspended fiber is non-empty.

33.7.3 Explicit Fiber vs. Implicit Fiber

It is convenient to take the position that every kernel thread in a program, including the default thread on which the program runs main(), has an initial default fiber whose stack is allocated by the host operating system. [Note: Thus, when main() instantiates a new std::fiber_context, it becomes the second fiber on that thread. —end note]

Upon resumption of a fiber, a non-empty std::fiber_context instance is passed in representing the newly-suspended previous fiber. This std::fiber_context instance might represent the thread’s default fiber, or it might represent an explicitly-launched fiber: a fiber explicitly instantiated by invoking std::fiber_context’s constructor.

In certain situations it is important to distinguish these two cases. In particular, it is Undefined Behavior to attempt to resume a thread’s default fiber on some other thread. We use the phrase explicit fiber or explicitly-launched fiber to designate a fiber instantiated by user code; conversely, implicit fiber designates the default fiber on any thread. An implicit fiber’s owning thread is the thread of which that fiber is the default fiber. An explicit fiber has no owning thread. Instead, when necessary, we speak of the thread on which a fiber was last resumed.

33.7.4 Implicit Top-Level Function

On every explicit fiber, the behavior is equivalent to calling the entry-function passed to std::fiber_context’s constructor from an implicit top-level function.

The conceptual top-level function must catch any C++ exception escaping from the fiber’s entry-function and call std::terminate.

Returning a std::fiber_context instance from the explicit fiber’s entry-function is equivalent to returning control to the implicit top-level function. The conceptual implicit top-level function is responsible for deallocating the explicit fiber’s stack memory.

Similarly, on every implicit fiber, the behavior is equivalent to passing control through an implicit top-level function above main() and above the entry-function for each std::thread. The conceptual top-level function for an implicit fiber does not deallocate the fiber’s stack memory, since the OS will do that.
If a non-empty std::fiber_context instance is returned to the conceptual top-level function (for either an explicit or implicit fiber), the fiber represented by that std::fiber_context instance is resumed.

33.7.5 Header <experimental/fiber_context> synopsis

```cpp
#include <fiber_context>

#define __cpp_lib_experimental_fiber_context 201907

namespace std {
    namespace experimental {
        namespace concurrency_v2 {
            class fiber_context;
        } // namespace concurrency_v2
    } // namespace experimental
} // namespace std
```

33.7.6 Class fiber_context

```cpp
namespace std {
    namespace experimental {
        namespace concurrency_v2 {
            class fiber_context {
                public:
                    fiber_context() noexcept;
                    template<typename Fn>
                    explicit fiber_context(Fn&& fn);
                    ~fiber_context();
                    fiber_context(fiber_context&& other) noexcept;
                    fiber_context& operator=(fiber_context&& other) noexcept;
                    fiber_context& operator=(const fiber_context& other) noexcept = delete;
                    fiber_context& operator=(const fiber_context& other) noexcept = delete;
                    fiber_context resume() &&;
                    template<typename Fn>
                    fiber_context resume_with(Fn&& fn) &&;
                    fiber_context resume_from_any_thread() &&;
                    template<typename Fn>
                    fiber_context resume_from_any_thread_with(Fn&& fn) &&;
                    bool can_resume() noexcept;
                    bool can_resume_from_any_thread() noexcept;
                    explicit operator bool() const noexcept;
                    bool valid() const noexcept;
                    void swap(fiber_context& other) noexcept;
                };
            } // namespace concurrency_v2
        } // namespace experimental
    } // namespace std
```

(constructor) constructs new fiber_context
fiber_context() noexcept

```cpp
template<typename Fn>
explicit fiber_context(Fn&& fn)
```

```cpp
fiber_context(fiber_context&& other) noexcept
```

```cpp
fiber_context(const fiber_context& other)=delete
```

1) this constructor instantiates an empty `std::fiber_context`. Its `empty()` method returns `true`.

2) takes an invocable object [func.def] as argument. The invocable must have signature `fiber_context fn(fiber_context&&)`. This constructor template shall not participate in overload resolution unless `Fn` is `Lvalue-Invocable` [func.wrap.func] for the argument type `std::fiber_context&&` and the return type `std::fiber_context`. SG1: Needs update to Invocable concept.

3) moves underlying state to new `std::fiber_context`

4) copy constructor deleted

Remarks:

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

— A newly constructed but not yet entered fiber may be resumed by any thread. For any explicit fiber, the thread that resumes it is constrained only by the last thread that previously resumed it – and that constraint is imposed by operations performed by the fiber, rather than by the `std::fiber_context` facility itself.

— The entry-function `fn` passed to `std::fiber_context` will be passed a synthesized `std::fiber_context` instance representing the suspended caller of `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()`.

~fiber_context()

Effects:

— destroys a `std::fiber_context` instance.

Requires:

— `empty()` returns `true`.

[Note: If a `std::fiber_context` instance to be destroyed is not yet empty, an application may pass to `resume_with()` or `resume_from_any_thread_with()` a function intended to convey to the suspended fiber the need to terminate voluntarily. For example, such a function may throw an exception, or set some state visible to the functions on that fiber. — end note]

operator= moves the `std::fiber_context` object

```cpp
fiber_context& operator=(fiber_context&& other) noexcept
```

```cpp
fiber_context& operator=(const fiber_context& other)=delete
```

Requires:

— `empty()` returns `true`.

Effects:

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

2) copy assignment operator deleted

Parameters:

`other` another `std::fiber_context` to assign to this object

Returns:

`*this`
Postconditions:
— other is made empty (empty() returns true)

**resume()** resumes a fiber

```cpp
template<typename Fn>
fiber_context resume_from_any_thread_with(Fn&& fn) &&
```

**Requires:**
— *this represents a non-empty fiber (empty() returns false)
— can_resume_from_this_thread() would return true

**Effects:**
— This member function template shall not participate in overload resolution unless Fn is Lvalue-Invocable [func.wrap.func] for the argument type std::fiber_context&& and the return type std::fiber_context.
  SG1: Needs update to Invocable concept.
— The execution context of the calling fiber is saved.
— The calling fiber is suspended.
— A new std::fiber_context instance representing the suspended caller is synthesized. For purposes of exposition, call it caller.
— The fiber represented by *this is resumed.
— The execution context of the resumed fiber is restored.
— The invocable fn is called on the newly-resumed fiber.
— caller is passed to fn.
— If fn throws an exception, the exception propagates on the newly-resumed fiber.
— Otherwise fn eventually returns a std::fiber_context instance. For purposes of exposition, call it returned. [Note: returned may or may not be the same as caller. — end note] [Note: returned may be empty. — end note]
— If this is the first entry to the fiber represented by *this, returned is passed to its entry-function.
— Otherwise, the fiber represented by *this previously suspended itself by calling one of resume(), resume_with(), resume_from_any_thread() or resume_from_any_thread_with(). returned is returned from whichever of the resume functions was called.

**Parameters:**
fn invocable object injected into resumed fiber

**Returns:**
fiber_context on resumption, resume_from_any_thread_with() returns a std::fiber_context representing the immediately preceding fiber: the fiber that resumed this one, thereby suspending itself

**Exceptions:**
— resume_from_any_thread_with() can throw any exception if, while suspended:
  • some other fiber calls resume_with() or resume_from_any_thread_with() to resume this suspended fiber, and
  • the function fn passed to resume_with() or resume_from_any_thread_with()— or some function called by fn— throws an exception
— Any exception thrown by the function fn passed to resume_from_any_thread_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_from_any_thread_with().

**Postconditions:**
— *this is made empty (empty() returns true)
Note: The intent of the names `resume_from_any_thread()` and `resume_from_any_thread_with()` is to clarify the direction in which cross-thread resumption occurs. The calling thread always directly resumes a suspended fiber: control is passed into the suspended fiber, and the currently-running fiber suspends. These method names mean that the fiber represented by `*this` will be resumed whether or not it was last resumed on the calling thread. —end note

Note: The returned `fiber_context` indicates via `empty()` whether the previous active fiber has terminated (returned from `entry_function`). —end note

Note: Because `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()` empties the instance on which it is called, no non-empty `std::fiber_context` instance ever represents the currently-running fiber. In order to express the state change explicitly, these methods are rvalue-reference qualified. —end note

Note: When calling any of these methods, it is conventional to replace the newly-emptied instance - the instance on which the method was called – with the new instance returned by that call. This helps to avoid subsequent inadvertent attempts to resume the old, empty instance. —end note

```
template<typename Fn>
fiber_context resume_with(Fn&& fn) &&
```

Note: The intent of the distinction between `resume()` and `resume_from_any_thread()`, as between `resume_with()` and `resume_from_any_thread_with()`, is both for validation and for code auditing. If an application only ever calls `resume()` or `resume_with()`, no fiber in that application will ever be resumed on a thread other than the one on which it was previously resumed. —end note

Requires:
— `*this` represents a non-empty fiber (`empty()` returns `false`)
— the calling thread is the same as the thread on which the fiber represented by `*this` was most recently resumed

Effects:
— `resume_with()` has the same semantics as `resume_from_any_thread_with()`.
— `fiber_context resume_from_any_thread()` &&

Effects: Equivalent to:
`resume_from_any_thread_with([](fiber_context&& caller){ return std::move(caller); })`

```
fiber_context resume() &&
```

Effects: Equivalent to:
`resume_with([](fiber_context&& caller){ return std::move(caller); })`

```
bool can_resume_from_this_thread() noexcept
```

Effects: query whether the calling thread can resume the suspended `std::fiber_context` instance by calling `resume_from_any_thread()` or `resume_from_any_thread_with()`.

Returns:
— `fiber_context::can_resume_from_this_thread()` returns `false` if `*this` is empty, or if it represents an implicit fiber, and the calling thread is not that fiber’s owning thread; otherwise `true`.

Remarks:
— You may resume a suspended implicit fiber on the same thread using any of `resume()`, `resume_with()`, `resume_from_any_thread()` or `resume_from_any_thread_with()`. Attempting to resume an implicit fiber from any thread other than its owning thread is Undefined Behavior.
— `can_resume_from_this_thread()` returns `true` if the calling thread is the same as the thread on which the fiber represented by `*this` was most recently resumed, or if the `std::fiber_context` instance represents an explicit fiber.
— `can_resume_from_this_thread()` must not be called concurrently from multiple threads. [Note: `can_resume_from_this_thread()` is not marked `const` because in at least one implementation, it requires an internal context switch. However, the stack operations are effectively read-only. —end note]
bool can_resume() noexcept

Returns: false if \*this is empty, or if the calling thread is not the same as the thread on which the fiber represented by \*this was most recently resumed. [Note: When can_resume() returns true, the std::fiber_context instance may be resumed by resume(), resume_with(), resume_from_any_thread() or resume_from_any_thread_with(). — end note]

Remarks:
— can_resume_from_this_thread() must not be called concurrently from multiple threads. [Note: can_resume_from_this_thread() is not marked const because in at least one implementation, it requires an internal context switch. However, the stack operations are effectively read-only. — end note]

empty() test whether std::fiber_context is empty

bool empty() const noexcept (1)

explicit operator bool() const noexcept (2)

Returns:
1) returns false if \*this represents a fiber of execution, true otherwise.
2) equivalent to (! empty())

[Note: Regardless of the number of std::fiber_context declarations, exactly one std::fiber_context instance represents each suspended fiber. — end note]

void swap(fiber_context& other) noexcept

Effects:
— Exchanges the state of \*this with other.

33.7.7 Function unwind_fiber()

[[ noreturn ]] void unwind_fiber(fiber_context&& other)

Effects: terminate the current running fiber in an implementation-defined manner.

Remarks:
— Terminating the fiber causes its stack variables to be destroyed.
— Once the running fiber has been terminated, other is returned to the fiber’s conceptual top-level function as described in Implicit Top-Level Function.

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

Returns:
— None: std::unwind_fiber() does not return
References

[1] SYS V AMD64 unwindng
[2] x64 Windows unwindng
[3] ARM64 Windows unwindng
[8] N3985: A proposal to add coroutines to the C++ standard library
[9] P0099R0: A low-level API for stackful context switching
[10] P0099R1: A low-level API for stackful context switching
[12] P0876R0: fibers without scheduler
[13] P0876R2: fibers without scheduler
[14] P0876R3: fibers without scheduler
[15] P0876R5: fibers without scheduler
[16] P0876R6: fibers without scheduler
[17] D0876R7: fibers without scheduler
[18] C++ Core Guidelines
[19] System V Application Binary Interface AMD64 Architecture Processor Supplement
[21] Library BoostCoroutine2
[22] Library Boost.Fiber
[23] Facebook’s microrouter
[24] Facebook’s Thrift
[25] Facebook’s folly::fibers
[26] Bloomberg’s quantum
[27] Habanero Extreme Scale Software Research Project
[28] Habanero HClib
[29] Library Synca