Doc number: P0666R1
Revises: P0666R1, P0159R0, N4392
Date: 2018-02-11
Project: Programming Language C++, Concurrency Working Group
Reply-to: **Olivier Giroux** <ogiroux@nvidia.com>

Revised Latches and Barriers for C++20

Context

This paper introduces changes to barrier objects that were requested at the Toronto 2017 meeting.

These changes include:

1. Adding a split arrive/wait method interface, to enable overlapped computation. *Note: though this was requested, the form in this proposal is new, exciting (!) and probably controversial.*
2. A simplified concept of participating threads, which had feedback it was complicated & not useful.
3. Improved generic handling of barrier types, especially for barriers with completion phases.
4. Some latch class method names have changed, to clarify that their semantics have now become aligned with those of the barriers. *Note: This was not requested, but the rationale used in the past was that the method names should differ because the semantics differed. Since this is no longer the case, it is proposed that they should match again.*

The previous revision of this paper proposed the unchanged section numbered 33.7 from P0159R0 (the Concurrency TS v1) for inclusion in the C++ standard. That proposal was not moved in Toronto 2017.

C++ Proposed Wording

Apply the following edits to N4713, the working draft of the Standard. The feature test macros `__cpp_lib_latch` and `__cpp_lib_barrier` should be created.

Modify 33.1 General **[thread.general]**

Table 140 – Thread support library summary

	Subclause	Header(s)
33.2	Requirements	
33.3	Threads	`<thread>`
33.4	Mutual exclusion	`<mutex> <shared_mutex>`
33.5	Condition variables	`<condition_variable>`

33.6	Futures	`<future>`
33.7	Latches and barriers	`<latch>` `<barrier>`

33.7 Latches and barriers [thread.coordination]

1 This section describes various concepts related to thread coordination, and defines the *coordination type* `latch`, `basic_barrier` and `barrier`. These types facilitate concurrent computation performed by a number of threads, in one or more phases.

2 In this subclause, a *synchronization point* represents a condition that a thread may wait for, potentially blocking until it is satisfied. A thread *arrives at the synchronization point* when it has an effect on the state of the condition, even if it does not cause it to become satisfied. The *set of participating threads* is the set of all threads that either arrive or wait at the synchronization point.

3 Each coordination type has an implementation-defined member class `future_type` that satisfies the requirements of MoveConstructible, MoveAssignable and Destructible, and meets the other requirements in this subclause. An instance of a `future_type` is *associated* with up to one synchronization point's condition, according to the effects of the member functions of the enclosing class. In this subclause, `f` denotes an instance of a `future_type`.

4 The expression `f.wait()` shall be well-formed and have the following semantics:

```
void wait() const;
```

Effects: If `*this` is associated with a synchronization point, blocks at the synchronization point until the condition to be satisfied. Otherwise, does not block.

5 Concurrent invocations of the member functions of coordination types, other than their destructors, and the member functions of their `future_type` do not introduce data races.

33.7.1 Latches [coordination.latch]:

6 A latch is a thread coordination mechanism that allows any number of threads to block until an *expected* count is summed (exactly) by threads that arrived at the latch. The expected count is set when the latch is constructed. An individual latch is a single-use object; once the count has been reached, the latch cannot be reused.

33.7.2 Header `<latch>` synopsis [latch.synopsis]:

```
namespace std {
  class latch {
   public:
    explicit constexpr latch(ptrdiff_t expected);
    ~latch();

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

    using future_type = implementation-defined;  // see 33.7
    future_type make_ready_future() const noexcept;
    future_type arrive();

    bool try_wait() const noexcept;
    void wait() const;
    void arrive_and_wait(ptrdiff_t n = 1);
   private:
    ptrdiff_t counter; // exposition only
  };
} // namespace std
```

33.7.3 Class `latch` [latch.class]:

7 Class `latch` maintains an internal `counter` that is initialized when the latch is created. Threads may block at the latch's synchronization point, waiting for `counter` to be decremented to `0`.

```
explicit constexpr latch(ptrdiff_t expected);
```

8 *Requires:* `expected >= 0`.

9 *Effects:* `counter = expected`.

10 *Remarks:* Initializes the latch with the `expected` count.

```
~latch();
```

11 *Requires:* No threads are blocked at the synchronization point.

12 *Effects:* Destroys the latch.

13 *Remarks:* May be called even if some threads have not yet returned from functions that block at the synchronization point, provided that they are unblocked. [*Note:* The destructor may block until all threads have exited invocations of `wait()` on this object, or instances of `future_type` associated with this object. — *end note*]

```
future_type arrive();
```

14 *Requires:* `counter >= update` and `update >= 0`.

15 *Effects:* Constructs an object of type `future_type` that is associated with the latch's synchronization point and the condition `counter == 0`, then atomically decrements `counter` by `update`.

16 *Synchronization:* Synchronizes with the returns from all calls unblocked by the effects, and invocations of `try_wait()` on this latch that return `true`.

17 *Returns:* The constructed object.

18 *Remarks*: Arrives at the synchronization point with `update` count.

```
bool try_wait() const noexcept;
```

19 *Returns:* `counter == 0`.

```
void wait() const;
```

20 *Effects:* If `counter == 0`, returns immediately. Otherwise, blocks the calling thread at the synchronization point, until `counter == 0`.

```
void arrive_and_wait(ptrdiff_t update);
```

21 *Effects:* Equivalent to: `arrive(update); wait();`.

33.7.4 Barrier types [coordination.barrier]:

22 A barrier provides a thread-coordination mechanism that allows an *expected* number of threads to block until the same number of threads have arrived at the barrier. The expected number is set when the barrier is constructed, but may change according to some functions' effects. Unlike latches, barriers are reusable; once

threads are released from a barrier's synchronization point, they can reuse the same barrier immediately. [*Note:* It is thus useful for managing repeated tasks, or phases of a larger task, that are handled by multiple threads. — *end note.*]

23 A barrier has a *completion phase* that is a (possibly empty) set of effects. When the member functions defined in this subclause *arrive at the barrier*, they have the following effects:

1. When the expected number of threads has arrived at the barrier, one of the participating threads executes the barrier type's completion phase.
2. When the completion phase is completed, all threads blocked at the synchronization point are unblocked. The end of the completion phase synchronizes with the returns from all calls unblocked by its completion.

33.7.5 Header `<barrier>` synopsis [barrier.synopsis]:

```
namespace std {

  template<class Function>
    class basic_barrier;

  using barrier = basic_barrier<implementation-defined>;
}
```

33.7.6 Class `basic_barrier` [barrier.class]:

```
template<class Function>
class basic_barrier {
 public:
  explicit basic_barrier(ptrdiff_t expected, Function completion = Function());
  ~basic_barrier();

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

  using future_type = implementation-defined;  // see 33.7
  future_type make_ready_future() const noexcept;
  future_type arrive();

  void arrive_and_wait();
  void arrive_and_drop();

 private:
  Function completion_; // exposition only
};
```

1 Template class `basic_barrier` is a barrier type with a completion phase controlled by a function object. The completion phase calls `completion_()`.

2 The template type argument `Function` shall be CopyConstructible, and instances of this type shall be Callable (C++14 §[func.wrap.func]) with no arguments and return type `void`.

`explicit basic_barrier(ptrdiff_t expected, Function completion);`

24 *Requires:* `expected >= 0`, and invoking `completion` shall not exit via an exception.

25 *Effects:* Initializes the barrier for `expected` number of threads, and initializes `completion_` with `std::move(completion)`. [*Note:* If `expected` is `0` the set of participating threads will be empty, and consequently this object may only be destroyed. — *end note*]

`~basic_barrier();`

26 *Requires:* No threads are blocked at the synchronization point.

27 *Effects:* Destroys the barrier.

28 *Remarks:* May be called even if some threads have not yet returned from functions that block at the synchronization point, provided that they have unblocked. [*Note:* The destructor may block until all threads have exited invocations of `wait()` on instances of `future_type` associated with this object. — *end note*]

`future_type make_ready_future() const noexcept;`

29 *Effects:* Constructs an object of type `future_type` that is not associated with a synchronization point.

30 *Returns:* The constructed object.

`future_type arrive();`

31 *Requires:* The expected number of tasks is not `0`.

32 *Effects:* Constructs an object of type `future_type` that is associated with the barrier's synchronization point and the condition that the expected number of threads arrive, then arrives at the synchronization point.

33 *Synchronization:* The call to `arrive()` synchronizes with the start of the completion phase.

34 *Returns:* The constructed object.

35 *Remarks*: This may cause the completion phase to start.

`void arrive_and_wait();`

36 *Equivalent to:* `arrive().wait()`.

`void arrive_and_drop();`

37 *Requires:* The expected number of tasks is not `0`.

38 *Effects:* Decrements the expected number of threads by `1`.

39 *Synchronization:* The call to `arrive_and_drop()` synchronizes with the start of the completion phase.

40 *Remarks*: This may cause the completion phase to start.

33.7.7 Barrier with predefined parameter [barrier.predef]:

`using barrier = basic_barrier<`*implementation-defined*`>;`

3 The class `barrier` is a barrier type with an empty completion phase.