Doc number: P0666R2
Revises: P0666R1, P0159R0, N4392
Date: 2018-05-06
Project: Programming Language C++, **Concurrency Working Group**
Reply-to: **Olivier Giroux** <ogiroux@nvidia.com>

Revised Latches and Barriers for C++20

Changes in this revision

These changes include:

1. Fixes requested in JAX.
2. Simplified split arrive/wait method interface to reduce contention.
3. Introduced "phase" wording to help explain the lifetime of arrive's return value.

C++ Proposed Wording

Apply the following edits to N4741, 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 types* `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 contribute to or 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.

3 Concurrent invocations of the member functions of coordination types, other than their destructors, do not introduce data races.

33.7.1 Latches [coordination.latch]:

4 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;

    void arrive(ptrdiff_t n = 1);
    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]:

5 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);`

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

7 *Effects:* `counter = expected`.

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

`~latch();`

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

10 *Effects:* Destroys the latch.

11 *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. — *end note*]

`void arrive(ptrdiff_t update);`

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

13 *Effects:* atomically decrements `counter` by `update`.

14 *Synchronization:* Synchronizes with the returns from all calls unblocked by the effects.

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

`bool try_wait() const noexcept;`

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

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

17 *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);
```

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

33.7.4 Barrier types [coordination.barrier]:

19 A barrier is a thread coordination mechanism that allows at most an *expected* count of threads to block until that count is summed (exactly) by threads that arrived at the barrier in each of its successive *phases*. Once threads are released from blocking at the synchronization point for a phase, they can reuse the same barrier immediately in its next phase. [*Note:* It is thus useful for managing repeated tasks, or phases of a larger task, that are handled by multiple threads. — *end note.*]

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

1. When the expected number of threads for this phase have arrived at the barrier, one of those threads executes the barrier type's completion step.
2. When the completion step is completed, all threads blocked at the synchronization point for this phase are unblocked and the barrier enters its next phase. The end of the completion step strongly happens before 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 value_type = implementation-defined;
  value_type arrive(ptrdiff_t);
  bool try_wait(value_type) const;
  void wait(value_type) const;

  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 step controlled by a function object. The completion step calls `completion_()`. Threads may block at the barrier's synchronization point for a phase, waiting for the expected sum contributions by threads that arrive in that phase.

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

3 `basic_barrier::value_type` is an integral type.

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

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

22 *Effects:* Initializes the barrier for `expected` number of threads in the first phase, and initializes `completion_` with `std::move(completion)`. [*Note:* If `expected` is `0` this object may only be destroyed. — *end note*]

```
~basic_barrier();
```

23 *Requires:* No threads are blocked at a synchronization point for any phase.

24 *Effects:* Destroys the barrier.

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

```
value_type arrive(ptrdiff_t update);
```

26 *Requires:* The expected count is not less than `update`.

27 *Effects:* Constructs an object of type `value_type` that is associated with the barrier's synchronization point for the current phase, then arrives `update` times at the synchronization point for the current phase.

28 *Synchronization:* The call to `arrive()` strongly happens before the start of the completion step for the current phase.

29 *Returns:* The constructed object.

30 *Remarks*: This may cause the completion step to start.

```
bool try_wait(value_type arrival) const;
```

31 *Requires:* `arrival` is associated with a synchronization point for the current or the immediately preceding phases of the barrier.

32 *Returns:* `true` if the synchronization point condition associated with `arrival` is satisfied, otherwise `false`.

```
void wait(value_type arrival) const;
```

33 *Requires:* `arrival` is associated with a synchronization point for the current or the immediately preceding phases of the barrier.

34 *Effects:* blocks at the synchronization point associated with `arrival` until the condition is satisfied.

```
void arrive_and_wait();
```

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

```
void arrive_and_drop();
```

36 *Requires:* The expected number of threads for the current phase is not `0`.

37 *Effects:* Decrements the expected number of threads for subsequent phases by `1`, then arrives at the synchronization point for the current phase.

38 *Synchronization:* The call to `arrive_and_drop()` strongly happens before the start of the completion step for the current phase.

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

33.7.7 Barrier with predefined parameter [barrier.predef]:

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

4 The class `barrier` is a barrier type with an empty completion step.