C++ Latches and Barriers

ISO/IEC JTC1 SC22 WG21 N4392 - 2015-03-03

Alasdair Mackintosh, alasdair@google.com, alasdair.mackintosh@gmail.com

Olivier Giroux, OGiroux@nvidia.com, ogiroux@gmail.com

Revision History

N3666	2013-04-18	Initial Version
N3817	2013-10-11	Clarify destructor behaviour. Add comment on templatised completion functions.
N3885	2013-01-21	Add Alternative Solutions section. (Not formally published)
N3998	2014-05-21	Add Concepts, simplify latch and barrier, add notifiying_barrier
N4204	2014-08-06	Minor revisions after Rapperswil meeting
D4281	2014-11-06	Revisions after LEWG feedback
XXXX	2015-02-24	Reformat for LWG review. Add deleted move constructors. Remove 'throws' clause from count_down
XXXX	2015-02-25	Improved wording and formatting after various comments from LEWG and LWG members. Remove 'Concepts' section. Add exposition variables
N4392	2015-03-03	Final cleanup. Remove 'Notes' section.

Introduction

Certain idioms that are commonly used in concurrent programming are missing from the standard libraries. Although many of these these can be relatively straightforward to implement, we believe it is more efficient to have a standard version.

In addition, although some idioms can be provided using mutexes, higher performance can often be obtained with atomic operations and lock-free algorithms. However, these algorithms are more complex to write, and are prone to error.

Other standard concurrency idioms may have difficult corner cases, and can be hard to implement correctly. For these reasons, we believe that it is valuable to provide these in the standard library.

Note: This paper uses the term 'thread' throughout. Where relevant, it should be updated to refer to execution agents when these are adopted in the standard. See N4231 and N4156.

Solution

We propose a set of commonly-used concurrency classes, some of which may be implemented using efficient lock-free algorithms where appropriate. This paper describes various concepts related to thread co-ordination, and defines the *latch*, *barrier* and *flex_barrier* classes. The remainder of this paper contains the proposed wording. We use '**N**' as a placeholder for the main section number.

N. Coordination Mechanisms [thread.coordination]

N.1 Terminology [thread.coordination.terminology]

In this sub-clause, a *synchronization point* represents a point at which a thread may block until a given condition has been reached.

N.2 Latches [thread.coordination.latch]

Latches are a thread coordination mechanism that allow one or more threads to block until an operation is completed. An individual latch is a single-use object; once the operation has been completed, the latch cannot be reused.

Header `<experimental/latch>` Synopsis

```
namespace std {
namespace experimental {
inline namespace concurrency_v1 {
  class latch {
   public:
    explicit latch(ptrdiff_t count);
    latch(const latch&) = delete;
    latch(latch&&) = delete;

    ~latch();

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

    void count_down_and_wait();
    void count_down(ptrdiff_t n);

    bool is_ready() const noexcept;
    void wait() const;

   private:
    ptrdiff_t counter_; // exposition only
  };
} // namespace concurrency_v1
} // namespace experimental
} // namespace std
```

N.2.1 Class `latch` [thread.coordination.latch.class]

A latch maintains an internal `counter_` that is initialized when the latch is created. Threads may block at a synchronization point waiting for `counter_` to be decremented to `0`. When `counter_` reaches `0`, all such blocked threads are released.

Calls to `countdown_and_wait()`, `count_down()`, `wait()`, and `is_ready()` behave as atomic operations.

```
explicit latch(ptrdiff_t count);
```

Requires: `count >= 0`.

Synchronization: None

Postconditions: `counter_ == count`.

```
~latch();
```

Requires: No threads are blocked at the synchronization point.

Remarks: May be called even if some threads have not yet returned from `wait()` or `count_down_and_wait()` provided that `counter_` is `0`. [*Note:* The destructor might not return until all threads have exited `wait()` or `count_down_and_wait()`. — *end note*]

[*Note:* It is the caller's responsibility to ensure that no other thread enters `wait()` after one thread has called the destructor. This may require additional co-ordination. — *end note*]

```
void count_down_and_wait();
```

Requires: `counter_ > 0`.

Effects: Decrements `counter_` by `1`. Blocks at the synchronization point until `counter_` reaches `0`.

Synchronization: Synchronizes with all calls that block on this latch and with all `is_ready` calls on this latch that return `true`.

Throws: Nothing.

```
void count_down(ptrdiff_t n);
```

Requires: `counter_ >= n` and `n >= 0`.

Effects: Decrements `counter_` by `n`. Does not block.

Synchronization: Synchronizes with all calls that block on this latch and with all `is_ready` calls on this latch that return `true`.

Throws: Nothing.

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

[*Editor's note*: SG1 seems to have a convention that blocking functions are never marked `noexcept` (e.g. `future::wait`) even if they never throw. LWG requests that SG1 check whether this pattern is intended, and update the `noexcept` clauses here accordingly — *end editor's note*]

Effects: If `counter_` is `0`, returns immediately. Otherwise, blocks the calling thread at the synchronization point until `counter_` reaches `0`.

Throws: Nothing.

```
is_ready() const noexcept;
```

Returns: `counter_ == 0`. Does not block.

N.3 Barrier types [thread.coordination.barrier]

Barriers are a thread coordination mechanism that allow a *set of participating threads* to block until an operation is completed. Unlike a latch, a barrier is re-usable: once the participating threads are released from a barrier's synchronization point, they can re-use the same barrier. It is thus useful for managing repeated tasks, or phases of a larger task, that are handled by multiple threads.

The *barrier types* are the standard library types `barrier` and `flex_barrier`. They shall meet the requirements set out in this sub-clause. In this description, `b` denotes an object of a barrier type.

Each barrier type defines a *completion phase* as a (possibly empty) set of effects. When the member functions defined in this sub-clause *arrive at the barrier's synchronization point*, they have the following effects:

1. The function blocks.
2. When all threads in the barrier's set of participating threads are blocked at its synchronization point, one participating thread is unblocked and executes the barrier type's completion phase.

3. When the completion phase is completed, all other participating threads are unblocked. The end of the completion phase synchronizes with the returns from all calls unblocked by its completion.

The expression `b.arrive_and_wait()` shall be well-formed and have the following semantics:

Requires: The current thread is a member of the set of participating threads.

Effects: Arrives at the barrier's synchronization point.

[*Note:* It is safe for a thread to call `arrive_and_wait()` or `arrive_and_drop()` again immediately. It is not necessary to ensure that all blocked threads have exited `arrive_and_wait()` before one thread calls it again. — *end note*]

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

Throws: Nothing.

The expression `b.arrive_and_drop()` shall be well-formed and have the following semantics:

Requires: The current thread is a member of the set of participating threads.

Effects: Either arrives at the barrier's synchronization point and then removes the current thread from the set of participating threads, or just removes the current thread from the set of participating threads. [*Note*: Removing the current thread from the set of participating threads can cause the completion phase to start. — *end note*]

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

Throws: Nothing

Notes: If all participating threads call `arrive_and_drop()`, any further operations on the barrier are undefined, apart from calling the destructor. If a thread that has called `arrive_and_drop()` calls another method on the same barrier, other than the destructor, the results are undefined.

Calls to `arrive_and_wait()`and `arrive_and_drop()` never introduce data races with themselves or each other.

Header <experimental/barrier> synopsis

```
namespace std {
namespace experimental {
inline namespace concurrency_v1 {
  class barrier;
  class flex_barrier;
} // namespace concurrency_v1
} // namespace experimental
} // namespace std
```

N.3.1 Class `barrier` [thread.coordination.barrier.class]

`barrier` is a barrier type whose completion phase has no effects. Its constructor takes a parameter representing the initial size of its set of participating threads.

```
class barrier {
 public:
  explicit barrier(ptrdiff_t num_threads);
  barrier(const barrier&) = delete;
  barrier(barrier&&) = delete;

  ~barrier();

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

  void arrive_and_wait();
  void arrive_and_drop();
};
```

`explicit barrier(ptrdiff_t num_threads);`

Requires: `num_threads >= 0`. [*Note:* If `num_threads` is zero, the barrier may only be destroyed. — *end note*]

Effects: Initializes the barrier for `num_threads` participating threads. [*Note:* The set of participating threads is the first `num_threads` threads to arrive at the synchronization point. —*end note*]

`~barrier();`

Requires: No threads are blocked at the synchronization point.

Effects: Destroys the barrier

N.3.2 Class **`flex_barrier`** [thread.coordination.flexbarrier.class]

`flex_barrier` is a barrier type whose completion phase can be controlled by a constructor parameter.

```
class flex_barrier {
 public:
  template <class F>
    flex_barrier(ptrdiff_t num_threads, F completion);
  explicit flex_barrier(ptrdiff_t num_threads);
  flex_barrier(const flex_barrier&) = delete;
  flex_barrier(flex_barrier&&) = delete;

  ~flex_barrier();

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

  void arrive_and_wait();
  void arrive_and_drop();

 private:
  function<ptrdiff_t()> completion_;   // exposition only
};
```

The completion phase calls `completion_()`. If this returns `-1`, then the set of participating threads is unchanged. Otherwise, the set of participating threads becomes a new set with a size equal to the returned value. [*Note*: If `completion_()` returns `0` then the set of participating threads becomes empty, and this object may only be destroyed. —*end note*]

```
template <class F>
flex_barrier(ptrdiff_t num_threads, F completion);
```

Requires:

- `num_threads >= 0`.
- `F` shall meet the requirements of `CopyConstructible`.
- `completion` shall be Callable (C++14 §[func.wrap.func]) with no arguments and return type convertible to `ptrdiff_t`.
- Invoking `completion` shall return a value greater than or equal to `-1` and shall not exit via an exception.

Effects: Initializes the `flex_barrier` with the set of participating threads, of size `num_threads`, and initializes `completion_` with `std::move(completion)`. [*Note*: The set of participating threads consists of the first `num_threads` threads that will arrive at the synchronization point. — *end note*]

Notes: If `num_threads` is zero the set of participating threads is empty, and this object may only be destroyed.

```
explicit flex_barrier(ptrdiff_t num_threads);
```

Requires: `num_threads >= 0.`

Effects: Has the same effect as creating a `flex_barrier` with `num_threads` and with a callable object whose invocation returns `-1` and has no side effects.

```
~flex_barrier();
```

Requires: No threads are blocked at the synchronization point.

Effects: Destroys the barrier.