Doc number: **P0126R1**
Revises: **P0126R0, N4195**
Date: 2016-02-12
Project: Programming Language C++, Concurrency Working Group
Reply-to: Olivier Giroux <ogiroux@nvidia.com>

std::synchronic<T>

Atomic objects make it easy to implement inefficient synchronization in C++. The first problem that users typically have, is poor system performance under oversubscription and/or contention. The second is high energy consumption under contention, regardless of oversubscription.

At issue is a trade-off centered on resource arbitration for synchronization, placing in tension:

- The focus of modern platform architecture is on lowering total energy use.
- The focus of performance-critical software is on minimizing latency.

Implementations could do significantly better with more semantic information. There exists different native support for efficient polling on all major software and hardware platforms. We now propose "synchronic" objects, an atomic library abstraction for this diverse support.

For more background, see P0126R0 and Futexes are Tricky.

A simplifying abstraction

Synchronic objects make it easier to implement scalable and efficient synchronization using atomic objects. The easiest way to use a synchronic object is to declare an expected atomic value for synchronization, and notify when an atomic object should be compared against this value.

For example:

```
//similar to std::latch (n4538)
//using std::hardware_false_sharing_size (n4523)
class example {
  ...
  void sync_up_my_team() {
    if(count.fetch_add(-1)!=1)
      while(!released.load());
      sync.expect(released, true);
    else
      released.store(true);
      sync.notify(released, true);
  }
  ...
  alignas(hardware_false_sharing_size) atomic<int> count;
  alignas(hardware_false_sharing_size) atomic<bool> released;
  synchronic<bool> sync;
};
```

C++ Proposed Wording

The proposed edits are with respect to the current working draft of the Standard.

Feature test macros

The `__cpp_lib_synchronic` feature test macro should be added.

29.2 Header `<atomic>` synopsis:

```
    // 29.8, fences
    extern "C" void atomic_thread_fence(memory_order) noexcept;
    extern "C" void atomic_thread_fence(memory_order) noexcept;

    //  29.9, synchronic operations
    enum notify_hint { notify_all, notify_one };
    enum expect_hint { expect_urgent, expect_delay };

    template <class T> struct synchronic;
  }
```

29.9 Synchronic objects **[atomics.synchronic]**

1 Synchronic objects provide low-level blocking primitives used to implement synchronization with atomic objects. Class `synchronic<T>` encapsulates an efficient algorithm to wait until a condition is met, a predicate associated with a single object of the corresponding class `atomic<T>`. This facility neither requires, nor provides mutual-exclusion between threads.

2 Concurrent executions of the `notify` and `expect` member functions do not introduce data races. If they invoke a user-provided function, that function may still introduce data races.

3 Executions of the `expect` member functions return when the condition is satisfied, or else block. While blocked, subsequent evaluations of the condition may be deferred until another thread invokes a `notify` member function with the same atomic object. [*Note:* an evaluation that is not deferred indefinitely is only eventually performed. This makes synchronic objects susceptible to transient values, an issue known as the ABA problem, resulting in continued blocking after the condition is temporarily met. – *End Note.*]

4 The implementation shall behave as if the start of each evaluation of the condition by executions of the `expect` functions and invocations of `notify` functions are executed in a single unspecified total order consistent with the "happens before" order.

29.9.1 Class `synchronic` **[atomics.synchronic.class]**

```
  namespace std {
    template <class T>
    class synchronic {
    public:
      synchronic();
      ~synchronic();
      synchronic(const synchronic&) = delete;
      synchronic& operator=(const synchronic&) = delete;
      synchronic(synchronic&&) = delete;
      synchronic& operator=(synchronic&&) = delete;

      void notify(A& object, T value,
                  memory_order order = memory_order_seq_cst,
                  notify_hint hint = notify_all) noexcept;
      void expect(A const& object, T desired,
                  memory_order order = memory_order_seq_cst,
                  expect_hint hint = expect_urgent) const noexcept;

      void notify(A& object, F&& func, notify_hint hint = notify_all);
      void expect(A const& object, F&& func,
                  expect_hint hint = expect_urgent) const;

      void expect_update(A const& object, T current,
                         memory_order order = memory_order_seq_cst,
                         expect_hint hint = expect_urgent) const noexcept;
      void expect_update_for(A const & object, T current,
                         chrono::duration<Rep, Period> const& rel_time,
                         expect_hint hint = expect_urgent) const;
      void expect_update_until(A const& object, T current,
                         chrono::time_point<Clock,Duration> const& abs_time,
                         expect_hint hint = expect_urgent) const;

    }
  }
```

1 In the following operation definitions:

- an *A* refers to a corresponding atomic class type.
- an *F* refers to a callable type.

`synchronic();`

2 *Effects:* Constructs an object of type `synchronic<T>`.

3 *Throws:* `system_error` (19.5.6).

`~synchronic();`

4 *Requires:* There shall be no threads blocked on `*this`. [*Note*: a `synchronic` object can be destroyed if all threads blocked on `this` have been notified. – *end note*]

5 *Effects:*

- May block until all invocations of `notify` return.
- Destroys the object.

```
void notify(A& object, T value, memory_order order,
    notify_hint hint) noexcept;
void notify(A& object, F&& func, notify_hint hint);
```

6 *Requires:* `func` is callable with the signature `void()` and does not invoke a member function on `this`.

7 *Effects:*

- Invokes `func` or `object.store(value, order)`.
- If `hint` is `notify_one` and any execution of `expect` member functions invoked with the same atomic object `object` are blocked, unblocks one of those executions.
- If `hint` is `notify_all`, unblocks all executions of `expect` member functions invoked with the same atomic object `object` that are blocked.

8 *Throws:* `system_error` (19.5.6) or any exception thrown by `func`.

```
void expect(A const& object, T desired, memory_order order,
    expect_hint hint) const noexcept;
void expect(A const& object, F&& pred, expect_hint hint) const;
void expect_update(A const& object, T current, memory_order order,
    expect_hint hint) const noexcept;
void expect_update_for(A const & object, T current,
    chrono::duration<Rep, Period> const& rel_time,
    expect_hint hint) const;
void expect_update_until(A const& object, T current,
    chrono::time_point<Clock,Duration> const& abs_time,
    expect_hint hint) const;
```

9 *Requires:* `pred` is callable with the signature `bool()` and does not invoke a member function on `this`.

10 *Effects:*

- Invokes `pred`, or `object.load(order)`. If a timed function is used, then `order` is `memory_order_relaxed`.
- Blocks until `pred` returns true, or the value of `object` is either equal to `desired` or not equal to `current`, or the absolute time-out specified by `abs_time` expires, or the relative time-out specified by `rel_time` expires, or may unblock spuriously if a timed function is used.

11 *Throws:* `system_error` (19.5.6) or any exception thrown by `pred`.

12 *Remarks:* the value of `hint` has only a performance effect.