Doc number: P0514R4
Revises: P0514R2, P0126R2, N4195
Date: 2018-05-03
Project: Programming Language C++, **Library Evolution Working Group**
Reply-to: **Olivier Giroux** <ogiroux@nvidia.com>

Efficient concurrent waiting for C++20

Changes in this revision

1. Applied feedback from JAX.
2. Proactively updated "synchronizes with" to "strongly happens before".

Abstract

Context

This revision proposes that a subset of the facilities from prior revisions be adopted for C++20.

The facilities included in this proposal have not seen substantial changes in the last year. The one facility that was not included in this proposal (~~synchronic~~ ~~condition_variable_atomic~~ synchronic) remains in flux, but will now be decoupled from the others.

Abstract

The C++ atomic objects make it all too easy to implement inefficient blocking synchronization, due to lack of support for waiting in a more efficient way than polling. One problem that results, is poor system performance under conditions of oversubscription or contention. Another problem is high energy consumption under contention, regardless of oversubscription conditions.

The specialized `atomic_flag` object does nothing to help with this problem, despite a name that suggests it is suitable for this use. Its interface is tightly-fit to the demands of the simplest spinlocks without contention mitigation beyond what timed back-off can achieve. What is needed are specialized atomic and synchronization facilities, likely to replace `atomic_flag` in practice.

Semaphores and a simple abstraction for waiting

Semaphores are lightweight synchronization primitives that control concurrent access to a shared resource. A binary semaphore, then, is analogous to a mutex with no thread ownership semantics. This concept is behind our new **proposed** type:

```
struct semaphore_mutex {
    void lock() {
        s.acquire();
    }
    void unlock() {
        s.release();
    }
private:
```

```
        std::binary_semaphore s(1);
    };
```

A counting semaphore type is also proposed alongside: `std::counting_semaphore`, to regulate shared access to a resource that is not mutually-exclusive but bounded by a maximum degree of concurrency. These types follow Dijkstra's semaphore concept (wiki).

In addition to new semaphore types, we also propose simpler atomic free functions that enable incremental change to pre-existing algorithms expressed in terms of atomics, to benefit from the same efficient support behind semaphores:

```
    struct simple_lock {
        void lock() {
            bool old;
            while(!b.compare_exchange_weak(old = false, true))
                std::atomic_wait(&b, true);
        }
        void unlock() {
            b = false;
            std::atomic_notify_one(&b);
        }
    private:
        std::atomic<bool> b = ATOMIC_VAR_INIT(false);
    };
```

Note: in high-quality implementations this necessitates a semaphore table owned by the implementation, which causes some unavoidable interference due to the aliasing of unrelated atomic updates.

Reference implementation

It's here - https://github.com/ogiroux/semaphore.

C++ Proposed Wording

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

32.2 Header `<atomic>` synopsis **[atomics.syn]**

```
 // 32.3, type aliases
  using atomic_int_fast64_t    = atomic<int_fast64_t>;
  using atomic_uint_fast64_t   = atomic<uint_fast64_t>;
  using atomic_int_fast_wait_t  = atomic<implementation-defined>;
  using atomic_uint_fast_wait_t = atomic<implementation-defined>;

 // 32.9, fences
  extern "C" void atomic_thread_fence(memory_order) noexcept;
  extern "C" void atomic_signal_fence(memory_order) noexcept;

 // 32.10, waiting and notifying functions
  template <class T>
    void atomic_notify_one(const volatile atomic<T>*);
```

```
  template <class T>
    void atomic_notify_one(const atomic<T>*);
  template <class T>
    void atomic_notify_all(const volatile atomic<T>*);
  template <class T>
    void atomic_notify_all(const atomic<T>*);
  template <class T>
    void atomic_wait(const volatile atomic<T>*,
                     typename atomic<T>::value_type);
  template <class T>
    void atomic_wait(const atomic<T>*, typename atomic<T>::value_type);
  template <class T>
    void atomic_wait_explicit(const volatile atomic<T>*,
                              typename atomic<T>::value_type,
                              memory_order);
  template <class T>
    void atomic_wait_explicit(const atomic<T>*,
                              typename atomic<T>::value_type, memory_order);
}
```

32.10 Waiting and notifying functions [atomics.wait]

1 The functions in this subclause provide a mechanism to wait for the value of an atomic object to change, more efficiently than can be achieved with polling. Waiting functions in this facility may block until they are unblocked by notifying functions, according to each function's effects. [*Note:* Programs are not guaranteed to observe transient atomic values, an issue known as the A-B-A problem, resulting in continued blocking if a condition is only temporarily met. – *End Note.*]

2 The functions `atomic_wait` and `atomic_wait_explicit` are waiting functions. The functions `atomic_notify_one` and `atomic_notify_all` are notifying functions.

```
using atomic_int_fast_wait_t   = atomic<implementation-defined>;
using atomic_uint_fast_wait_t  = atomic<implementation-defined>;
```

3 The type aliases `atomic_int_fast_wait_t` and `atomic_uint_fast_wait_t` are integral atomic types. Implementations should ensure that invocations of waiting and notifying functions with these types have the lowest performance overhead among integer types.

```
template <class T>
  void atomic_notify_one(const volatile atomic<T>* object);
template <class T>
  void atomic_notify_one(const atomic<T>* object);
```

4 *Effects*: unblocks up to execution of a waiting function that blocked after observing the result of an atomic operation X, if there exists another atomic operation Y, such that X precedes Y in the modification order of `*object`, and Y happens-before this call.

```
template <class T>
  void atomic_notify_all(const volatile atomic<T>* object);
template <class T>
  void atomic_notify_all(const atomic<T>* object);
```

5 *Effects*: unblocks each execution of a waiting function that blocked after observing the result of an atomic operation X, if there exists another atomic operation Y, such that X precedes Y in the modification order of `*object`, and Y happens-before this call.

```
template <class T>
```

```
  void atomic_wait_explicit(const volatile atomic<T>* object,
                            typename atomic<T>::value_type old,
                            memory_order order);
template <class T>
  void atomic_wait_explicit(const atomic<T>* object,
                            typename atomic<T>::value_type old,
                            memory_order order);
```

6 *Requires*: The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

7 *Effects*: Repeatedly performs the following steps, in order:

1. Evaluates `object->load(order) != old` then, if the result is `true`, returns.
2. Blocks until an implementation-defined condition has been met. [*Note*: Consequently, it may unblock for reasons other than a call to a notifying function. - *end note*]

```
template <class T>
  void atomic_wait(const volatile atomic<T>* object,
                   typename atomic<T>::value_type old);
template <class T>
  void atomic_wait(const atomic<T>* object,
                   typename atomic<T>::value_type old);
```

8 *Effects*: Equivalent to: `atomic_wait_explicit(object, old, memory_order_seq_cst);`

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	Semaphores	`<semaphore>`

33.7 Semaphores [thread.semaphore]

1 Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.

2 A counting semaphore is a semaphore object that models a non-negative resource count. A binary semaphore is a semaphore object that has only two states, also known as available and unavailable. [*Note*: A binary semaphore should be more efficient than a counting semaphore with a unit magnitude count. – *end note*]

33.7.1 Header `<semaphore>` synopsis [semaphore.syn]:

```
namespace std {

  template<ptrdiff_t max_value>
```

```
    class basic_semaphore;

  using counting_semaphore = basic_semaphore<implementation-defined>;
  using binary_semaphore = basic_semaphore<1>;
}
```

33.7.2 Class template `basic_semaphore` [semaphore.basic]:

1 Class `basic_semaphore` maintains an internal `counter` that is initialized when the semaphore is created. Threads may block waiting until `counter >= 1`.

2 Semaphores permit concurrent invocation of the `release`, `acquire`, `try_acquire`, `try_acquire_for`, and `try_acquire_until` member functions.

```
namespace std {

  template<ptrdiff_t least_max_value>
  class basic_semaphore {
  public:
    static constexpr ptrdiff_t max() noexcept;

    explicit constexpr basic_semaphore(ptrdiff_t);
    ~basic_semaphore();

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

    void release(ptrdiff_t = 1);
    void acquire();
    bool try_acquire();
    template <class Clock, class Duration>
      bool try_acquire_until(chrono::time_point<Clock, Duration> const&);
    template <class Rep, class Period>
      bool try_acquire_for(chrono::duration<Rep, Period> const&);
  private:
    ptrdiff_t counter; // exposition only
  };
}
```

```
static constexpr ptrdiff_t max() noexcept;
```

1 *Returns:* The maximum value of `counter`. This value shall not be less than that of the template argument `least_max_value`. [*Note*: The value may exceed `least_max_value`. – *end note*]

```
explicit constexpr binary_semaphore(ptrdiff_t desired);
```

2 *Requires*: `desired >= 0` and `desired <= max()`.

3 *Effects*: Initializes `counter` with the value `desired`.

```
~binary_semaphore();
```

4 *Requires*: For every function call that blocks on `counter`, a function call that will cause it to unblock and return shall happen before this call. [*Note:* This relaxes the usual rules, which would have required all wait calls to happen before destruction. — *end note*]

5 *Effects*: Destroys the object.

```
void release(ptrdiff_t update = 1);
```

6 *Requires*: `update >= 0`, and `counter + update <= max()`.

7 *Effects*: `counter += update`, executed atomically. If any threads are blocked on `counter`, unblocks them.

8 *Synchronization*: Strongly happens before invocations of `try_acquire()` that observe the result of the effects.

```
bool try_acquire();
```

9 *Effects*:

a) With low probability, returns immediately. [Note: An implementation should ensure that `try_acquire()` does not consistently return `false` in the absence of contending acquisitions. — end note.]

b) Otherwise, `if(counter >= 1) counter -= 1`, executed atomically.

10 *Returns*: `true` if `counter` was decremented, otherwise `false`.

```
void acquire();
```

11 *Effects*: Repeatedly performs the following steps, in order:

a) Evaluates `try_acquire()` then, if the result is `true`, returns.

b) Blocks until `counter >= 1`.

```
template <class Clock, class Duration>
  bool try_acquire_until(chrono::time_point<Clock, Duration> const& abs_time);

template <class Rep, class Period>
  bool try_wait_for(chrono::duration<Rep, Period> const& rel_time);
```

12 *Effects*: Repeatedly performs the following steps, in order:

c) Evaluates `try_acquire()`. If the result is `true`, returns `true`.

d) Blocks until the timeout expires or `counter >= 1`. If the timeout expired, returns `false`.

13 *Throws*: Timeout-related exceptions (33.2.4).

33.7.3 Semaphores with predefined parameters [semaphore.predef]:

```
using counting_semaphore = basic_semaphore<implementation-defined>;
```

1 The name `counting_semaphore` introduces a counting semaphore type with a maximum value that should be at least as large as the maximum number of threads the implementation can support.

```
using binary_semaphore = basic_semaphore<1>;
```

2 The name `binary_semaphore` introduces a binary semaphore type with a required maximum value of `1`.