Doc number: P0514R2
Revises: P0514R1
Date: 2017-10-09
Project: Programming Language C++, Concurrency Working Group
Reply-to: **Olivier Giroux** <ogiroux@nvidia.com>

Efficient waiting for concurrent programs.

We propose to create new specialized atomic types that likely replace `atomic_flag` in practice, and new atomic free functions that provide useful and efficient waiting functionality for other atomic types.

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

The current `atomic_flag` object does nothing to help with this problem, despite its name that suggests it is suitable for this use. Its interface is tightly-fitted to the demands of the simplest spinlocks without contention or energy mitigation beyond what timed back-off can achieve.

Presenting a simple abstraction for scalable waiting.

On its own, a binary semaphore is analogous to a lock without thread ownership. It is natural, therefore, that our `std::binary_semaphore` object can easily be adapted to serve the role of a lock:

```
struct semaphore_lock {
    void lock() {
        s.acquire();
    }
    void unlock() {
        s.release();
    }
private:
    std::binary_semaphore s(1);
};
```

This example uses the binary semaphore type. A counting semaphore type is also provided, which permits the simultaneous release and acquisition of multiple credits to the semaphore.

New atomic free functions are provided to enable pre-existing uses of atomics to benefit from the same efficient waiting implementation that is behind the semaphore:

```
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 that in high-quality implementations this necessitates a semaphore table owned by the implementation, which causes some unavoidable interference due to aliasing unrelated atomic updates. For greater control over this sort of interference, it is also possible to supply a semaphore owned by the program, using the `condition_variable_atomic` type, restricted for this use:

```
struct improved_simple_lock {
    void lock() {
        bool old;
        while(!b.compare_exchange_weak(old = false, true))
            s.wait(&b, true);
    }
    void unlock() {
        b = false;
        s.notify_one(&b);
    }
private:
    std::atomic<bool> b = ATOMIC_VAR_INIT(false);
    std::condition_variable_atomic s;
};
```

A reference implementation is provided for your evaluation.

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

Please see P0514R0, P0514R1, P0126 and N4195 for additional analysis not repeated here.

C++ Proposed Wording

Apply the following edits to the working draft of the Standard.

The feature test macro `__cpp_lib_semaphore` should be added.

Modify 32.2 Header <atomic> synopsis [atomics.syn]

```
 // 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_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);
  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);
}
```

Add 32.10 Waiting and notifying functions [atomics.waitnotify]

1 This section provides 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.*]

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

2 *Effects*: unblocks at least one execution of a waiting function that blocked after observing the result of preceding operations in `*object`'s modification order.

```
template <class T>
```

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

3 *Effects*: unblocks all executions of a waiting function that blocked after observing the result of preceding operations in `*object`'s modification order.

```
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);
```

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

5 *Effects*: Each execution is performed as:

1. Evaluates `object->load(order) != old` then, if the result is `true`, returns.
2. Blocks.
3. Unblocks when:
   - As a result of invoking `atomic_notify_all` or `atomic_notify_one`, as described in that function's effects.
   - At the implementation's discretion.
4. Each time the execution unblocks, it repeats.

```
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);
```

6 *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>

Modify 33.5 Condition variables [thread.condition]

1 Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class `condition_variable` provides a condition variable that can only wait on an object of type `unique_lock<mutex>`, allowing maximum efficiency on some platforms. Class `condition_variable_any` provides a general condition variable that can wait on objects of user-supplied lock types. Class `condition_variable_atomic` provides a specialized condition variable that evaluates predicates from a single object of class `atomic<T>`, without using a lock.

2 Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

3 The execution of `notify_one` and `notify_all` shall be atomic. The execution of `wait`, `wait_for`, and `wait_until` shall be performed in up to three atomic parts:

1. the release of ~~the~~ any user-supplied lock ~~mutex~~, or the evaluation of a predicate over an object of class `atomic<T>`, and entry into the waiting state;
2. the unblocking of the wait; and
3. the reacquisition of ~~the~~ any user-supplied lock.

Modify 33.5.1 Header <condition_variable> synopsis [condition_variable.syn]

```
namespace std {
  class condition_variable;
  class condition_variable_any;
  class condition_vatiable_atomic;
  void notify_all_at_thread_exit(condition_variable& cond,
                                 unique_lock<mutex> lk);
  enum class cv_status { no_timeout, timeout };
}
```

Add 33.5.5 Class condition_variable_atomic [thread.condition.condvaratomic]

1 Class `condition_variable_atomic` is used with an object of class `atomic<T>` without the need to hold a lock. It is unspecified whether operations on class `condition_variable_atomic` are lock-free.

```
namespace std {
  class condition_variable_atomic {
  public:

    condition_variable_atomic();
    ~condition_variable_atomic();

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

    template <class T>
      void notify_one(const atomic<T>&) noexcept;
    template <class T>
      void notify_one(const volatile atomic<T>&) noexcept;
    template <class T>
      void notify_all(const atomic<T>&) noexcept;
    template <class T>
      void notify_all(const volatile atomic<T>&) noexcept;
```

```
    template <class T>
      void wait(const volatile atomic<T>&, typename atomic<T>::value_type,
                memory_order = memory_order_seq_cst);
    template <class T>
      void wait(const atomic<T>&, typename atomic<T>::value_type,
                memory_order = memory_order_seq_cst);
    template <class T, class Predicate>
      void wait(const volatile atomic<T>&, typename atomic<T>::value_type,
                Predicate pred, memory_order = memory_order_seq_cst);
    template <class T, class Predicate>
      void wait(const atomic<T>&, typename atomic<T>::value_type,
                Predicate pred, memory_order = memory_order_seq_cst);
    template <class T, class Clock, class Duration>
      bool wait_until(const volatile atomic<T>&, typename atomic<T>::value_type,
                      chrono::time_point<Clock, Duration> const&,
                      memory_order = memory_order_seq_cst);
    template <class T, class Clock, class Duration>
      bool wait_until(const atomic<T>&, typename atomic<T>::value_type,
                      chrono::time_point<Clock, Duration> const&,
                      memory_order = memory_order_seq_cst);
    template <class T, class Clock, class Duration, class Predicate>
      bool wait_until(const volatile atomic<T>&, typename atomic<T>::value_type,
                      chrono::time_point<Clock, Duration> const&,
                      Predicate pred, memory_order = memory_order_seq_cst);
    template <class T, class Clock, class Duration, class Predicate>
      bool wait_until(const atomic<T>&, typename atomic<T>::value_type,
                      chrono::time_point<Clock, Duration> const&,
                      Predicate pred, memory_order = memory_order_seq_cst);
    template <class T, class Rep, class Period>
      bool wait_for(const volatile atomic<T>&, typename atomic<T>::value_type,
                    chrono::duration<Rep, Period> const&,
                    memory_order = memory_order_seq_cst);
    template <class T, class Rep, class Period>
      bool wait_for(const atomic<T>&, typename atomic<T>::value_type,
                    chrono::duration<Rep, Period> const&,
                    memory_order = memory_order_seq_cst);
    template <class T, class Rep, class Period, class Predicate>
      bool wait_for(const volatile atomic<T>&, typename atomic<T>::value_type,
                    chrono::duration<Rep, Period> const&,
                    Predicate pred, memory_order = memory_order_seq_cst);
    template <class T, class Rep, class Period, class Predicate>
      bool wait_for(const atomic<T>&, typename atomic<T>::value_type,
                    chrono::duration<Rep, Period> const&,
                    Predicate pred, memory_order = memory_order_seq_cst);
  };
}

condition_variable_atomic();
```

1 *Effects:* Constructs an object of type `condition_variable_atomic`.

2 *Throws:* `system_error` when an exception is required (33.2.2).

3 *Error conditions:*

— `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

```
~condition_variable_atomic();
```

4 *Requires*: There shall be no thread blocked on `*this`. [*Note:* That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must take care to ensure that no threads wait on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. — *end note*]

5 *Effects*: Destroys the object.

```
void notify_one(const volatile atomic<T>& object) noexcept;
void notify_one(const atomic<T>& object) noexcept;
```

6 *Effects:* If any threads are blocked waiting for `*this` and `object`, unblocks one of those threads.

```
void notify_all(const volatile atomic<T>& object) noexcept;
void notify_all(const atomic<T>& object) noexcept;
```

7 *Effects:* Unblocks all threads that are blocked waiting for `*this` and `object`.

```
template <class T>
  void wait(const volatile atomic<T>& object, typename atomic<T>::value_type old,
            memory_order order = memory_order_seq_cst);
template <class T>
  void wait(const atomic<T>& object, typename atomic<T>::value_type old,
            memory_order order = memory_order_seq_cst);
template <class T, class Predicate>
  void wait(const volatile atomic<T>& object, typename atomic<T>::value_type old,
            Predicate pred, memory_order order = memory_order_seq_cst);
template <class T, class Predicate>
  void wait(const atomic<T>& object, typename atomic<T>::value_type old,
            Predicate pred, memory_order order = memory_order_seq_cst);
```

8 *Effects*: Each execution is performed as:

1. Evaluates the condition that applies:
   - `object.load(order) != old` or
   - `pred(object.load(order))`.
2. If the result `true`, returns.
3. Blocks on `*this` and `object`.
4. Unblocks when:
   - As a result of a notify operation, as described in that function's effects.
   - At the implementation's discretion.
5. Each time the execution unblocks, it repeats.

```
template <class T, class Clock, class Duration>
  bool wait_until(const volatile atomic<T>& object,
                  typename atomic<T>::value_type old,
                  chrono::time_point<Clock, Duration> const& abs_time,
                  memory_order order = memory_order_seq_cst);
template <class T, class Clock, class Duration>
  bool wait_until(const atomic<T>& object, typename atomic<T>::value_type old,
                  chrono::time_point<Clock, Duration> const& abs_time,
                  memory_order order = memory_order_seq_cst);
template <class T, class Clock, class Duration, class Predicate>
  bool wait_until(const volatile atomic<T>& object,
```

```
                  chrono::time_point<Clock, Duration> const& abs_time,
                  Predicate pred, memory_order order = memory_order_seq_cst);
template <class T, class Clock, class Duration, class Predicate>
  bool wait_until(const atomic<T>& object,
                  chrono::time_point<Clock, Duration> const& abs_time,
                  Predicate pred, memory_order order = memory_order_seq_cst);
template <class T, class Rep, class Period>
  bool wait_for(const volatile atomic<T>& object, typename atomic<T>::value_type old,
                chrono::duration<Rep, Period> const& rel_time,
                memory_order order = memory_order_seq_cst);
template <class T, class Rep, class Period>
  bool wait_for(const atomic<T>& object, typename atomic<T>::value_type old,
                chrono::duration<Rep, Period> const& rel_time,
                memory_order order = memory_order_seq_cst);
template <class T, class Rep, class Period, class Predicate>
  bool wait_for(const volatile atomic<T>& object,
                chrono::duration<Rep, Period> const& rel_time,
                Predicate pred, memory_order order = memory_order_seq_cst);
template <class T, class Rep, class Period, class Predicate>
  bool wait_for(const atomic<T>& object,
                chrono::duration<Rep, Period> const& rel_time,
                Predicate pred, memory_order order = memory_order_seq_cst);
```

9 *Effects*: Each execution is performed as:

1. Evaluates the condition that applies:
   - `object.load(order) != old` or
   - `pred(object.load(order))`.
2. If the result is `true`, or spuriously, returns.
3. Blocks on `*this` and `object`.
4. Unblocks when:
   - As a result of a notify operation, as described in that function's effects.
   - The timeout expires.
   - At the implementation's discretion.
5. Each time the execution unblocks, it repeats.

10 *Returns*: The result of the condition evaluation in step 1.

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

Add 33.7 Semaphores [thread.semaphores]

1 Semaphores are lightweight synchronization primitives that control concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, are more efficient than condition variables. Class `binary_semaphore` has two states, also known as available and unavailable. Class `counting_semaphore` models a non-negative count.

2 Semaphores permit concurrent invocation of the `acquire`, `acquire_for`, `acquire_until` and `release` member functions.

3 Semaphore construction and destruction need not be synchronized.

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

```
namespace std {
  class binary_semaphore;
  class counting_semaphore;
}
```

Add 33.7.2 Class binary_semaphore [semaphore.binary]:

```
namespace std {
  class binary_semaphore {
  public:
    using count_type = implementation-defined; // see 33.7.2.1
    static constexpr count_type max = 1;

    binary_semaphore(count_type = 0);
    ~binary_semaphore();

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

    void release();
    void acquire();
    void 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&);
  };
}
```

```
using count_type = implementation-defined;
```

1 An unsigned integral type able to represent the range between zero and `max`, inclusive.

```
constexpr binary_semaphore(count_type desired = 0);
```

2 *Requires*: `desired` is no greater than `max`.

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

```
~binary_semaphore();
```

4 *Requires*: There are no threads blocked on `*this`. [*Note*: Returns from invocations of waiting functions do not need to happen before destruction, however the notification by signaling functions to unblock the waiting functions must happen before destruction. This is a weaker requirement than normal. – *end note*]

5 *Effects*: Destroys the object.

```
void release();
```

6 *Requires*: The value pointed to by `this` is less than `max`.

7 *Effects*: Atomically increments the value pointed to by `this` by 1. Unblocks at least one execution of a waiting function that blocked after observing the result of preceding operations in the object's modification order.

8 *Synchronization*: Synchronizes-with invocations of waiting functions that unblock as a result of the effects.

```
bool try_acquire();
```

9 *Effects*: Subtracts 1 from the value pointed to by `this` then, if the result is positive or zero, atomically replaces the value with the result. An implementation may spuriously fail to replace the value if there are contending invocations in other threads.

10 *Returns*: `true` if the value was replaced, otherwise `false`.

```
void acquire();
```

11 *Effects*: Each execution is performed as:

1. Evaluates `try_acquire()` then, if the result is `true`, returns.
2. Blocks on `*this`.
3. Unblocks when:
   - As a result of a release operation, as described in that function's effects.
   - At the implementation's discretion.
4. Each time the execution unblocks, it repeats.

```
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*: Each execution is performed as:

1. Evaluates `try_acquire()` then, if the result is `true` or spuriously, returns.
2. Blocks on `*this`.
3. Unblocks when:
   - As a result of a release operation, as described in that function's effects.
   - The timeout expires.
   - At the implementation's discretion.
4. Each time the execution unblocks, it repeats.

13 *Returns*: The result of the invocation in step 1.

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

Add 33.7.3 Class counting_semaphore [semaphore.counting]:

```
namespace std {
  class counting_semaphore {
  public:
    using count_type = implementation-defined; // see 33.7.3.1
    static constexpr count_type max = implementation-defined; // see 33.7.3.2

    counting_semaphore(count_type = 0);
    ~counting_semaphore();

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

    void release(count_type = 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&);
  };
}

using count_type = implementation-defined;
```

1 An unsigned integral type able to represent the range between zero and `max`, inclusive.

```
static constexpr count_type max = implementation-defined;
```

2 The maximum value that the semaphore can hold.

```
constexpr counting_semaphore(count_type desired = 0);
```

3 *Requires*: `desired` is no greater than `max`.

4 *Effects*: Initializes the object with the value `desired`.

```
~counting_semaphore();
```

5 *Requires*: There are no threads blocked on `*this`. [*Note*: Returns from invocations of waiting functions do not need to happen before destruction, however the notification by signaling functions to unblock the waiting functions must happen before destruction. This is a weaker requirement than normal. – *end note*]

6 *Effects*: Destroys the object.

```
void release(count_type count = 1);
```

7 *Requires*: The value pointed to by `this` is less than `max + count - 1`.

8 *Effects*: Atomically increments the value pointed to by `this` by `count`. Unblocks executions of waiting functions that blocked after observing the result of preceding operations in the object's modification order.

9 *Synchronization*: Synchronizes-with invocations of waiting functions that unblock as a result of the effects.

```
bool try_acquire();
```

10 *Effects*: Subtracts 1 from the value pointed to by `this` then, if the result is positive or zero, atomically replaces the value with the result. An implementation may spuriously fail to replace the value if there are contending invocations in other threads.

11 *Returns*: `true` if the value was replaced, otherwise `false`.

```
void acquire();
```

12 *Effects*: Each execution is performed as:

5. Evaluates `try_acquire()` then, if the result is `true`, returns.
6. Blocks on `*this`.
7. Unblocks when:
   - As a result of a release operation, as described in that function's effects.
   - At the implementation's discretion.
8. Each time the execution unblocks, it repeats.

```
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);
```

15 *Effects*: Each execution is performed as:

5. Evaluates `try_acquire()` then, if the result is `true` or spuriously, returns.
6. Blocks on `*this`.
7. Unblocks when:
   - As a result of a release operation, as described in that function's effects.
   - The timeout expires.
   - At the implementation's discretion.
8. Each time the execution unblocks, it repeats.

16 *Returns*: The result of the invocation in step 1.

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