Doc number: P0514R1
Revises: P0514R0
Date: 2017-6-14
Project: Programming Language C++, Concurrency Working Group
Reply-to: **Olivier Giroux** <ogiroux@nvidia.com>

Adding `std::*_semaphore` to the atomics clause.

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 semlock {
    void lock() {
        s.acquire();
    }
    void unlock() {
        s.release();
    }
private:
    std::experimental::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::experimental::atomic_wait(&b, old);
```

```
    }
    void unlock() {
        b = false;
        std::experimental::atomic_signal(&b);
    }
private:
    std::atomic<bool> b(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 atomic semaphore type, restricted for this use:

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

Note that `atomic_semaphore` has no public interface, it is a semaphore in name ***and*** implementation only.

A reference implementation is provided for your evaluation.

It's here - https://github.com/ogiroux/semaphore - though as of press time it is not up-to-date.

See P0514R0 for past performance analysis that has not been invalidated.

Note: the `synchronic<T>` interface of either P0126 or N4195 is no longer recommended.

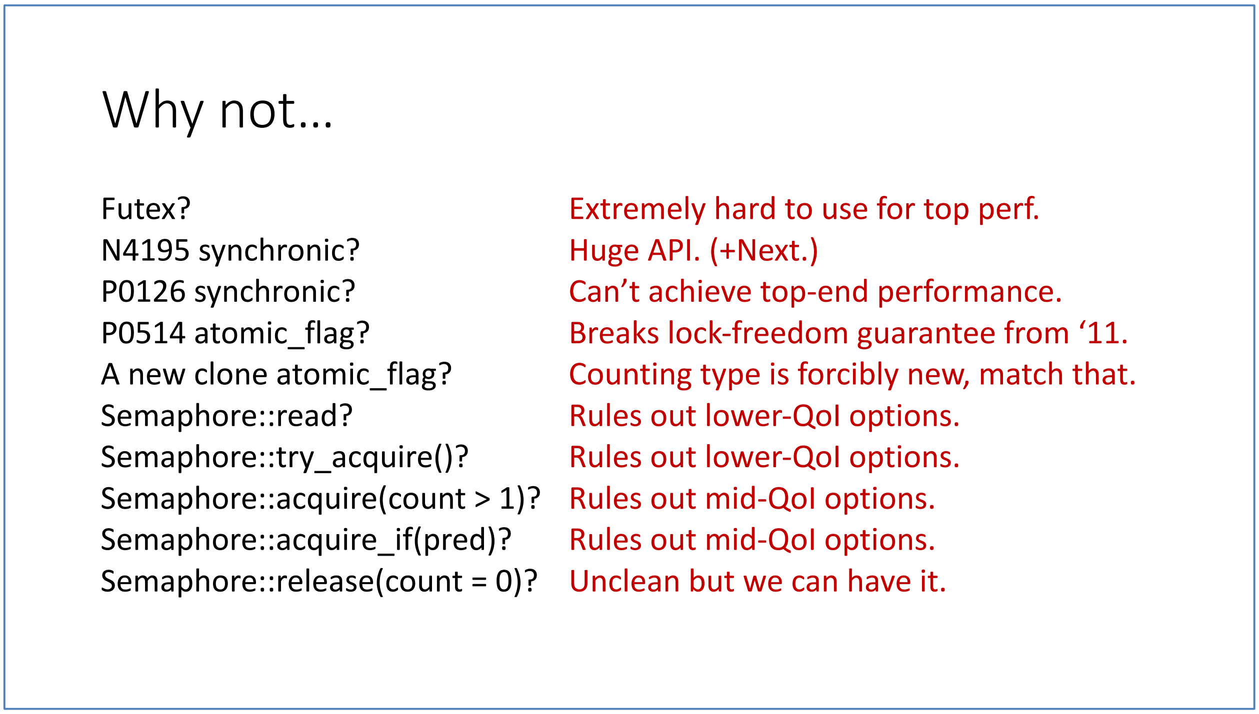
In short, the highest performance is not achievable with the most recent `synchronic<T>` interface because additional atomic operations are imposed by the abstraction. Specifically, two atomics are needed to synchronize and manage contention, whereas an optimized implementation may be able to fuse them into one.

This new approach provides strictly more implementation freedom, including the freedom to fuse contention-management with synchronization. The implementation is not made any simpler, note.

Note: the extended `atomic_flag` interface of P0514R0 is no longer recommended.

Lock-freedom is guaranteed to `atomic_flag` and could not be preserved with the extension.

Note: this slide from the Kona meeting summarizes the above, and a few more.



Except that we recommend, now, to include both "*try_acquire(…)*" and "*acquire(count)*" equivalents (now dubbed `try_wait` and `wait`) instead and let them degenerate to either zero-time timed operations or a loop over scalar operations.

Note: this is proposed for addition to clause 32 (atomics) instead of 33 (thread support).

Why? Several reasons. Some semaphore functions take memory_order operands, operations on semaphores do not introduce data-races without synchronization, memory consistency effects of semaphores match read-modify-write operations on atomic objects, semaphore semantics refer to a modification order for the integral counter that the semaphore models, some semaphores functions take atomic objects as operands, some atomic free functions take semaphore enums as operands, all semaphore objects are designed to leave room open for C compatibility. Clause 33 wouldn't be an awful bad fit, but not as good a fit.

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, semaphore type and operations
  enum semaphore_notify {
    semaphore_notify_all,
    semaphore_notify_one,
    semaphore_notify_none
  };
  class binary_semaphore;
  class counting_semaphore;
  class atomic_semaphore;

 // 32.10.1, atomic free functions for waiting
  template <class T>
  void atomic_signal_explicit(const atomic<T>*, semaphore_notify);
  template <class T>
  void atomic_signal(const atomic<T>*);
  template <class T>
  void atomic_wait_explicit(const atomic<T>*,
                            typename atomic<T>::value_type, memory_order);
  template <class T>
  void atomic_wait(const atomic<T>*, typename atomic<T>::value_type);
  template <class T>
  bool atomic_try_wait_explicit(const atomic<T>*,
                                typename atomic<T>::value_type, memory_order);
  template <class T>
  bool atomic_try_wait(const atomic<T>*, typename atomic<T>::value_type);

}
```

Create 32.10 Semaphore types and operations [atomics.semaphore]:

```
namespace std {

  class binary_semaphore {

    using count_type = implementation-defined;  // see 32.10.1
    static constexpr count_type max = 1;

    void signal(memory_order = memory_order_seq_cst) noexcept;
    void signal(semaphore_notify, memory_order = memory_order_seq_cst) noexcept;
    void wait(memory_order = memory_order_seq_cst) noexcept;
    bool try_wait(memory_order = memory_order_seq_cst) noexcept;
    template <class Clock, class Duration>
    bool try_wait_until(chrono::time_point<Clock, Duration> const&,
                        memory_order = memory_order_seq_cst) noexcept;
    template <class Rep, class Period>
    bool try_wait_for(chrono::duration<Rep, Period> const&,
                      memory_order = memory_order_seq_cst) noexcept;

    constexpr binary_semaphore(count_type = 0) noexcept;
    ~binary_semaphore();
    binary_semaphore(const binary_semaphore&) = delete;
    binary_semaphore& operator=(const binary_semaphore&) = delete;
  };

  struct counting_semaphore {

    using count_type = implementation-defined; // see 32.10.1
```

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

    void signal(memory_order = memory_order_seq_cst) noexcept;
    void signal(count_type, memory_order = memory_order_seq_cst) noexcept;
    void signal(count_type, semaphore_notify,
                memory_order = memory_order_seq_cst) noexcept;
    void wait(memory_order = memory_order_seq_cst) noexcept;
    bool try_wait(memory_order = memory_order_seq_cst) noexcept;
    template <class Clock, class Duration>
    bool try_wait_until(chrono::time_point<Clock, Duration> const&,
                        memory_order = memory_order_seq_cst) noexcept;
    template <class Rep, class Period>
    bool try_wait_for(chrono::duration<Rep, Period> const&,
                      memory_order = memory_order_seq_cst) noexcept;

    constexpr counting_semaphore(count_type = 0) noexcept;
    ~counting_semaphore();
    counting_semaphore(const counting_semaphore&) = delete;
    counting_semaphore& operator=(const counting_semaphore&) = delete;
  };

  struct atomic_semaphore {

    template <class T>
    void signal(const atomic<T>*, semaphore_notify = semaphore_notify_all) noexcept;
    template <class T>
    void wait(const atomic<T>*, typename atomic<T>::value_type,
              memory_order = memory_order_seq_cst) noexcept;
    template <class T>
    bool try_wait(const atomic<T>*, typename atomic<T>::value_type,
                  memory_order = memory_order_seq_cst) noexcept;
    template <class T, class Clock, class Duration>
    bool try_wait_until(const atomic<T>*, typename atomic<T>::value_type,
                        chrono::time_point<Clock, Duration> const&,
                        memory_order = memory_order_seq_cst) noexcept;
    template <class T, class Rep, class Period>
    bool try_wait_for(const atomic<T>*, typename atomic<T>::value_type,
                      chrono::duration<Rep, Period> const&,
                      memory_order = memory_order_seq_cst) noexcept;

    constexpr atomic_semaphore() noexcept;
    ~atomic_semaphore();
    atomic_semaphore(const atomic_semaphore&) = delete;
    atomic_semaphore& operator=(const atomic_semaphore&) = delete;
  };

}
```

1 Semaphores are non-negative integral atomic objects with specialized operations that efficiently block the invoking thread as long as they would result in a negative value, or until a timeout has elapsed. Semaphores are widely useful to efficiently implement concurrent control constructs such as locks, barriers and queues.

2 Class `binary_semaphore` has two states, also known as available and unavailable, while class `counting_semaphore` holds arbitrary positive integral values within a specified representable range. Operations on these types do not introduce data-races, are considered atomic read-modify-write operations, and are not guaranteed to be lock-free (4.7).

3 Class `atomic_semaphore` refers to a separate atomic object in the program to define its state. This facility enables the combination of efficient waiting algorithms with atomic objects whose usage is not limited to the semaphore concept. [*Note:* Programs using this facility 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.*]

4 For the following definitions:

- Member functions that match the name `signal`, and non-member functions that match the pattern `atomic_signal`..., are signaling functions.
- Member functions that match the name `wait` or `try_wait_for` or `try_wait_until`, and non-member functions that match the pattern `atomic_wait`..., are waiting functions.

5 Waiting functions may block until they are unblocked by signaling functions, according to each function's effects.

32.10.1 Operations on semaphore types [atomics.semaphore.operations]:

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

1 An integral type able to represent all of the valid values of the semaphore object.

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

2 The maximum value of the semaphore object. If any operation on a semaphore would result in a greater value then the result is undefined, even if `count_type` could represent it exactly.

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

constexpr counting_semaphore(count_type desired = 0) noexcept;

constexpr atomic_semaphore() noexcept;
```

3 *Effects*: Initializes the object with the value `desired`, if specified, or a default state otherwise. Initialization is not an atomic operation (4.7).

```
~binary_semaphore();

~counting_semaphore();

~atomic_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 binary_semaphore::signal(semaphore_notify notify, memory_order order =
memory_order_seq_cst) noexcept;
```

```
void counting_semaphore::signal(count_type count, memory_order order =
memory_order_seq_cst) noexcept;

void counting_semaphore::signal(count_type count, semaphore_notify notify,
memory_order order = memory_order_seq_cst) noexcept;
```

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

7 *Effects*:

1. Atomically increments the value pointed to by `this` by 1 or `count`, if specified. Memory is affected according to the value of `order`.
2. If `notify` is `semaphore_notify_all`, unblocks all executions of waiting functions that blocked after observing the result of preceding operations in the object's modification order.
3. If `notify` is `semaphore_notify_one`, unblocks at least one execution of a waiting function that blocked after observing the result of preceding operations in the object's modification order.

```
void binary_semaphore::signal(memory_order order = memory_order_seq_cst) noexcept;
void counting_semaphore::signal(memory_order order = memory_order_seq_cst) noexcept;
```

8 *Effects*: Equivalent to: `signal(semaphore_notify_all, order);`

```
template <class T>
void atomic_signal_explicit(const atomic<T>* object, semaphore_notify notify);

template <class T>
void atomic_semaphore::signal(const atomic<T>* object, semaphore_notify notify =
semaphore_notify_all) noexcept;
```

9 *Effects*:

1. If `notify` is `semaphore_notify_all`, unblocks all executions of waiting functions that blocked after observing the result of preceding operations in `*object`'s modification order.
2. If `notify` is `semaphore_notify_one`, 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_signal(const atomic<T>* object);
```

10 *Effects*: Equivalent to: `atomic_signal_explicit(object, semaphore_notify_all);`

```
bool binary_semaphore::try_wait(memory_order order = memory_order_seq_cst) noexcept;

bool counting_semaphore::try_wait(memory_order order = memory_order_seq_cst) noexcept;
```

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

12 *Effects*: Subtracts 1 or `count`, if specified, from the value pointed to by `this` then, if the result is positive or zero, atomically replaces the value with the result. Memory is affected according to the value of `order`.

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

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

template <class T>
bool atomic_semaphore::try_wait(const atomic<T>* object, typename
atomic<T>::value_type old,
              memory_order order = memory_order_seq_cst);
```

14 *Effects*: Equivalent to: `return object->load(order) != old;`

```
template <class T>
bool atomic_try_wait(const atomic<T>* object, typename atomic<T>::value_type old);
```

15 *Effects*: Equivalent to:

```
return atomic_try_wait_explicit(object, old, memory_order_seq_cst);
```

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

void binary_semaphore::wait(memory_order order = memory_order_seq_cst) noexcept;

void counting_semaphore::wait(memory_order order = memory_order_seq_cst) noexcept;

template <class T>
void atomic_semaphore::wait(const atomic<T>* object, typename atomic<T>::value_type
old, memory_order = memory_order_seq_cst) noexcept;

template <class Clock, class Duration>
bool binary_semaphore::try_wait_until(chrono::time_point<Clock, Duration> const&
abs_time, memory_order order = memory_order_seq_cst) noexcept;

template <class Clock, class Duration>
bool counting_semaphore::try_wait_until(chrono::time_point<Clock, Duration> const&
abs_time, memory_order order = memory_order_seq_cst) noexcept;

template <class T, class Clock, class Duration>
bool atomic_semaphore::try_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) noexcept;

template <class Rep, class Period>
bool binary_semaphore::try_wait_for(chrono::duration<Rep, Period> const& rel_time,
memory_order order = memory_order_seq_cst) noexcept;

template <class Rep, class Period>
bool counting_semaphore::try_wait_for(chrono::duration<Rep, Period> const& rel_time,
memory_order order = memory_order_seq_cst) noexcept;

template <class T, class Rep, class Period>
bool atomic_semaphore::try_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) noexcept;
```

16 Each waiting function has a corresponding *try-waiting* function. For member functions named `wait`, `try_wait_until` or `try_wait_for`, the try-waiting function is named `try_wait`. For the non-

member template function named `atomic_wait_explicit`, the try-waiting function is named `atomic_try_wait_explicit`.

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

18 *Effects*: Each execution of a waiting function is performed as:

1. Invokes the try-waiting function with the values of the parameters, except for timeouts, as the arguments of the function call, in order. If it returned `true`, returns.
2. If a timeout is specified, may return spuriously.
3. Blocks.
4. Unblocks when:
   – As a result of some signaling operations, as described in that function's effects.
   – The timeout expires.
   – At the implementation's discretion.
5. Each time the execution unblocks, it repeats.

19 *Returns*: if a timeout is specified, the value returned by the last invocation of `try_wait`, otherwise nothing.

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

20 *Effects*: Equivalent to:

```
atomic_wait_explicit(object, old, memory_order_seq_cst);
```