Doc number: D0514R0
Revises: None.
Date: 2016-11-06
Project: Programming Language C++, Concurrency Working Group
Reply-to: **Olivier Giroux** <ogiroux@nvidia.com>

Enhancing `std::atomic_flag` for waiting.

TL;DR summary of the goals:

1 The `atomic_flag` type ~~provides~~ combines the classic test-and-set functionality with the ability to block until the object is in a specified state. It has two states, set and clear.

We propose to make `atomic_flag` more useful and efficient, without loss of compatibility.

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 backoff can achieve.

Presenting a simple abstraction for scalable waiting.

Our proposed enhancements for `atomic_flag` objects make it easier to implement scalable and efficient synchronization using atomic objects.

For example:

```
struct atomic_flag_lock {
    void lock() {
        while (f.test_and_set())
            f.wait(false);
    }
    void unlock() {
        f.clear();
    }
private:
    std::experimental::atomic_flag f = ATOMIC_FLAG_INIT;
};
```

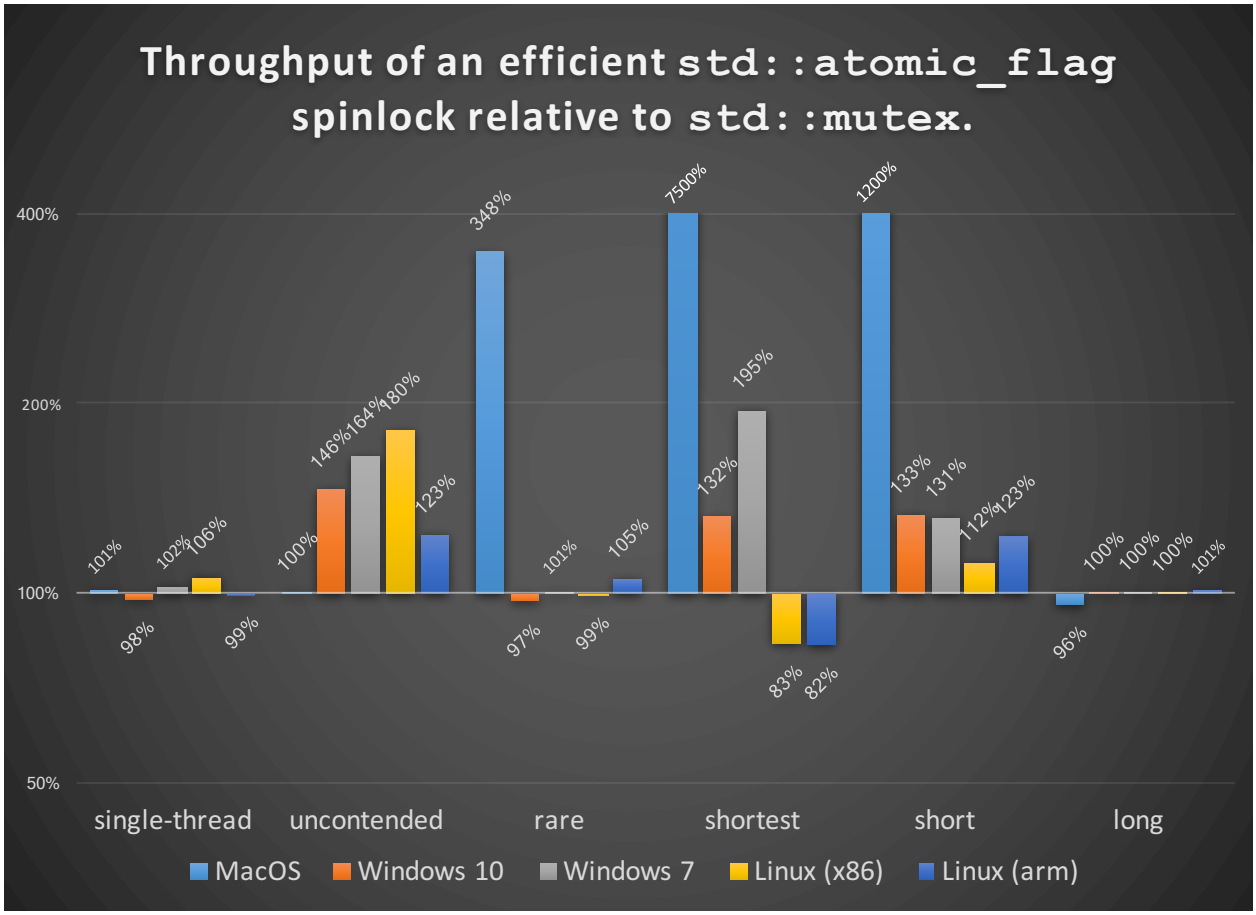
A reference implementation is provided for your evaluation.

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

We show a gently-optimized version of the previous program correspdonging to the data plot (below) that shows aggregate throughput (in lock grants per unit time) under different operating conditions:

```
using align = std::hardware_destructive_interference_size;
struct alignas(align) atomic_flag_lock {
  void lock() {
    while (f.test_and_set(std::memory_order_acquire))
      f.wait(false, std::memory_order_relaxed);
  }
  void unlock() {
    f.clear(std::memory_order_release);
  }
private:
  std::experimental::atomic_flag f = ATOMIC_FLAG_INIT;
};
```

(We apologize that memory model and alignment optimizations are required in the real world.)



Operating conditions: Single-threaded, uncontended, and rare use 1 lock per thread. | Rare has threads acquire randomly-chosen locks. | Shortest, short, long use 1 lock.
All but the single-threaded operating condition use the maximum number of physical threads.
System information: MacOS, Linux (x86), Windows: i7-4850HQ | Linux (arm): Jetson TX1.
= Repeatability error of approx. 5%,. = YMMV. =

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.

The requirements placed on legacy platforms are not (much) worse.

Legacy, or lower-quality implementations, will need to ensure that they can atomically read the underlying type. Although the current interface of `atomic_flag` doesn't expose this ability to users, we believe that there exist no platform that support `atomic_flag` and could not expose a `load` capability.

Higher-quality implementations will want to use an `atomic<int8_t>`, or better, and make use of the full expressiveness of this type in order to efficiently deal with contention.

There is no obvious ABI problem.

We do not foresee a need to break the ABI for this feature because good implementations are available that fit in a single byte on all modern systems, and the prior interface is still supported. Modest simplifications to the implementation are possible if an `atomic<int32_t>` can be used, nevertheless.

This extension was foreseen when `atomic_flag` was introduced.

For example, wait functions figure in N2145, N2324 and N2393 but are not proposed for inclusion. The arguments given are twofold: 1) that the authors did not expect the type to be used outside of lock implementations for atomic types that aren't lock-free, and 2) that the implementation quality foreseen at the time was low. Notably these papers don't mention platform concerns.

The extension preserves the guarantee of lock-freedom.

Just in case you were wondering, it is the case.

Example implementation of the functionality in the prior paper using features of this paper.

Standardization of this functionality is not proposed in this paper.

```
template <class T, class V>
void atomic_notify(
  std::experimental::atomic_flag& f,
  std::atomic<T>& a,
  V newval,
  std::memory_order order = std::memory_order_seq_cst,
  std::experimental::atomic_notify notify = std::experimental::atomic_notify::all) {

    a.store(newval); //requires sc
    if (f.test()) //requires sc
        f.set(false, std::memory_order_relaxed, notify);
}

template <class T, class V>
void atomic_wait(
  std::experimental::atomic_flag& f,
  std::atomic<T> const& a,
  V current,
  std::memory_order order = std::memory_order_seq_cst) {

    for (int i = 0; i < 32; ++i, std::this_thread::yield())
        if (a.load(order) != current)
            return;
    while (1) {
        f.set(true, std::memory_order_seq_cst,
              std::experimental::atomic_notify::none); //requires sc
        if (a.load() != current) //requires sc
            return;
        f.wait(false, std::memory_order_relaxed);
    }
}
```

C++ Proposed Wording

Apply the following edits to the working draft of the Standard. The feature test macro `__cpp_lib_atomic_flag_wait` should be added.

Add to [atomics.syn]:

```
  // 29.7, flag type and operations
  enum class atomic_notify {
    all, one, none
  };
  struct atomic_flag;
  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*,
    memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit_notify(volatile atomic_flag*,
    memory_order, atomic_notify) noexcept;
  bool atomic_flag_test_and_set_explicit_notify(atomic_flag*,
    memory_order, atomic_notify) noexcept;
  void atomic_flag_clear(volatile atomic_flag*) noexcept;
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit_notify(volatile atomic_flag*,
    memory_order, atomic_notify) noexcept;
  void atomic_flag_clear_explicit_notify(atomic_flag*, memory_order,
    atomic_notify) noexcept;
  bool atomic_flag_test(const volatile atomic_flag*) noexcept;
  bool atomic_flag_test(const atomic_flag*) noexcept;
  bool atomic_flag_test_explicit(const volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_explicit(const atomic_flag*, memory_order) noexcept;
  void atomic_flag_wait(const volatile atomic_flag*, bool);
  void atomic_flag_wait(const atomic_flag*, bool);
  void atomic_flag_wait_explicit(const volatile atomic_flag*, bool, memory_order);
  void atomic_flag_wait_explicit(const atomic_flag*, bool, memory_order);
  #define ATOMIC_FLAG_INIT see below
```

Add to [atomics.flag]:

```
namespace std {
  typedef struct atomic_flag {
    bool test_and_set(memory_order order = memory_order_seq_cst,
        atomic_notify notify = atomic_notify::all) noexcept;
    bool test_and_set(memory_order order = memory_order_seq_cst,
        atomic_notify notify = atomic_notify::all) volatile noexcept;
    void clear(memory_order order = memory_order_seq_cst,
        atomic_notify notify = atomic_notify::all) noexcept;
    void clear(memory_order order = memory_order_seq_cst,
        atomic_notify notify = atomic_notify::all) volatile noexcept;
    void set(bool state, memory_order order = memory_order_seq_cst,
        atomic_notify notify = atomic_notify::all) noexcept;
    void set(bool state, memory_order order = memory_order_seq_cst,
        atomic_notify notify = atomic_notify::all) volatile noexcept;
    bool test(memory_order order = memory_order_seq_cst) const noexcept;
    bool test(memory_order order = memory_order_seq_cst) const volatile noexcept;
    void wait(bool set, memory_order order = memory_order_seq_cst) const noexcept;
    void wait(bool set,
        memory_order order = memory_order_seq_cst) const volatile noexcept;
    template <class Clock, class Duration>
    bool wait_until(bool set, chrono::time_point<Clock, Duration> const& abs_time,
```

```
      memory_order order = memory_order_seq_cst) const;
    template <class Clock, class Duration>
    bool wait_until(bool set, chrono::time_point<Clock, Duration> const& abs_time,
      memory_order order = memory_order_seq_cst) const volatile;
    template <class Rep, class Period>
    bool wait_for(bool set, chrono::duration<Rep, Period> const& rel_time,
      memory_order order = memory_order_seq_cst) const;
    template <class Rep, class Period>
    bool wait_for(bool set, chrono::duration<Rep, Period> const& rel_time,
      memory_order order = memory_order_seq_cst) const volatile;

    atomic_flag(__base_t init) noexcept : atom(init) { }
    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;
  } atomic_flag;

  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*,
    memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit_notify(volatile atomic_flag*,
    memory_order, atomic_notify) noexcept;
  bool atomic_flag_test_and_set_explicit_notify(atomic_flag*,
    memory_order, atomic_notify) noexcept;
  void atomic_flag_clear(volatile atomic_flag*) noexcept;
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit_notify(volatile atomic_flag*,
    memory_order, atomic_notify) noexcept;
  void atomic_flag_clear_explicit_notify(atomic_flag*, memory_order,
    atomic_notify) noexcept;
  bool atomic_flag_test(const volatile atomic_flag*) noexcept;
  bool atomic_flag_test(const atomic_flag*) noexcept;
  bool atomic_flag_test_explicit(const volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_explicit(const atomic_flag*, memory_order) noexcept;
  void atomic_flag_wait(const volatile atomic_flag*, bool);
  void atomic_flag_wait(const atomic_flag*, bool);
  void atomic_flag_wait_explicit(const volatile atomic_flag*, bool, memory_order);
  void atomic_flag_wait_explicit(const atomic_flag*, bool, memory_order);

  #define ATOMIC_FLAG_INIT see below
}
```

2 The `atomic_flag` type ~~provides~~ combines the classic test-and-set functionality, with the ability to block until the object is in a specified state. It has two states, set and clear.

3 Operations on an object of type `atomic_flag` shall be lock-free. [*Note*: Hence the operations should also be address-free. No other type requires lock-free operations, so the `atomic_flag` type is the minimum hardware-implemented type needed to conform to this International standard. The remaining types can be emulated with `atomic_flag`, though with less than ideal properties. — *end note*]

4 The atomic_flag type shall have standard layout. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor.

5 The macro `ATOMIC_FLAG_INIT` shall be defined in such a way that it can be used to initialize an object of type atomic_flag to the clear state. The macro can be used in the form:

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

It is unspecified whether the macro can be used in other initialization contexts. For a complete static-duration object, that initialization shall be static. Unless initialized with ATOMIC_FLAG_INIT, it is unspecified whether an atomic_flag object has an initial state of set or clear.

6 The `set_and_set` and `clear` member functions are notifying functions. The `wait`, `wait_for`, and `wait_until` member functions are waiting functions. Executions of waiting functions may block until they are unblocked by a notifying function, according to each function's effects.

7 [*Note:* Programs using `atomic_flag` waiting functions may be susceptible to transient values, an issue known as the ABA problem, resulting in continued blocking if a condition is only temporarily met. – *End Note.*]

```
bool atomic_flag_test_and_set(volatile atomic_flag* object) noexcept;
bool atomic_flag_test_and_set(atomic_flag* object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag* object,
  memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag* object,
  memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit_notify(volatile atomic_flag* object,
  memory_order order, atomic_notify notify) noexcept;
bool atomic_flag_test_and_set_explicit_notify(atomic_flag* object,
  memory_order, atomic_notify notify) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst,
  atomic_notify notify = atomic_notify::all) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst,
  atomic_notify notify = atomic_notify::all) volatile noexcept;
```

8 *Effects*:

1. Atomically sets the value pointed to by `object` or by `this` to true. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).
2. If the value of the object is changed by the effects and `notify` is `atomic_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 the value of the object is changed by the effects and `notify` is `atomic_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.

9 *Returns*: Atomically, the value of the object immediately before the effects.

```
void atomic_flag_clear(volatile atomic_flag* object) noexcept;
void atomic_flag_clear(atomic_flag* object) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag* object,
  memory_order order) noexcept;
void atomic_flag_clear_explicit(atomic_flag* object, memory_order order) noexcept;
void atomic_flag_clear_explicit_notify(volatile atomic_flag* object,
  memory_order order, atomic_notify notify) noexcept;
void atomic_flag_clear_explicit_notify(atomic_flag* object, memory_order order,
  atomic_notify notify) noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst,
  atomic_notify notify = atomic_notify::all) noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst,
  atomic_notify notify = atomic_notify::all) volatile noexcept;
```

10 *Requires*: The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

11 *Effects*:

1. Atomically sets the value pointed to by `object` or by `this` to false. Memory is affected according to the value of `order`.
2. If the value of the object is changed by the effects and `notify` is `atomic_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 the value of the object is changed by the effects and `notify` is `atomic_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 set(bool state, memory_order order = memory_order_seq_cst,
  atomic_notify notify = atomic_notify::all) noexcept;
void set(bool state, memory_order order = memory_order_seq_cst,
  atomic_notify notify = atomic_notify::all) volatile noexcept;
```

12 *Effects*: Equivalent to:

```
if (state)
  test_and_set(order, notify);
else
  clear(order, notify);
```

```
bool atomic_flag_test(const volatile atomic_flag* object) noexcept;
bool atomic_flag_test(const atomic_flag* object) noexcept;
bool atomic_flag_test_explicit(const volatile atomic_flag* object,
  memory_order order) noexcept;
bool atomic_flag_test_explicit(const atomic_flag* object,
  memory_order order) noexcept;
bool atomic_flag::test(memory_order order = memory_order_seq_cst) const noexcept;
bool atomic_flag::test(
  memory_order order = memory_order_seq_cst) const volatile noexcept;
```

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

14 *Effects*: Memory is affected according to the value of `order`.

15 *Returns*: Atomically returns the value pointed to by `object` or by `this`.

```
void atomic_flag_wait(const volatile atomic_flag* object, bool set);
void atomic_flag_wait(const atomic_flag* object, bool set);
void atomic_flag_wait_explicit(const volatile atomic_flag* object,
  bool set, memory_order order);
void atomic_flag_wait_explicit(const atomic_flag* object, bool set,
  memory_order order);
void atomic_flag::wait(bool set,
  memory_order order = memory_order_seq_cst) const noexcept;
void atomic_flag::wait(bool set,
  memory_order order = memory_order_seq_cst) const volatile noexcept;
template <class Clock, class Duration>
bool atomic_flag::wait_until(bool set,
  chrono::time_point<Clock, Duration> const& abs_time,
  memory_order order = memory_order_seq_cst) const;
template <class Clock, class Duration>
```

```
bool atomic_flag::wait_until(bool set,
  chrono::time_point<Clock, Duration> const& abs_time,
  memory_order order = memory_order_seq_cst) const volatile;
```

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

1. Evaluates `test(order) == set` then, if it is satisfied, returns.
2. If `wait_until` was invoked, may return spuriously.
3. Blocks.
4. Unblocks when:
   - As a result of some notifying operations, as described in that function's effects.
   - The absolute timeout expires.
   - At the implementation's discretion.
5. Each time the execution unblocks, it repeats.

17 *Returns*: The result of `test(order) == set`, or false if spuriously.

```
template <class Rep, class Period>
bool wait_for(bool set, chrono::duration<Rep, Period> const& rel_time,
  memory_order order = memory_order_seq_cst) const;
template <class Rep, class Period>
bool wait_for(bool set, chrono::duration<Rep, Period> const& rel_time,
  memory_order order = memory_order_seq_cst) const volatile;
```

18 *Effects:* Equivalent to:

```
wait_for(set,chrono::steady_clock::now() + rel_time, order);
```