Efficient waiting for concurrent programs

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. We propose to create new specialized atomic operations, and thread synchronization object types, that likely replace atomic_flag in practice.

A simple abstraction for scalable 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: std::binary_semaphore.

Objects of class binary_semaphore are easily adapted to serve the role of a mutex:

```cpp
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.

Moving beyond new semaphore types, we propose atomic free functions that enable pre-existing algorithms expressed in terms of atomics to benefit from the same efficient support behind semaphores:

```cpp
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 of unrelated atomic updates. For greater control over this sort of interference, we introduce the final type in this proposal: class condition_variable_atomic.

With this last facility, we can manage false sharing of synchronization state and achieve higher performance:

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;
};

Reference implementation


Please see P0514R0, P0514R1, P0126 and N4195 for additional analysis not repeated here.
C++ Proposed Wording

Apply the following edits to N4687, the working draft of the Standard.

The feature test macro __cpp_lib_semaphore should be added.

Modify 32.2 Header <atomic> synopsis

```cpp
// 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

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.]

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

```cpp
template <class T>
    void atomic_notify_one(const volatile atomic<T>* object);
template <class T>
    void atomic_notify_one(const atomic<T>* object);
```
Effects: unblocks up to one thread 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.

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

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

Effects: unblocks each thread 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.

```cpp
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);
```

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

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]

```cpp
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);
```

Effects: Equivalent to:

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

Modify 33.1 General

Table 140 – Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.2 Requirements</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>33.3 Threads</td>
<td>&lt;mutex&gt; &lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>33.4 Mutual exclusion</td>
<td>&lt;condition_variable&gt;</td>
</tr>
<tr>
<td>33.5 Condition variables</td>
<td>&lt;future&gt;</td>
</tr>
<tr>
<td>33.6 Futures</td>
<td>&lt;semaphore&gt;</td>
</tr>
</tbody>
</table>

Modify 33.5 Condition variables
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 type types. Class `condition_variable_atomic` provides a specialized condition variable that evaluates predicates over a single object of class `atomic<T>`, without using a lock.

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

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

```cpp
namespace std {
    class condition_variable;
    class condition_variable_any;
    class condition_variable_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`

```
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>&, Predicate pred,
memory_order = memory_order_seq_cst);
template <class T, class Predicate>
void wait(const atomic<T>&, 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 Predicate, class Clock, class Duration>
bool wait_until(const volatile atomic<T>&, Predicate pred,
chrono::time_point<Clock, Duration> const&,
memory_order = memory_order_seq_cst);
template <class T, class Predicate, class Clock, class Duration>
bool wait_until(const atomic<T>&, Predicate pred,
chrono::time_point<Clock, Duration> const&,
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 Predicate, class Rep, class Period>
bool wait_for(const volatile atomic<T>&, Predicate pred,
chrono::duration<Rep, Period> const&,
memory_order = memory_order_seq_cst);
template <class T, class Predicate, class Rep, class Period>
bool wait_for(const atomic<T>&, Predicate pred,
chrono::duration<Rep, Period> const&,
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: For every function call that blocks on *this, 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 notify_one(const volatile atomic<T>& object) noexcept;
void notify_one(const atomic<T>& object) noexcept;

Effects: If any threads are blocked on *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;

Effects: Unblocks all threads that are blocked on *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, Predicate pred,
      memory_order order = memory_order_seq_cst);
template <class T, class Predicate>
void wait(const atomic<T>& object, Predicate pred,
      memory_order order = memory_order_seq_cst);

Effects: Repeatedly performs the following steps, in order:
   a) For the overloads that take Predicate, evaluate pred(object.load(order)), and for the other, evaluate object.load(order) != old. If the result is true, returns.
   b) Blocks on *this and object 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, 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 Predicate, class Clock, class Duration>
bool wait_until(const volatile atomic<T>& object, Predicate pred,
      chrono::time_point<Clock, Duration> const& abs_time,
      memory_order order = memory_order_seq_cst);
template <class T, class Predicate, class Clock, class Duration>
bool wait_until(const atomic<T>& object, Predicate pred,
      chrono::time_point<Clock, Duration> const& abs_time,
      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 Predicate, class Rep, class Period>
bool wait_for(const volatile atomic<T>& object, Predicate pred,
      chrono::duration<Rep, Period> const& rel_time,
      memory_order order = memory_order_seq_cst);
memory_order order = memory_order_seq_cst);  

template <class T, class Predicate, class Rep, class Period>
bool wait_for(const atomic<T>& object, Predicate pred,  
chrono::duration<Rep, Period> const& rel_time,  
memory_order order = memory_order_seq_cst);  

Effects: Repeatedly performs the following steps, in order:
  a) For the overloads that take Predicate, evaluate pred(object.load(order)), and for the other, evaluate object.load(order) != old. If the result is true, or with low probability if the result is false, returns the result.
  b) Blocks on *this and object until the timeout expires or an implementation-defined condition has been met. If the timeout expired, returns false. [Note: consequently, it may unblock for reasons other than a call to a notifying function. - end note]

Throws: Timeout-related exceptions (33.2.4).

Add 33.7 Semaphores

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, may be more efficient than condition variables. Class counting_semaphore models a non-negative resource count. Class binary_semaphore has only two states, also known as available and unavailable, and may be even more efficient than class counting_semaphore.

For purposes of determining the existence of a data race, all member functions of binary_semaphore and counting_semaphore (other than construction and destruction) behave as atomic operations on *this.

Add 33.7.1 Header <semaphore> synopsis

namespace std {
  class binary_semaphore;
  class counting_semaphore;
}

Add 33.7.2 Class binary_semaphore

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();
    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:
    count_type counter; // exposition only
  };
}
using count_type = implementation-defined;

1   An integral type able to represent any value of type int between zero and max, inclusive.

static constexpr count_type max = 1;

2   The maximum value that the semaphore can hold.

constexpr binary_semaphore(count_type desired = 0);

3   Requires: desired is not negative, and no greater than max.
4   Effects: Initializes counter with the value desired.

~binary_semaphore();

5   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 ]
6   Effects: Destroys the object.

void release();

7   Requires: counter < max.
8   Effects: Atomically increments counter by 1 then, if any threads are blocked on counter, unblocks at least one among them.
9   Synchronization: Synchronizes with invocations of try_acquire() that observe the result of the effects.

bool try_acquire();

10  Effects: Atomically, subtracts 1 from counter then, if the result is positive or zero, updates counter 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: 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);
a) Evaluates `try_acquire()`. If the result is `true`, returns `true`.

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

Throws: Timeout-related exceptions (33.2.4).

Add 33.7.3 Class `counting_semaphore` [semaphore.counting]:

```cpp
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&);

        private:
            count_type counter; // exposition only
        };
    }
}
```

An integral type able to represent any value of type `int` between zero and `max`, inclusive.

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

The maximum value that the semaphore can hold. [Note: `max` should be at least as large as the maximum number of threads the implementation can support. - end note]

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

**Requires:** `desired` is not negative, and no greater than `max`.

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

```cpp
~counting_semaphore();
```

**Requires:** For every function call that blocks on `*this`, 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 ]

```cpp
Effects: Destroys the object.
```

```cpp
void release(count_type update = 1);
```
**Requires:** update > 0, and counter + update <= max.

**Effects:** Atomically increments the counter by update. If any threads are blocked on counter, unblocks at least update among them.

**Synchronization:** Synchronizes with invocations of try_acquire() that observe the result of the effects.

```cpp
bool try_acquire();
```

**Effects:** Atomically, decrements counter by 1 then, if the result is positive or zero, updates counter with the result. An implementation may spuriously fail to replace the value if there are contending invocations in other threads.

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

```cpp
void acquire();
```

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

- c) Evaluates try_acquire(). If the result is true, returns.
- d) Blocks until counter >= 1.

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

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

**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.

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