Doc number: **P0126R0**
Revises: **N4195**
Date: 2015-09-24
Project: Programming Language C++, Concurrency Working Group
Reply-to: Olivier Giroux <ogiroux@nvidia.com>
Samples: https://github.com/ogiroux/synchronic.

std::synchronic<T>

Atomic objects make it easy to write custom synchronization primitives with efficiency problems. One particularly thorny issue is the poor performance of the overall system when it is oversubscribed and/or contention is high. Another is the high power consumed under contention, even when there is no oversubscription of the system.

The central problem is the utilization of system resources by synchronizing threads, which is a balance between two conflicting realities that are equally important:

- The focus of modern platform architecture is on lowering total energy use, and to do so for platform APIs originally designed for uniprocessor multi-tasking.
- The focus of performance-critical software is on minimizing latency on the critical-path, especially on now-ubiquitous multiprocessor systems.

There is yet no standard vocabulary to bridge these two realities. In spite of this, there is at least some native (different) support for efficient polling on every major platform, both software and hardware. We now propose “synchronic” variables, a portable library abstraction to balance low-latency and low-energy synchronization.

Good Spin Loops are Difficult to Write

A bad spin loop is easy to write and performs well in a reasonably wide regime envelope – two facts that conspire to make them ubiquitous in practice. What makes a bad spin loop ‘bad’ is that it does not behave gracefully in the face of long delays (causing high energy use) or under high degrees of contention (impeding progress on the critical path). The polar opposite, a condition variable, behaves gracefully in all regimes but incur a high minimum cost in comparison.

Simple synchronization algorithms follow one of these two curves with wide areas of resource waste between them:

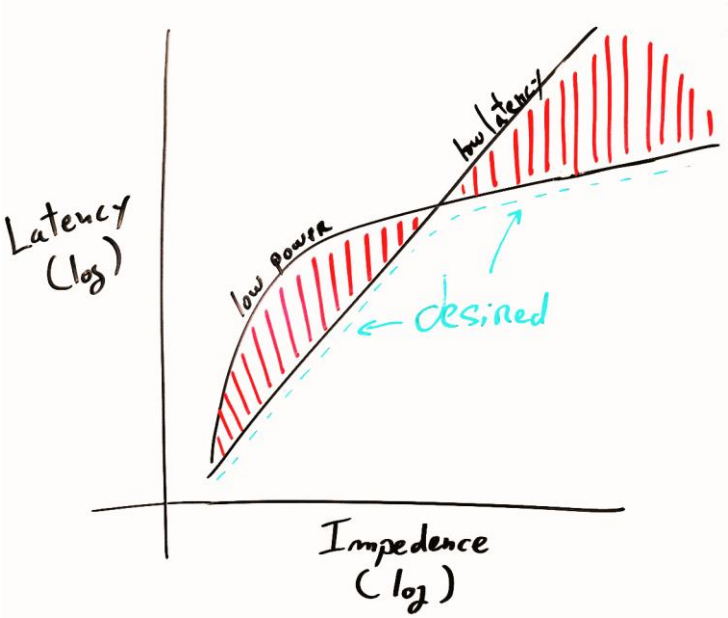

Figure 1. Synchronization algorithms have exponential overheads under contention but bad spin loops stand out as being exceptionally exponential.

The following example outlines an implementation of the mutex concept dubbed Test-and-Test-And-Set (TTAS), ubiquitously found on Internet forums and in production:

```
struct ttas_mutex {
    ttas_mutex() : locked(false) { }
    void lock() {
        while(1) {
            bool state = false;
            if(locked.compare_exchange_weak(state, true,
                                            memory_order_acquire))
                break;
            while(locked.load(memory_order_relaxed)==state)
                ; //see below why this is emphasized
        }
    }
    void unlock() {
        locked.store(false, memory_order_release);
    }
    atomic<bool> locked;
};
```

This algorithm's nested loop reduces cache thrashing over an implementation without the loop, which improves its performance under contention. However, it fails to improve either the power consumption of the algorithm under delay or the performance when the system is oversubscribed. The reason for these issues is that the threads that issue the highlighted operations appear to make useful progress to the system's arbiters.

To achieve the desired operational profile, this example must be modified with a hybrid of these techniques: unmitigated spinning (shown), hardware thread yielding, randomized timed exponential back-off, and invocations of platform API's like `SYS_futex` for minimum-energy

waiting. Implementations of `std::mutex` tend to do this correctly, but the technique is beyond most users.

While we agree that users don't need to re-implement `std::mutex`, they need to be able to implement their own (customized) primitives with efficiency comparable to `std::mutex`.

Condition Variables Can Be Improved Upon

When you want to block until something happens, then using condition variables seem like the primitive of choice. Unfortunately, that is usually a poor way to get the job done because, on average, condition variables incur one order-of-magnitude greater overhead than is achievable with hybrid methods. Even, those layered on condition variables themselves.

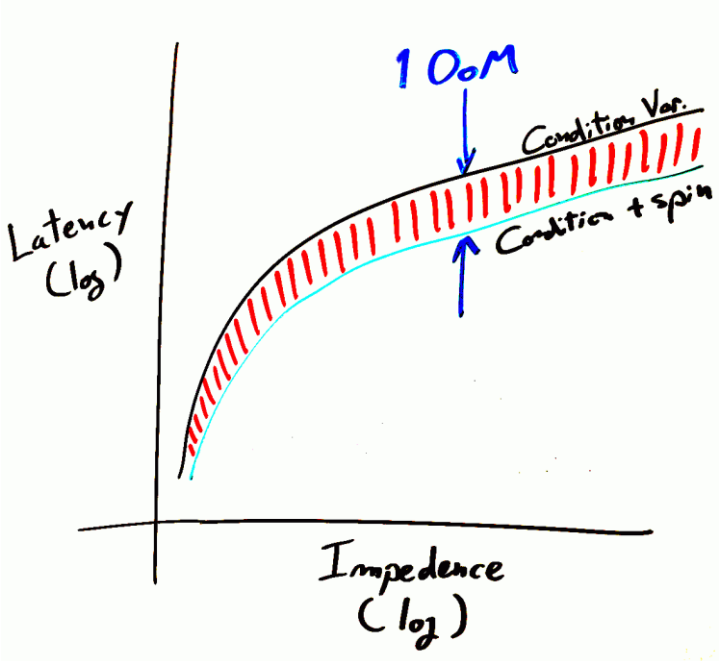

Figure 2. For the same job, it's entirely feasible to outperform platform condition variables by 2-10x.

The reasons for the poor performance are two-fold:

1. Condition variable APIs require mutual-exclusion of execution, on the assumption that the condition predicate is ***not*** applied to an atomic object.
2. Condition variables tend to be implemented with system calls, partly because of that mutual-exclusion requirement and partly because of implementation folklore.

As a result, it takes significant effort and skill to leverage condition variables in a high-performance contet. In a reference implementation for this proposal, synchronic objects, the most complicated underlying algorithm is the one that is based on condition variables. This particular algorithm is a parade of anti-patterns: seq-locks, store-acquire and empty (but vital) critical sections that would upset anyone's intuition.

This is substantially the same wisdom captured in the saying "Futexes Are Tricky."

A simplifying abstraction

We believe that synchronic objects make it easier to implement scalable and power-efficient synchronization on top of C++ atomic variables. The simplest way to use synchronic objects is to express a synchronization value that is expected for continued progress, and notify when this value occurs.

The interface for this purpose can be summarized as follows:

```
template <class T>
struct synchronic {
     ...
    void notify(std::atomic<T>& atom, T val) noexcept;
    void expect(std::atomic<T> const& atom, T val) const;
};
```

Where it is specified that:

- `notify` writes `val` into `atom` with `memory_order_release`.
- `expect` blocks until a read of `atom` with `memory_order_acquire` returns `val`. If `expect` blocks, it may avoid reading `atom` until the next invocation of `notify` by another thread associated with the same object `atom`.

To illustrate how this is used, we show a partial implementation of the `std::latch` object in the concurrency TS (http://wg21.link/n4538):

```
class latch {
  ...
  void count_down_and_wait() {
    if(count.fetch_add(-1,std::memory_order_acq_rel)!=1)
      while(!released.load(std::memory_order_acquire));
      sync.expect(released, true);
    else
      released.store(true, std::memory_order_release);
      sync.notify(released, true);
  }
  ...
  atomic<int> count;
  atomic<bool> released;
  synchronic<bool> sync;
};
```

Synchronic objects **[thread.synchronic]**

Synchronic objects provide efficient waiting operations for synchronization over simple atomic objects. A synchronic object encapsulates an algorithm for waiting and its associated acceleration data structures.

Synchronic objects permit concurrent invocations of the `notify`, `expect` and `expect_update` member functions.

Invocations of `expect` functions may block indefinitely unless a `notify` function is invoked by another thread for the object `object`. Conversely, invocations of `expect` functions that return do not guarantee that a `notify` function has been invoked by any other thread because implementations may return whenever the condition is true.

Invocations of the `expect` functions that are unblocked by the invocation of a `notify` function may re-evaluate the user-provided predicate and block again. If the value of the synchronic object is transient, threads may only unpredictably unblock.

> [*Note:* synchronic objects functions are susceptible to issues with transient values, also known as the ABA problem, resulting in continued blocking of threads that could potentially be unblocked. Users of synchronic objects should ensure that either transient values do not occur or that the program does not depend on threads unblocking when transient values occur. – *End Note.*]

Synchronic object construction and destruction need not be synchronized.

Header `synchronic` synopsis

```
namespace std {
  template <class T> struct synchronic;

  enum notify_hint { notify_all, notify_one };
  enum expect_hint { expect_urgent, expect_delay };
}
```

Class `synchronic`

```
namespace std {
  template <class T>
  class synchronic {
  public:
    synchronic();
    ~synchronic();
    synchronic(const synchronic&) = delete;
    synchronic& operator=(const synchronic&) = delete;
    synchronic& operator=(const synchronic&) volatile = delete;

    void notify(A& object, T value,
                notify_hint hint = notify_all) noexcept;
    void expect(A const& object, T desired,
                expect_hint hint = expect_urgent) const noexcept;

    void notify(A& object, F&& func, notify_hint hint = notify_all);
```

```
    void expect(A const& object, F&& func,
                expect_hint hint = expect_urgent) const;

    void expect_update(A const& object, T current,
                       expect_hint hint = expect_urgent) const noexcept;
    void expect_update_for(A const & object, T current,
                       chrono::duration<Rep, Period> const& rel_time,
                       expect_hint hint = expect_urgent) const;
    void expect_update_until(A const& object, T current,
                       chrono::time_point<Clock,Duration> const& abs_time,
                       expect_hint hint = expect_urgent) const;

  }
}
```

In the following operation definitions:

- an *A* refers to the corresponding atomic type.
- an *F* refers to a callable type.

`synchronic();`

Effects: Constructs an object of type `synchronic<T>`.

Throws: `system_error` when an exception is required (30.2.2).

Error conditions:

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

`~synchronic();`

Requires: There shall be no threads blocked on `*this`. [*Note*: `synchronic` objects can be destroyed if all threads blocked on an object have been notified. – *end note*]

Effects: Destroys the object.

```
void notify(A& object, T value,
    notify_hint hint = notify_all) noexcept;
void notify(A& object, F&& func,
    notify_hint hint = notify_all);
```

Requires: the object `func` is callable with the signature `void()`.

Effects:

- invokes `func` or `object.store(value, memory_order_release)`.
- if `hint` is `notify_one` and any threads are blocked waiting for `object`, unblocks one of those threads.
- if `hint` is `notify_all`, unblocks all threads that are blocked waiting for `object`.

Synchronization: each invocation of `notify` *synchronizes-with* invocations of `expect` that unblock as a result.

```
void expect(A const& object, T desired,
    expect_hint hint = expect_urgent) const noexcept;
void expect(A const& object, F&& pred,
    expect_hint hint = expect_urgent) const;
void expect_update(A const& object, T current,
    expect_hint hint = expect_urgent) const noexcept;
void expect_update_for(A const & object, T current,
    chrono::duration<Rep, Period> const& abs_time,
    expect_hint hint = expect_urgent) const;
void expect_update_until(A const& object, T current,
    chrono::time_point<Clock,Duration> const& rel_time,
    expect_hint hint = expect_urgent) const;
```

Requires: the object `pred` is callable with the signature `bool()`.

Effects:

- invokes `pred` or `object.load(memory_order_acquire)`.
- blocks the thread until the predicate `pred` is true, or the value of `object` is either equal to `desired` or not equal to `current`, or the absolute time-out specified by `abs_time` expires, or the relative time-out specified by `rel_time` expires, or spuriously if a timed function is used.

Remarks: the value of `hint` has no effect.

Throws: Timeout-related exceptions (30.2.4).