Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0660R3**
Date: 2018-06-06
Reply to: Nicolai Josuttis (nico@josuttis.de), Herb Sutter (hsutter@microsoft.com),
Anthony Williams (anthony@justsoftwaresolutions.co.uk)
Audience: SG1, LEWG, LWG
Prev. Version: P0660R2

A Cooperatively Interruptible Joining Thread, Rev 3

History

New in Rev 3

- Interrupt API no longer for `std::thread`, which means no ABI change of `std::thread`
  - Note that due to new TLS data to signal an interrupt, *all* started threads can use interrupts (and have extended footprints)
- Rename `wait_interruptable()`/`iwait()` to `wait_or_throw()`
- Rename `ready()` to `valid()`
- Interrupt tokens don't require to be `ready()`/`valid()` on calls like `is_interrupted()`
- Clean-up API as suggested in SG1 discussions in Rapperswil 2018

New in Rev 2

- Integrate all decisions/pools and proposals from SG1 discussion in Albuquerque 2018
- Interrupt API also for `std::thread`
- Rename `ithread` to `jthread` (only adding the auto join in the destructor)
- Keep interrupt API simple:
  - Monotonic (to reset the token you can use `exchange_interrupt_token()`)
- Different CV interface:
  - Add a `cv_status interrupted`
  - Add `wait_interruptable()` and `wait_until()` overloads

New in Rev 1

- Updated terminology
- API clean-ups
- Synopsis of the proposed wording
- A proposed API for waiting condition variables

Motivation

For C++17, in Jacksonville 2016 we had an evening session
http://wiki.edg.com/bin/view/Wg21jacksonville/P0206R0
with a very clear directive:

> Add an auto-joining (in destructor) thread type under the understanding that its name will be changed and there will be LEWG review.
> SF F N A SA
> 10 11 1 2 0
>
> Include it in C++17
> SF F N A SA
> 9 5 8 2 0

This clear directive was broken.

Even worse, there is still no proposal to go the path strongly requests here.

And it seems we see more and more the consequences of breaking our own intent: Several guidelines recommend not to use std::thread for this reason, others teach to use std::thread with care or just complain about the bad API. For example:

- High Integrity C++ spec by Programming Research recommends:

  > *18.2.1 Use high_integrity::thread in place of std::thread*
  >
  > *The destructor of std::thread will call std::terminate if the thread owned by the class is still joinable. By using a wrapper class a default behavior can be provided.*

  … followed by full source code for the wrapper.

- Similarly, the C++ Core Guidelines also recommends:

  > *CP.25: Prefer gsl::joining_thread over std::thread*
  >
  > *Reason A joining_thread is a thread that joins at the end of its scope.*

  … and provide a full implementation.

This is the community voting with their feet, almost to the point of rebellion.
It should be a big red flag that the major style guides are consistently repeating this advice and (this part is very rare) multiple such guides going to the trouble of providing actual code to use instead. At the very least, it points out that the status quo is leading to divergence (the above are two different wrapper types).

For this reason, this paper proposes a standard thread class that "does the expected thing":

- Proposing a thread class that uses RAII-style join() if still joinable at destruction time.
- Adding an API to cooperatively signal thread interruption so that a join() in the destructor might not wait forever.

For a full discussion of the motivation see P0660R0 and P0660R1.

The important thing is that this is a long requested feature. The reason that many projects can't switch from boost::thread to standard threads and that several guidelines warn against the direct usage of std::thread.

Key Design Decisions

We need a new thread class, because the change to join in the destructor is not compatible with the existing behavior. Also adding just a template argument with a default value breaks binary compatibility.

However, note that the vote in SG1 discussing this topic in Albuquerque 2018 was to provide the new interrupt mechanism also for std::thread, which however causes ABI breaks on std::thread objects. Nevertheless, the interrupt mechanism proposed here can be used in all started threads.

We roughly do the following:

- Introduce a type `std::interrupt_token` to signal interrupts
- Provide a new thread class `std::jthread`, with the same API as `std::thread` with the following differences:
  - Add an API to signal interrupts and use it in a RAII-style:
    - Constructor for a passed task/callback creates an interrupt token that can be used with the jthread object to dignal interrupts and is copied to the started thread to deal with interrupt signals.
    - Silently catches a std::interrupted exception of the started thread.
    - Destructor calls interrupt() and join() if still joinable. Same applies to the jthread object if a new thread is move assigned.
- Extend std::this_thread to check for interrupts (which is also then possible to be used for any thread started with std::thread or async()).
- Extend the `std::condition_variable` API to deal with interrupts.

There is no functional and ABI change of std::thread objects and the existing CV interface.

Class std::interrupt_token

A simple helper class `std::interrupt_token` is introduced, which

- can signal an interrupt (once)
- allows to check for an interrupt
- is cheap to copy

The interrupt mechanism is initializing by passing an initial interrupt state (usually by initializing with `false` so that no interrupt is signaled yet; but `true` is also possible):

```
std::interrupt_token it{false};
```

A default constructor is also provided, which does not initializing the interrupt mechanism to make default initialized interrupt tokens cheap:

```
std::interrupt_token it;     // cheap, but interrupt API disabled
```

You can check whether the interrupt API can be used:

- bool **valid()**
  - signals whether the interrupt mechanism was enabled (interrupt token initialized with a Boolean value)

With `valid()==true` you can call the following member functions to signal and check for interrupts (otherwise we get undefined behavior):

- bool **interrupt()**
  - signals an interrupt (and returns whether an interrupt was signaled before)
- bool **is_interrupted()**
  - yields whether an interrupt was signaled yet
- void **throw_if_interrupted()**
  - throws an `std::interrupted` exception if an interrupt was signaled (yet).

If the interrupt token is not valid

- `interrupt()` is UB (undefined behavior)
- `is_interrupted()` yields false
- `throw_if_interrupted()` doesn't throw

In addition, `operator==` and `operator!=` can be used to check whether two interrupt_token objects refer to the same signaled interrupt. This operation e.g. helpful to check whether an interrupt token was replaced.

All functions are thread-safe in the strong sense: They can be called concurrently without introducing data races and will behave as if they executed in a single total order consistent with the SC order. (Thus, internally we use atomic flags and `atomic_exchange()` or `atomic_load()`).

Implementation Hints

An easy way to implement `std::interrupt_token` is to make it a wrapped `shared_ptr<atomic<bool>>` (see below).

With such an implementation a token is pretty cheap to copy (just increments the reference count). If this is not good enough, you can pass it by reference (without any danger provided it is on the stack).

Class std::interrupted

For interrupt tokens, the `throw_if_interrupted()` function throw a new standard exception class **`std::interrupted`**, if interruption was signaled.

Usually, this exception will be automatically be caught by the thread, where the check was performed, which usually is the thread that checks whether it was interrupted. So the exception only has one task: end the current thread without any warning or error (if not caught elsewhere).

Note: Class `std::interrupted` is intentionally *not* derived from `std::exception`.
The reason:

- This exception semantically represents “terminate the thread” instead of an error.
- This exception should not pollute general existing handling of `std::exception`. Existing (third-party) code called while an interrupt is handled should simply not handle this exception. If code has to react on exceptions in general, it anyway already should have a `catch(…)` clause (e.g. to clean-up and rethrow).
- The exception will by default not terminate the program because it only ends the current thread, which should be a thread interrupted by another thread.
- The exception does not break backward compatibility because it will not occur unless somebody explicitly added code to interrupt a running thread.

Note: we already have `std::nested_exception` outside the std::exception tree.

Namespace std::this_thread

The API for **`std::this_thread`** is in general **extended** to be able to cooperate with and check for requested thread interrupts:

```
namespace this_thread {
  // basic interrupt handling:
  static interrupt_token get_interrupt_token() noexcept;
  static bool is_interrupted() noexcept {
    return get_interrupt_token().is_interrupted();
  }
  static void throw_if_interrupted() {
    get_interrupt_token().throw_if_interrupted();
  }

  // (temporarily) replace old/original interrupt token:
  static interrupt_token
           exchange_interrupt_token(const interrupt_token&) noexcept;
}
```

`std::exchange_interrupt_token()` is provided to be able to (temporarily) replace the original token by another interrupt token (e.g. to temporarily disable handling interrupt signals).

Note that this API is also available for threads started as `std::thread` or with `std::async()`.
In that case, by default:

- `get_interrupt_token()` returns a default constructor token, so that
- `is_interrupted()` always yields false
- `throw_if_interrupted()` has no effect

However, if the interrupt token was replaced, `get_interrupt_token()` yields the passed token, which might return `true` on `is_interrupted()` and might throw on `throw_if_interrupted()`.

Note that replacing the token instead of un-interrupting the token avoids lifetime and ABA problems.

Class `std::jthread`

We propose a new class **`std::jthread`**:

- It is fully compatible to `std::thread` with its whole API (including all static functions) with the following modifications:
  - Constructor for a passed task/callback creates an interrupt token that is copied to the started thread
  - Silently catches a `std::interrupted` exception of the started thread.
  - Destructor calls `interrupt()` and `join()` if still joinable. Same applies to the thread if a new thread is move assigned.
- It also provides the supplementary API for cooperative interrupts:

```
class jthread
{
  public:
    …
    // supplementary interrupt API:
    interrupt_token get_original_interrupt_token() const noexcept;
    bool interrupt() noexcept {
      return get_original_interrupt_token().interrupt();
    }
    bool is_interrupted() const noexcept {
      return get_original_interrupt_token().is_interrupted();
    }
};
```

Note that `get_original_interrupt_token()` can even be called for a non-joinable jthread. That means that interrupts can even be signaled after `detach()` was called:

```
std::jthread t1([]{…});
…
t1.detach();
…
t1.interrupt();    // signals interrupt on detached thread
```

Calling `get_interrupt_token()` for a default constructed jthread results in undefined behavior.

The whole `std::jthread` API is in principle implementable on top of the existing standard concurrency library. However, with OS support better performance is possible.

An example implementation is available at: www.josuttis.de/jthread.

How To Use Interrupt Tokens in Threads

The basic interface of `std::jthread` supports the following examples:

```
std::jthread t([] {
                   while (!std::this_thread::is_interrupted()) {
                      //...
                   }
                });
```

Or:

```
std::jthread t([] {
                   while (…)
                      // …
                      std::this_thread::throw_if_interrupted();
                      //...
                   }
                });

// optionally (if not called, called by the destructor):
t.interrupt();
t.join();
```

Without calling `interrupt()` and `join()` (i.e. if t is still joinable and the destructor of t is called), the destructor of `std::jthread` itself calls `interrupt()` and then `join()`. Thus, the destructor waits for a cooperative end of the started thread.

The same happens if a new `jthread` is move assigned and the existing objects is still joinable.

Note that the mechanism does never cancel the thread directly or calls a cancelling low-level thread function.

If `interrupt()` is called, the next check for an interrupt by the started thread with

```
std::this_thread::is_interrupted()
```

yields true. Alternatively, a checkpoint such as

```
std::this_thread::throw_if_interrupted()
```

throws `std::interrupted`. If the exception is not caught inside the called thread, it ends the started thread silently without calling `terminate()` (any other uncaught exception inside the called thread still results into `terminate()`).

Instead of calling `t.interrupt()`, you can also call:

```
auto it = t.get_original_interrupt_token();
…
it.interrupt();
```

to cheaply pass a token to other places that might interrupt. The tokens are neither bound to the lifetime of the interrupting or the interrupted thread.

Also `std::this_thread::get_interrupt_token()` yields an interrupt token in the started thread which you can also use to check for interrupts.

The implicitly created interrupt token for `std::jthread` can only be used once to signal interruption. This design was chosen because it's simple and avoids lifetime and ABA issues.

You can replace the interrupt token **in the called thread** with `exchange_interrupt_token()` for the following reasons:

- Temporarily disable signaling interruptions
- Provide other tokens to signal interrupt from somewhere else

Note that the original `std::jthread` object will still only signal interrupts to the original interrupt token of the called thread. So to deal with signaled interrupts, the called thread has to checks against its original interrupt token.

For example:

```
std::jthread t1([]{
                    …
                    while (!std::this_thread::is_interrupted) {
                      …
                      // temporarily disable “standard” interrupt handling using this_thread:
                      auto origIT =
                        std::this_thread::exchange_interrupt_token(
                               std::interrupt_token{});
                      …
                      // check for signaled interrupt:
                      origIT.thrwo_if_interrupted();
                      …
                      // restore original interrupt handling for this_thread:
                      std::this_thread::exchange_interrupt_token(origIT);
                      …
                     }
                     …
                  }
               });
```

How Threads are Initialized Using Interrupt Tokens

A basic bootstrap of the interrupt objects would be:

```
std::interrupt_token interruptor{false};
std::interrupt_token interruptee(interruptor};
```

Note that by passing false the interrupt mechanism is initialized (the default constructor will not initialize the interrupt mechanism, which is significantly cheaper if the mechanism is not needed).
To initialize the mechanism later you can assign another interrupt token:

```
std::interrupt_token interruptor;
interruptor = std::interrupt_token{false};
```

Then, you can signal an interrupt as follows:

```
interruptor.interrupt();
…
```

While a started thread, which usually gets the token as interruptee, from time to time can check for interrupts:

```
interruptee.throw_if_interrupted();
// and/or:
if (interuptee.is_interrupted()) …
```

Note that the API does not distinguish between interruptor and interruptee. It's only the programming logic that does.

Class `std::jthread` would use interrupt tokens internally this way. Thus, the constructor of a thread would perform the necessary bootstrap to create the API for the calling thread (being the interrupter) and the started thread (being the interruptee).

In principle the started thread would get the interrupt token as part of the TLS (it is possible to pass it as static thread_local data member in class jthread, though). The rough implementation idea is as follows:

```
class jthread {
  …
  private:
    //*** API for the starting thread:
    interrupt_token _thread_it{};                // interrupt token for started thread
    ::std::thread _thread{::std::thread{}};      // started thread (if any)

    //*** API for the started thread (simulated TLS stuff):
    inline static thread_local
      interrupt_token _this_thread_it{};         // interrupt token for this thread
};

// THE constructor that starts the thread:
template <typename Callable, typename... Args>
jthread::jthread(Callable&& cb, Args&&... args)
 : _thread_it{false},                            // initialize interrupt token
   _thread{[](interrupt_token it, auto&& cb, auto&&... args) {   // called thread
                // pass the interrupt_token to the started thread
                _this_thread_it = std:move(_thread_it);
                …    // invoke the callback with the passed args
           }
           _thread_it,    // not captured due to possible races if immediately set
           ::std::forward<Callable>(cb),   // pass callable
           ::std::forward<Args>(args)...   // pass arguments for callable
         }
{
}
```

Convenient Interruption Points for Blocking Calls

So far, this API allows to provide the interrupt mechanism as safe inter-thread communication.

Another question is whether and where to give the ability that the started thread automatically checks for interrupts while it is blocking/waiting.

For a full discussion of the motivation of using interrupts in blocking/waiting functions, see P0660R0.

In Toronto in 2017, SG1 voted to have some support for it:

> Must include some blocking function support in v1.

SF	F	N	A	SA
3	6	6	3	0

While there are simple workarounds in several cases (timed waits), at least support for condition variables seems to be critical because their intent is not to waste CPU time for polling and an implementations needs OS support.

Note that we do not want to change the existing API of waiting/blocking functions (including exceptions that can be thrown). Instead, we have to extend the existing API's by new overloads and or classes.

First, `cv_status` should get a new value `interrupted` to be able to distinguish it from.

Then, we propose according to the in SG1 discussion in Albuquerque 2018 adding `wait_interruptable()` and some additional `wait_until()` overloads that allow to also pass an interrupt token.

The usage would for example be:

```
bool ready = false;
std::mutex readyMutex;
std::condition_variable readyCV;

std::jthread t1([&ready, &readyMutex, &readyCV]{
                  std::unique_lock<std::mutex> lg(readyMutex);
                  while (!ready) {
                   auto status =
                   readyCV.wait_until(lg,
                                      std::this_thread::get_interrupt_token());
                   if (status == std::cv_status::interrupted) {
                     return;   // cancel the thread due to interrupt
                   }
                  }
                  …
                 });
…
t1.interrupt();   // signals interrupt, which terminates wait_or_throw() and the thread
```

Or just:

```
{
  bool ready = false;
  std::mutex readyMutex;
  std::condition_variable readyCV;

  std::jthread t1([&] {
                      std::unique_lock<std::mutex> lg(readyMutex);
                      readyCV.wait_or_throw(lg, [&] { return ready; });
                      …
                  });
  …
}   // destructor of t1 signals interrupt, which terminates wait_or_throw() and the thread
```

Alternatively, you can check the return value of this new overload of `wait_until()` against `std::cv_status::interrupted`.

We propose:

- New overloads of wait_until() for all existing wait() and wait_until() functions, which
  - have an additional interrupt_token parameter at the end and
  - unblock on a signaled interrupt and
  - if no predicate is passed always return cv_status, which might have the new value std::cv_status::interrupted.
- New overloads of wait_for() for all existing wait_for() functions, which
  - have an additional interrupt_token parameter at the end and
  - unblock on a signaled interrupt and
  - if no predicate is passed always return cv_status, which might have the new value std::cv_status::interrupted.
- Convenience functions wait_or_throw() that throw std::interrupted on signaled interrupts.

See the new signatures as part of the proposed wording.

API of std::jthread

Basically, an std::jthread should provide the same interface as std::thread plus the supplementary interrupt API:

```
class jthread
{
  public:
    // - cover full API of std::thread to be able to switch from std::thread to std::jthread:

    // note: use std::thread types:
    using id = ::std::thread::id;
    using native_handle_type = ::std::thread::native_handle_type;

    // construct/copy/destroy:
    jthread() noexcept;
    // THE constructor that starts the thread:
    // - NOTE: should SFINAE out copy constructor semantics
    template <typename Callable, typename... Args,
              typename = enable_if_t<!is_same_v<decay_t<Callable>, jthread>>>
    explicit jthread(Callable&& cb, Args&&... args);
    ~jthread();

    jthread(const jthread&) = delete;
    jthread(jthread&&) noexcept = default;
    jthread& operator=(const jthread&) = delete;
    jthread& operator=(jthread&&) noexcept = default;

    // members:
    void swap(jthread&) noexcept;
    bool joinable() const noexcept;
    void join();
    void detach();

    id get_id() const noexcept;
    native_handle_type native_handle();

    // static members:
    static unsigned hardware_concurrency() noexcept {
      return ::std::thread::hardware_concurrency();
    };

    // supplementary API:
    interrupt_token get_original_interrupt_token() const noexcept;
    bool interrupt() noexcept {
      return get_original_interrupt_token().interrupt();
    }
};
```

Note that `native_handle()` and `get_id()` return `std::thread` types.

We could also add `is_interrupted()` and `throw_if_interrupted()` here for the calling thread, but `interrupt()` is the only “native” interface, which we want to support directly. For anything else you can use `get_original_interrupt_token()`.

We could also provide a **`get_thread()`** helper, which (a bit dangerous) would return a reference to the wrapped `std::thread`.

Interrupt Handling API

The basic interrupt handling API, first defines the type for interrupt exceptions:

```
class interrupted final
{
  public:
    explicit interrupted();
    const char* what() const noexcept;
};
```

Note that the class is final because we see no sense to derive from it.

An example implementation of interrupt_token might look as follows:

```
class interrupt_token {
 private:
  std::shared_ptr<std::atomic<bool>> _ip{nullptr};
 public:
  // default constructor is cheap:
  explicit interrupt_token() = default;
  // enable interrupt mechanisms by passing a bool (usually false):
  explicit interrupt_token(bool b)
   : _ip{new std::atomic<bool>{b}} {
  }
  bool valid() const {
    return _ip != nullptr;
  }
  // interrupt handling:
  bool interrupt() noexcept {
    return valid() && _ip->exchange(true);
  }
  bool is_interrupted() const noexcept {
    return valid() && _ip->load();
  }
  void throw_if_interrupted() {
    if (valid() && _ip->load()) {
      throw ::std::interrupted();
    }
  }
  // two interrupt tokens are equal if their refer to the same interrupt signal:
  friend bool operator== (const interrupt_token& lhs,
                          const interrupt_token& rhs) {
    return lhs._ip == rhs._ip;
  }
  friend bool operator!= (const interrupt_token& lhs,
                          const interrupt_token& rhs) {
    return !(lhs==rhs);
  }
};
```

API for std::this_thread

```
namespace std {
 namespace this_thread {
  static bool is_interrupted() noexcept;
  static void throw_if_interrupted();
  static interrupt_token get_interrupt_token() noexcept;
  static interrupt_token
           exchange_interrupt_token(const interrupt_token&) noexcept;
 }
}
```

Proposed Wording

(All against N4660)

Full proposed wording at work

Add as a new chapter:

```
namespace std {
      class interrupted final
      {
        public:
          explicit interrupted();
          const char* what() const noexcept;
      };
}
```

Add as a new chapter:

```
namespace std {
 class interrupt_token {
  public:
   explicit interrupt_token();       // cheap non-initialization
   explicit interrupt_token(bool);   // initialization of interrupt mechanism
   interrupt_token(interrupt_token&&) noexcept;
   interrupt_token& operator=(interrupt_token&&) noexcept;

   bool valid() const;
   bool interrupt() noexcept;
   bool is_interrupted() const noexcept;
   void throw_if_interrupted();
 };

 bool operator== (const interrupt_token& rhs, const interrupt_token& lhs);
 bool operator!= (const interrupt_token& rhs, const interrupt_token& lhs);
}
```

Formulate guarantees regarding happens before relations (especially on interrupt() and is_interrupted() if the token is valid).

Regarding the move operations the token is moved:

`interrupt_token(iinterrupt_token&& t) noexcept;`

Effects: Move constructs a interrupt_token instance from t.
Postconditions: *this shall contain the old value of t. t shall be not interruptible. t.valid() == false.

`interrupt_token& operator=(interrupt_token&& r) noexcept;`

Effects: Equivalent to `interrupt_token{std::move(t)}.swap(*this)`.
Returns: `*this`.

Add to 33.3.1 Header <thread> synopsis [thread.syn]

Extend std::this_thread:

```
namespace std {
 …
 namespace this_thread {
  …
  static bool is_interrupted() noexcept;
  static void throw_if_interrupted();
  static interrupt_token get_interrupt_token() noexcept;
  static interrupt_token
            exchange_interrupt_token(const interrupt_token&) noexcept;
 }
}
```

Add as a new chapter in parallel after class thread:

33.3.2 Class jthread [thread.jthread.class]

```
namespace std {
 class jthread
 {
  // standardized API as std::thread:
  public:
    // types:
    using id = ::std::thread::id;
    using native_handle_type = ::std::thread::native_handle_type;

    // construct/copy/destroy:
    jthread() noexcept;
    template <typename F, typename... Args>
      explicit jthread(F&& f, Args&&... args);
    ~jthread();

    jthread(const jthread&) = delete;
    jthread(jthread&&) noexcept;
    jthread& operator=(const jthread&) = delete;
    jthread& operator=(jthread&&) noexcept;

    // members:
    void swap(jthread&) noexcept;
    bool joinable() const noexcept;
    void join();
    void detach();

    id get_id() const noexcept;
    native_handle_type native_handle();

    // static members:
    static unsigned hardware_concurrency() noexcept {
      return ::std::thread::hardware_concurrency();
    };

    // - supplementary interrupt API:
    interrupt_token get_original_interrupt_token() const noexcept;
    bool interrupt() noexcept {
```

```
      return get_original_interrupt_token().interrupt();
    }
 };
}
```

Formulate the following differences to std::thread:

- Constructor for a passed task/callback creates an interrupt token that is copied to the started thread
  - Note that this token can never be updated/replaced for the std::jthread object.
- Silently catches a std::interrupted exception of the started thread.
- Destructor calls interrupt() and joint() is still joinable. Same applies to the thread if a new thread is move assigned.

Extend in **33.5.1 Header <condition_variable> synopsis [condition_variable.syn]**

```
namespace std {
  …
  enum class cv_status { no_timeout, timeout, interrupted };
}
```

Add in **33.5.3 Class condition_variable [thread.condition.condvar]**

```
namespace std {
 class condition_variable {
 public:
  …

  void wait(unique_lock<mutex>& lock);
  template <class Predicate>
   void wait(unique_lock<mutex>& lock, Predicate pred);
  // throw std::interrupted on interrupt:
  void wait_or_throw(unique_lock<mutex>& lock);
  template <class Predicate>
   void wait_or_throw(unique_lock<mutex>& lock, Predicate pred);

  template <class Clock, class Duration>
   cv_status wait_until(unique_lock<mutex>& lock,
                        const chrono::time_point<Clock, Duration>& abs_time);
  template <class Clock, class Duration, class Predicate>
   bool wait_until(unique_lock<mutex>& lock,
                   const chrono::time_point<Clock, Duration>& abs_time,
                   Predicate pred);
  // return std::cv_status::interrupted on interrupt:
  cv_status wait_until(unique_lock<mutex>& lock,
                       const interrupt_token& itoken);
  template <class Predicate>
   bool wait_until(unique_lock<mutex>& lock,
                        Predicate pred,
                        const interrupt_token& itoken);
  template <class Clock, class Duration>
   cv_status wait_until(unique_lock<mutex>& lock,
                        const chrono::time_point<Clock, Duration>& abs_time
                        const interrupt_token& itoken);
  template <class Clock, class Duration, class Predicate>
   bool wait_until(unique_lock<mutex>& lock,
```

```
                        const chrono::time_point<Clock, Duration>& abs_time
                        Predicate pred,
                        const interrupt_token& itoken);

  template <class Rep, class Period>
   cv_status wait_for(unique_lock<mutex>& lock,
                      const chrono::duration<Rep, Period>& rel_time);
  template <class Rep, class Period, class Predicate>
   bool wait_for(unique_lock<mutex>& lock,
                 const chrono::duration<Rep, Period>& rel_time,
                 Predicate pred);
  // return std::cv_status::interrupted on interrupt:
  template <class Rep, class Period>
   cv_status wait_for(unique_lock<mutex>& lock,
                      const chrono::duration<Rep, Period>& rel_time,
                      const interrupt_token& itoken);
  template <class Rep, class Period, class Predicate>
   bool wait_for(unique_lock<mutex>& lock,
                      const chrono::duration<Rep, Period>& rel_time,
                      Predicate pred,
                      const interrupt_token& itoken);
  };
}
```

As usual, the overloads taking a predicate yield as a Boolean value whether the predicate is fulfilled.

with the following wording (differences to wait() highlighted):

```
  void wait_interruptable(unique_lock<mutex>& lock);
```

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

- no other thread is waiting on this condition_variable object or
- lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

Effects:

- Atomically calls lock.unlock() and blocks on `*this`.
- When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
- The function will unblock when signaled by a call to notify_one(), a call to notify_all(), , `itoken.is_interrupted()` yields true, or spuriously.

Remarks: If the function fails to meet the postcondition, terminate() shall be called (18.5.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. —*end note*]

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Throws: `std::interrupted()` if `this_thread::is_interrupted()`.

```
template <class Clock, class Duration>
cv_status wait_until(unique_lock<mutex>& lock,
                     const chrono::time_point<Clock, Duration>& abs_time,
                     const interrupt_token& itoken);
```

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

- no other thread is waiting on this condition_variable object or
- lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

Effects:

- Atomically calls lock.unlock() and blocks on `*this`.
- When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
- The function will unblock when signaled by a call to notify_one(), a call to notify_all(), expiration of the absolute timeout (33.2.4) specified by `abs_time`, `itoken.is_interrupted()` yields true, or spuriously.

- If the function exits via an exception, lock.lock() shall be called prior to exiting the function.

Remarks: If the function fails to meet the postcondition, terminate() shall be called (18.5.1). [*Note:* This can happen if the re-locking of the mutex throws an exception. —*end note*]
Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.
Returns: cv_status::timeout if the absolute timeout (33.2.4) specified by abs_time expired, `cv_status::interrupted` if `itoken.is_interrupted()` yields true, otherwise cv_status::no_timeout.
Throws: Timeout-related exceptions (33.2.4).

Other wait_or_throw(), wait_until(), wait_for() functions accordingly.

Feature Test Macro

This is a new feature so that it shall have the following feature macro:

```
__cpp_lib_jthread
```

Acknowledgements

Thanks to all who incredibly helped me to prepare this paper, such as all people in the C++ concurrency and library working group.

Especially, we want to thank: Hans Boehm, Olivier Giroux, Pablo Halpern, Howard Hinnant, Alisdair Meredith, Gor Nishanov, Ville Voutilainen, Anthony Williams, Jeffrey Yasskin.