A Cooperatively Interruptible Joining Thread, Rev 5

New in R5

As requested at the SG1 meeting in Seattle 2018:
- Removed exception class `std::interrupted` and the `throw_if_interrupted()` API.
- Removed all TLS extensions and extensions to `std::this_thread`.
- Added support to let `jthread` call a callable that either takes the interrupt token as additional first argument or doesn’t get it (taking just all passed arguments).

New in R4

- Removed interruptible CV waiting members that don’t take a predicate.
- Removed adding a new `cv_status` value `interrupted`.
- Added CV members for interruptible timed waits.
- Renamed CV members that wait interruptible.
- Several minor fixes (e.g. on `noexcept`) and full proposed wording.

Purpose

This is the proposed wording for a cooperatively interruptible joining thread.


A default implementation exists at: http://github.com/josuttis/jthread. Note that the proposed functionality can be fully implemented on top of the existing C++ standard library without special OS support.

Basis examples

- A `jthread` automatically signals an interrupt at the end of its lifetime to the started thread (if still joinable) and joins:
  ```cpp
  void testJThreadWithToken()
  {
    std::jthread t([] (std::interrupt_token itoken) {
      while (!itoken.is_interrupted()) {
        //...
      }
    });
  } // jthread destructor signals interrupt and therefore ends the started thread and joins
  ```

  The interrupt could also be explicitly signaled with `t.interrupt()`.

- If the started thread doesn’t take an interrupt token, the destructor still has the benefit of calling `join()` (if still joinable):
  ```cpp
  void testJThreadJoining()
  {
    std::jthread t([] {
      //...
    });
  } // jthread destructor calls join()
  ```

  This is a significant improvement over `std::thread` where you had to program the following to get the same behavior (which is common in many scenarios):
void compareWithStdThreadJoining()
{
    std::thread t([] {
        //...
    });
    try {
        //...
    } catch (...) {
        j.join();
        throw; // rethrow
    }
    t.join();
}

— An extended CV API enables to interrupt CV waits using the passed interrupt token (i.e. interrupting the CV wait without polling):

void testInterruptibleCVWait()
{
    bool ready = false;
    std::mutex readyMutex;
    std::condition_variable readyCV;
    std::jthread t([&ready, &readyMutex, &readyCV] (std::interrupt_token it) {
        while (...) {
            ...
            {
                std::unique_lock lg{readyMutex};
                readyCV.wait_until(lg,
                    [&ready] {
                        return ready;
                    },
                    it); // also ends wait if it interrupted
            }
            ...
        }
    });
    ...
} // jthread destructor signals interrupt and therefore unblocks the CV wait and ends the started thread

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: Lewis Baker, Hans Boehm, Olivier Giroux, Pablo Halpern, Howard Hinnant, Alisdair Meredith, Gor Nishanov, Ville Voutilainen, and Jonathan Wakely.

Proposed Wording
All against N4762.

[Editorial note: This proposal uses the LaTeX macros of the draft standard. To adopt it please ask for the LaTeX source code of the proposed wording.]
30 Thread support library

30.1 General

The following subclauses describe components to create and manage threads, perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 1.

Table 1 — Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>30.3</td>
<td>Threads &lt;thread&gt;</td>
</tr>
<tr>
<td>30.4</td>
<td>Interrupt Tokens &lt;interrupt_token&gt;</td>
</tr>
<tr>
<td>30.5</td>
<td>Joining Threads &lt;jthread&gt;</td>
</tr>
<tr>
<td>30.6</td>
<td>Mutual exclusion &lt;mutex&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>30.7</td>
<td>Condition variables &lt;condition_variable&gt;</td>
</tr>
<tr>
<td>30.8</td>
<td>Futures &lt;future&gt;</td>
</tr>
</tbody>
</table>

30.2 Requirements [thread.req]

...

30.3 Threads [thread.threads]

...
30.4 Interrupt Tokens

30.4 describes components that can be used to asynchronously signal an interrupt. The interrupt can only be signaled once.

30.4.1 Header <interrupt_token> synopsis

namespace std {
    // 30.4.2 class interrupt_token
    class interrupt_token;
}

30.4.2 Class interrupt_token

The class interrupt_token implements semantics of shared ownership of an interrupt state (an atomic token to signal an interrupt). An interrupt can only be signaled once. All owners can signal an interrupt, provided the token is valid. All owners can check whether an interrupt was signaled. The last remaining owner of the interrupt state automatically releases the resources associated with the interrupt state.

namespace std {
    class interrupt_token {
        public:
            // 30.4.2.1 create, copy, destroy:
            explicit interrupt_token() noexcept;
            explicit interrupt_token(bool initial_state);
            interrupt_token(const interrupt_token&) noexcept;
            interrupt_token(interrupt_token&&) noexcept;
            interrupt_token& operator=(const interrupt_token&) noexcept;
            interrupt_token& operator=(interrupt_token&&) noexcept;
            ~interrupt_token();
            void swap(interrupt_token&) noexcept;

            // 30.4.2.5 interrupt handling:
            bool valid() const noexcept;
            bool is_interrupted() const noexcept;
            bool interrupt();
    }
}

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

Calls to interrupt() and is_interrupted() are atomic operations (6.8.2.1p3 ??) on the interrupt state contained in the interrupt_token object. Hence concurrent calls to these functions do not introduce data races. A call to interrupt() synchronizes with any call to interrupt() and is_interrupted() that observes the interrupt.

[Note: The implementation of the managed interrupt state shall ensure that future extensions to interrupt tokens are possible without breaking binary compatibility (i.e. make the shared interrupt state a polymorphic type) — end note]

30.4.2.1 interrupt_token constructors

interrupt_token() noexcept;

Effects: Constructs a new interrupt_token object that can’t be used to signal interrupts. [Note: Therefore, no resources have to be associated for the state. — end note]

Ensure: valid() == false.

interrupt_token(bool initial_state) noexcept;

Effects: Constructs a new interrupt_token object that can signal interrupts via an atomic associated interrupt state.

Ensure: valid() == true and is_interrupted() == initial_state.
interrupt_token(const interrupt_token& rhs) noexcept;

Effects: If rhs is not valid, constructs an interrupt_token object that is not valid; otherwise, constructs an interrupt_token that shares the ownership of the interrupt state with rhs.

Ensures: valid() == rhs.valid() and is_interrupted() == rhs.is_interrupted() and *this == rhs.

interrupt_token(interrupt_token&& rhs) noexcept;

Effects: Move constructs an object of type interrupt_token from rhs.

Ensures: *this shall contain the old value of rhs and rhs.valid() == false.

30.4.2.2 interrupt_token destructor

~interrupt_token();

Effects: If valid() and *this is the last owner of the interrupt state, releases the resources associated with the interrupt state.

30.4.2.3 interrupt_token assignment

interrupt_token& operator=(const interrupt_token& rhs) noexcept;

Effects: Equivalent to: interrupt_token(rhs).swap(*this);

Returns: *this.

interrupt_token& operator=(interrupt_token&& rhs) noexcept;

Effects: Equivalent to: interrupt_token(std::move(rhs)).swap(*this);

Returns: *this.

30.4.2.4 interrupt_token swap

void swap(interrupt_token& rhs) noexcept;

Effects: Swaps the state of *this and rhs.

30.4.2.5 interrupt_token members

bool valid() const noexcept;

Returns: true if the interrupt token can be used to signal interrupts.

bool is_interrupted() const noexcept;

Returns: true if initialized with true or initialized with false and interrupt() was called by one of the owners.

bool interrupt();

Effects: If !valid() or is_interrupted() the call has no effect. Otherwise, signals an interrupt so that is_interrupted() == true. [Note: Signaling an interrupt includes notifying all condition_variables temporarily registered via a an interruptable wait (30.7.1.2) — end note]

Ensures: !valid() || is_interrupted()

Returns: The value of is_interrupted() prior to the call.

30.4.2.6 interrupt_token comparisons

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

Returns: !lhs.valid() && !rhs.valid() or whether lhs and rhs refer to the same interrupt state (copied or moved from the same initial interrupt_token object).

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

Returns: !(lhs==rhs).
30.5 Joining Threads

30.5 describes components that can be used to create and manage threads with the ability to signal interrupts to cooperatively cancel the running thread.

30.5.1 Header <jthread> synopsis

```cpp
#include <interrupt_token>
namespace std {
    // 30.5.2 class jthread
    class jthread;

    void swap(jthread& x, jthread& y) noexcept;
}
```

30.5.2 Class jthread

The class `jthread` provides a mechanism to create a new thread of execution. The functionality is the same as for class `thread` with the additional ability to signal an interrupt and to automatically `join()` the started thread.

[Editorial note: This color signals differences to class `std::thread`.]

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

            // construct/copy/destroy
            jthread() noexcept;
            template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
            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(); // see ??

            // interrupt token handling
            interrupt_token get_original_interrupt_token() const noexcept;
            bool interrupt() noexcept;

            // static members
            static unsigned int hardware_concurrency() noexcept;

        private:
            interrupt_token itoken;
            // exposition only
        }
    }
}
```

30.5.2.1 jthread constructors

`jthread()` noexcept;

1. Effects: Constructs a `jthread` object that does not represent a thread of execution.
2. Ensures: `get_id() == id()` and `itoken.valid() == false`.

§ 30.5.2.1 6
template<class F, class... Args> explicit jthread(F&& f, Args&&... args);

Requires: F and each T_i in Args shall satisfy the Cpp17MoveConstructible requirements. INVOKE(DECAY_COPY(std::forward<F>(f)), itoken, DECAY_COPY(std::forward<Args>(args))...) or INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...) (??) shall be a valid expression.

Remarks: This constructor shall not participate in overload resolution if remove_cvref_t<F> is the same type as std::jthread.

Effects: Initializes itoken with false and constructs an object of type jthread. The new thread of execution executes INVOKE(DECAY_COPY(std::forward<F>(f)), itoken, DECAY_COPY(std::forward<Args>(args))...) if that expression is well-formed, otherwise INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...) with the calls to DECAY_COPY being evaluated in the constructing thread. Any return value from this invocation is ignored. [Note: This implies that any exceptions not thrown from the invocation of the copy of f will be thrown in the constructing thread, not the new thread. — end note] If the invocation with INVOKE() terminates with an uncaught exception, terminate() shall be called.

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.

Ensures: get_id() != id(). itoken.valid() == true. *this represents the newly started thread. [Note: Note that the calling thread can signal an interrupt only once, because it can’t replace this interrupt token. — end note]

Throws: system_error if unable to start the new thread.

Error conditions:

30.5.2.2 jthread destructor

~jthread();

Effects: Constructs an object of type jthread from x, and sets x to a default constructed state.

Ensures: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction. itoken yields the value of x.itoken prior to the start of construction and x.itoken.valid() == false.

30.5.2.3 jthread assignment

jthread& operator=(jthread&& x) noexcept;

Effects: If joinable(), calls interrupt() and join(). Assigns the state of x to *this and sets x to a default constructed state.

Ensures: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment. itoken yields the value of x.itoken prior to the assignment and x.itoken.valid() == false.

Returns: *this.

30.5.2.4 jthread interrupt members

interrupt_token get_original_interrupt_token() const noexcept

Effects: Equivalent to: return itoken;

bool interrupt() noexcept;

Effects: Equivalent to: return itoken.interrupt();
30.6 Mutual exclusion

...

30.7 Condition variables

30.7.1 Class condition_variable

namespace std {
    class condition_variable {
        public:
            condition_variable();
            ~condition_variable();

            condition_variable(const condition_variable&) = delete;
            condition_variable& operator=(const condition_variable&) = delete;

            void notify_one() noexcept;
            void notify_all() noexcept;
        // 30.7.1.1 noninterruptable waits:
            void wait(unique_lock<mutex>& lock);
            template<class Predicate>
                void wait(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);

            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);
        // 30.7.1.2 interrupt_token waits:
            template <class Predicate>
                bool wait_until(unique_lock<mutex>& lock,
                    Predicate pred, 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, 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, interrupt_token itoken);

            using native_handle_type = implementation-defined; /* see ??
            native_handle_type native_handle(); /* see ??
        
    }
}

1 The class condition_variable shall be a standard-layout class (??).

condition_variable();

2 Effects: Constructs an object of type condition_variable.

3 Throws: system_error when an exception is required (??).
Error conditions:

— resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

~condition_variable();

Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. The user should take care to ensure that no threads wait on *this once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of wait, wait_for, or wait_until that take a predicate. — end note]

Effects: Destroys the object.

void notify_one() noexcept;

Effects: If any threads are blocked waiting for *this, unblocks one of those threads.

void notify_all() noexcept;

30.7.1.1 Noninterruptable waits [thread.condition.wait]

Effects: Unblocks all threads that are blocked waiting for *this.

void wait(unique_lock<mutex>& lock);

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(2.1) no other thread is waiting on this condition_variable object or
(2.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until) threads.

Effects:
(3.1) Atomically calls lock.unlock() and blocks on *this.
(3.2) When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
(3.3) The function will unblock when signaled by a call to notify_one() or a call to notify_all(), or spuriously.

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

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

Throws: Nothing.

template<class Predicate>
void wait(unique_lock<mutex>* lock, Predicate pred);

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(7.1) no other thread is waiting on this condition_variable object or
(7.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until) threads.

Effects: Equivalent to:
while (!pred())
wait(lock);

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

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

Throws: Any exception thrown by pred.
template<class Clock, class Duration>
   cv_status wait_until(unique_lock<mutex>& lock,
   const chrono::time_point<Clock, Duration>& abs_time);

   // 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, 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 (??) specified by abs_time, 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 (??). [Note: This
   // can happen if the re-locking of the mutex throws an exception. — end note]

   // Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.
   // Returns: cv_status::timeout if the absolute timeout (??) specified by abs_time expired, otherwise
   // cv_status::no_timeout.
   // Throws: Timeout-related exceptions (??).

   template<class Rep, class Period>
   cv_status wait_for(unique_lock<mutex>& lock,
   const chrono::duration<Rep, Period>& rel_time);

   // 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, wait_until) threads.

   // Effects: Equivalent to:
   // return wait_until(lock, chrono::steady_clock::now() + rel_time);

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

   // Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.
   // Returns: cv_status::timeout if the relative timeout (??) specified by rel_time expired, otherwise
   // cv_status::no_timeout.
   // Throws: Timeout-related exceptions (??).

   template<class Clock, class Duration, class Predicate>
   bool wait_until(unique_lock<mutex>& lock,
   const chrono::time_point<Clock, Duration>& abs_time,
   Predicate pred);

   // 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, wait_until) threads.

   // Effects: Equivalent to:
   // while (!pred())
   //   if (wait_until(lock, abs_time) == cv_status::timeout)
   //     return pred();
   //   return true;

   // Remarks: If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This
   // can happen if the re-locking of the mutex throws an exception. — end note]
Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered. — end note]

Throws: Timeout-related exceptions (??) or any exception thrown by pred.

```cpp
template<class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
              const chrono::duration<Rep, Period>& rel_time,
              Predicate pred);
```

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, wait_until) threads.

Effects: Equivalent to:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

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

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

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note]

Throws: Timeout-related exceptions (??) or any exception thrown by pred.
30.7.1.2 interrupt_token waits

The following functions ensure to get notified if an interrupt is signaled for the passed `interrupt_token`. In that case they return (returning `false` if the predicate evaluates to `false`).

*[Editorial note: Because all signatures here in the effects clause call `is_interrupted()`, we don't need wording that the calls synchronize with `interrupt()`.*]

```cpp
template <class Predicate>
bool wait_until(unique_lock<mutex>& lock,
Predicate pred,
interrupt_token itoken);
```

*[Editorial note: This color signals differences to the corresponding `wait()` function without the interrupt token parameter.]*

1 **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`, `wait_until`) threads.

2 **Effects**: Registers `*this` to get notified when an interrupt is signaled on `itoken` during this call and then equivalent to:

   ```cpp
   while(!pred() && !itoken.is_interrupted()) {
     cv.wait(lock, 
     [&pred, &itoken] {
       return pred() || itoken.is_interrupted();
     });
   }
   return pred();
   ```

3 **[Note: The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered or an interrupt was signaled. — end note]***

4 **Ensures**: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

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

6 **Throws**: Any exception thrown by `pred`.

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

*[Editorial note: This color signals differences to the corresponding `wait_until()` function without the interrupt token parameter.]*

7 **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`, `wait_until`) threads.

8 **Effects**: Registers `*this` to get notified when an interrupt is signaled on `itoken` during this call and then equivalent to:

   ```cpp
   while(!pred() && !itoken.is_interrupted() && Clock::now() < abs_time) {
     cv.wait_until(lock,
     abs_time,
     [&pred, &itoken] {
       return pred() || itoken.is_interrupted();
     });
   }
   return pred();
   ```

9 **[Note: The returned value indicates whether the predicate evaluated to `true` regardless of whether the timeout was triggered or an interrupt was signaled. — end note]***
Ensures: \( \text{lock.owns_lock()} \) is true and \( \text{lock.mutex()} \) is locked by the calling thread.

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

Throws: Timeout-related exceptions (??) or any exception thrown by \text{pred}.

\[
\text{template } \langle \text{class Rep, class Period, class Predicate} \rangle \\
\text{bool wait_for(unique_lock<mutex>}& \text{lock,} \\
\quad \text{const chrono::duration<Rep, Period>& } \text{rel_time,} \\
\quad \text{Predicate } \text{pred,} \\
\quad \text{interrupt_token } \text{itoken);} \\
\text{[Editorial note: This color signals differences to the corresponding } \text{wait_for()} \text{ function without the interrupt token parameter. ]}
\]

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

\(\begin{align*}
\text{(13.1) } & \quad \text{no other thread is waiting on this condition_variable object or} \\
\text{(13.2) } & \quad \text{lock.mutex()} \text{ returns the same value for each of the lock arguments supplied by all concurrently waiting (via } \text{wait, wait_for, wait_until} \text{ threads.}
\end{align*}\)

Effects: Equivalent to:

\[
\text{return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred),} \\
\quad \text{std::move(itoken));}
\]

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered or an interrupt was signaled. — end note]

[Note: There is no blocking if \text{pred()} is initially true, even if the timeout has already expired. — end note]

Ensures: \( \text{lock.owns_lock()} \) is true and \( \text{lock.mutex()} \) is locked by the calling thread.

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

Throws: Timeout-related exceptions (??) or any exception thrown by \text{pred}.