Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 P0660R7
Date: 2018-11-09
Reply to: Nicolai Josuttis (nico@josuttis.de),
Lewis Baker (lbaker@fb.com)
Billy O'Neal (bion@microsoft.com)
Herb Sutter (hsutter@microsoft.com),
Anthony Williams (anthony@justsoftwaresolutions.co.uk)
Audience: SG1, LEWG, LWG
Prev. Version: `www.wg21.link/P0660R6`, `www.wg21.link/P1287R0`

Interrupt Tokens and a Joining Thread, Rev 7

New in R7

— Adopt `www.wg21.link/P1287` as discussed in the SG1 meeting in San Diego 2018, which includes:
  — Add callbacks for interrupt tokens.
  — Split into `interrupt_token` and `interrupt_source`.

New in R6

— User `condition_variable_any` instead of `consition_variable` to avoid all possible races, deadlocks, and unintended undefined behavior.

— Clarify future binary compatibility for interrupt handling (mention requirements for future callback support and allow `bad_alloc` exceptions on waits.

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 `jhread` 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.

For a full discussion fo the motivation, see `www.wg21.link/p0660r0` and `www.wg21.link/p0660r1`.

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:

```
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):

```
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_any 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: 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 [thread]

30.1 General [jthread.general]

1 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

	Subclause	Header(s)
30.2	Requirements	
30.3	Threads	`<thread>`
30.4	Interrupt Tokens	`<interrupt_token>`
30.5	Joining Threads	`<jthread>`
30.6	Mutual exclusion	`<mutex>` `<shared_mutex>`
30.7	Condition variables	`<condition_variable>`
30.8	Futures	`<future>`

30.2 Requirements [thread.req]

...

30.3 Threads [thread.threads]

...

30.4 Interrupt Tokens [thread.interrupt_token]

1 30.4 describes components that can be used to asynchonously signal an interrupt. The interrupt can only be signaled exactly once by one of multiple `interrupt_sources` to one or multiple `interrupt_tokens`. Callbacks can be registered as `interrupt_tokens` to be called when the interrupt is signaled.

For this, classes `interrupt_source` and `interrupt_token` implement semantics of shared ownership of an associated atomic interrupt state (an atomic token to signal an interrupt). The last remaining owner of the interrupt state automatically releases the resources associated with the interrupt state.

2 Calls to `interrupt()`, `is_interrupted()`, `is_valid()`, and `is_interruptible()` are atomic operations (6.8.2.1p3 **??**) on the interrupt state contained in the interrupt state 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.

30.4.1 Header <interrupt_token> synopsis [thread.interrupt_token.syn]

```
namespace std {
  // 30.4.4 class interrupt_token
  template <typename Callback> class interrupt_callback;
  class interrupt_source;
  class interrupt_token;
}
```

30.4.2 Class `interrupt_callback` [interrupt_callback]

1

```
namespace std {
  template <typename Callback>
  class interrupt_callback {
  public:
    // 30.4.2.1 create, copy, destroy:
    interrupt_callback(interrupt_token it, Callback&& cb);
    ~interrupt_callback();

    interrupt_callback(const interrupt_callback&) = delete;
    interrupt_callback(interrupt_callback&&) = delete;
    interrupt_callback& operator=(const interrupt_callback&) = delete;
    interrupt_callback& operator=(interrupt_callback&&) = delete;
  }
}
```

30.4.2.1 `interrupt_callback` constructors [interrupt_callback.constr]

`interrupt_callback(interrupt_token it, Callback&& cb) noexcept;`

1 *Requires:* `cb` is a callable object taking no parameters.

2 *Effects:* Constructs a new `interrupt_callback` object that can be used to be called when an interrupt is signaled at `it`. If `it.is_interrupted()` `cb` is immediatelly called.

30.4.3 Class `interrupt_source` [interrupt_source]

1 The class `interrupt_source` implements semantics of signaling interrupts to `interrupt_tokens` (30.4.4). All owners can signal an interrupt, provided the token is valid. An interrupt can only be signaled once. All owners can check whether an interrupt was signaled.

```
namespace std {
  class interrupt_source {
  public:
    // 30.4.3.1 create, copy, destroy:
    explicit interrupt_source() noexcept;
    explicit interrupt_source(nullptr_t);

    interrupt_source(const interrupt_source&) noexcept;
    interrupt_source(interrupt_source&&) noexcept;
    interrupt_source& operator=(const interrupt_source&) noexcept;
    interrupt_source& operator=(interrupt_source&&) noexcept;
```

```
      ~interrupt_source();
      void swap(interrupt_source&) noexcept;

      // 30.4.3.5 interrupt handling:
      interrupt_token get_token() const noexcept;
      bool is_valid() const noexcept;
      bool is_interrupted() const noexcept;
      bool interrupt();
    }
  }

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

[*Note*: Implementations are expected to implement interruption in terms of a type-erased facility that allows any destructible and invocable object to be called by `interruption_source::interrupt()` in a future version of C++. — *end note*]

30.4.3.1 interrupt_source constructors [interrupt_source.constr]

`interrupt_source() noexcept;`

1 *Effects:* Constructs a new `interrupt_source` object that can signal interrupts.

2 *Ensures:* `is_valid() == true` and `is_interrupted() == false`.

`interrupt_source(nullptr_t) noexcept;`

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

4 *Ensures:* `is_valid() == false`.

`interrupt_source(const interrupt_source& rhs) noexcept;`

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

6 *Ensures:* `is_valid() == rhs.is_valid()` and `is_interrupted() == rhs.is_interrupted()` and `*this == rhs`.

`interrupt_source(interrupt_source&& rhs) noexcept;`

7 *Effects:* Move constructs an object of type `interrupt_source` from `rhs`.

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

30.4.3.2 interrupt_source destructor [interrupt_source.destr]

`~interrupt_source();`

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

30.4.3.3 interrupt_source assignment [interrupt_source.assign]

`interrupt_source& operator=(const interrupt_source& rhs) noexcept;`

1 *Effects:* Equivalent to: `interrupt_source(rhs).swap(*this);`

2 *Returns:* `*this`.

`interrupt_source& operator=(interrupt_source&& rhs) noexcept;`

3 *Effects:* Equivalent to: `interrupt_source(std::move(rhs)).swap(*this);`

4 *Returns:* `*this`.

30.4.3.4 interrupt_source swap [interrupt_source.swap]

`void swap(interrupt_source& rhs) noexcept;`

1 *Effects:* Swaps the state of `*this` and `rhs`.

30.4.3.5 `interrupt_source` members [interrupt_source.mem]

`interrupt_token get_token() const noexcept;`

1 *Effects:* If `!is_valid()`, constructs an `interrupt_token` object that is not valid; otherwise, constructs an `interrupt_token` object `it` that shares the ownership of the interrupt state with `*this`.

2 *Ensures:* `is_valid() == it.is_valid()` and `is_interrupted() == it.is_interrupted()`.

`bool is_valid() const noexcept;`

3 *Returns:* `true` if the interrupt source can be used to signal interrupts. [*Note*: Returns `false` if the object was created with the `nullptr` fro the values was moved away. — *end note*]

`bool is_interrupted() const noexcept;`

4 *Returns:* `true` if `is_valid()` and `interrupt()` was called by one of the owners.

`bool interrupt();`

5 *Effects:* If `!is_valid()` or `is_interrupted()` the call has no effect. Otherwise, signals an interrupt so that `is_interrupted() == true` and all registered callbacks are synchronously called. [*Note*: Signaling an interrupt includes notifying all condition variables of type `condition_variable_any` temporarily registered during an interruptable wait (**??**) — *end note*]

6 *Ensures:* `!is_valid() || is_interrupted()`

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

30.4.3.6 `interrupt_source` comparisons [interrupt_source.cmp]

`bool operator== (const interrupt_source& lhs, const interrupt_source& rhs);`

1 *Returns:* `!lhs.is_valid() && !rhs.is_valid()` or whether `lhs` and `rhs` refer to the same interrupt state (copied or moved from the same initial `interrupt_source` object).

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

2 *Returns:* `!(lhs==rhs)`.

30.4.4 Class `interrupt_token` [interrupt_token]

1 The class `interrupt_token` implements semantics getting interrupts signaled from the `interrupt_source` object they were created from. All tokens can check whether an interrupt was signaled. When an interrupt is signaled, which is possible only once, any registered `interrupt_callback` (30.4.2) is called. Registering a callback after an interrupt was already signaled calls the callback immediately.

```
namespace std {
  class interrupt_token {
  public:
    // 30.4.4.1 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;

    // 30.4.4.5 interrupt handling:
    bool is_interrupted() const noexcept;
    bool is_interruptible() const noexcept;
  }
}

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

30.4.4.1 `interrupt_token` constructors [interrupt_token.constr]

`interrupt_token() noexcept;`

1 *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*]

2 *Ensures:* `is_interruptible() == false`.

`interrupt_token(const interrupt_token& rhs) noexcept;`

3 *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`.

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

`interrupt_token(interrupt_token&& rhs) noexcept;`

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

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

30.4.4.2 `interrupt_token` destructor [interrupt_token.destr]

`~interrupt_token();`

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

30.4.4.3 `interrupt_token` assignment [interrupt_token.assign]

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

30.4.4.4 `interrupt_token` swap [interrupt_token.swap]

`void swap(interrupt_token& rhs) noexcept;`

1 *Effects:* Swaps the state of `*this` and `rhs`.

30.4.4.5 `interrupt_token` members [interrupt_token.mem]

`bool is_interrupted() const noexcept;`

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

`bool is_interruptible() const noexcept;`

3 *Returns:* `true` if the interrupt token did or still can receive an interrupt signal so that registered callbacks can be called (immediately or later). [*Note*: Returns `false` if registering a callback doesn't make any sense because it can't be called (anymore). (e.g., because it is not interrupted yet and there is no more associated `interrupt_source` (30.4.3)). — *end note*]

30.4.4.6 `interrupt_token` comparisons [interrupt_token.cmp]

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

1 *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);`

2 *Returns:* `!(lhs==rhs)`.

30.5 Joining Threads [thread.jthreads]

1 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 [thread.jthread.syn]

```
#include <interrupt_token>

namespace std {
  // 30.5.2 class jthread
  class jthread;

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

30.5.2 Class `jthread` [thread.jthread.class]

1 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`.]

```
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();
    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_interrupt_source() const noexcept;
    bool interrupt() noexcept;
    // static members
    static unsigned int hardware_concurrency() noexcept;
  private:
    interrupt_token isource;                  // exposition only
  };
}
```

30.5.2.1 `jthread` constructors [thread.jthread.constr]

`jthread() noexcept;`

1 *Effects:* Constructs a `jthread` object that does not represent a thread of execution.

2 *Ensures:* `get_id() == id()` and `isource.valid() == false`.

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

3 *Requires:* `F` and each T_i in `Args` shall satisfy the *Cpp17MoveConstructible* requirements. `INVOKE(DECAY_COPY(std::forward<F>(f)), isource, 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.

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

5 *Effects:* Initializes `isource` and constructs an object of type `jthread`. The new thread of execution executes `INVOKE(DECAY_COPY(std::forward<F>(f)), isource,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.

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

7 *Ensures:* `get_id() != id()`. `isource.is_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*]

8 *Throws:* `system_error` if unable to start the new thread.

9 *Error conditions:*

(9.1) — `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

`jthread(jthread&& x) noexcept;`

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

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

30.5.2.2 `jthread` destructor [thread.jthread.destr]

`~jthread();`

1 If `joinable()`, calls `interrupt()` and `join()`. Otherwise, has no effects. [*Note*: Operations on `*this` are not synchronized. — *end note*]

30.5.2.3 `jthread` assignment [thread.jthread.assign]

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

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

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

3 *Returns:* `*this`.

30.5.2.4 `jthread` interrupt members [thread.jthread.interrupt]

`interrupt_token get_interrupt_source() const noexcept`

1 *Effects:* Equivalent to: `return isource;`

`bool interrupt() noexcept;`

2 *Effects:* Equivalent to: `return isource.interrupt();`

30.6 Mutual exclusion [thread.mutex]

...

30.7 Condition variables [thread.condition]

...

30.7.1 Header <condition_variable> synopsis [condition_variable.syn]

...

30.7.2 Non-member functions [thread.condition.nonmember]

...

30.7.3 Class condition_variable [thread.condition.condvar]

...

30.7.4 Class condition_variable_any [thread.condition.condvarany]

...

```
namespace std {
  class condition_variable_any {
  public:
    condition_variable_any();
    ~condition_variable_any();

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

    void notify_one() noexcept;
    void notify_all() noexcept;

    // 30.7.4.1 noninterruptable waits:
    template<class Lock>
      void wait(Lock& lock);
    template<class Lock, class Predicate>
      void wait(Lock& lock, Predicate pred);

    template<class Lock, class Clock, class Duration>
      cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
    template<class Lock, class Clock, class Duration, class Predicate>
      bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time,
                      Predicate pred);
    template<class Lock, class Rep, class Period>
      cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
    template<class Lock, class Rep, class Period, class Predicate>
      bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);

    // 30.7.4.2 interrupt_token waits:
    template <class Lock, class Predicate>
      bool wait_until(Lock& lock,
                      Predicate pred,
                      interrupt_token itoken);
    template <class Lock, class Clock, class Duration, class Predicate>
      bool wait_until(Lock& lock,
                      const chrono::time_point<Clock, Duration>& abs_time
                      Predicate pred,
                      interrupt_token itoken);
    template <class Lock, class Rep, class Period, class Predicate>
      bool wait_for(Lock& lock,
                    const chrono::duration<Rep, Period>& rel_time,
                    Predicate pred,
                    interrupt_token itoken);
  };
}
```

`condition_variable_any();`

1 *Effects:* Constructs an object of type `condition_variable_any`.

2 *Throws:* `bad_alloc` or `system_error` when an exception is required (??).

3 *Error conditions:*

(3.1) — `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

(3.2) — `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

`~condition_variable_any();`

4 *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*]

5 *Effects:* Destroys the object.

`void notify_one() noexcept;`

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

`void notify_all() noexcept;`

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

30.7.4.1 Noninterruptable waits [thread.condvarany.wait]

```
template<class Lock>
  void wait(Lock& lock);
```

1 *Effects:*

(1.1) — Atomically calls `lock.unlock()` and blocks on `*this`.

(1.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.

(1.3) — The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, or spuriously.

2 *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*]

3 *Ensures:* `lock` is locked by the calling thread.

4 *Throws:* Nothing.

```
template<class Lock, class Predicate>
  void wait(Lock& lock, Predicate pred);
```

5 *Effects:* Equivalent to:

```
while (!pred())
  wait(lock);
```

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

6 *Effects:*

(6.1) — Atomically calls `lock.unlock()` and blocks on `*this`.

(6.2) — When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.

(6.3) — 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.

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

7 *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*]

8 *Ensures:* `lock` is locked by the calling thread.

9 *Returns:* `cv_status::timeout` if the absolute timeout (**??**) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

10 *Throws:* Timeout-related exceptions (**??**).

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

11 *Effects:* Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

12 *Returns:* `cv_status::timeout` if the relative timeout (**??**) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

13 *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*]

14 *Ensures:* `lock` is locked by the calling thread.

15 *Throws:* Timeout-related exceptions (**??**).

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

16 *Effects:* Equivalent to:

```
while (!pred())
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
return true;
```

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

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

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

19 *Effects:* Equivalent to:

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

30.7.4.2 Interruptable waits [thread.condvarany.interruptwait]

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`). [*Note*: Because all signatures here call `is_interrupted()`, their calls synchronize with `interrupt()`. — *end note*]

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

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

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

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

3 *Ensures:* Exception or `lock` is locked by the calling thread.

4 *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*]

5 *Throws:* `std::bad_alloc` if memory for the internal data structures could not be allocated, or any exception thrown by `pred`.

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

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

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

7 [*Note*: There is no blocking if `pred()` is initially `true`, itoken is not valid or already interrupted, or if the timeout has already expired. — *end note*]

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

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*]

10 *Ensures:* Exception or `lock` is locked by the calling thread.

11 *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*]

12 *Throws:* `std::bad_alloc` if memory for the internal data structures could not be allocated, any timeout-related exception (**??**), or any exception thrown by `pred`.

```
template <class Lock, class Rep, class Period, class Predicate>
  bool wait_for(Lock& lock,
                const chrono::duration<Rep, Period>& rel_time,
                Predicate pred,
```

```
interrupt_token itoken);
```

13 *Effects:* Equivalent to:

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

30.8 Futures [futures]

...