Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 P0660R8
Date: 2019-01-13
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/P0660R7, www.wg21.link/P1287R0

Stop Tokens and a Joining Thread, Rev 8

New in R8

As requested at the LEWG meeting in San Diego 2018:

- Terminology (especially rename `interrupt_token` to `stop_token`).
- Add a deduction guide for `stop_callback`
- Add `std::nostopstate_t` to create stop stop tokens that don't share a stop state
- Several clarifications in wording

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

— At the end of its lifetime a `jthread` automatically signals a request to stop the started thread (if still joinable) and joins:

```
void testJThreadWithToken()
{
    std::jthread t([] (std::stop_token stoken) {
                     while (!stoken.stop_requested()) {
                       //...
                     }
                   });
    //...
} // jthread destructor signals requests to stop and therefore ends the started thread and joins
```

The stop could also be explicitly requested with `t.request_stop()`.

— If the started thread doesn't take an stop 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 stop 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::stop_token st) {
                   while (...) {
                     ...
                     {
                       std::unique_lock lg{readyMutex};
                       readyCV.wait_until(lg,
                                          [&ready] {
                                             return ready;
                                          },
                                          st);  // also ends wait on stop request for st
                     }
                     ...
                   }
                 });
```

```
  ...
} // jthread destructor signals stop request 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
```

Design Discussion

Problems with "interrupt"

Earlier versions of this paper used the names `interrupt_token`, `interrupt_source` and `interrupt_callback` to refer to the abstraction used to signal interrupt.

However, the term "interrupt" already has common usage in industry and typically refers to something which can be interrupted and then return back to the non-interrupted state.

For example, hardware interrupts are raised when some event happens and then once the interrupt is handled the system returns back to the non-interrupted state, allowing the interrupt to be raised again.

The boost::thread library also uses the term "interrupt" to refer to an operation that can be raised many times and when the interrupt is handled the state is reset back to non-interrupted.

This is different from the semantics of the abstraction proposed in this paper which has the semantics that once it has been signalled it never returns to the non-signalled state. Thus the term "interrupt" seems inappropriate and is likely to lead to confusion.

Alternative names

There was some discussion in at LEWG at San Diego about alternative names for `interrupt_token` and there were two candidates: `cancellation_token` and `stop_token`.

The term `cancellation_token` has precedent in other C++ libraries. For example, Microsoft's PPL uses the names 'cancellation_token', 'cancellation_token_source' and 'cancellation_registration'.

The use of the "cancel" term also has precedent in the Networking TS which defines methods such as `basic_waitable_timer::cancel()` and `basic_socket::cancel()` and makes use of `std::errc::operation_canceled` as an error code in response to a request to cancel the operation.

However, some concerns were raised about the potential for confusion if a `std::jthread::cancel()` method were added as some may confuse this as somehow being related to the semantics of `pthread_cancel()` which is able to cancel a thread at an arbitrary point rather than cooperatively at well-defined cancellation points.

A straw poll was taken in LEWG at San Diego and the group favoured `stop_token`.

A suggestion was also made to introduce the use of the term "request" to more clearly communicate the asynchronous and cooperative nature of the abstraction. This suggestion has been adopted.

As a result the proposed names for the types and methods are now as follows:

```
class stop_token {
public:
  ...
  [[nodiscard]] bool stop_requested() const noexcept;
  [[nodiscard]] bool stop_possible() const noexcept;
};

class stop_source {
public:
  ...
  [[nodiscard]] bool stop_requested() const noexcept;
  [[nodiscard]] bool stop_possible() const noexcept;
  bool request_stop() const noexcept;
};

template<Invocable Callback>
class stop_callback {
public:
  ...
};
```

Callback Registration/Deregistration

An important capability for asynchronous use-cases for `stop_token` is the ability to attach a callback to the `stop_token` that will be called if a request to stop is made. The motivations for this are discussed in more detail in P1287R0.

Registration of a callback is performed by constructing a `stop_callback` object, passing the constructor both a `stop_token` and a `Invocable` object that is invoked if/when a call to `request_stop()` is made.

For example:

```
void cancellable_operation(std::stop_token stoken = {})
{
  auto handle = begin_operation();
  std::stop_callback cb{ stoken, [&] { cancel_operation(handle); }};
  ...
  auto result = end_operation(handle);
}
```

When a `stop_callback` object is constructed, if the `stop_token` has already received a request to stop then the callback is immediately invoked inside the constructor. Otherwise, the callback is registered with the `stop_token` and is later invoked if/when some thread calls `request_stop()` on an associated `stop_source`.

The callback registration is guaranteed to be performed atomically. If there is a concurrent call to `request_stop()` from another thread then either the current thread will see the request to stop and immediately invoke the callback on the current thread or the other thread will see the callback registration and will invoke the callback before returning from `request_stop()`.

When the `stop_callback` object is destructed the callback is deregistered from the list of callbacks associated with the `stop_token`'s shared state the callback is guaranteed not to be called after the `stop_callback` destructor returns.

Note that there is a potential race here between the callback being deregistered and a call to `request_stop()` being made on another thread which could invoke the callback. If the callback has not yet started executing on the other thread then the callback is deregistered and is never called. Otherwise, if the callback has already started executing on another thread then the call to `stop_callback()` will block the current thread until the callback returns.

If the call to the `stop_callback` destructor is made from within the the invocation of the callback on the same thread then the destructor does not block waiting for the callback to return as this would cause a deadlock. Instead, the destructor returns immediately without waiting for the callback to return.

Other Hints

The terminology was carfully selected with the following reasons

- With a stop token we neither "interrupt" nor "cancel" something. We request a stop that cooperatively has to get handled.
- `stop_possible()` helps to avoid addign new callbacks or checking for stop states. The name was selected to have a common and pretty self-explanatory name that is shared by both `stop_source`s and `stop_token`s.

The deduction guide for `stop_callback`s enables constructing a `stop_callback` with an lvalue callable:

```
auto lambda = []{};
std::stop_callback cb{ token, lambda };  // captures by reference
```

Adding a new callback is `noexcept` (unless moving the passed function throws).

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, Tony Van Eerd, 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	Stop Tokens	`<stop_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 Stop Tokens [thread.stop_token]

1 30.4 describes components that can be used to asynchonously request an that an operation stop execution in a timely manner, typically because the result is no longer required.

2 A `stop_token` can be passed to an operation which can either actively poll the token to check if there has been a request to stop or can register a callback using the `stop_callback` class which will be called in the event that a request to stop is made. A request to stop can be made via any one of potentially multiple associated `stop_source`s and this request will be visible to all associated `stop_token`s. Once a request to stop has been made it cannot be reverted and second and subsequent requests to stop are no-ops.

3 Callbacks registered via a `stop_callback`s object is called when a request to stop is first made by any of the `stop_source` objects associated with the `stop_token` used to construct the `stop_callback`.

4 To support this, classes `stop_source`, `stop_token` and `stop_callback` implement semantics of shared ownership of an associated atomic stop state. The last remaining owner of the stop state automatically releases the resources associated with the stop state.

5 Calls to `request_stop()`, `stop_requested()`, and `stop_possible()` are atomic operations (6.8.2.1p3 **??**) on the shared stop state. Hence concurrent calls to these functions do not introduce data races. A call to `request_stop()` that returns `false` (i.e. the first call) synchronizes with a call to `stop_requested()` on an associated `stop_token` or `stop_source` that returns `true`.

30.4.1 Header `<stop_token>` synopsis [thread.stop_token.syn]

```
namespace std {
  // 30.4.4 class stop_token
  class stop_token;
  // 30.4.3 class stop_source
  class stop_source;
  // 30.4.2 class stop_callback
  template <Invocable Callback>
    requires MoveConstructible<Callback>
  class stop_callback;
}
```

30.4.2 Class `stop_callback` [stop_callback]

1

```
namespace std {
  template <Invocable Callback>
    requires MoveConstructible<Callback>
  class stop_callback {
  public:
    // 30.4.2.1 create, destroy:
    explicit stop_callback(const stop_token& st, Callback&& cb)
        noexcept(std::is_nothrow_move_constructible_v<Callback>);
    explicit stop_callback(stop_token&& st, Callback&& cb)
        noexcept(std::is_nothrow_move_constructible_v<Callback>);
    ~stop_callback();

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

  private:
    // exposition only
    Callback callback;
  };

  template <typename Callback>
  stop_callback(const stop_token&, Callback&&) -> stop_callback<Callback>;

  template <typename Callback>
  stop_callback(stop_token&&, Callback&&) -> stop_callback<Callback>;
}
```

```
template<typename _Callback>
stop_callback(const stop_token&, _Callback&&) -> stop_callback<_Callback>;

template<typename _Callback>
stop_callback(stop_token&&, _Callback&&) -> stop_callback<_Callback>;
```

30.4.2.1 `stop_callback` constructors and destructor [stop_callback.constr]

```
explicit stop_callback(const stop_token& st, Callback&& cb)
  noexcept(std::is_nothrow_move_constructible_v<Callback>);
explicit stop_callback(stop_token&& st, Callback&& cb)
  noexcept(std::is_nothrow_move_constructible_v<Callback>);
```

1 *Effects:* Initialises `callback` with `static_cast<Callback&&>(cb)`. If `st.stop_requested()` is `true` then immediately invokes `static_cast<Callback&&>(callback)` with zero arguments on the current thread before the constructor returns. Otherwise, the callback is registered with the shared stop state of `st` such that `static_cast<Callback&&>(callback)` is invoked by first call to `src.request_stop()` on a `stop_source` instance, `src`, that references the same atomic stop state as `st`. If invoking the callback throws an unhandled exception then `std::terminate()` is called. *Throws:* Any exception thrown by the initialization of `callback`.

```
~stop_callback();
```

3 *Effects:* Deregisters the callback from the associated atomic stop state. If `callback` is concurrently executing on another thread then the destructor shall block until the invocation of `callback` returns before calling `callback`'s destructor. The destructor shall not block waiting for the execution of another callback registered with the same atomic stop state to finish. A subsequent call to `src.request_stop()` on a `stop_source`, `src`, with the same associated stop state shall not invoke `callback` once the destructor has returned.

30.4.3 Class `stop_source` [stop_source]

1 The class `stop_source` implements the semantics of signaling a request to stop to `stop_token`s (30.4.4) sharing the same atomic stop state. All `stop_source`s sharing the same atomic stop state can request a stop. Once a request to stop has been made it cannot be undone. A subsequent request to stop is a no-op.

```
namespace std {
  // 30.4.3.1 no-shared-stop-state indicator
  struct nostopstate_t{see below};
  inline constexpr nostopstate_t nostopstate(unspecified);

  class stop_source {
  public:
    // 30.4.3.2 create, copy, destroy:
    stop_source();
    explicit stop_source(nullptr_t) noexcept;

    stop_source(const stop_source&) noexcept;
    stop_source(stop_source&&) noexcept;
    stop_source& operator=(const stop_source&) noexcept;
    stop_source& operator=(stop_source&&) noexcept;
    ~stop_source();
    void swap(stop_source&) noexcept;

    // 30.4.3.6 stop handling:
    [[nodiscard]] stop_token get_token() const noexcept;
    [[nodiscard]] bool stop_possible() const noexcept;
    [[nodiscard]] bool stop_requested() const noexcept;
    bool request_stop() const noexcept;

    friend bool operator== (const stop_source& lhs, const stop_source& rhs) noexcept;
    friend bool operator!= (const stop_source& lhs, const stop_source& rhs) noexcept;
  };
}
```

30.4.3.1 No-shared-stop-state indicator [stop_source.nostopstate]

```
struct nostopstate_t{see below};
inline constexpr nostopstate_t nullopt(unspecified);
```

1 The struct `nostopstate_t` is an empty class type used as a unique type to indicate the state of not containing a shared stop state for `stop_source` objects. In particular, `stop_source` has a constructor with `nostopstate_t` as a single argument; this indicates that a stop source object not sharing a stop state shall be constructed.

2 Type `nostopstate_t` shall not have a default constructor or an initializer-list constructor, and shall not be an aggregate.

30.4.3.2 `stop_source` constructors [stop_source.constr]

```
stop_source();
```

1 *Effects:* Constructs a new `stop_source` object that can be used to request stops.

2 *Ensures:* `stop_possible() == true` and `stop_requested() == false`.

3 *Throws:* `bad_alloc` If memory could not be allocated for the shared atomic stop state.

```
explicit stop_source(nullptr_t) noexcept;
```

4 *Effects:* Constructs a new `stop_source` object that can't be used to request stops. [*Note*: Therefore, no resources have to be associated for the state. — *end note*]

5 *Ensures:* `stop_possible() == false`.

```
stop_source(const stop_source& rhs) noexcept;
```

6 *Effects:* If `rhs.stop_possible() == true`, constructs an `stop_source` that shares the ownership of the stop state with `rhs`.

7 *Ensures:* `stop_possible() == rhs.stop_possible()` and `stop_requested() == rhs.stop_requested()` and `*this == rhs`.

```
stop_source(stop_source&& rhs) noexcept;
```

8 *Effects:* Move constructs an object of type `stop_source` from `rhs`.

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

30.4.3.3 `stop_source` destructor [stop_source.destr]

```
~stop_source();
```

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

30.4.3.4 `stop_source` assignment [stop_source.assign]

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

30.4.3.5 `stop_source` swap [stop_source.swap]

```
void swap(stop_source& rhs) noexcept;
```

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

30.4.3.6 `stop_source` members [stop_source.mem]

```
[[nodiscard]] stop_token get_token() const noexcept;
```

1 *Effects:* If `!stop_possible()`, constructs an `stop_token` object that does not share a stop state. Otherwise, constructs an `stop_token` object `st` that shares the ownership of the stop state with `*this`.

2 *Ensures:* `stop_possible() == st.stop_possible()` and `stop_requested() == st.stop_requested()`.

```
[[nodiscard]] bool stop_possible() const noexcept;
```

3 *Returns:* `true` if the stop source can be used to request stops. [*Note*: Returns `false` if the object was created with the `nullptr` or the values were moved away. — *end note*]

```
[[nodiscard]] bool stop_requested() const noexcept;
```

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

```
bool request_stop() const noexcept;
```

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

6 *Ensures:* `!stop_possible() || stop_requested()`

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

30.4.3.7 `stop_source` comparisons [stop_source.cmp]

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

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

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

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

30.4.4 Class `stop_token` [stop_token]

1 The class `stop_token` provides an interface for querying whether a request to stop has been made (`stop_requested()`) or can ever be made (`stop_possible()`) from an associated `stop_source` object. A `stop_token` can also be passed to a `stop_callback` constructor to register a callback to be called when a request to stop has been made from an associated `stop_source`.

```
namespace std {
  class stop_token {
  public:
    // 30.4.4.1 create, copy, destroy:
    stop_token() noexcept;

    stop_token(const stop_token&) noexcept;
    stop_token(stop_token&&) noexcept;
    stop_token& operator=(const stop_token&) noexcept;
    stop_token& operator=(stop_token&&) noexcept;
    ~stop_token();
    void swap(stop_token&) noexcept;

    // 30.4.4.5 stop handling:
    [[nodiscard]] bool stop_requested() const noexcept;
    [[nodiscard]] bool stop_possible() const noexcept;

    friend bool operator== (const stop_token& lhs, const stop_token& rhs) noexcept;
    friend bool operator!= (const stop_token& lhs, const stop_token& rhs) noexcept;
  };
}
```

30.4.4.1 `stop_token` constructors [stop_token.constr]

```
stop_token() noexcept;
```

1 *Effects:* Constructs a new `stop_token` object that can never receive a request to stop. [*Note*: Therefore, no resources have to be associated for the state. — *end note*]

2 *Ensures:* `stop_possible() == false` and `stop_requested() == false`.

```
stop_token(const stop_token& rhs) noexcept;
```

3 *Effects:* If `rhs.stop_possible() == false`, constructs a `stop_token` object that can never receive a request to stop. Otherwise, constructs an `stop_token` that shares the ownership of the stop state with `rhs`.

4 *Ensures:* `stop_possible() == rhs.stop_possible()` and `stop_requested() == rhs.stop_requested()` and `*this == rhs`.

```
stop_token(stop_token&& rhs) noexcept;
```

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

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

30.4.4.2 `stop_token` destructor [stop_token.destr]

```
~stop_token();
```

1 *Effects:* If `*this` is the last owner of the atomic stop state, releases the resources associated with the atomic stop state.

30.4.4.3 `stop_token` assignment [stop_token.assign]

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

30.4.4.4 `stop_token` swap [stop_token.swap]

```
void swap(stop_token& rhs) noexcept;
```

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

30.4.4.5 `stop_token` members [stop_token.mem]

```
[[nodiscard]] bool stop_requested() const noexcept;
```

1 *Returns:* `true` if `request_stop()` was called on and associated `stop_source`, otherwise `false`.
2 *Synchronization:* If `true` is returned then synchronizes with the first call to `request_stop()` on an associated `stop_source`.

```
[[nodiscard]] bool stop_possible() const noexcept;
```

3 *Returns:* `false` if a subsequent call to `stop_requested()` will never return `true`. [*Note*: To return `true` either a call to `request_stop()` on an associated `stop_source` must have already been made or there must still be associated `stop_source` objects in existence on which a call to `request_stop()` could potentially be made in future. — *end note*]

30.4.4.6 `stop_token` comparisons [stop_token.cmp]

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

1 *Returns:* `true` if both `lhs` and `rhs` both have no shared stop state or refer to the same stop state (copied or moved from the same initial `stop_source` object).

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

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 request stops to cooperatively cancel the running thread.

30.5.1 Header <jthread> synopsis [thread.jthread.syn]

```
#include <stop_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 request a stop 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();
    [[nodiscard]] id get_id() const noexcept;
    [[nodiscard]] native_handle_type native_handle();     // see ??

    // stop token handling
    [[nodiscard]] stop_token get_stop_source() const noexcept;
    [[nodiscard]] bool request_stop() noexcept;

    // static members
    [[nodiscard]] static unsigned int hardware_concurrency() noexcept;

  private:
    stop_token ssource;                 // 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 `ssource.stop_possible() == 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)), ssource, *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 `ssource` and constructs an object of type `jthread`. The new thread of execution executes *INVOKE*(*DECAY_COPY*(std::forward<F>(f)), ssource.get_token(),*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()`. `ssource.stop_possible() == true`. `*this` represents the newly started thread. [*Note*: Note that the calling thread can request a stop only once, because it can't replace this stop 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. `ssource` yields the value of `x.ssource` prior to the start of construction and `x.ssource.stop_possible() == false`.

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

```
~jthread();
```

1 If `joinable()`, calls `request_stop()` 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 `request_stop()` 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. `ssource` yields the value of `x.ssource` prior to the assignment and `x.ssource.stop_possible() == false`.

3 *Returns:* `*this`.

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

```
[[nodiscard]] stop_token get_stop_source() const noexcept
```

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

```
[[nodiscard]] bool request_stop() noexcept;
```

2 *Effects:* Equivalent to: `return ssource.request_stop();`

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

  };
}
```

`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 requested 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 requested 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 a stop is requested for the passed `stop_token`. In that case they return (returning `false` if the predicate evaluates to `false`). [*Note*: Because all signatures here call `stop_requested()`, their calls synchronize with `request_stop()`. — *end note*]

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

1 *Effects:* Registers `*this` to get notified when a stop is requested on `stoken` during this call and then equivalent to:

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

2 [*Note*: The returned value indicates whether the predicate evaluated to `true` regardless of whether a stop was requested. — *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,
                  stop_token stoken);
```

6 *Effects:* Registers `*this` to get notified when a stop is requested on `stoken` during this call and then equivalent to:

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

7 [*Note*: There is no blocking, if `pred()` is initially `true`, stoken is not stop_possible, a stop was already requested, or 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 a stop was requested. — *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,
```

```
            stop_token stoken);
```

13 *Effects:* Equivalent to:

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

30.8 Futures [futures]

...