Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0660R1**
Date: 2018-03-10
Reply to: Nicolai Josuttis (nico@josuttis.de), Herb Sutter (hsutter@microsoft.com),
Anthony Williams (anthony@justsoftwaresolutions.co.uk)
Audience: SG1, LEWG, LWG
Prev. Version: P0660R0

A Cooperatively Interruptible Joining Thread, Rev 1

Motivation

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

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

This clear directive was broken.

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

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

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

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

  … followed by full source code for the wrapper.

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

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

  … and provide a full implementation.

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

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

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

New in Rev 1

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

Motivation

For a full discussion of the motivation see P0660R0.

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

Key Design Decisions

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

So we start to introduce a new family of API's, all starting with "i" for "interruptible (one letter is key because programmers should not use std::thread just for the shorter name).

Class `std::ithread`

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

- It is fully compatible to `std::thread` with its whole API except the destructor, so that programmers can easily switch from `std::thread` to this class. The only difference is that the destructor signals interruption and joins instead of just calling terminate() if the thread is still joinable.
- It provides a supplementary API for cooperative interrupts:
  - The calling thread can call `interrupt()` (directly or via an interrupt token).
  - The called thread can (and should) from time to time check for requested interruption (again directly or via an interrupt token).

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

```
namespace this_thread {
  static bool is_interrupted() noexcept;
  static void throw_if_interrupted();
}
```

In general (i.e., for threads started with `std::thread` or `std::async()`, these functions always yield false or do not throw, respectively. For interrupted `std::ithread`'s the functions throw or yield true.

Class std::interrupt_token

`std:ithread` uses a simple helper class **`std::interrupt_token`** to signal interrupts, which

- Is cheap to copy
- can signal an interrupt
- allows to check for an interrupt
- can also remove the signal to interrupt

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

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

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

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

You can check whether the interrupt API can be used:

- bool **ready()**
  - signals whether the interrupt mechanism was initialized

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

- bool **interrupt()**
  - signals an interrupt (and returns whether an interrupt was signaled before)
- bool **is_interrupted()**
  - yields whether an interrupt was signaled yet
- bool **is_interrupted_and_reset()**
  - removes any signaled interrupt and returns whether an interrupt was signaled before
- void **throw_if_interrupted()**
  - throws a std::interrupted exception if an interrupt was signaled (yet).
- void **throw_if_interrupted_and reset()**
  - throws a std::interrupted exception if an interrupt was signaled (yet) after removing the signaled interrupt

The `throw_if_interrupted*()` functions throw a new standard exception class **`std::interrupted`**. It is intentionally ***not*** derived from `std::exception` to not pollute general existing handling of `std::exception`. These exceptions will be automatically be caught by the started `std::ithread` so that they just end the started thread without any warning or error if not caught elsewhere.

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

Provide operator == to check whether two interrupt_tokens use the same interrupt signal or are both not ready?

Implementation Hints

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

- Is pretty cheap to copy (just increments the reference count).
  - If this is not good enough, you can pass it by reference (without any danger provided it is on the stack).

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

A first example implementation is available at: www.josuttis.de/ithread

Why (These) Interruption Tokens?

For a full discussion of the motivation of interrupt tokens in general, see P0660R0.

For simplicity and safety we decided to:

- Use only one type
- Guarantee that the token are reference-counted (valid independent from the lifetime of any thread).

Although, we can think of additional features for the interrupt API such as

- registering one or multiple callbacks,
- holding additional information such as default timeouts for waits

we keep it simple, pretty cheap, and safe.

How to use std::ithread

The basic interface of `std::ithread` supports the following example:

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

Or:

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

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

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

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

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

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

yields true. Alteratively, a checkpoint such as

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

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

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

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

to cheaply pass a token to other places that might interrupt. The tokens are not bound to the lifetime of the ithread (but not to the lifetime of the called thread).

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

How std::ithread uses Interrupt Tokens

A basic bootstrap of the interrupt objects would be:

```
std::interrupt_token interruptor{false};
std::interrupt_token interruptee(interruptor};
…
interruptor.interrupt();
…
// usually after a token is passed to the thread that might get interrupted:
interruptee.throw_if_interrupted();
// and/or:
if (interuptee.is_interrupted()) …
```

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

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

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

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

// THE constructor that starts the thread:
template <typename Callable, typename... Args>
ithread::ithread(Callable&& cb, Args&&... args)
 : _thread_it{interrupt_token{false}},                  // initialize interrupt token
   _thread{[&] (auto&& cb, auto&&... args) {   // called lambda in the thread
                  // pass the interrupt_token to the started thread
                  _this_thread_it = _thread_it;
                  …
           }}
{
}
```

Convenient Interruption Points for Blocking Calls

This API allows to provide the interrupt mechanism as safe inter-thread communication.

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

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

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

> Must include some blocking function support in v1.

SF	F	N	A	SA
3	6	6	3	0

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

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

So, optionally, we proposed the following API:

```
namespace std {
  class condition_variable {
   public:
    template <class Predicate>
      void wait(interrupt_token,
                unique_lock<mutex>& lock, Predicate pred);
    template <class Predicate>
      void iwait(unique_lock<mutex>& lock, Predicate pred);

  };
}
```

The specification would be that:

- the **`wait()`** overload might also have a spurious wakeup if for the passed interrupt token an interrupt was signaled (and the thread was started as an std::ithread)
- the new **`iwait()`** function might in addition throw std::interrupted if an interrupt was signaled (and the thread was started as an std::ithread)

In principle, corresponding overloads/supplements are possible for other blocking/waiting functions.

API of std::ithread

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

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

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

    // construct/copy/destroy:
    ithread() noexcept;
    // THE constructor that starts the thread:
    // - NOTE: should SFINAE out copy constructor semantics
    template <typename Callable, typename... Args,
```

```
                typename = enable_if_t<!is_same_v<decay_t<Callable>, ithread>>>
    explicit ithread(Callable&& cb, Args&&... args);
    ~ithread();

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

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

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

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

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

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

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

We could also add `throw_if_interrupted()` and `throw_if_interrupted_and_reset()` here, but that doesn't seem to be very useful.

Interrupt Handling API

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

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

An example implementation of interrupt_token might look as follows:

```
class interrupt_token {
 private:
  std::shared_ptr<std::atomic<bool>> _ip{nullptr};
 public:
  // default constructor is cheap:
  explicit interrupt_token() = default;
  // enable interrupt mechanisms by passing a bool (usually false):
  explicit interrupt_token(bool b)
   : _ip{new std::atomic<bool>{b}} {
  }
  // interrupt handling:
  bool interrupt() noexcept {
    assert(_ip != nullptr);
    return _ip->exchange(true);
  }
  bool is_interrupted() const noexcept {
    assert(_ip != nullptr);
    return _ip->load();
  }
  bool is_interrupted_and_reset() noexcept {
    assert(_ip != nullptr);
    return _ip->exchange(false);
  }
  void throw_if_interrupted() {
    assert(_ip != nullptr);
    if (_ip->load()) {
      throw ::std::interrupted();
    }
  }
  void throw_if_interrupted_and_reset() {
    assert(_ip != nullptr);
    if (_ip->exchange(false)) {
      throw ::std::interrupted();
    }
  }
};
```

API for Interruptible Blocking Convenience Functions

See above

API for std::this_thread

```
namespace std {
 namespace this_thread {
  static interrupt_token get_interrupt_token() noexcept;
  static bool is_interrupted() noexcept;
  static bool is_interrupted_and_reset() noexcept;
  static void throw_if_interrupted();
  static void throw_if_interrupted_and_reset();
 }
}
```

Names

Of course, the proposal raises several questions about names.

To list some alternatives:

Name used here	Purpose	Alternatives	Remarks
ithread	cooperatively interruptible joining thread	jthread, task	name should be short to support convenient replacement of std::thread
throw_if_interrupted()	throws exception if interruption was signaled	interrupt_point()	
interrupt_token	Checp-to-copy API to deal with interrupts	interrupt_promise, interrupt_token, interrupt_future, interrupt_source	Could be same or different types
iwait(),	global convenient functions using the thread local interrupt future		Naming should be consistent with ithread

In general, to help application programmers, the prefix should always be consistent and not sometimes “interrupt_...” and sometimes “interruptible_...” or “interruptions_...”.

Proposed Wording

(All against N4660)

Full proposed wording at work

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

```
namespace std {
 …
 namespace this_thread {
  static interrupt_token get_interrupt_token() noexcept;
  static bool is_interrupted() noexcept;
  static bool is_interrupted_and_reset() noexcept;
  static void throw_if_interrupted();
  static void throw_if_interrupted_and_reset();
 }
```

Add as a new chapter in parallel to class thread:

33.3.2 Class ithread [thread.ithread.class]

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

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

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

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

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

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

```
    // - supplementary interrupt API:
    interrupt_token get_interrupt_token() const noexcept;
    bool interrupt() noexcept {
      return get_interrupt_token().interrupt();
    }
    bool is_interrupted() const noexcept {
      return get_interrupt_token().is_interrupted();
    }
    bool is_interrupted_and_reset() noexcept {
      return get_interrupt_token().is_interrupted_and_reset();
    }
};
```

Add as a new chapter:

```
namespace std {

 class interrupt_token {
  public:
   explicit interrupt_token();      // cheap non-initialization
   explicit interrupt_token(bool)   // initialization of interrupt mechanism

   bool ready() const;
   bool interrupt() noexcept;
   bool is_interrupted() const noexcept;
   bool is_interrupted_and_reset() noexcept;
   void throw_if_interrupted();
   void throw_if_interrupted_and_reset();
 };
```

Optionally, add in **33.5.3 Class condition_variable [thread.condition.condvar]**

```
namespace std {
  class condition_variable {
   public:
    …
    template <class Predicate>
      void wait(interrupt_token,
                unique_lock<mutex>& lock, Predicate pred);
    template <class Predicate>
      void iwait(unique_lock<mutex>& lock, Predicate pred);
  };
}
```

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

template <class Predicate>
void wait(interrupt_token itok, unique_lock<mutex>& lock, 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`, or `wait_until`) threads.

Effects: Equivalent to:

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

where `mywait(lock)` performs the following:

- 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()` or a call to `notify_all()`, or `itok::is_interrupted()`, or spuriously.

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

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

Throws: `std::interrupted()` or any exception thrown by pred.

```
template <class Predicate>
void iwait(unique_lock<mutex>& lock, Predicate pred);
```

Effects: Equivalent to:

```
wait(std::this_thread::get_interrupt_token(), lock, pred);
```

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, I want to thank: Howard Hinnant, Hans Boehm, Anthony Williams, Herb Sutter, Ville Voutilainen, Jeffrey Yasskin.