Project:	ISO JTC1/SC22/WG21: Programming Language C++
Doc No:	WG21 **P0660R0**
Date:	2017-06-19
Reply to:	Nicolai Josuttis (nico@josuttis.de)
Audience:	Concurrency, LWG
Prev. Version:	

A Cooperatively Interruptible Joining Thread

In Jacksonville 2016 we had an evening session
http://wiki.edg.com/bin/view/Wg21jacksonville/P0206R0
with the following outcome:

> 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

Unfortunately, this decision was reverted later in Oulu in a smaller group so that we didn't get what the majority voted for.
In addition, there are constantly requests to support interruption for started threads, which is implemented in Boost.Thread and was discussed but rejected during the standardization of std::thread for C++11.

It's time now to fulfill the need of an easy to use basic thread class for application programmers based on the experience of the existing Boost thread classes. Not providing this currently hinders projects to switch from Boost to the C++ standard library.

As a side effect, this paper introduces an API for cooperative concurrent/distributed interrupts. It's a solution on the program level, not on the OS level.

Key Proposal

Introduce a class **std::ithread** with the following key features:

- Basic API like std::thread
- 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.
    - Some convenience functions are provided to support this for typical blocking calls.
- The destructor requests for a cooperative end of the started thread if it is still joinable by calling interrupt() and then join().

std:ithread uses the following API for cooperative interrupts between multiple threads:

- **std::interrupt_promise** provides the source to trigger/signal interrupts.
  It allows to ask for
  - an **std::interrupt_token**, which is a cheap to copy object to trigger/signal interrupts from multiple places.
    - It can only be used as long as the interrupt promise is valid.
  - the corresponding **std::interrupt_future**, which provides the API to check whether an interruption was triggered/signaled.
    - When checking for interrupts, an **std::interrupted** exception is thrown, if an interruption was signaled/triggered.

The whole proposal does not require any additional OS support.

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

How to use std::ithread

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

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

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

If `interrupt()` is called, the next check for an interrupt by the started thread with `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. Note however that the tokens are bound to the lifetime of the ithread (but not to the lifetime of the called thread).

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.

How to Implement Interruptions

Boost.Thread already provides an API to interrupt threads cooperatively. But the author, Anthony Williams, commented:

> Boost has interruption because it was proposed for C++0x, and I thought it was a good idea. ... It is still there because some people use it and removing it would break backwards compatibility.
>
> I would not do it this way again, and do not recommend adding it to C++ in the future.

As an alternative he proposes interruption tokens:

> I'm quite liking the idea of an "Interruption Token" which you can pass round. Any thread can trigger the interruption token, which will can then be explicitly checked for interruption via token.check_for_interruption() or similar, or passed into blocking calls to allow those calls to be interrupted. We could therefore have condition_variable::wait() overloads that take interruption_token objects to allow interruption, for example.

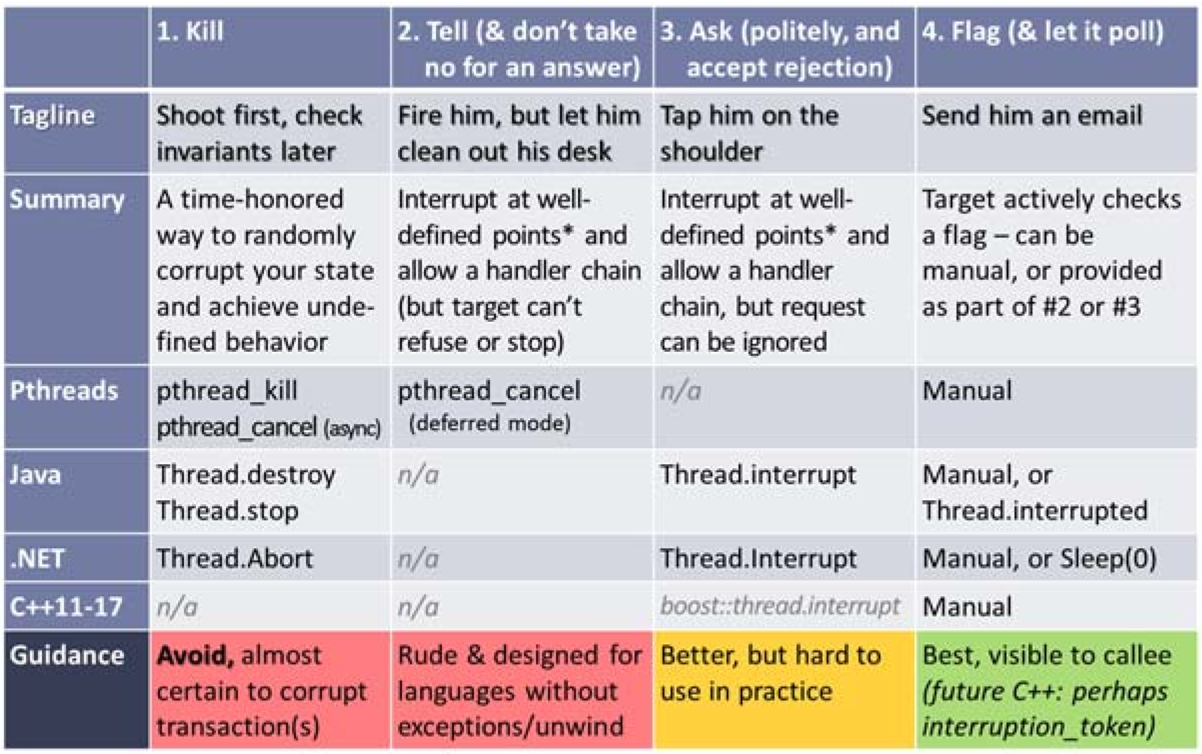
That direction is exactly what this paper proposes.

Why Interruption Tokens?

Herb Sutter elaborated on the interrupt token proposal as follows:

> Let me strongly support Anthony on interruption tokens. It is the state of the art, and the only interruption mechanism I know of that has a chance to get consensus. (FWIW, Microsoft also has existing practice to contribute as PPL provides the same, called cancellation_tokens; example: https://msdn.microsoft.com/en-us/library/dd984117.aspx?f=255&MSPPError=-2147217396 .)

and discussed some alternative options as follows:



	1. Kill	2. Tell (& don't take no for an answer)	3. Ask (politely, and accept rejection)	4. Flag (& let it poll)
Tagline	Shoot first, check invariants later	Fire him, but let him clean out his desk	Tap him on the shoulder	Send him an email
Summary	A time-honored way to randomly corrupt your state and achieve unde-fined behavior	Interrupt at well-defined points* and allow a handler chain (but target can't refuse or stop)	Interrupt at well-defined points* and allow a handler chain, but request can be ignored	Target actively checks a flag – can be manual, or provided as part of #2 or #3
Pthreads	pthread_kill pthread_cancel (async)	pthread_cancel (deferred mode)	*n/a*	Manual
Java	Thread.destroy Thread.stop	*n/a*	Thread.interrupt	Manual, or Thread.interrupted
.NET	Thread.Abort	*n/a*	Thread.Interrupt	Manual, or Sleep(0)
C++11-17	*n/a*	*n/a*	*boost::thread.interrupt*	Manual
Guidance	**Avoid,** almost certain to corrupt transaction(s)	Rude & designed for languages without exceptions/unwind	Better, but hard to use in practice	Best, visible to callee *(future C++: perhaps interruption_token)*

> **Option1**(Asynchronous interruption (e.g., pthread_kill, pthread_cancel async mode, Java Thread.destroy/stop, .NET Thread.Abort): I assume we agree that #1 is totally broken because it is necessarily undefined behavior, these are typically deprecated/banned on the platforms that provide them, including Posix where last time looked the official docs do not deprecate pthread_kill/cancel(async) but Butenhof's book[1] is very clear that there is no such thing as killing or async-cancelling a single thread and these are equivalent to killing the whole process because they lead to memory corruption across the process.
>
> **Options 2 and 3** are similar: Interrupt at well-defined interruption points (e.g., throw from a wait/sleep/join point), ...
>
>> Option 2: ... but the target cannot refuse or stop ending the thread/task (pthread_cancel deferred mode, you get to run installed cancellation handlers but you cannot refuse the request and keep going)
>>
>> Option 3: ... and the target can handle it and continue (Java Thread.interrupt, .NET Threat.Interrupt, boost::thread.interrupt)
>
> This is less bad than #1 because it is not automatic undefined behavior (good!), but I think we know from experience that this is still unusable in practice. The reason is that people keep forgetting to check for those exceptions; no matter how much they are trained, they cannot remember that every wait/sleep/join might throw an InterruptedException/thread_interrupted. The whole .NET Frameworks, written by people who invented their Thread.Interrupt and were disciplined about framework quality and safety, is interruption-unsafe because they could not consistently remember, and will always be because it is prohibitive to get it right and stay clean, which means that essentially all .NET code cannot use interruption safely (because virtually all .NET code uses the frameworks).

[1] Butenhof, D., "Programming with POSIX Threads," Addison Wesley, 1997.

So anything that says "wait [or sleep, or join, etc.] throws an exception if there's an interruption request" is unusable. Even if it had been built in from the beginning it would be hard to use correctly.

But definitely we cannot add it retroactively to the standard library wait/sleep/join points so that these can now throw a new kind of exception that will certainly destabilize code that is not expecting it today. …

That is, we cannot consistently add throwing interrupted_exceptions to our std:: wait/sleep/join operations (interruption points), which is a necessary part of Option 3.

For example:

- Some **future::wait()**/**condition_variable::wait()** overloads are currently guaranteed not to throw. Allowing them to throw interrupted_exception would be a breaking change and typically corrupt the interrupted thread (interrupt an in-progress update before invariants are restored) and therefore in general corrupt the program (at least the parts that can transitively depend on that invariant to do further processing).
- **thread::sleep_for()**/**sleep_until()** is allowed to throw only exceptions thrown by clock, time_point, or duration -- and the ones of those in std:: are guaranteed not to throw. For code that does not use non-std:: clock/time_point/duration, allowing these sleep functions to throw interrupted_exception would turn a non-throwing function into a throwing one, with the same issue as future::wait. For code that uses non-std:: throwing clock/time_point/duration, it adds a new exception not possible today and will break their code if they are catching the specific types of exceptions known to be throwing from the types they are using.
- **thread::join()** is currently guaranteed to throw only system_error. Allowing it to throw interrupted_exception could break code that only catches system_error. (As a workaround we could throw a system_error that wraps interrupted_exception but now we still don't have a consistent model.)

In general, even for wait/sleep/join functions that already can throw, enabling them to throw a new kind of exception can affect code that is checking and thinking about only specific exception types that were advertised before.

Option 4: Interruptee polls explicitly (e.g., Java Thread.interrupted, PPL cancellation_token, or roll your own cooperative mechanism to use a flag/semaphore/condition_variable to ask the task to stop)

This actually works because interruption never happens unless the programmer writing the interruptee code is actively asking about it -- which means that by construction they are thinking about it as they write their code, so they have a great chance of making the code correct for interruption. It avoids #1's UB by avoiding injecting async interruptions, and it avoids #2/3's silent exception injection that programmers can never remember and forget to harden their code against despite years of training.

So that's why I think the interruption token model is the only interruption model that has a chance to be standardized IMO. And it's actually a good one. But anything like #1-2-3 will get shot down because they just do not work in practice (and I will actively help shoot them down; we need to learn from those mistakes and not repeat them).

Hans Boehm commented

I like the alternative being floated here better than the others,

and later:

I now think the token-based solution (which was once proposed by Microsoft, but then I think not followed through on) is better.

API of Interrupt Tokens

There are a couple of design decisions, when implementing an API for interrupt token. In principle many arguments apply as for cancellation tokens, which you e.g. can find in .NET:

https://blogs.msdn.microsoft.com/pfxteam/2009/05/22/net-4-cancellation-framework/

First, it seems to be a good approach to separate between the API to trigger interrupts and the API to check for interrupts. To quote https://blogs.msdn.microsoft.com/pfxteam/2009/05/22/net-4-cancellation-framework/:

> Two new types form the basis of the framework: A *CancellationToken* is a struct that represents a 'potential request for cancellation'. This struct is passed into method calls as a parameter and the method can poll on it or register a callback to be fired when cancellation is requested. A *CancellationTokenSource* is a class that provides the mechanism for initiating a cancellation request and it has a Token property for obtaining an associated token. It would have been natural to combine these two classes into one, but this design allows the two key operations (initiating a cancellation request vs. observing and responding to cancellation) to be cleanly separated. In particular, methods that take only a CancellationToken can observe a cancellation request but cannot initiate one.

But we also have to take into account, which lifetime guarantees we give to the API.

Of course, we can share all necessary information among all potential interrupters and interruptees, implemented with a shared_ptr approach to hold all data until the last interrupt stakeholder dies. But this proposal tried to avoid that to save (heap) resources.

Instead the proposed interrupt API is as follows:

In principle, we create a promise/future pair:

- The promise is for the interrupters
- The future for the interruptee

wrapped by classes that hide (some) implementation details and can carry additional information.

A basic bootstrap of the interrupt objects would be:

```
std::interrupt_promise ip;
std::interrupt_token it = ip.get_interrupt_token();
std::interrupt_future ifut = ip.get_interrupt_future();
…
it.interrupt();
…
// usually after ifut is passed to the thread that might get interrupted:
ifut.throw_if_interrupted();
```

That way:

- The lifetime of all potential interrupters is decoupled from the lifetime of the interruptee.
- interrupt_token allows to cheaply copy and pass the API to interrupt to multiple places.
    - To avoid the need to allocated shared (heap) resources, calling interrupt() from an interrupt token is only valid if the interrupt promise still is alive (the tokens refer to the promise).
    - We could alternatively inside use a shared pointer to the promise. But that better should be an API on top.
- interrupt_promise and interrupt_future only have move semantics.
- Currently there can only be one interruptee, which also avoids the need to allocated shared resources to handle interrupts in multiple places.
  Note however, that we can ask throw_if_interrupted() multiple times (the interrupt_future wraps the future to call get only once).
    - We could allow to use shared futures to be able to copy interuptees, but that's not implemented, yet.

  Note however, that we can ask throw_if_interrupted() multiple times (the interrupt_future wraps the future to ensure that get() is called only once).

Of course we can merge two of the three objects into two:
interrupt_promise and interrupt_token could be the same class. That would mean that we first create an interrupt token and then an interrupt future from it.

Note however, that is exactly the opposite from the bootstrap .NET has, where they first create the "CancellationTokenSource", which is usable as the interruptee, and then create interrupt tokens from it. The order is different here, because we decouple with promise/future and always have to create the promise first.

Thus, we still can decide on the following:

- Should we split into two or three interrupt classes?
- Should interrupt token decouple their lifetime from interruptees?
- Should we support multiple interruptees?
- Should we even decouple lifetime of tokens from promises?

Also we can think of additional features for the interrupt API such as registering one or multiple callbacks.

One addition this proposal has, which we will discuss later, is the ability to define the interval interruptible blocking convenience functions use to check for interrupts.

How std::ithread uses Interrupt Tokens

Class std::ithread would use interrupt tokens internally. Thus, the constructor of a thread would perform the necessary bootstrap to create the API for the interrupter (creating an interrupt_promise) and the interruptee (creating the corresponding interrupt_future and passing it to the calls thread as TLS).

If the interrupt token approach would not support decoupling of interrupter and interruptee, ithread would create the necessary promise/future pair (I started with that and it gets pretty ugly).

That is for a started std::ithread `t`:

- **`t.get_interrupt_token().interrupt()`** calls set_value() for the promise
    - o a convenience function allows to call just: `t.interrupt()`
- The called thread has a static thread_local API to be able to check whether the thread was interrupted.
  For example, **`std::this_thread::throw_if_interrupted()`** calls
    std::ithread::threadLocalInterruptFuture.throw_if_interrupted();

How to define Interruption Points?

As API of threads to check for interruptions the interruptee would use a function such as

```
interruptFuture.throw_if_interrupted()
```

to throw an exception if an interrupt was triggered/signaled.

As written, class std::ithread would map that to a convenient global call:

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

as it stores its interrupt future in its TLS by something like the following:

```
class ithread {
    inline thread_local static ::std::interrupt_future _ifuture;
  public:
    static void throw_if_interrupted() {
      ithread::_ifuture.throw_if_interrupted();
    }
  // …
};
namespace this_thread {
  static void throw_if_interrupted() {
    ithread::throw_if_interrupted();
  }
}
```

In principle, also threads started with std::thread() could call this_thread::throw_if_interrupted(), but as their thread_local interrupt_future isn't valid, the call would have no effect.

Nevertheless, std::thread could use the interrupt API directly to call for a passed interrupt future:

```
myInterruptFuture.throw_if_interrupted()
```

Convenient Interruption Points for Blocking Calls

The question came up, whether we should provide additional places to allow cooperative interruptions because as Hans Boehm wrote:

I really don't want to write the waits with timeouts just to test whether my thread was interrupted.

And Boost already provides the following additional interruption points (see https://www.justsoftwaresolutions.co.uk/threading/thread-interruption-in-boost-thread-library.html):

```
boost::thread::join()
boost::thread::timed_join()
boost::condition_variable::wait()
boost::condition_variable::timed_wait()
boost::condition_variable_any::wait()
boost::condition_variable_any::timed_wait()
boost::this_thread::sleep()
```

The standard has slightly different API's but this defines the general places where the ability to get interrupted probably makes sense.

However as discussed above for **Options 2 and 3** 3, we do not want to change the existing API (including exceptions that can be thrown). Instead, we could overload the existing blocking API's for dealing with interrupt futures and/or provide global helpers.

So, in general, instead of calling

```
obj.wait();
```

to start an interruptible wait programmers would call:

```
obj.wait(interruptFuture);
```

which is roughly implemented as follows:

```
do {
  interruptFuture.throw_if_interrupted();
}
while (obj.wait_for(100ms));
```

One question is, where the check interval (here "100ms") should come from. I suggest that it is part of the interrupt API. That is, when bootstrapping the interrupt objects, you can define the interval. A useful default might be provided. That is, the interruptible wait() would really be implemented as follows:

```
do {
  interruptFuture.throw_if_interrupted();
}
while (obj.wait_for(interruptFuture.get_check_interval()));
```

The same way, even timed waits would start a more fine grained loop.

Issues with Convenience Interruption Points

Note however, that for some cases we have to make semantic design decisions.

- For example, **future::wait()** has an issue:

  For futures, wait() start a deferred thread.
  But if we map that to a loop of wait_for() calls this does NOT start a new thread.
  So, we have to decide what it means to call an interruptible wait for deferred futures.

- Also condition variables require further attention as Hans and Anthony pointed out:

  Hans: Otherwise the only way to check for interruption during a cv wait is by timing out regularly and checking. That's rather ugly. Both Posix and Java interrupt condition variable waits; Posix interrupts a bunch of other blocking calls as well.

  Anthony: If you transfer wait() calls into a loop of wait_for() calls, periodically checking for the interrupt, you may miss notifications that happen after the first wait_for returns, but before the next call starts. If you have a predicate, you can treat it as a spurious wake, and check the predicate, but without a predicate once you've woken from the underlying call, you have to return to the caller.

Fortunately, we already have support for spurious wakeups in standard CVs, by passing the condition predicate or the lock guard to check the condition. Thus, we can provide something like (here globally defined without passing the interrupt future):

```
template <typename CV, typename LG, typename Pred>
void condition_variable::wait(interrupt_future& ifut,
                              CV& cv, LG&& lg, Pred&& pred)
{
    do {
      ifut.throw_if_interrupted();
    }
    while (cv.wait_for(::std::forward<LG>(lg),
                       ifut.get_check_interval(),
                       pred));
}
```

API of std::ithread

Basically, an std::ithread should provide the same interface as std::thread:

```
class ithread {
  public:
    // construct/copy/destroy/swap:
    ithread() noexcept;
    template <class F, class... Args>
      explicit ithread(F&& f, Args&&...   args);
    ~ithread();
    ithread(const ithread&) = delete;
    ithread(ithread&&) noexcept;
    ithread& operator=(const ithread&) = delete;
    ithread& operator=(ithread&&) noexcept;
    void swap(ithread&) noexcept;

    // members:
    bool joinable() const noexcept;
    void join();
    void detach();
    thread::id get_id() const noexcept;
    thread::native_handle_type native_handle();
    //…
};
```

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.

The supplementary API would be as follows:

```
class ithread {
  public:
   // …
   interrupt_token get_interrupt_token();

   bool interrupt() {   // returns whether it was already interrupted
      return get_interrupt_token().interrupt();
   }
};
```

Interrupt Handling API

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

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

Then as a helper the cheap-to-copy API to signal/trigger interrupts:

```
class interrupt_token
{
  public:
    bool interrupt();    // returns whether it was already interrupted
};
```

The basic object to initialize an interrupt context (and needed by interrupt tokens) is defined as:

```
class interrupt_promise
{
  public:
    // constructor (with default convenient interval)
    template <typename D = ::std::chrono::milliseconds>
    interrupt_promise(D ci = ::std::chrono::milliseconds(100));

    // only move support:
    interrupt_promise (const interrupt_promise&) = delete;
    interrupt_promise& operator= (const interrupt_promise&)  = delete;
    interrupt_promise (interrupt_promise&&) = default;
    interrupt_promise& operator= (interrupt_promise&&) = default;
    … // swap() etc.

    // key API:
    ::std::interrupt_future get_interrupt_future();
    ::std::interrupt_token get_interrupt_token();
    bool interrupt() {   // returns whether it was already interrupted
      return get_interrupt_token().interrupt();
    }
}
```

API for interruptees:

```
class interrupt_future
{
  public:
    // constructors and assignments:
    interrupt_future() = default;

    // only move support:
    interrupt_future (const interrupt_future&) = delete;
    interrupt_future& operator= (const interrupt_future&)  = delete;
    interrupt_future (interrupt_future&&) = default;
    interrupt_future& operator= (interrupt_future&&) = default;

    // key API:
    void throw_if_interrupted();     // might throw std::interrupted
    system_clock_duration get_check_interval();  // for blocking convenience funcs
};
```

The only real constructor initializing the object real data is private for the interrupt_promise when calling get_interrupt_future() there.

API for Interruptible Blocking Convenience Functions

As written, we could and should overload all relevant blocking calls by a version taking an interrupt_future as additional first argument.

For example:

```
  class condition_variable {
    public:
    // …

    template <class Predicate>
    void wait(interrupt_future& ifut,
              unique_lock<mutex>& lock, Predicate pred);
    template <class Clock, class Duration, class Predicate>
    bool wait_until(interrupt_future& ifut,
                    unique_lock<mutex>& lock,
                    const chrono::time_point<Clock, Duration>& abs_time,
                    Predicate pred);
    template <class Rep, class Period, class Predicate>
    bool wait_for(interrupt_future& ifut,
                  unique_lock<mutex>& lock,
                  const chrono::duration<Rep, Period>& rel_time,
                  Predicate pred);
};
```

We still have to discuss whether the overloads for the non-predicate version makes sense here.

We could also provide global helpers. For example:

```
      template <typename CV, typename LG, typename Pred>
      void wait(::std::interrupt_future& ifut,
                CV& cv, LG&& lg, Pred&& pred)
      {
          do {
            ifut.throw_if_interrupted();
          }
          while (cv.wait_for(::std::forward<LG>(lg),
                             ifut.get_check_interval(),
                             ::std::forward<Pred>(pred)));
      }

      template <typename CV, typename LG, typename Pred>
      void iwait(CV& cv, LG&& lg, Pred&& pred)
      {
        wait(::std::this_thread::get_interrupt_future(),
             cv,
             ::std::forward<LG>(lg), ::std::forward<Pred>(pred));
      }
```

For the moment I skip all the other overloads and convenience functions, but in principle the implementations look the same.

Note as discussed above we have to decide about semantic details such as how to deal with deferred futures.

API for std::this_thread

For the started thread we also have to provide at least:

```
namespace this_thread {
  void throw_if_interrupted() {
    ithread::throw_if_interrupted();
  }
}
```

The same way we might provide:

- `get_interrupt_future()`
- `get_check_interval()`
- …

and convenient blocking calls such as:

```
namespace this_thread {
  template <class Clock, class Duration>
  void sleep_until(interrupt_future& ifut,
                   const chrono::time_point<Clock, Duration>& abs_time);
  template <class Rep, class Period>
  void sleep_for(interrupt_future& ifut,
                 const chrono::duration<Rep, Period>& rel_time);
  }

  // optionally:
  template <class Clock, class Duration>
  void isleep_until(const chrono::time_point<Clock, Duration>& tp) {
    sleep_until(::std::get_interrupt_future(), tp);
  }
  template <class Rep, class Period>
  void isleep_for(const chrono::duration<Rep, Period>& d) {
    sleep_for(::std::get_interrupt_future(), d);
  }
}
```

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_promise, interrupt_token, interrupt_future		…, interrupt_source, interruptee, …	
iwait(), isleep_until(), isleep_for()	global convenient functions using the thread local interrupt future		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 N4618)

t.b.d.

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.