Document No.	P1287R0
Date	2018-10-08
Reply To	Lewis Baker <lbaker@fb.com> Kirk Shoop <kirkshoop@fb.com>
Audience	SG1, LEWG

Supporting async use-cases for `interrupt_token`

Abstract

The jthread paper (P0660R4) which seeks to introduce a joinable thread, `std::jthread`, also proposes to add a new type `std::interrupt_token` that can be used to interrupt blocking wait operations on a `std::condition_variable`.

The motivation for adding `std::interrupt_token` in the jthread paper seems to be primarily as a mechanism to allow interrupting a thread's blocking operations so that we can safely join the thread in the jthread destructor. The destructor signals the `interrupt_token` and then calls `.join()` on the thread.

However, there is also a need to be able to interrupt operations other than `std::condition_variable::wait()`. With the pending introduction of coroutines, executors and async networking and I/O into the C++ standard there will be a growing number of asynchronous operations that will also need to support being interrupted.

The current interface of `interrupt_token` as proposed provides only the ability to poll for interruption. While this approach can work well for synchronous or parallel computations that are actively executing and that can periodically check for interruption, the polling model does not work well for asynchronous operations that may be suspended, waiting for some operation to complete or some event to occur. Such operations typically need to execute some logic to actively interrupt the operation.

In order to support the use-case of interrupting asynchronous operations there needs to be some way for that operation to subscribe for notification that an `interrupt_token` has been interrupted.

This paper proposes adding the ability to attach multiple, independent callbacks to an `interrupt_token` for the lifetime of an RAII object of type `interrupt_callback<Callback>`.

This paper also seeks to raise concerns about the design of `interrupt_token` acting both as a source of interruption signals and interface for responding to interruption signals. An alternative design is proposed that separates the concerns of signalling interruption from responding to interruption.

Callbacks

For `interrupt_token` to be usable for interrupting asynchronous operations there needs to be some way for those operations to be actively interrupted when interruption is requested.

For example, to interrupt an asynchronous I/O request on Windows you need to call `CancelIoEx()` to request that the OS cancel the operation.

In another example, if a coroutine was asynchronously waiting for a timer to elapse and we wanted to interrupt this operation then we would need to actively perform some action to cancel this timer and promptly resume the coroutine. If the only facility we have in `interrupt_token` is the ability to poll for cancellation then this would mean the timer would need to periodically wake up and check to see if it has been cancelled, increasing both the runtime cost and introducing latency in responding to interruption.

The proposed solution to this is to add the ability to register callbacks that are associated with the `interrupt_token` such that the first thread to call `interrupt_token::interrupt()` then executes all callbacks that were registered at the time inside the call to `interrupt()`.

The executed callbacks are then able to perform some action to actively interrupt whatever operation is currently being waited on.

Proposed API

The proposed change to the API of `interrupt_token` is the introduction of a new `interrupt_callback<Callback>` type that acts as a RAII object that is responsible for registering the callback with the `interrupt_token` on construction and deregistering the callback on destruction.

Synopsis:

```
// <interrupt_token>
```

```
#include <concepts>

namespace std
{
  class interrupt_token;

  // Exposition only
  class __interrupt_callback_base
  {
  public:
    // Type erased call to callback.
    virtual void __call() noexcept = 0;
  };

  template<Invocable Callback>
  class interrupt_callback : private __interrupt_callback_base
  {
  public:
    // Construction registers the callback with the interrupt_token
    interrupt_callback(
      interrupt_token&& it, Callback&& callback);
    interrupt_callback(
      const interrupt_token& it, Callback&& callback);

    // Deregisters the callback from the interrupt_token.
    ~interrupt_callback();

    // Interrupt callbacks are not copyable or movable
    interrupt_callback(interrupt_callback&&) = delete;
    interrupt_callback(const interrupt_callback&) = delete;
    interrupt_callback& operator=(interrupt_callback&&) = delete;
    interrupt_callback& operator=(
      const interrupt_callback&&) = delete;

  private:

    // Exposition only
    virtual void __call() noexcept override
    {
      callback_();
    }

    // Exposition only
```

```
    interrupt_token it_;
    Callback callback_;
  };

  template<typename Callback>
  interrupt_callback(interrupt_token&&, Callback&&)
    -> interrupt_callback<Callback>;

  template<typename Callback>
  interrupt_callback(const interrupt_token&, Callback&&)
    -> interrupt_callback<Callback>;
}
```

Example usage:

Example: Cancelling a timer

```
#include <interrupt_token>

task<void> async_sleep(
  std::chrono::milliseconds duration,
  std::interrupt_token it)
{
  auto timer = timer::create(duration);

  // Register a callback to be run if interruption is requested.
  // Callback template parameter is deduced due to deduction
guides.
  std::interrupt_callback cb{ it, [&timer] { timer.cancel(); } };

  // Asynchronously wait for time to elapse.
  // This will resume early if timer.cancel() is called.
  co_await timer;

  // Callback automatically deregistered at end of scope.
}
```

Semantics:

`bool std::interrupt_token::interrupt() noexcept;`
Effects: If `!valid() || is_interrupted()` the call has no effect. Otherwise, signals an interrupt so that `is_interrupted() == true` and then executes the callbacks of all `interrupt_callback` objects currently associated with the same `interrupt_token` state (`interrupt_token` objects copied or moved from the same initial `interrupt_token` object).

Ensures: `!valid() || is_interrupted()`
Returns: The value of `is_interrupted()` prior to the call.

```
template<typename Callback>
std::interrupt_callback<Callback>::interrupt_callback(
  interrupt_token&& it,
  Callback&& callback);
template<typename Callback>
std::interrupt_callback<Callback>::interrupt_callback(
  const interrupt_token& it,
  Callback&& callback);
```

Effects: Copies '`callback`' into the `interrupt_callback` object. If `it.is_interrupted()` is true on entry to the constructor then invokes the callback in the current thread before the constructor returns. Otherwise, associates this callback with the `interrupt_token` so that if some thread subsequently calls `it.interrupt()` then this callback object will be invoked.
Synchronisation: Guarantees that, if there is a concurrent call to `it.interrupt()` on another thread that either the other thread will 'synchronise with' the callback registration and will execute the callback before the call to `it.interrupt()` returns, or this thread will 'synchronise with' the call to `it.interrupt()` and will execute the callback inline before the constructor returns.
Exceptions: Throws any exception thrown by Callback's move-constructor or `std::bad_alloc` if memory could not be allocated for the callback registration. The callback will not be invoked or have been invoked if the call to the constructor exits with an exception.

```
template<typename Callback>
std::interrupt_callback<Callback>::~interrupt_callback();
```

Effects: Deregisters the callback from the associated `interrupt_token`. A subsequent call to `.interrupt()` on the `interrupt_token` will not execute this callback.
Synchronisation: If another thread has made a concurrent call to `interrupt_token::interrupt()` then either the callback will be deregistered prior to the other thread executing the callback and the callback will not run, or the other thread will execute the callback and the destructor will not return until after the callback has finished executing.
Exceptions: None. If the `Callback` destructor throws an exception then `std::terminate()` is called.

Design Discussion

Why do we need to support multiple callbacks?

There were some suggestions at the ad-hoc Executors meeting in Bellevue that we may be able to get away with an implementation that supports only a single callback if we can have the

callbacks chain execution on to each other. eg. if the most recently registered callback executes the previously registered callback, and so on until the end of the chain is reached.

This design would be sufficient if callbacks were registered and never deregistered or if callbacks were only ever registered with strictly nested lifetimes. However, as we will often want to temporarily register an interrupt callback only for the duration of an operation this rules out the first case. And since an `interrupt_token` is able to be copied, it can potentially be passed to multiple functions or coroutines that execute concurrently and thus the lifetimes of the callback registrations may not be strictly nested.

Note that use-cases of `interrupt_token` with coroutines could conceivably need to handle many thousands to millions of concurrent interruptible operations registering callbacks with an interrupt token at any one point in time.

Where do the callbacks run?

There are several options for where the callbacks should execute. It could be in some runtime-defined thread (eg. using `std::async()`), or we could require that the callback registration provide an executor that will execute the callback, or it could be on the thread that calls `.interrupt()`.

The simplest solution, and the solution proposed by this paper, is to execute the callbacks inline within the first call to `.interrupt()` on the `interrupt_token`.

If there multiple calls to `.interrupt()` then only the first call will execute the callback. Subsequent calls to `.interrupt()` will be a no-op and will return immediately.

Avoiding heap allocations for callbacks

When a client of the `interrupt_token` wants to register a callback, the invocable object/function-pointer for the callback needs to be stored somewhere and type-erased so that the `interrupt_token::interrupt()` implementation can call it.

By having the `interrupt_callback` object be templated on the callback-type it can then store the callback object inline inside the `interrupt_callback` object along with any other implementation-specific state required to be able to register the callback with the `interrupt_token`. This can avoid the need to store the callback on the heap (eg. as it might if using `std::function<void()>` to store the callback).

One possible implementation that would not require any additional heap-allocations to register a callback would be to store two additional pointers in the `interrupt_callback` object and have the constructor insert itself into an intrusive doubly-linked list in the `interrupt_token` state (using appropriate synchronisation).

Another possible implementation is the one used by the cppcoro[1] library's `cancellation_token` type. This implementation allocates a pool of pointers to callback objects internally in the `interrupt_token` state and then uses a lock-free algorithm to atomically register the callback object with a slot in this pool. This allows lock-free registration and (mostly) lock-free deregistration of callbacks. Deregistering a callback will block only when there is a concurrent call to .interrupt() on another thread.

Memory used by this pool of pointers is not reclaimed until the last `interrupt_token` referencing the shared-state is destroyed, however, this memory is reused by subsequent callback registrations.

Other implementation strategies that make use of deferred reclaimation[2] techniques may be possible here. This is an area for future research.

Overhead of supporting callbacks

The overhead of supporting callbacks in the `interrupt_token` interface in the case where callbacks are never registered can be made negligible for polling-only use-cases.

For example, the shared state in the cppcoro implementation looks like this:

```
 struct cancellation_state
 {
   // Bit 0 - cancellation requested
   // Bit 1 - cancellation notification complete
   // Bits 2-32 - Ref count for cancellation_source objects
   // Bits 33-63 - Ref count for cancellation_token objects
   std::atomic<std::uint64_t> m_state;
   std::atomic<cancellation_registration_state*>
m_registrationState;
 };
```

The allocation of the data-structure used to store callbacks is deferred until the first callback is registered. If you never register a callback against the `interrupt_token` then the storage overhead is simply that of a single pointer.

This means that for polling-only use-cases of `interrupt_token` the implementation can be made as space efficient as a `std::shared_ptr<std::atomic<bool>>`.

---

[1] https://github.com/lewissbaker/cppcoro

[2] See Hazard Pointers and RCU facilities proposed in P0566 and P1122.

What happens if callbacks throw an exception?

If a callback that was registered with the `interrupt_token` were to throw an exception when called then we need to decide what happens with that exception.

If the exception were to propagate out of the call to `interrupt_token::interrupt()` that was executing the callback then this could have the potential to introduce the exception into a context that is remote from and knows nothing about how to handle the exception.

Further, this raises the question of what the `interrupt()` implementation should do if there were multiple callbacks to execute and the first callback threw an exception. Ideally it should continue notifying the other registered callbacks as otherwise their operations would not be interrupted. But then what if those other callbacks also throw an exception? Should the exceptions be aggregated and rethrown once all callbacks had been executed?

The simplest solution here seems to be to require that the callback not throw an exception. If the implementation of a callback that is attempting to interrupt some operation that can potentially fail then it seems logical to have that callback handle that error and either do it's best to interrupt the operation or call `std::terminate()` if such failure was fatal or unrecoverable.

The proposed semantics, then, is to require that the callback be invoked in a noexcept context so that if the callback did throw an exception then this would result in `std::terminate()` being called.

Risks of introducing callbacks

One of the risks of adding support for callbacks to the design of `interrupt_token` is that it means that code that registers a callback can potentially introduce execution of arbitrary code into the execution context of another thread which calls `.interrupt()` on the token.

Without care, it can be easy for this to potentially introduce deadlocks if the implementation of those callbacks are not lock-free. For example, if a callback tries to acquire a mutex lock but the calling thread already holds the mutex lock while calling `.interrupt()` then this will lead to a deadlock. The example implementation of an interruptible condition_variable::wait_for().

For code where this is a potential problem, the calling code may need to execute the callbacks on a separate context. This can be done by scheduling the invocation of the callbacks to a different execution context. eg. using an executor.

There are two approaches to this:

- The caller can schedule the call to `.interrupt()` onto an executor.
  eg. `executor.defer([token] { token.interrupt(); });`
  This will schedule once and then execute each of the callbacks in-turn on the executor.

- The individual callbacks that are not lock-free can schedule execution onto another execution context in the event they would need to block inside the callback. eg.

```
interrupt_callback cb{ token, [executor, &] {
  executor.defer([&] { /* interruption logic here */ });
}};
```

Both approaches are possible with the proposed design. However, without higher-level knowledge of the context in which `.interrupt()` will be called, callback implementations will generally need to act defensively and use the second approach to ensure they are lock free.

Interruptible `condition_variable::wait_for()` using callbacks

The jthread paper[3] proposes to add overloads of `condition_variable` wait operations that can be interrupted via an `interrupt_token`.

There are two main strategies to implementing an interruptible `condition_variable::wait()`. The first is to have the wait operation periodically wake up and check the state of the `interrupt_token` to see if it has been interrupted. The second is to register a list of `condition_variable` operations to be interrupted with the `interrupt_token` so that the call to `.interrupt()` can actively interrupt them.

An implementation that wanted to use the second approach would need to create a mechanism internally that was effectively equivalent to registering callbacks. The implementation could potentially be optimised as it could assume that the callbacks are only ever interrupting `condition_variable` wait operations, but it would still need to store a collection of operations to interrupt.

Instead, if we were to make this callback facility a part of the public `interrupt_token` API then we could build the interruptible `condition_variable::wait_for()` operation on top of this callback facility and the existing standard library public interface.

This would also make it possible to later extend the ability to interrupt other kinds of operations in the standard library (eg. `std::semaphore::acquire()`, `std::mutex::lock()`) without needing to modify the implementation of `interrupt_token` and worry about ABI compatibility/breakage.

Below is a potential implementation of an interruptible `condition_variable::wait_for()` operation that uses `interrupt_callback` and the public `condition_variable` APIs. Note that particular standard library implementations may be able to provide a more efficient implementation by taking advantage of internal implementation details.

---

[3] P0660R4 jthread paper presented at Bellevue 2018

A correct implementation is subtle and difficult to get right due to the potential for either introducing deadlock or missed wake-ups. For this reason, it is important that the standard library provide interruptible implementations of these methods rather than relying on users to implement them correctly using `interrupt_callback` themselves.

Example: A potential implementation of interruptible `condition_variable::wait_for()` using `interrupt_callback`.

```
 template<typename _Pred>
 bool std::condition_variable::wait_for(
   std::unique_lock<std::mutex>& __lock,
   std::interrupt_token __it,
   _Pred __pred)
 {
   // Don't incur overhead of interrupt_callback if we can never be
   // interrupted.
   if (!__it.valid())
   {
     return this->wait(__lock, std::move(__pred));
   }

   if (__it.is_interrupted()) return false;
   if (__pred()) return true;

   enum class _State {
     __waiting,
     __running,
     __interrupting,
     __interrupted
   };

   std::atomic<_State> __state{ _State::__running };
   std::mutex& __mutex = *__lock.mutex();

   std::future<void> __asyncNotification;

   std::interrupt_callback __cb{ __it,
     [&__mutex, &__state, &__asyncNotification, this] noexcept
     {
       if (__state.exchange(_State::__interrupting) ==
             _State::__waiting)
       {
         // We need to make sure that we acquire the mutex lock
before
```

```
          // signalling the condition_variable via notify_all() so
that
          // we are sure that the thread calling wait_for() has
          // been enqueued onto the condition_variable's wait-list.
          // otherwise, the call to notify_all() may not wake up the
          // wait_for() thread.
          //
          // However, if the thread that called __it.interrupt()
          // currently holds the mutex and is executing the callback
          // inline then it is undefined behaviour if we attempt to
          // acquire the lock again in the same thread.
          // So to guard against this we need to defer waiting for
the
          // mutex to be acquired to another thread.
          //
          // Implementations that are able to determine whether the
          // current thread holds the mutex lock may be able to
          // avoid this step and just call this->notify_all()
          // in this case instead.

          // Assign the resulting future to a variable in the scope
          // of the wait_for() function so that the wait_for()
          // function will not exit until the async notification
          // operation has finished executing.
          __asyncNotification = std::async(
            std::launch::async,
            [this, &__mutex] noexcept
            {
              std::lock_guard __lock{ __mutex };
              // Notify the wait_for() function that we have acquired
              // the mutex and that we will run to completion without
              // further blocking.
              __state.store(_State::__interrupted);
              this->notify_all();
            });
        }
      }
    };

    while (true)
    {
      // Transition from __running to __waiting before we call
wait().
```

```
    auto __oldState = __state.exchange(_State::__waiting);
    if (__oldState == _State::__interrupting)
    {
      // Interrupt callback has executed and transitioned the
      // state to __interrupting. Since it will have seen our
      // our previous state as __running it will not have
      // attempted to acquire the mutex and thus will eventually
      // run to completion.
      // This means that it is safe to exit this function and
      // detach the callback which will wait for the callback to
      // finish executing.
      return false;
    }

    assert(__oldState == _State::__running);

    this->wait(__lock);

    __oldState = __state.exchange(_State::__running);
    if (__oldState == _State::__interrupting)
    {
      // Callback thread has run and saw our state as __waiting and
      // so is now currently waiting to acquire the mutex lock so
      // that it can wake us up.
      //
      // We can't let the '__cb' or '__asyncNotification'
destructor
      // run while we hold the lock since we will block waiting for
      // the callback to run to completion first which will
deadlock.
      //
      // Instead, we voluntarily release the mutex lock here to
allow
      // the callback to acquire the mutex and we wait until it has
      // signalled that it has acquired the mutex by setting the
      // state to __interrupted and waking us up again.
      this->wait(__lock, [&__state]
        {
          // State will only be modified by callback while it is
          // holding the mutex now so it is safe to use relaxed
          // memory order.
          return __state.load(std::memory_order_relaxed) ==
            _State::__interrupted);
```

```
          });
        return false;
      }
      else if (__oldState == _State::__interrupted)
      {
        // Callback has executed and acquired the mutex and will
        // continue to run to completion (if it hasn't already).
        // Safe to let the '__cb' destructor run and block waiting
for
        // the callback to finish executing.
        return false;
      }
      else if (__pred())
      {
        // We have set the state to __running. If the interrupt
        // callback does run concurrently then it will see the
        // __running state and will run to completion without
        // waiting. So it's safe to return and deregister the
        // callback here.
        return true;
      }
    }
  }
```

Splitting Signalling and Responding to Interruption

The jthread proposal proposes a single type that can be used for both signalling interruption by calling the `token.interrupt()` method as well as responding to interruption by querying token.

The concern with this design is that it becomes difficult to isolate responsibility for signalling interruption from those operations that only need to respond to interruption. It is not possible to pass an `interrupt_token` into an opaque interruptible function and guarantee that it will not call `.interrupt()` on that token.

For example, say we had an interruptible operation and we wanted to cancel the operation after a certain timeout had elapsed, or when the `interrupt_token` passed into this function was interrupted.

Example: A naive implementation of a timeout.

```
task<void> do_work(interrupt_token itoken);
```

```
task<bool> timeout_example(interrupt_token itoken)
{
  auto [_, result] = co_await when_all(
    [&]() -> task<>
    {
      // Interrupt do_work() when async_sleep() completes.
      scope_guard interruptOnExit = [&] { itoken.interrupt(); };
      co_await async_sleep(100ms, itoken);
    }(),
    [&]() -> task<>
    {
      // Interrupt async_sleep() when do_work() completes.
      scope_guard interruptOnExit = [&] { itoken.interrupt(); };
      co_await do_work(itoken);
    });

  co_return result;
}
```

The problem with this approach is that this function has now called `.interrupt()` on the `interrupt_token` that was passed in to it. This means that if that same interrupt token had been (or will be) passed into some other operation then it too would also now be interrupted.

```
task<void> higher_level_call(interrupt_token itoken)
{
  bool succeeded = co_await timeout_example(itoken);
  if (!succeeded) {
    // Whoops! 'itoken' will now be in the interrupted state.
    // So our fallback code will now be immediately interrupted
    // even though our caller did not request interruption.
    co_await do_something_else(itoken);
  }
}
```

This makes it difficult to reason about the interruption behaviour of code as any opaque function that you pass an `interrupt_token` into may potentially call `.interrupt()` on that token.

Proposed API

If we want to prevent code that we pass an `interrupt_token` into from signalling the interrupt then we need to split the facilities for signalling an interrupt from the facilities from responding to an interrupt request so that we can pass an object that allows the function to respond to interruption but that does not allow signalling an interrupt.

The following interface is proposed as an alternative:

```
namespace std
{
  class interrupt_token
  {
  public:
    interrupt_token() noexcept; // can never be interrupted
    interrupt_token(const interrupt_token&) noexcept;
    interrupt_token(interrupt_token&&) noexcept;
    ~interrupt_token();
    interrupt_token& operator=(interrupt_token&&) noexcept;
    interrupt_token& operator=(const interrupt_topken&) noexcept;

    bool is_interrupted() const noexcept;
    bool is_interruptible() const noexcept;

    // Comparison operators/swap() omitted for brevity
  };

  class interrupt_source
  {
  public:
    interrupt_source(); // constructs a new shared-state
    interrupt_source(const interrupt_source&) noexcept;
    interrupt_source(interrupt_source&&) noexcept;
    interrupt_source& operator=(const interrupt_source&) noexcept;
    interrupt_source& operator=(interrupt_source&&) noexcept;

    bool is_interrupted() const noexcept;
    bool is_interruptible() const noexcept;

    // Request interruption of all interrupt_token objects obtained
    // via get_token() or by copying a token returned from
    // get_token().
    bool interrupt() const noexcept;

    interrupt_token get_token() const noexcept;

    // Comparison operators/swap() omitted for brevity
  };

  template<typename Callback>
```

```
  class interrupt_callback
  {
    // as per "Callbacks" section above.
  };
}
```

The `timeout_example()` revisited

With this interface used in conjunction with the `interrupt_callback` facility from the first section of this paper it now becomes possible to safely pass an `interrupt_token` into a function without needing to worry about whether that function could potentially signal an interrupt on the token passed in (it cannot).

Example: The above `timeout_example()` modified to work with `interrupt_source`

```
task<> do_work(interrupt_token it);

task<bool> timeout_example(interrupt_token it)
{
  // Introduce a new interrupt_source that can be used to interrupt
  // child operations independently of operations outside of this
  // scope.
  interrupt_source interrupter;

  // Attach a callback to the incoming interrupt_token so that
  // the child operations are interrupted if 'it' is interrupted.
  interrupt_callback cb{ it, [&] { interrupter.interrupt(); }};

  auto [_, result] = co_await when_all(
    [&]() -> task<>
    {
      // Interrupt do_work() once async_sleep() completes.
      scope_guard interruptOnExit = [&] { interrupter.interrupt();
};
      co_await async_sleep(100ms, interrupter.get_token());
    }(),
    [&]() -> task<bool>
    {
      // Interrupt async_sleep() once do_work() finishes.
      scope_guard interruptOnExit = [&] { interrupter.interrupt();
};
      co_return co_await do_work(interrupter.get_token());
    }());
```

```
  co_return result;
}
```

By splitting the two ends of the interruption signal into distinct types, we are forcing the implementation of the `timeout_example()` function to explicitly introduce a new interruption scope by creating a new `interrupt_source` object on which they can call `.interrupt()`.

This encourages writing functions that are better-encapsulated and isolated from each other with respect to interruption.

An optimisation opportunity with `is_interruptible()`

The proposed API above includes an `is_interruptible()` method on both the `interrupt_token` and on the `interrupt_source` objects.

The `is_interruptible()` method replaces the existing `valid()` method on `interrupt_token` that was proposed in the jthread paper but generalises it to also cover detecting the case where an `interrupt_token` was originally constructed from a valid `interrupt_source` but where there are no longer any more `interrupt_source` objects that reference the shared interrupt state and thus there is no possibility of anything signalling an interrupt.

This will allow code to execute a fast-path that avoids overheads necessary to handle interruption in the case that there are no more `interrupt_source` objects that could signal interruption in addition to the case where a default-constructed `interrupt_token` was passed in (which was a case already handled well by `valid()`).

Naming / Bikeshedding

This paper proposes the names `interrupt_source` and `interrupt_callback` for the proposed new abstractions in addition to the existing `interrupt_token` abstraction proposed in the jthread paper.

These names should be considered as initial suggestions and are open for discussion.

If the committee decides to pursue the modifications proposed in this paper then it may be worthwhile to re-evaluate the naming of these abstractions in light of the additional facilities.

A (non-exhaustive) list of alternatives to be considered:

- `interrupt_token` `interrupt_source` `interrupt_callback`
- `interrupt_receiver` `interrupt_signaller` `interrupt_subscription`

- `cancellation_token cancellation_source cancellation_registration`

Conclusion

There is great value in having a general purpose vocabulary type that can be used for signalling and responding to interrupt requests within the standard library.

However, the current design of `interrupt_token` as proposed in the jthread paper D0660R5 does not address the needs for interruptible asynchronous operations, such as asynchronous I/O, that require active steps to be taken to interrupt them.

Further, the lack of separation between the interfaces for signalling an interrupt and responding to an interrupt request makes it difficult and error-prone to write well-encapsulated and isolated interruptible operations.

This paper proposes a facility that allows callbacks to be registered with an `interrupt_token` for the scope of an operation by constructing an `interrupt_callback` object.

This paper also proposes splitting the responsibilities of signalling an interrupt and responding to an interrupt into two separate classes; `interrupt_source` and `interrupt_token`, to allow developers to write interruptible functions that are more likely to be correct and easier to reason about.

The committee is encouraged to consider the enhancements put forward in this proposal in conjunction with the jthread paper P0660R5.