Toward Boost Async 0.2

Vicente J. Botet Escriba

Table of Contents

> **Warning**
>
> Async is not a part of the Boost libraries.

Overview

Description

Boost.Async is a C++ library to allow the calling of functions and functors in an asynchronous manner, thereby making it easier to improve the level of concurrency and parallelism in your applications. It provides:

- An asynchronous execution framework working with `AsynchronousExecutor` and `AsynchronousCompletionToken`. It includes some generic functions and several `AsynchronousExecutor` and `AsynchronousCompletionToken`:
  - fork and fork_all to execute asynchronously functions
  - fork_after: request an `AsynchronousExecutor` to execute a function asynchronously once each one of `AsynchronousCompletionToken` in the dependency tuple parameter are ready. It is similar to the async_with_dependencies proposed Peter Dimov.

  - generic get, join, ... free functions to synchronize on an `AsynchronousCompletionToken`
  - generic get_all, join_all, ... free functions to synchronize on multiple `AsynchronousCompletionToken`
  - generic wait_for_all, wait_for_any to execute asynchronously functions and wait for the completion of all or any of them.
- Some `AsynchronousExecutor` and `AsynchronousCompletionToken` models
  - immediate executors: executes synchronously a function on the current thread. Often used for test purposes
  - basic_threader: can be seen as a thread factory executing asynchronously a function on the returned thread.
  - launchers: Lanchers can be seen as a future factory executing asynchronously a function on a hidden thread.
  - threader/joiner: A Threader runs a unary function in its own thread. A Threader can be seen as a Joiner factory executing asynchronously a function on a thread encapsulated on the returned Joiner. The joiner is used to synchronize with and pick up the result from a function or to manage the encapsulated thread.
  - `tp::pool` and `tp::task` customization as an `AsynchronousExecutor` and an `AsynchronousCompletionToken` respectively. `tp::pool` can be seen as a `tp::task` factory executing asynchronously a function on a pool of threads.
  - a generic asynchronous_executor_decorator which allows to decorate the function to be evaluated asynchronously.

References

- The threader-joiner classes are based on the original idea of Kevlin Henney N1833 - Preliminary Threading Library Proposal for TR2

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by `<>` to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Async
#include <boost/async/async.hpp>

// Create a namespace aliases
namespace basync = boost::async;
```

Motivation

Asynchronous Executors and Asynchronous Completion Token Handles

In N1833 - Preliminary Threading Library Proposal for TR2 Kevlin Henney introduces the concept of `threader` an asynchronous executor and a function `thread()` that evaluates a function asynchronously and returns an asynchronous completion token `joiner`, able to join but also to get the value of the function result.

In N2185 - Proposed Text for Parallel Task Execution Peter Dimov introduces a `fork()` function able to evaluate a function asynchronously and returns a `future` handle.

In N2276 - Thread Pools and Futures Anthony William introduces `launch_in_thread` and `launch_in_pool` function templates which evaluates a function asynchronously either in a specific `thread` or a thread pool and returns a `unique_future` handle.

In Boost.ThreadPool Oliver Kowalke proposes a complete implementation of a thread `pool` with a `submit()` function which evaluates a function asynchronously and returns a `task` handle.

Behind all these proposals there is a concept of asynchronous executor, fork-like function and the asynchronous completion token handle.

Table 1. AE/ACT/fork-like relationship

Proposal	executor	fork-like	ACT handle
Boost.Thread	`thread` class	`thread()` constructor	`boost::thread`
Boost.ThreadPool	`tp::pool`	`submit()`	`tp::task`
N2276	`boost::thread`	`launch_in_thread()`	`unique_future<T>`
N2276	thread_pool	`launch_in_pool()`	`unique_future<T>`
N2185	thread pool	`fork()`	`future<T>`
N1833	`threader`	`thread()`	`joiner<T>`

The asynchronous completion token models can follow two interfaces, the thread interface and the future interface. Some asynchronous completion tokens handle allow to recover the result of the evaluation of the function, others allow to manage the underlying thread of execution.

It seems natural to make a generic __fork__ function that will evaluate a function asynchronously with respect to the calling thread and returns an ACT handle. The following meta-function associates an ACT handle to an asynchronous executor.

```
template <typename AE, typename T>
struct asynchronous_completion_token {
    typedef typename AE::template handle<T>::type type;
};
```

The result of forking a nullary function by an asynchronous executor is given by the metafunction result_of::fork<AE,F>

```
namespace result_of {
    template <typename AE,typename F>
    struct __c_fork__ {
        typedef typename boost::result_of<F()>::type result_type;
        typedef typename asynchronous_completion_token<AE, result_type>::type type;
    };
}
```

The default implementation of fork delegates on fork asynchronous executor function.

```
template< typename AE, typename F >
typename result_of::fork<AE, F>::type fork( AE& ae, F fn ) {
    return ae.fork(fn);
}
```

Forking n-ary functions rely on the nullary version and bind.

```
template< typename AE, typename F, typename A1, ..., typename An >
typename asynchronous_completion_token<AE,
            typename boost::result_of<F(A1,..., An)>::type >::type
fork( AE& ae, F fn, A1 a1, ..., An an ) {
    return ae.fork( bind( fn, a1, ..., an ) );
}
```

We can define a basic_threader which just returns a new thread as follows:

```
class basic_threader {
public:
    template <typename T>
    struct handle {
        typedef boost::thread type;
    };

    template <typename F>
    boost::thread fork(F f) {
        thread th(f);
        return boost::move(th);
    }
};
```

The library includes also a launcher class that creates a thread and returns a unique_future when forking

```
class launcher {
public:
    template <typename T>
    struct handle {
        typedef unique_future<T> type;
    };
    template <typename F>
    unique_future<typename result_of<F()>::type>
    fork(F f) {
        typedef typename boost::result_of<F()>::type result_type;
        packaged_task<result_type> tsk(f);
        unique_future<result_type> res = tsk.get_future();
        thread th(boost::move(tsk));
        return res;
    }
};
```

and a shared_launcher class that creates a thread and returns a shared_future when forking.

Given the sequential example:

```
double f( double a, int n )
{
    double r = 0.0;

    for( int i = 1; i <= n; ++i )
    {
        double x = 1.0 / i;
        r += std::pow( x, a );
    }

    return r;
}

int main()
{
    double m1 = f( 1.0, 1000000 );
    double m2 = f( 1.0, 5000000 );
    double m3 = f( 2.2, 1000000 );
    double m4 = f( 2.2, 5000000 );

    std::cout << m2 - m1 + m3 - m4 << std::endl;
}
```

The library allows a programmer to switch to parallel execution as follows:

```
int main()
{
    launcher l;
    boost::unique_future<double> fm1 = basync::fork( l, f, 1.0, 1000000 );
    boost::unique_future<double> fm2 = basync::fork( l, f, 1.0, 5000000 );
    boost::unique_future<double> fm3 = basync::fork( l, f, 2.2, 1000000 );
    boost::unique_future<double> fm4 = basync::fork( l, f, 2.2, 5000000 );

    std::cout << fm2.get() - fm1.get() + fm3.get() - fm4.get() << std::endl;
}
```

The question now is how we can adapt the example to an existing asynchronous executor such as the Boost.ThreadPool library. We need to specialize the template class asynchronous_completion_token to state which is the `AsynchronousCompletionToken` associate to the `tp::pool`.

```
namespace boost { namespace async {

template <typename Channel, typename T>
struct asynchronous_completion_token<boost::tp::pool<Channel>,T> {
    typedef boost::tp::task<T> type;
};

}}
```

and also to specialize the fork function as the default one requires a fork member function and `tp::pool` provides a `submit()` member function`

```
namespace boost { namespace async {

template< typename Channel, typename F >
result_of::fork<boost::tp::pool<Channel>, F>::type
fork<boost::tp::pool<Channel>,F>( boost::tp::pool<Channel>& ae, F fn ) {
    return ae.submit(fn);
}
}
}
```

Evidently these specializations must be done on the `boost::async` namespace.

As the preceding is ilegal in C++03 we need to use an auxiliary class to define the default behaviour of the fork function

```
namespace partial_specialization_workaround {
    template< typename AE, typename F >
    struct fork {
        static typename result_of::fork<AE,F>::type apply(AE& ae, F fn ) {
            return ae.fork(fn);
        }
    };
}
template< typename AE, typename F >
typename result_of::fork<AE,F>::type
fork( AE& ae, F fn ) {
    return partial_specialization_workaround::fork<AE,F>::apply(ae,fn);
}
```

And specialize partially the partial_specialization_workaround::fork class

```
namespace boost { namespace async {
    namespace partial_specialization_workaround {
        template< typename Channel, typename F >
        struct fork<boost::tp::pool<Channel>,F> {
            static typename result_of::fork<boost::tp::pool<Channel>, F>::type
            apply( boost::tp::pool<Channel>& ae, F fn ) {
                return ae.submit(fn);
            }
        };
    }
}}
```

Note that only the _fork_ function needs to be specialized. The library provides the other overloadings.

We can write the preceding main function in a more generic way

```
template < typename AE>
void do(AE& ae)
{
    typedef basync::result_of::fork<AE, int(*)(double, int) >::type auto_type;
    auto_type fm1 = basync::fork(ae, f, 1.0, 1000000 );
    auto_type fm2 = basync::fork(ae, f, 1.0, 5000000 );
    auto_type fm3 = basync::fork(ae, f, 2.2, 1000000 );
    auto_type fm4 = basync::fork(ae, f, 2.2, 5000000 );

    std::cout << fm2.get() - fm1.get() + fm3.get() - fm4.get() << std::endl;
}

int main()
{
    launcher ae;
    do(ae);
}
```

and we can switch from using the launcher or the tp::pool just by changing one line

```
int main()
{
    boost::tp::pool<> ae(boost::tp::poolsize(6))
    do(ae);
}
```

Instead of defining a type, the user can make use of BOOST_AUTO once the associated files included on the threadpool sub-directory.

```
BOOST_AUTO(fm1, basync::fork(ae, f, 1.0, 1000000 ));
```

As a extreme case the library provides a immediate executor which allows to execute synchronously the function on the current thread. This can be used for test purposes. Note that this executor can not be used when there are dependencies between the children `AsynchronousCompletionToken` and the parent `AsynchronousCompletionToken`.

The library allows also to fork several functions at one time

```
result_of::fork_all<AE, int(*)(), int(*)(), int(*)()>::type handles = ba↵
sync::fork_all(ae, f, g, h);
std::cout << get<1>(res).get() - get<0>(res).get() + get<2>(res).get() << std::endl;
```

The result of the fork_all operation is a fusion tuple of asynchronous completion token handles. The user can apply any fusion algorithm on this tuple as for example

```
bool b = fusion::none(handles, fct::interruption_requested());
```

The asynchronous completion token models follows two interfaces, the thread interface and the unique_/shared_future interface.

To make common tasks easier the library provides some functors in the name space fct: for the thread interface as

- fct::join
- fct::join_until
- fct::join_for
- fct::detach
- fct::interrupt

- fct::interrupt_requested

and for the future operations as

- fct::get
- fct::wait
- fct::wait_until
- fct::wait_for
- fct::is_ready
- fct::has_value
- fct::has_exception

Here is an example for get:

```
namespace fct {
    struct get {
        template<typename ACT>
        typename ACT::result_type operator()(ACT& t) const {
            return t.get();
        }
    };
}
```

In addition the library provides some non member functions that are the result of applying these functors to the tuple using a fusion algorithm:

- join_all
- join_all_until
- join_all_for
- detach_all
- interrupt_all
- interrupt_requested_on_all
- get_all
- wait_all
- wait_all_until
- wait_all_for
- are_all_ready
- have_all_value
- have_all_exception

Next follows how get_all is defined.

```
template <typename MovableTuple>
typename result_of::get_all<Sequence>::type
get_all(Sequence& t) {
    return fusion::transform(t, fct::get());
}
```

The library defines in a systematic way the result_of of a function as a metafunction having the same name as the function on the namespace result_of, as the Boost.Fusion library does.

```
namespace result_of {
    template <typename Sequence>
    struct get_all {
        typedef typename fusion::result_of::transform<Sequence, fct::get>::type type
    };
}
```

So the user can do the following

```
result_of::fork_all<AE, int(*)(), int(*)(), int(*)()>::type res =
    basync::fork_all(ae, f, g, h);
result_of::get_all<result_of::fork_all<AE, int(*)(), int(*)(), int(*)()>::type>::type values =
    basync::get_all(handles);
```

or using a typedef

```
typedef result_of::fork_all<AE, int(*)(), int(*)(), int(*)()>::type auto_type;
auto_type handles = basync::fork_all(ae, f, g, h);
result_of::get_all<auto_type>::type values= basync::get_all(handles);
```

Note that the notation can be shortened by using the C++0x auto keyword.

```
auto res = basync::fork_all(ae, f, g, h);
auto values = basync::get_all(handles);
```

or using BOOST_AUTO

```
BOOST_AUTO(res, basync::fork_all(ae, f, g, h));
BOOST_AUTO(values, basync::get_all(handles));
```

Last but not least the library provides also some sugaring functions like wait_for_all that forks and wait for the completion of all the functions.

```
result_of::wait_for_all<AE, int(*)(), int(*)(), int(*)()>::type res =
    basync::wait_for_all(ae, f, g, h);
std::cout << get<1>(res) - get<0>(res) + get<2>(res) << std::endl;
```

and wait_for_any, which works only with functions that return the same type or are convertible to the same type, and return the index and the value of any of the completed functions.

```
result_of::wait_for_any<AE, int(*)(), int(*)(), int(*)()>::type res =
    basync::wait_for_any(ae, f, g, h);
std::cout << "function " << res.first
    << " finshed first with result=" << res.second << std::endl;
```

The current implementation use the wait_for_any function so any AE must provide a way to get a unique|shared_future from its `AsynchronousCompletionToken`.

The library defines a functor allowing the user to specialize it

```
template <typename AE>
struct get_future {
    template <typename T>
    shared_future<T>& operator()(typename asynchronous_completion_token<AE,T>::type& act)
    { return act.get_future(); }
};
```

Resuming a simple way to define a new AsynchronousExecutor is to define a class as

```
struct AsynchronousExecutor {
    template <typename T>
    struct handle {
        typedef implementation-specific-type-modeling-a-ACT type;
    };

    template <typename F>
    typename handle<typename result_of<F()>::type>::type
    fork(F f);
};
```

Threader/Joiner

See the N1833 - Preliminary Threading Library Proposal for TR2 where Kevlin Henney introduces the concept of threader as an asynchronous executor and a function thread that evaluates a function asynchronously and returns an asynchronous completion token joiner, able to join but also to get the value of the function result.

The main specificity is that here we make a difference between unique_joiner (move-only) and shared_joiner and as consequence unique_threader and shared_threader.

The second specificity concerns the fact joiners can detach, terminate, ... on destruction.

[/

Users'Guide

[/

Getting Started

Installing Async

Getting Boost.Async

You can get the last stable release of **Boost.Async** by downloading `async.zip` from the Boost Vault

You can also access the latest (unstable?) state from the Boost Sandbox.

Building Boost.Async

Boost.Async is a header only library.

Build Requirements

Boost.Async depends on Boost. You must use either Boost version 1.39.x or the version in SVN trunk. In particular, **Boost.Async** depends on:

Boost.Bind	for bind, ...
Boost.Config	for ??? and abi_prefic_sufix, ...
Boost.Fusion	for tuples, and sequence algorithms ...
Boost.MPL	for transform, ...
Boost.Preprocesor	to simulate variadic templates , ...
Boost.SmartPtr	for shared_ptr, ...
Boost.Threads	for thread, mutex, condition_variable, ...
Boost.TypeTrais	for is_void, remove_references, ...
Boost.TypeOf	to register the ACT types.
Boost.Utility	for result_of, enable_if...

In addition it depends on the following libraries that are not yet accepted on Boost

Boost.Futures	for futures
Boost.ThreadPool	Only when using the `AsynchronousExecutor` boost::tp::pool and the `AsynchronousCompletionToken` boost::tp::task

And also will depend on a near future, conditionally, on the following libraries that are even not submitted to Boost.

Boost.Chrono	for time and duration
Boost.Move	to emulate the move semantic.
Boost.SmartPtr.UniquePtr	for unique_ptr, ...

Exceptions safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

Currently, **Boost.Async** has been tested in the following compilers/platforms:

- GCC 3.4.4 Cygwin
- GCC 3.4.6 Linux
- GCC 4.1.2 Linux

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Async Hello World!

This is a little bit more than a Hello World! example. It will also say Bye, Bye!

```
#include <boost/async/basic_threader.hpp>
#include <iostream>

namespace basync = boost::async;

void my_thread() {
    std::cout << "Hello World!" << std::endl;
}

int main() {
    boost::async::basic_threader ae;
    basync::wait_for_all(ae, my_thread);
    return 0;
}
```

```
Hello World!
```

Multiple algorithms

This example shows how to launch several algorithms and wait only for the more efficient.

```
#include <boost/async/typeof/threader.hpp>
#include <boost/async/wait_for_any.hpp>
#include <iostream>

namespace basync = boost::async;

int my_thread1() {
    sleep(3);
    std::cout << "1 thread_id=" << boost::this_thread::get_id() << std::endl;
}

int my_thread2() {
    sleep(1);
    std::cout << "2 thread_id=" << boost::this_thread::get_id() << std::endl;
}

int my_thread3() {
    sleep(2);
    std::cout << "3 thread_id=" << boost::this_thread::get_id() << std::endl;
}

int main() {
    basync::shared_threader ae;
    BOOST_AUTO(res,basync::wait_for_any(ae, my_thread1, my_thread2, my_thread3));
    std::cout << "Algotithm " << result.first+1 << " finished the first. result=" << res↵
ult.second << std::endl;
    return 0;
}
```

This results on the following output

```
3 thread_id=0x9c03f8
2 thread_id=0x9c0850
1 thread_id=0x9d0c40
Algotithm 2 finished the first. result=0
```

Tutorial

AE/ACT framework

References

N1833 - Preliminary Threading Library Proposal for TR2

N2185 - Proposed Text for Parallel Task Execution

N2276 - Thread Pools and Futures

N2802: A plea to reconsider detach-on-destruction for thread objects

Boost.ThreadPool (O. Kowalke)

Boost.Futures (A. Williams)

Boost.Thread (A. Williams)

async (Edd)

Glossary

AE Asynchronous executor

ACT Asynchronous completion token

Reference

Concepts

Asynchronous Completion Token Concepts

Concept ACT

Description

An `AsynchronousCompletionToken` allows to wait for the completion of an asynchronous executed operation. An `AsynchronousCompletionToken` should be `Movable` or `CopyConstructible`. The completion of the `AsynchronousCompletionToken` is undefined at this level. Different models could signal this completion when setting a value or an exception.

Notation

act	An AsynchronousCompletionToken
f	A Nullary function with type F
abs_time	A system_time
rel_time	A DurationType
b	A bool

Expression requirements

A type models a AsynchronousCompletionToken if, the following expressions are valid:

Expression	Return type	Runtime Complexity
wait(act)	void	Constant
b = wait_until(act, abs_time)	bool	Constant
b = wait_for(act, rel_time)	bool	Constant

Meta Expressions

Expression	Type	Compile Time Complexity
act_traits<ACT>::move_dest_type	Any	Constant
act_traits<ACT>::move_result	MPL boolean	Constant
is_movable<ACT>::type	MPL boolean	Constant
has_future_if<ACT>::type	MPL boolean	Constant
has_thread_if<ACT>::type	MPL boolean	Constant

Expression Semantics

Expression	Semantics
wait(act)	Blocks until the act completes
b = wait_until(act,abs_time)	Blocks until the act completes or abs_time is reached
b = wait_for(act,rel_time)	Blocks until the act completes or rel_time has been elapsed

Expression wait(act)

Effects:	Blocks until the act completes.
Synchronization:	The completion of act happens before wait() returns.
Throws:	the stored exception, if an exception was stored and not retrieved before.
Postconditions:	is_ready(act) == true.

Thread safety: unsafe

Expression b = wait_until(act,abs_time)

```
bool wait_until(const system_time& abs_time);
template<typename TimeDuration>
bool wait_for(TimeDuration const& rel_time);
```

Effects: Blocks until the act completes or abs_time is not reached.

Synchronization: The completion of the act happens before wait() returns.

Returns: true only if the function returns because act is ready.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: is_ready() == true.

Thread safety: unsafe

Expression b = wait_for(act,rel_time)

Effects: blocks until the act completes or rel_time has elapsed.

Synchronization: The completion of the act happens before wait() returns.

Returns: true only if the function returns because act is ready.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: is_ready() == true.

Thread safety: unsafe

Models

- unique_future
- shared_future
- unique_joiner
- shared_joiner
- tp::task
- boost::thread

Concept FutureBasedACT

The completion of the FutureBasedACT is undefined at this level but occurs usually after a set_value or set_exception on the associated promise.

Description

An FutureBasedACT is a AsynchronousCompletionToken that associates a value expected on the its completion.

Notation

act An AsynchronousCompletionToken

`cact`	An const `AsynchronousCompletionToken`
`f`	A `Nullary` function with type F
`abs_time`	A `system_time`
`rel_time`	A `DurationType`
`b`	A bool
`v`	`act_traits<typeof(act)>::move_dest_type`

Expression requirements

A type models an `FutureBasedACT` if, in addition to being an `AsynchronousCompletionToken`, the following expressions are valid:

Expression	Return type	Runtime Complexity
`v = get(act)`	`act_traits<typeof(act)>::move_dest_type`	Constant
`b = is_ready(cact)`	bool	Constant
`b = has_exception(cact)`	bool	Constant
`b = has_value(cact)`	bool	Constant

Expression Semantics

Expression	Semantics
`v = get(act)`	Blocks until `act` contains a value and returns the stored value
`b = is_ready(cact)`	Is true only if `cact` holds a value or an exception ready for retrieval.
`b = has_exception(cact)`	Is true only if `cact` contains an exception.
`b = has_value(cact)`	Is true only if `cact` contains a value

Expression `v=get(act)`

Effects:	Retrieves the value returned by the Nullary function.
Synchronization:	The completion of the `act` happens before get() returns.
Returns:	Depending on the nature of the ACT returns a `act_traits<ACT>::move_dest_type`.
Throws:	the stored exception, if an exception was stored and not retrieved before.
Postconditions:	if `act_traits<ACT>::move_result` is `true` it is unspecified what happens when `get()` is called a second time on the same shared_joiner.
Thread safety:	unsafe

Expression `b = is_ready(cact)`

Returns:	true only if `cact` holds a value or an exception ready for retrieval.
Remark:	if `act_traits<ACT>::move_result` is true the return value could be unspecified after a call to `get(act)`.

Expression `b = has_exception(cact)`

Returns: true only if `is_ready(cact)` == true and `cact` contains an exception.

Expression `b = has_value(cact)`

Returns: true only if `is_ready(cact)` == true and `cact` contains a value.

Models

- `unique_future`
- `shared_future`
- `unique_joiner`
- `shared_joiner`
- `tp::task`

Concept `ThreadBasedACT`

The completion of the `ThreadBasedACT` is undefined at this level but occurs usually after a function finish.

Description

An `ThreadBasedACT` is a `AsynchronousCompletionToken` that provides a thread like interface.

Notation

`act`	An `AsynchronousCompletionToken`
`cact`	A const `AsynchronousCompletionToken`
`cact`	An const `AsynchronousCompletionToken`
`f`	A `Nullary` function with type F
`abs_time`	A `system_time`
`rel_time`	A `DurationType`
`b`	A bool
`id`	An `act_traits<AsynchronousCompletionToken>::id_type`

Expression requirements

A type models an `FutureBasedACT` if, in addition to being an `AsynchronousCompletionToken`, the following expressions are valid:

Expression	Return type	Runtime Complexity
`b = joinable(cact)`	bool	Constant
`join(act)`	void	Constant
`b = join_until(act, abs_time)`	bool	Constant
`b = join_for(act, rel_time)`	bool	Constant
`detach(act)`	void	Constant
`interrupt(act)`	void	Constant
`b = interruption_requested(cact)`	bool	Constant
`id = get_id(cact)`	`act_traits<AsynchronousCompletionToken>::id_type`	Constant

Expression Semantics

Expression	Semantics
`b = joinable(cact)`	true if `cact` refers to a 'thread of execution', false otherwise
`join(act)`	waits for the associated 'thread of execution' to complete
`b = join_until(act, abs_time)`	waits for the associated 'thread of execution' to complete or the time `wait_until` has been reach.
`b = join_for(act, rel_time)`	waits for the associated 'thread of execution' to complete or the specified duration `rel_time` has elapsed
`detach(act)`	the associated 'thread of execution' becomes detached, and no longer has an associated one
`interrupt(act)`	request that the associated 'thread of execution' be interrupted the next time it enters one of the predefined interruption points with interruption enabled, or if it is currently blocked in a call to one of the predefined interruption points with interruption enabled
`b = interruption_requested(cact)`	true if interruption has been requested for the associated 'thread of execution', false otherwise.
`id = get_id(cact)`	an instance of `act_traits<AsynchronousCompletionToken>::id_type` that represents the associated 'thread of execution'.

Expression `b=joinable(act)`

Returns: true if `act` refers to a 'thread of execution', false otherwise

Throws: Nothing

Expression `join()`

Preconditions: `get_id(act)!=boost::async::get_current_id<ACT>()`

Effects: If `act` refers to a thread of execution, waits for that 'thread of execution' to complete.

Postconditions: If `act` refers to a 'thread of execution' on entry, that 'thread of execution' has completed. `act` no longer refers to any 'thread of execution'.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `join()` is one of the predefined *interruption points*.

Expression `b=join_until(act)|b=join_for(act)`

```
bool join_until(const system_time& wait_until);

template<typename TimeDuration>
bool join_for(TimeDuration const& rel_time);
```

Preconditions: `get_id(act)!=boost::async::get_current_id<ACT>()`

Effects: If `act` refers to a 'thread of execution', waits for that thread of execution to complete, the time `wait_until` has been reach or the specified duration `rel_time` has elapsed. If `act` doesn't refer to a 'thread of execution', returns immediately.

Returns: `true` if `act` refers to a thread of execution on entry, and that thread of execution has completed before the call times out, `false` otherwise.

Postconditions: If `act` refers to a thread of execution on entry, and `timed_join` returns `true`, that thread of execution has completed, and `act` no longer refers to any thread of execution. If this call to `timed_join` returns `false`, `*this` is unchanged.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `join_until()` is one of the predefined *interruption points*.

Expression `detach(act)`

Effects: If `act` refers to a 'thread of execution', that 'thread of execution' becomes detached, and no longer has an associated thread object.

Postconditions: `act` no longer refers to any 'thread of execution'.

Throws: Nothing

Expression `get_id(cact)`

Returns: If `act` refers to a 'thread of execution', an instance of `act_traits<AsynchronousCompletionToken>::id_type` that represents that `AsynchronousCompletionToken`. Otherwise returns a default-constructed `act_traits<AsynchronousCompletionToken>::id_type`.

Throws: Nothing

Expression `interrupt(act)`

Effects: If `act` refers to a 'thread of execution', request that the 'thread of execution' will be interrupted the next time it enters one of the predefined *interruption points* with interruption enabled, or if it is currently *blocked* in a call to one of the predefined *interruption points* with interruption enabled .

Throws: Nothing

Expression `h = native_handle(act)`

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Models

- `unique_joiner`
- `shared_joiner`
- `boost::thread`
- `tp::task`

Asynchronous Executors Concepts

Concept `AsynchronousExecutor`

Description

An `AsynchronousExecutor` executes asynchronously a function and returns an `AsynchronousCompletionToken` when calling the fork function on it.

Notation

`ae`	An `AsynchronousExecutor`
`f`	A `Nullary` function with type F
`act`	An `AsynchronousCompletionToken`

Expression requirements

A type models a `AsynchronousExecutor` if, the following expressions are valid:

Expression	Return type	Runtime Complexity
`__fork__(ae, f)`	`AsynchronousCompletionToken`	Constant
`get_future<AE>()(act)`	inherits from `unique_future|shared_future`	Constant
`asynchronous_completion_token<AE, T>::type`	Model of `AsynchronousCompletionToken` satisfying `__act_value<ACT>::type` is `T`	Constant

Expression Semantics

Expression	Semantics
`act = __fork__(ae, f)`	request `ae` to execute asynchronously the function `f`and returns an `AsynchronousCompletionToken`
`get_future<AE>()(act)`	gets a reference to a inherits from `unique_future|shared_future`

Constraints

The following constraints applies:

- `act_value<AsynchronousCompletionToken>::type == boost::result_of<F()::type>`

Models

- `basic_threader`
- `unique_threader`
- `shared_threader`
- `launcher`
- `shared_launcher`
- `scheduler`
- `tp::pool`

Concept `IntrinsicAsynchronousExecutor`

Description

The default fork implementation put some requirements in its `AsynchronousExecutor` parameter. This concept is related to this. An `IntrinsicAsynchronousExecutor` is `AsynchronousExecutor` that works well with the default implementation of _fork_.

Notation

`ae` An `IntrinsicAsynchronousExecutor`

`f` A `Nullary` function

Expression requirements

A type models an `IntrinsicAsynchronousExecutor` if, in addition to being an `AsynchronousExecutor`, the following expressions are valid:

Expression	Return type	Runtime Complexity
`ae.fork(f)`	`handle<boost::result_of<F()>::type`	Constant

Meta Expressions

Expression	Model Of	Compile Time Complexity
`handle<boost::result_of<F()>::type`	`AsynchronousCompletionToken`	Constant

Expression Semantics

Expression	Semantics
`ae.fork(f)`	executes asynchronously the function `f` and returns a `handle`

Models

- `basic_threader`
- `unique_threader`
- `shared_threader`
- `launcher`

- `shared_launcher`
- `scheduler`

AE/ACT Framework Reference

Header `<boost/async/act_traits.hpp>`

Includes all the `AsynchronousCompletionToken` and `AsynchronousExecutor` traits.

```
namespace boost {
namespace async {

    template<typename ACT>
    struct act_traits;

    template <typename ACT>
    struct is_movable;

    template <typename ACT>
    struct has_future_if;

    template <typename ACT>
    struct has_thread_if;

    template <typename AE, typename T>
    struct asynchronous_completion_token;

    template <typename AE>
    struct get_future;

}
}
```

AE operations

Header `<boost/async/fork.hpp>`

```
namespace boost { namespace async {
    namespace result_of {
        template <typename AE, typename F, typename A1, ..., typename An>
        struct fork;
            typedef typename AE::handle<typename result_of<F(A1, ..., An)> >::type type;
        };
    }

    template< typename AE, typename F, typename A1, ..., typename An >
    typename  asynchronous_completion_token<AE,
        typename boost::result_of<F(A1,..., An)>::type >::type
    fork( AE& ae, F fn, A1 a1, ..., An an );


    template< typename F, typename A1, ..., typename An >
    typename  asynchronous_completion_token<default_asynchronous_executor,
        typename boost::result_of<F(A1,..., An)>::type >::type
    fork( F fn, A1 a1, ..., An an );
}}
```

Metafunction `result_of::fork<AE,F>`

A metafunction returning the result type of applying _fork_ to an asynchronous executor and a Nullary functor.

```
namespace result_of {
    template <typename AE, typename F, typename A1, ..., typename An>
    struct fork;
        typedef typename AE::handle<typename result_of<F(A1, ..., An)> >::type type;
    };
}
```

Table 2. fork Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchronousExecutor`	Operation's argument
`F`	A model of n-ary function	Operation's argument
`Ak`	A model of n-ary function	n-ary function argument type for argument k

Expression: result_of::fork<AE,F,A1,...,An>::type

Return type: AE::handle<typename result_of<F(A1,...,An)> >::type

Non member function `fork()`

```
template< typename AE, typename F, typename A1 , ... typename An >
typename result_of::fork<AE,F, A1, An> >::type> >::type
fork( AE& ae, F fn, A1 a1 , ..., An an );
```

Table 3. fork Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchronousExecutor`	Operation's argument
`F`	A model of n-ary function	Operation's argument
`Ak`	A model of n-ary function	n-ary function argument type for argument k

Requires: The expression fn(a1, ..., an) must be valid and have a type convertible to R, where R is typename result_of<F(A1, ..., An)>::type..

Efect: Request the `AE` to creates a thread of execution for the function `fn` Request the asynchronous evaluation the expression `fn(a1, ..., an)` with respect to the calling thread to the asynchronous executor `ae` and places its result in an object h of type AE::handle<R>::type as if by using h.set_value(fn(a1, ..., an)). If the expression fn() throws an exception e, places e into h as if by using h.set_exception(current_exception()).

Returns: the AE handle h.

Header `<boost/async/fork_after.hpp>`

Defines a free function `fork_after` which request the asynchronous evaluation a function with respect to the calling thread to the asynchronous executor `ae` after the completion of some `AsynchronousCompletionToken`. The result is an `AsynchronousCompletionToken` wrapping the `AsynchronousCompletionToken` associated to the `AsynchronousExecutor`.

The default implementation forks a helper task which waits the completion of the `AsynchronousCompletionToken` 's only then evaluates the function. A user adapting another `AsynchronousExecutor` could want to specialize the `fork_after` free function. As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `fork_after` member function `fork_after` calls to the static operation `apply` on a class with the same name in the namespace `partial_specialization_workaround`. So the user can specialize partially this class.

```
namespace boost { namespace async {
    template <typename ACT>
    struct act_traits<act_wrapper<ACT> >;

    template <typename ACT>
    struct is_movable<act_wrapper<ACT> > : is_movable<ACT>{};

    template <typename ACT>
    struct has_future_if<act_wrapper<ACT> > : has_future_if<ACT> {};

    template <typename ACT>
    struct has_thread_if<act_wrapper<ACT> > : has_thread_if<ACT>{};

    template <typename ACT>
    struct act_wrapper;

    namespace result_of {
        template <typename AE,typename F>
        struct fork_after {
            typedef act_wrapper<typename asynchronous_completion_token<AE, typename boost::res↵
ult_of<F()>::type>::type> type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename AE, typename F, typename D >
        struct fork_after {
            static typename result_of::fork_after<AE,F>::type
            apply(AE& ae, F fn, D& d);
        };
    }

    template< typename AE, typename F, typename D>
    typename result_of::fork_after<AE,F>::type
    fork_after( AE& ae, F fn, D& d);

    template< typename AE, typename D, typename F, typename A1, ..., typename An  >
    act_wrapper< typename asynchronous_completion_token<AE, typename boost::res↵
ult_of<F(A1,..., An)>::type >::type >
    after_completion_fork( AE& ae, D& d, F fn, A1 a1, ..., An an );


}}
```

Partial Specialization Template Class `act_traits<act_wrapper<ACT>>`

act_wrapper inherits the traits of its wrapped `AsynchronousCompletionToken`.

```
template <typename ACT>
struct act_traits<act_wrapper<ACT> > : act_traits<ACT>{};
```

Partial Specialization Template Class `is_movable<act_wrapper<ACT> >`

act_wrapper inherits the traits of its wrapped `AsynchronousCompletionToken`.

```
template <typename ACT>
struct is_movable<act_wrapper<ACT> > : is_movable<ACT>{};
```

Template Class `act_wrapper<>`

```
template <typename ACT>
struct act_wrapper {
    typedef typename act_traits<act_wrapper<ACT> >::move_dest_type move_dest_type;
    act_wrapper();
    void wait_initialized();
    void set(ACT& other);
    void set(boost::detail::thread_move_t<ACT> other);

    void wait();
    bool wait_until(const system_time& abs_time);
    template <typename Duration>
    bool wait_for(ACT& act, Duration rel_time);
    move_dest_type get();
    bool is_ready();
    bool has_value();
    bool has_exception();

    void detach();
    bool joinable();
    void join();
    bool join_until(const system_time& abs_time);
    template <typename Duration>
    bool join_for(ACT& act, Duration rel_time);
    void interrupt();
    bool interruption_requested();
};
```

Metafunction `result_of::fork<AE,F>`

A metafunction returning the result type of applying `fork_after` to an asynchronous executor and a Nullary functor.

```
namespace result_of {
    template <typename AE,typename F>
    struct fork_after {
        typedef act_wrapper<typename asynchronous_completion_token<AE, typename boost::res↵
ult_of<F()>::type>::type> type;
    };
}
```

Table 4. fork Parameters

Parameter	Requirement	Description
AE	A model of `AsynchronousExecutor`	Operation's argument
F	A model of n-ary function	Operation's argument

Expression: result_of::fork_after<AE,F>::type

Return type: act_wrapper<typename asynchronous_completion_token<AE, typename boost::result_of<F()>::type>::type>

Static Member Function `partial_specialization_workaround::fork_after<>::apply()`

```
namespace partial_specialization_workaround {
    template< typename AE, typename F, typename D >
    struct fork_after {
        static typename result_of::fork_after<AE,F>::type
        apply(AE& ae, F fn, D& d);
    };
}
```

Non member function `fork_after()`

```
template< typename AE, typename F, typename D>
typename result_of::fork_after<AE,F,D>::type
fork_after( AE& ae, F fn, D& d);
```

Table 5. fork Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchronousExecutor`	Operation's argument
`F`	A model of n-ary function	Operation's argument
`D`	A model of a fusion `Sequence` of `AsynchronousCompletionToken`	Dependent `AsynchronousCompletionToken`

Requires: The expression fn() must be valid and have a type convertible to R, where R is typename result_of<F()>::type..

Efect: Request the asynchronous evaluation the expression `fn()` with respect to the calling thread to the asynchronous executor `ae` after the completion of all the `AsynchronousCompletionToken` in `d` and places its result in an object h of type AE::handle<R>::type as if by using h.set_value(fn()). If the expression fn() throws an exception e, places e into h as if by using h.set_exception(current_exception()).

Returns: the AE handle h.

Non member function `after_completion_fork()`

```
template< typename AE, typename D, typename F, typename A1, ..., typename An  >
act_wrapper< typename asynchronous_completion_token<AE, typename boost::res↲
ult_of<F(A1,..., An)>::type >::type >
after_completion_fork( AE& ae, D& d, F fn, A1 a1, ..., An an );
```

Table 6. fork Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchronousExecutor`	Operation's argument
`D`	A model of a fusion `Sequence` of `AsynchronousCompletionToken`	Dependent `AsynchronousCompletionToken`
`F`	A model of n-ary function	Operation's argument
`Ak`	A model of n-ary function	n-ary function argument type for argument k

Requires: The expression fn(a1, ..., an) must be valid and have a type convertible to R, where R is typename result_of<Fn()>::type..

Efect: Request the `AE` to creates a thread of execution for the function `fn` Request the asynchronous evaluation the expression `fn(a1, ..., an)` with respect to the calling thread to the asynchronous executor `ae` after the completion of all the `AsynchronousCompletionToken` in `d` and places its result in an object h of type AE::handle<R>::type as if by using h.set_value(fn(a1, ..., an)). If the expression fn() throws an exception e, places e into h as if by using h.set_exception(current_exception()).

Returns: the AE handle h.

Header `<boost/async/fork_all.hpp>`

```
namespace boost { namespace async {
    namespace result_of {
        template <typename AE, typename T>
        struct fork_all;
        template <typename AE, typename F1, ..., typename Fn>
        struct fork_all <AE,fusion::tuple<F1, ..., Fn> >{
            typedef  fusion::tuple<
                typename result_of::fork<AE,F1>::type,
                ...
                typename result_of::fork<AE,Fn>::type
            > type;
        };
    }

    template< typename AE, typename F1, ...,  typename Fn>
    typename result_of::fork_all<AE, mpl::tuple<F1, ..., Fn> >::type
    fork_all( AE& ae, F1 f1, ..., Fn fn );

    template< typename F1, ...,  typename Fn>
    typename result_of::fork_all<default_asynchronous_executor, F1, ..., Fn>::type
    fork_all( F1 f1, ..., Fn fn );
}}
```

Metafunction `result_of::fork_all<AE,F1, ..., Fn>`

A metafunction returning the result type of applying fork_all to an asynchronous executor and n Nullary functors.

```
namespace result_of {
    template <typename AE, typename T>
    struct fork_all;
    template <typename AE, typename F1, ..., typename Fn>
    struct fork_all <AE,fusion::tuple<F1, ..., Fn> >{
        typedef  fusion::tuple<
            typename result_of::fork<AE,F1>::type,
            ...
            typename result_of::fork<AE,Fn>::type
        > type;
    };
}
```

Table 7. fork_all Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchrousExecutor`	Operation's argument
`Fk`	A model of nullary function	Operation's argument

Expression: `result_of::fork_all<AE,F1,...,Fn>::type`

Return type: a fusion tuple of the result of forking each `Fk` by the `AE`

Non member function `fork_all()`

```
template< typename AE, typename F1, ...,  typename Fn>
typename result_of::fork_all<AE, mpl::tuple<F1, ..., Fn> >::type
fork_all( AE& ae, F1 f1, ..., Fn fn );

template< typename F1, ...,  typename Fn>
typename result_of::fork_all<default_asynchronous_executor, F1, ..., Fn>::type
fork_all( F1 f1, ..., Fn fn );
```

Table 8. fork Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchrousExecutor`	Operation's argument
`Fk`	A model of nullary function	Operation's argument

Returns: a fusion tuple of the result of forking each `fk` by the `ae`

Efect: Request the `AE` to creates a n thread of execution one for the function `fk`.

Header `<boost/async/wait_for_all.hpp>`

```
namespace boost {
namespace async {
    namespace result_of {
        template <typename AE, typename F1, ..., typename Fn>
        struct wait_for_all {
            typedef  fusion::tuple<
                typename result_of<F1()>::type,
                ...
                typename result_of<Fn()>::type,
            > type;
        };
    }

    template< typename AE, typename F1, ...,  typename Fn>
    typename result_of::wait_for_all<AE, F1, ..., Fn>::type
    wait_for_all( AE& ae, F1 f1, ..., Fn fn );
}
}
```

Metafunction `result_of::wait_for_all<AE,F1, ..., Fn>`

A metafunction returning the result type of applying get_all to a Sequence of asynchronous executor handles.

```
namespace result_of {
    template <typename AE, typename F1, ..., typename Fn>
    struct wait_for_all {
        typedef  fusion::tuple<
            typename result_of<F1()>::type,
            ...
            typename result_of<Fn()>::type,
        > type;
    };
}
```

Table 9. wait_for_all Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchrousExecutor`	Operation's argument
`Fk`	A model of nullary function	Operation's argument

Expression: `result_of::wait_for_all<AE, F1, ..., Fn>::type`

Return type: a fusion tuple of the result of applying get to each one of the asynchronous executors handles resulting of forking each function `Fk` by `AE`

Non member function `wait_for_all`

```
template< typename AE, typename F1, ...,  typename Fn>
typename result_of::wait_for_all<AE, F1, ..., Fn>::type
wait_for_all( AE& ae, F1 f1, ..., Fn fn );
```

Returns: a fusion tuple of the result of applying get to each one of the asynchronous executors handles resulting of forking each function `fk` by `ae`.

Effect: Request the `AE` to creates a n thread of execution one for the function `fk` and blocks until all the AE handles are ready.

Header `<boost/async/wait_for_any.hpp>`

```
namespace boost { namespace async {
    namespace result_of {
        template <typename AE, typename F1, ..., typename Fn>
        struct wait_for_any {
            // requires typename result_of<F1()>::type == typename result_of<Fk()>::type
            typedef  std::pair<unsigned,typename result_of<F1()>::type> type;
        };
    }

    template< typename AE, typename F1, ...,  typename Fn>
    typename result_of::wait_for_any<AE, F1, ..., Fn>::type
    wait_for_any( AE& ae, F1 f1, ..., Fn fn );
}}
```

Metafunction `result_of::wait_for_all<AE,F1, ..., Fn>`

A metafunction returning the a pair: the index of the first function executed by the AE and the result type of applying get on an asynchronous executor handles.

```
namespace result_of {
    template <typename AE, typename F1, ..., typename Fn>
    struct wait_for_any {
        // requires typename result_of<F1()>::type == typename result_of<Fk()>::type
        typedef  std::pair<unsigned,typename result_of<F1()>::type> type;
    };
}
```

Table 10. wait_for_all Parameters

Parameter	Requirement	Description
`AE`	A model of `AsynchrousExecutor`	Operation's argument
`Fk`	A model of nullary function	Operation's argument

Expression: `result_of::wait_for_any<AE, F1, ..., Fn>::type`

Return type: a pair: the index of the first function executed by the AE and the result type of applying get on an asynchronous executor handles created by `ae` to fork each `fk`

Non member function `wait_for_any`

```
template< typename AE, typename F1, ...,  typename Fn>
typename result_of::wait_for_any<AE, F1, ..., Fn>::type
wait_for_any( AE& ae, F1 f1, ..., Fn fn );
```

Returns: a fusion tuple of the result of applying get to each one of the asynchronous executors handles resulting of forking each function `fk` by `ae`.

Effect: Request the `AE` to creates a n thread of execution one for the function `fk` and blocks until all the AE handles are ready.

Header `<boost/async/algorithm.hpp>`

Include all the `AsynchronousExecutor/AsynchronousCompletionToken` framework functions.

```
#include <boost/async/fork.hpp>
//#include <boost/async/lazy_fork.hpp>
#include <boost/async/fork_after.hpp>
#include <boost/async/fork_all.hpp>
#include <boost/async/wait_for_all.hpp>
#include <boost/async/wait_for_any.hpp>

#include <boost/async/algorithm/join.hpp>
#include <boost/async/algorithm/join_until.hpp>
//#include <boost/async/algorithm/join_all_for.hpp>
#include <boost/async/algorithm/joinable.hpp>
#include <boost/async/algorithm/detach.hpp>
#include <boost/async/algorithm/interrupt.hpp>
#include <boost/async/algorithm/interruption_requested.hpp>

#include <boost/async/algorithm/join_all.hpp>
#include <boost/async/algorithm/join_all_until.hpp>
//#include <boost/async/algorithm/join_all_for.hpp>
#include <boost/async/algorithm/are_all_joinable.hpp>
#include <boost/async/algorithm/detach_all.hpp>
#include <boost/async/algorithm/interrupt_all.hpp>
#include <boost/async/algorithm/interruption_requested_on_all.hpp>

#include <boost/async/algorithm/wait.hpp>
#include <boost/async/algorithm/wait_until.hpp>
//#include <boost/async/algorithm/wait_all_for.hpp>
#include <boost/async/algorithm/get.hpp>
#include <boost/async/algorithm/get_until.hpp>
#include <boost/async/algorithm/is_ready.hpp>
#include <boost/async/algorithm/has_value.hpp>
#include <boost/async/algorithm/has_exception.hpp>

#include <boost/async/algorithm/wait_all.hpp>
#include <boost/async/algorithm/wait_all_until.hpp>
//#include <boost/async/algorithm/wait_all_for.hpp>
#include <boost/async/algorithm/get_all.hpp>
//#include <boost/async/algorithm/get_all_until.hpp>
//#include <boost/async/algorithm/get_all_for.hpp>
#include <boost/async/algorithm/are_all_ready.hpp>
#include <boost/async/algorithm/have_all_value.hpp>
#include <boost/async/algorithm/have_all_exception.hpp>
```

Future based ACT operations

Header `<boost/async/algorithm/wait.hpp>`

Defines a free function `wait` which waits the `AsynchronousCompletionToken` passed as parameter. The default implementation applies the `wait` member function to the `AsynchronousCompletionToken`. A user adapting another `AsynchronousCompletionToken` could need to specialize the `wait` free function if the `AsynchronousCompletionToken` do not provides a wait function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `wait` member function, `wait` calls to the static operation apply on a class with the same name in the namespace `partial_specialization_workaround`. So the user can specialize partially this class.

The template parameter ACT must be a model of `FutureBasedACT`.

```
namespace boost {
namespace async {

    namespace result_of {
        template <typename ACT> struct wait {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename ACT> struct wait {
            static typename result_of::wait<ACT>::type apply( ACT& act ) {
                return act.wait();
            }
        };
    }

    template <typename ACT>
    typename boost::enable_if<has_future_if<ACT>,
        typename result_of::wait<ACT>::type
    >::type
    wait(ACT& act) {
        return partial_specialization_workaround::wait<ACT>::apply(act);
    }

}
}   // namespace boost
```

Header `<boost/async/algorithm/wait_until.hpp>`

Defines a free function `wait_until` which wait until the `AsynchronousCompletionToken` passed as parameter is ready or the given time is reached. The default implementation applies the `wait_until` member function to the `AsynchronousCompletionToken`. A user adapting another `AsynchronousCompletionToken` could need to specialize the `wait_until` free function if the `AsynchronousCompletionToken` do not provides a wait_until function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `wait_until` member function, `wait_until` calls to the static operation apply on a class with the same name in the namespace `partial_specialization_workaround`. So the user can specialize partially this class.

Defines a free function `wait_for` which wait until the `AsynchronousCompletionToken` passed as parameter is ready or the given time is elapsed. The default implementation applies the `wait_for` member function to the `AsynchronousCompletionToken`. A user adapting another `AsynchronousCompletionToken` could need to specialize the `wait_for` free function if the `AsynchronousCompletionToken` do not provides a `wait_for` function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `wait_until` member function, `wait_for` calls to the static operation apply on a class with the same name in the namespace `partial_specialization_workaround`. So the user can specialize partially this class.

The template parameter ACT must be a model of `FutureBasedACT`.

```
namespace boost { namespace async {
    namespace result_of {
        template <typename ACT> struct wait_until {
            typedef bool type;
        };
        template <typename ACT, typename Duration> struct wait_for {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename ACT> struct wait_until {
            static typename result_of::template wait_until<ACT>::type apply( ACT& act, const sys↵
tem_time& abs_time );
        };
        template< typename ACT, typename Duration> struct wait_for {
            static typename result_of::template wait_for<ACT,Duration>::type apply( ACT& act, Dur↵
ation abs_time );
        };
    }

    template <typename ACT>
    typename boost::enable_if<has_future_if<ACT>,
        typename result_of::template wait_until<ACT>::type
    >::type wait_until(ACT& act, const system_time& abs_time);

    template <typename ACT, typename Duration>
    typename boost::enable_if<has_future_if<ACT>,
        typename result_of::template wait_for<ACT,Duration>::type
    >::type wait_for(ACT& act, Duration rel_time);

}
}   // namespace boost
```

Header `<boost/async/algorithm/wait_all.hpp>`

Defines a free function `wait_all` which waits the completion of all the `AsynchronousCompletionToken` in the sequence passed as parameter.

```
namespace boost {
namespace async {

    namespace fct {
        struct wait {
            typedef void result_type;
            template<typename ACT>
            void operator()(ACT& act) const;
        };
    }

    namespace result_of {
        template <typename Sequence>
        struct wait_all {
            typedef typename fusion::result_of::for_each<Sequence, fct::wait>::type type;
        };
    }

    template <typename Sequence>
    typename result_of::wait_all<Sequence>::type
    wait_all(Sequence& t);

}
}   // namespace boost
```

Header `<boost/async/algorithm/wait_all_until.hpp>`

Defines two free function `wait_all_until` and `wait_all_for` which waits the completion of all the `AsynchronousCompletionToken` in the sequence passed as parameter or a given time is reached or elapsed respectively.

```
namespace boost {
namespace async {

    namespace fct {
        struct wait_until {
            wait_until(const system_time& abs_time);
            template<typename ACT>
            bool operator()(ACT& act) const;

        struct wait_for {
            template <typename Duration>
            wait_for(const Duration& rel_time);
            template<typename ACT>
            bool operator()(ACT& act) const;
    }

    namespace result_of {
        template <typename Sequence>
        struct wait_all_until {
            typedef bool type;
        };

        template <typename Sequence>
        struct wait_all_for {
            typedef bool type;
        };
    }

    template <typename Sequence>
    typename result_of::wait_all_until<Sequence const>
    wait_all_until(Sequence const& t, const system_time& abs_time);

    template <typename Sequence, typename Duration>
    typename result_of::wait_all_for<Sequence>
    wait_all_for(Sequence& t, const Duration& rel_time);

}
}
```

Header `<boost/async/algorithm/are_all_ready.hpp>`

Defines a free function `are_all_ready` which states if all the `AsynchronousCompletionToken` in a sequence of `AsynchronousCompletionToken` are ready. The current implementation applies the `is_ready` free function for each `AsynchronousCompletionToken`.

```
namespace boost { namespace async {
    namespace fct {
        struct is_ready {
            typedef bool result_type;
            template<typename ACT> bool operator()(ACT& act) const;
        };
    }

    namespace result_of {
        template <typename Sequence> struct are_all_ready {
          typedef typename fusion::result_of::template all<Sequence, fct::is_ready>::type type;
        };
    }

    template <typename Sequence> bool are_all_ready(Sequence& t);
}}
```

Thread based ACT operations

Header `<boost/async/algorithm/detach.hpp>`

Defines a free function `detach()` which `detach()` the `AsynchronousCompletionToken` passed as parameter. The default implementation applies the `detach()` member function to the `AsynchronousCompletionToken`. A user adapting another `AsynchronousCompletionToken` could need to specialize the `detach()` free function if the `AsynchronousCompletionToken` do not provides a detach function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `detach()` member function detach calls to the static operation apply on a class with the same name in the namespace `partial_specialization_workaround`. So the user can specialize partially this class.

The template parameter ACT must be a model of `ThreadBasedACT`.

```
namespace boost { namespace async {
    namespace result_of {
        template <typename ACT> struct detach {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename ACT> struct detach {
            static typename result_of::detach<ACT>::type apply( ACT& act );
        };
    }

    template <typename ACT>
    typename boost::enable_if<has_thread_if<ACT>,void>::type
    detach(ACT& act);
}}
```

Header `<boost/async/algorithm/detach_all.hpp>`

Defines a free function `detach_all` which detach all the `AsynchronousCompletionToken` in the sequence passed as parameter.

```
namespace boost { namespace async {
    namespace fct {
        struct detach {
            typedef void result_type;
            template<typename ACT>
            void operator()(ACT& act) const;
        };
    }

    namespace result_of {
        template <typename Sequence>
        struct detach_all {
            typedef typename fusion::result_of::for_each<Sequence, fct::detach>::type type;
        };
    }

    template <typename Sequence>
    void detach_all(Sequence& t);
}}
```

Header <boost/async/algorithm/are_all_joinable.hpp>

Defines a free function `are_all_joinable` which states if all the `AsynchronousCompletionToken` in a sequence of `AsynchronousCompletionToken` are `joinable`.

```
namespace boost { namespace async {
    namespace fct {
        struct joinable {
            typedef bool result_type;

            template<typename ACT>
            bool operator()(ACT& act) const;
        };
    }

    namespace result_of {
        template <typename Sequence>
        struct are_all_joinable {
            typedef typename fusion::result_of::all<Sequence, fct::joinable>::type type;
        };
    }

    template <typename Sequence>
    bool are_all_joinable(Sequence& t);
}}
```

AE/ACT Models Reference

Header `<boost/async/future_traits.hpp>`

```
namespace boost { namespace async {

    template<typename T>
    struct act_traits<unique_future<T> >
    {
#ifdef BOOST_HAS_RVALUE_REFS
        typedef typename boost::mpl::if_<boost::is_fundamental<T>,T,T&&>::type move_dest_type;
#else
        typedef typename boost::mpl::if_<boost::is_convertible<T&,boost::de↵
tail::thread_move_t<T> >,boost::detail::thread_move_t<T>,T>::type move_dest_type;
#endif
    };

    template<typename T>
    struct act_traits<unique_future<T&> >
    {
        typedef T& move_dest_type;
    };

    template<>
    struct act_traits<unique_future<void> >
    {
        typedef void move_dest_type;
    };

    template<typename T>
    struct act_traits<shared_future<T> >
    {
#ifdef BOOST_HAS_RVALUE_REFS
        typedef typename boost::mpl::if_<boost::is_fundamental<T>,T,T&&>::type move_dest_type;
#else
        typedef typename boost::mpl::if_<boost::is_convertible<T&,boost::de↵
tail::thread_move_t<T> >,boost::detail::thread_move_t<T>,T>::type move_dest_type;
#endif
    };

    template<typename T>
    struct act_traits<shared_future<T&> >
    {
        typedef T& move_dest_type;
    };

    template<>
    struct act_traits<shared_future<void> >
    {
        typedef void move_dest_type;
    };

    template<typename T>
    struct act_traits<unique_future<T&> >
    {
        typedef T& move_dest_type;
    };
    template <typename R>
    struct is_movable<unique_future<R> > : mpl::true_{};

    template <typename R>
    struct has_future_if<unique_future<R> > : mpl::true_{};
```

```
template <typename R>
struct has_thread_if<unique_future<R> > : mpl::false_{};


template <typename R>
struct is_movable<shared_future<R> > : mpl::true_{};

template <typename R>
struct has_future_if<shared_future<R> > : mpl::true_{};

template <typename R>
struct has_thread_if<shared_future<R> > : mpl::true_{};

namespace partial_specialization_workaround {
    template <typename R> struct join<unique_future<R> > {
        static typename result_of::template join<unique_future<R> >::type
        apply( unique_future<R>& act) {
            return act.wait();
        }
    };
    template <typename R> struct join<shared_future<R> > {
        static typename result_of::template join<shared_future<R> >::type
        apply( shared_future<R>& act) {
            return act.wait();
        }
    };
    template <typename R> struct join_until<unique_future<R> > {
        static typename result_of::template join_until<unique_future<R> >::type
        apply( unique_future<R>& act, const system_time& abs_time ) {
            return act.timed_wait_until(abs_time);
        }
    };
    template <typename R> struct join_until<shared_future<R> > {
        static typename result_of::template join_until<shared_future<R> >::type
        apply( shared_future<R>& act, const system_time& abs_time ) {
            return act.timed_wait_until(abs_time);
        }
    };
    template <typename R, typename Duration> struct join_for<unique_future<R>, Duration> {
        static typename result_of::template join_for<unique_future<R>,Duration>::type
        apply( unique_future<R>& act, Duration rel_time ) {
            return act.timed_wait(rel_time);
        }
    };
    template <typename R, typename Duration> struct join_for<shared_future<R>, Duration> {
        static typename result_of::template join_for<shared_future<R>,Duration>::type
        apply( shared_future<R>& act, Duration rel_time ) {
            return act.timed_wait(rel_time);
        }
    };
    template <typename R> struct wait_until<unique_future<R> > {
        static typename result_of::template wait_until<unique_future<R> >::type
        apply( unique_future<R>& act, const system_time& abs_time ) {
            return act.timed_wait_until(abs_time);
        }
    };
    template <typename R> struct wait_until<shared_future<R> > {
        static typename result_of::template wait_until<shared_future<R> >::type
        apply( shared_future<R>& act, const system_time& abs_time ) {
            return act.timed_wait_until(abs_time);
        }
    };
```

```
        template <typename R, typename Duration> struct wait_for<unique_future<R>, Duration> {
            static typename result_of::template wait_for<unique_future<R>,Duration>::type
            apply( unique_future<R>& act, Duration rel_time ) {
                return act.timed_wait(rel_time);
            }
        };
        template <typename R, typename Duration> struct wait_for<shared_future<R>, Duration> {
            static typename result_of::template wait_for<shared_future<R>,Duration>::type
            apply( shared_future<R>& act, Duration rel_time ) {
                return act.timed_wait(rel_time);
            }
        };
    }

}}
```

Header `<boost/async/basic_threader.hpp>`

`basic_threader` is an `AsynchronousExecutor` with a thread as `ThreadBasedACT`.

```
namespace boost { namespace async {
    class basic_threader {
    public:
        thread::native_handle_attr_type& attr();

        template <typename T> struct handle {
            typedef thread type;
        };

        template <typename F> thread fork(F f);
    };

    template<>
    struct act_traits<thread >  {
            typedef void move_dest_type;
    };

    namespace partial_specialization_workaround {
        template <>
        struct wait<thread> {
            static result_of::wait<thread>::type apply( thread& act) {
                return act.join();
            }
        };

        template <>
        struct wait_until<thread> {
            static result_of::wait_until<thread>::type apply( thread& act, const sys↵
tem_time& abs_time ) {
                return act.timed_join(abs_time);
            }
        };
        template <typename Duration>
        struct wait_for<thread, Duration> {
            static typename result_of::template wait_for<thread,Duration>::type ap↵
ply( thread& act, Duration abs_time ) {
                return act.timed_join(abs_time);
            }
        };

        template <>
```

```
        struct join_until<thread> {
            static result_of::join_until<thread>::type apply( thread& act, const sys↲
tem_time& abs_time ) {
                return act.timed_join(abs_time);
            }
        };
        template <typename Duration>
        struct join_for<thread, Duration> {
            static typename result_of::template join_for<thread,Duration>::type ap↲
ply( thread& act, Duration abs_time ) {
                return act.timed_join(abs_time);
            }
        };
    }

}}
```

Header `<boost/async/launcher.hpp>`

A launcher is an `AsynchronousExecutor` with a future as `FutureBasedACT` so we can get the value associated to it.

The library defines two kind of launchers: unique_launcher and shared_launcher that respectively have a unique_future and a shared_future as `AsynchronousCompletionToken`

```
#include <boost/async/fork.hpp>
namespace boost { namespace async {
    class launcher;
    class shared_launcher;

    namespace partial_specialization_workaround {
        template <typename R> struct join<unique_future<R> > {
            static typename result_of::template join<unique_future<R> >::type
            apply( unique_future<R>& act) {
                return act.wait();
            }
        };
        template <typename R> struct join<shared_future<R> > {
            static typename result_of::template join<shared_future<R> >::type
            apply( shared_future<R>& act) {
                return act.wait();
            }
        };
        template <typename R> struct join_until<unique_future<R> > {
            static typename result_of::template join_until<unique_future<R> >::type
            apply( unique_future<R>& act, const system_time& abs_time ) {
                return act.timed_wait_until(abs_time);
            }
        };
        template <typename R> struct join_until<shared_future<R> > {
            static typename result_of::template join_until<shared_future<R> >::type
            apply( shared_future<R>& act, const system_time& abs_time ) {
                return act.timed_wait_until(abs_time);
            }
        };
        template <typename R, typename Duration> struct join_for<unique_future<R>, Duration> {
            static typename result_of::template join_for<unique_future<R>,Duration>::type
            apply( unique_future<R>& act, Duration rel_time ) {
                return act.timed_wait(rel_time);
            }
        };
        template <typename R, typename Duration> struct join_for<shared_future<R>, Duration> {
            static typename result_of::template join_for<shared_future<R>,Duration>::type
            apply( shared_future<R>& act, Duration rel_time ) {
                return act.timed_wait(rel_time);
            }
        };
        template <typename R> struct wait_until<unique_future<R> > {
            static typename result_of::template wait_until<unique_future<R> >::type
            apply( unique_future<R>& act, const system_time& abs_time ) {
                return act.timed_wait_until(abs_time);
            }
        };
        template <typename R> struct wait_until<shared_future<R> > {
            static typename result_of::template wait_until<shared_future<R> >::type
            apply( shared_future<R>& act, const system_time& abs_time ) {
                return act.timed_wait_until(abs_time);
            }
        };
        template <typename R, typename Duration> struct wait_for<unique_future<R>, Duration> {
            static typename result_of::template wait_for<unique_future<R>,Duration>::type
            apply( unique_future<R>& act, Duration rel_time ) {
                return act.timed_wait(rel_time);
            }
        };
        template <typename R, typename Duration> struct wait_for<shared_future<R>, Duration> {
            static typename result_of::template wait_for<shared_future<R>,Duration>::type
            apply( shared_future<R>& act, Duration rel_time ) {
```

```
                return act.timed_wait(rel_time);
            }
        };
    }

}}
```

Class launcher

Thread launcher using a common configuration managed with the thread attributes and returning on the fork operation a unique_future to the resulting type of the call to the threaded function.

```
class launcher {
public:
    thread::native_handle_attr_type& attr();

    template <typename T>
    struct handle {
        typedef unique_future<T> > type;
    };

    template <typename F>
    unique_future<typename result_of<F()>::type>
    fork(F f);
};
```

Member function launcher::attributes

Reference to the thread attributes accesor.

```
thread::native_handle_attr_type& attributes();
```

Returns: A reference to the thread attributes.

Complexity: constant.

Metafunction launcher::handle<>

Metafunction that returns the result type of the fork function applied to a launcher and the value type.

```
template <typename T>
struct handle {
    typedef unique_future<T> > type;
};
```

Expression: L::handle<T>::type

Return type: A unique_future<T>.

Complexity: constant.

Member function lancher::fork

```
template <typename F>
unique_future<typename result_of<F()>::type>
fork(F f);
```

Returns: A unique_future to the result of calling a function F.

Effects: create a thread executing the function f. The result of the function will be stored on the resulting future.

Class `shared_launcher`

Thread shared_launcher using a common configuration managed with the thread attributes and returning on the fork operation a unique_future to the resulting type of the call to the threaded function.

```
class shared_launcher {
public:
    thread::native_handle_attr_type& attr();

    template <typename T>
    struct handle {
        typedef unique_future<T> > type;
    };

    template <typename F>
    unique_future<typename result_of<F()>::type>
    fork(F f);
};
```

Member function `shared_launcher::attributes`

Reference to the thread attributes accesor.

```
thread::native_handle_attr_type& attributes();
```

Returns: A reference to the thread attributes.

Complexity: constant.

Metafunction `shared_launcher::handle<>`

Metafunction that returns the result type of the fork function applied to a shared_launcher and the value type.

```
template <typename T>
struct handle {
    typedef unique_future<T> > type;
};
```

Expression: L::handle<T>::type

Return type: A unique_future<T>.

Complexity: constant.

Member function `lancher::fork`

```
template <typename F>
unique_future<typename result_of<F()>::type>
fork(F f);
```

Returns: A unique_future to the result of calling a function F.

Effects: create a thread executing the function f. The result of the function will be stored on the resulting future.

Header `<boost/async/threader.hpp>`

A threader is an `AsynchronousExecutor` with an `AsynchronousCompletionToken` that model `ThreadBasedACT` and `FutureBasedACT`.

The library defines two kind of threaders: unique_threader and shared_threader that respectively have a unique_joiner and a shared_joiner as `AsynchronousCompletionToken`

```
#include <boost/async/fork.hpp>
namespace boost {
namespace async {
    template <typename ResultType>
    class unique_joiner;

    template <typename ResultType>
    void swap(unique_joiner<ResultType>& lhs, unique_joiner<ResultType>& rhs);

    class unique_threader;

    template <typename ResultType>
    class shared_joiner;

    template <typename ResultType>
    void swap(shared_joiner<ResultType>& lhs, shared_joiner<ResultType>& rhs);

    class shared_threader;
}
}
```

Template Class `unique_joiner<>`

```
template <typename ResultType>
class unique_joiner {
    typedef unique_joiner this_type;
public:
    unique_joiner(const unique_joiner& rhs) = delete;
    unique_joiner& operator=(const unique_joiner& rhs) = delete;

    typedef ResultType result_type;

    template <typename Nullary>
    unique_joiner(thread::native_handle_attr_type& attr, Nullary f);
    template <typename Nullary>
    unique_joiner(Nullary f);

    unique_joiner(boost::detail::thread_move_t<unique_joiner> x);
    unique_joiner& operator=(boost::detail::thread_move_t<unique_joiner> x);
    operator boost::detail::thread_move_t<unique_joiner>();
    boost::detail::thread_move_t<unique_joiner> move();

    void swap(this_type& x);

    bool joinable() const;
    void join();
    bool join_until(const system_time& abs_time);
    template<typename TimeDuration>
    inline bool join_for(TimeDuration const& rel_time);

    result_type get();
    result_type operator()();

    bool is_ready() const;
    bool has_exception() const;
    bool has_value() const;

    void wait() const;
    bool wait_until(const system_time& abs_time) const;
    template<typename TimeDuration>
    inline bool wait_for(TimeDuration const& rel_time) const;

    thread::id get_id() const;
    void detach();
    void interrupt();
    bool interruption_requested() const;

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    unique_future<result_type> get_future();

};
```

unique_joiner Destructor

```
~unique_joiner();
```

Effects: If `*this` has an associated thread of execution, calls `detach()`. Destroys `*this`.

Throws: Nothing.

Member function `swap()`

```
void swap(unique_joiner& other);
```

Effects: Exchanges the threads of execution associated with `*this` and `other`, so `*this` is associated with the thread of execution associated with `other` prior to the call, and vice-versa.

Postconditions: `this->get_id()` returns the same value as `other.get_id()` prior to the call. `other.get_id()` returns the same value as `this->get_id()` prior to the call.

Throws: Nothing.

Member Function `get()|operator()()`

```
result_type get();
result_type operator()();
```

Effects: Retrieves the value returned by the Nullary function.

Sychronization: The completion of the call to the operator()() the Nullary function happens before get() returns.

Returns: If the result type R is a reference, returns the stored reference. If R is void, there is no return value. Otherwise, returns an rvalue-reference to the value stored in the asynchronous result.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: It is unspecified what happens when get() is called a second time on the same unique_joiner.

Thread safety: unsafe

Member Function `is_ready()`

```
bool is_ready() const;
```

Returns: true only if the associated state holds a value or an exception ready for retrieval.

Remark: the return value is unspecified after a call to get().

Member Function `has_exception()`

```
bool has_exception() const;
```

Returns: true only if is_ready() == true and the associated state contains an exception.

Member Function `has_value()`

```
bool has_value() const;
```

Returns: true only if is_ready() == true and the associated state contains a value.

Member Function `wait()`

```
void wait();
```

Effects: Blocks until the Nullariry function ends.

Sychronization: The completion of the call to the operator()() the Nullary function happens before wait() returns.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: is_ready() == true.

Thread safety: unsafe

Member Function `wait_until()|wait_for()`

```
bool wait_until(const system_time& abs_time);
template<typename TimeDuration>
bool wait_for(TimeDuration const& rel_time);
```

Effects: Blocks until the Nullariry function ends.

Sychronization: The completion of the call to the operator()() the Nullary function happens before wait() returns.

Returns: If the result type R is a reference, returns the stored reference. If R is void, there is no return value. Otherwise, returns an rvalue-reference to the value stored in the asynchronous result.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: is_ready() == true.

Thread safety: unsafe

Member function `joinable()`

```
bool joinable() const;
```

Returns: `true` if `*this` refers to a thread of execution, `false` otherwise.

Throws: Nothing

Member function `join()`

```
void join();
```

Preconditions: `this->get_id()!=boost::this_thread::get_id()`

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete.

Postconditions: If `*this` refers to a thread of execution on entry, that thread of execution has completed. `*this` no longer refers to any thread of execution.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `join()` is one of the predefined *interruption points*.

Member function `join_until()|join_for()`

```
bool join_until(const system_time& wait_until);

template<typename TimeDuration>
bool join_for(TimeDuration const& rel_time);
```

Preconditions: `this->get_id()!=boost::this_thread::get_id()`

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete, the time `wait_until` has been reach or the specified duration `rel_time` has elapsed. If `*this` doesn't refer to a thread of execution, returns immediately.

Returns: `true` if `*this` refers to a thread of execution on entry, and that thread of execution has completed before the call times out, `false` otherwise.

Postconditions: If `*this` refers to a thread of execution on entry, and `timed_join` returns `true`, that thread of execution has completed, and `*this` no longer refers to any thread of execution. If this call to `timed_join` returns `false`, `*this` is unchanged.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `timed_join()` is one of the predefined *interruption points*.

Member function `detach()`

```
void detach();
```

Effects: If `*this` refers to a thread of execution, that thread of execution becomes detached, and no longer has an associated `boost::thread` object.

Postconditions: `*this` no longer refers to any thread of execution.

Throws: Nothing

Member function `get_id()`

```
thread::id get_id() const;
```

Returns: If `*this` refers to a thread of execution, an instance of `boost::thread::id` that represents that thread. Otherwise returns a default-constructed `boost::thread::id`.

Throws: Nothing

Member function `interrupt()`

```
void interrupt();
```

Effects: If `*this` refers to a thread of execution, request that the thread will be interrupted the next time it enters one of the predefined *interruption points* with interruption enabled, or if it is currently *blocked* in a call to one of the predefined *interruption points* with interruption enabled .

Throws: Nothing

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Non-member function swap()

```
void swap(unique_joiner& lhs,unique_joiner& rhs);
```

Effects: lhs.swap(rhs).

Template Class unique_threader

```
class unique_threader {
public:
    thread::native_handle_attr_type& attributes();

    template <typename T>
    struct handle {
        typedef unique_joiner<T> type;
    };

    template <typename F>
    unique_joiner<typename result_of<F()>::type>
    fork(F f);

};
```

Member function unique_threader::attributes()

Reference to the thread attributes accesor.

```
thread::native_handle_attr_type& attributes();
```

Returns: A reference to the thread attributes.

Complexity: constant.

Metafunction unique_threader::handle<>

Metafunction that returns the result type of the fork function applied to a unique_threader and the value type.

```
template <typename T>
struct handle {
    typedef unique_joiner<T> type;
};
```

Expression: L::handle<T>::type

Return type: A unique_joiner<T>.

Complexity: constant.

Member function unique_threader::fork

```
template <typename F>
unique_joiner<typename result_of<F()>::type>
fork(F f);
```

Returns: A unique_joiner to the result of calling a function F.

Effects: create a thread executing the function f. The result of the function will be stored on the resulting unique_joiner.

Template Class `shared_joiner<>`

```
template <typename ResultType>
class shared_joiner {
    typedef shared_joiner this_type;
public:
    shared_joiner(const shared_joiner& rhs);
    shared_joiner& operator=(const shared_joiner& rhs);

    typedef ResultType result_type;

    template <typename Nullary>
    shared_joiner(thread::native_handle_attr_type& attr, Nullary f);
    template <typename Nullary>
    shared_joiner(Nullary f);

    shared_joiner(boost::detail::thread_move_t<shared_joiner> x);
    shared_joiner& operator=(boost::detail::thread_move_t<shared_joiner> x);
    operator boost::detail::thread_move_t<shared_joiner>();
    boost::detail::thread_move_t<shared_joiner> move();

    void swap(this_type& x);

    bool joinable() const;
    void join();
    bool join_until(const system_time& abs_time);
    template<typename TimeDuration>
    inline bool join_for(TimeDuration const& rel_time);

    result_type get();
    result_type operator()();

    bool is_ready() const;
    bool has_exception() const;
    bool has_value() const;

    void wait() const;
    bool wait_until(const system_time& abs_time) const;
    template<typename TimeDuration>
    inline bool wait_for(TimeDuration const& rel_time) const;

    thread::id get_id() const;
    void detach();
    void interrupt();
    bool interruption_requested() const;

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    shared_future<result_type> get_future();
};
```

shared_joiner Destructor

```
~shared_joiner();
```

Effects: If `*this` has an associated thread of execution, calls `detach()`. Destroys `*this`.

Throws: Nothing.

Member function `swap()`

```
void swap(shared_joiner& other);
```

Effects: Exchanges the threads of execution associated with `*this` and `other`, so `*this` is associated with the thread of execution associated with `other` prior to the call, and vice-versa.

Postconditions: `this->get_id()` returns the same value as `other.get_id()` prior to the call. `other.get_id()` returns the same value as `this->get_id()` prior to the call.

Throws: Nothing.

Member Function `get()|operator()()`

```
result_type get();
result_type operator()();
```

Effects: Retrieves the value returned by the Nullary function.

Sychronization: The completion of the call to the operator()() the Nullary function happens before get() returns.

Returns: If the result type R is a reference, returns the stored reference. If R is void, there is no return value. Otherwise, returns an rvalue-reference to the value stored in the asynchronous result.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: It is unspecified what happens when get() is called a second time on the same shared_joiner.

Thread safety: unsafe

Member Function `is_ready()`

```
bool is_ready() const;
```

Returns: true only if the associated state holds a value or an exception ready for retrieval.

Remark: the return value is unspecified after a call to get().

Member Function `has_exception()`

```
bool has_exception() const;
```

Returns: true only if is_ready() == true and the associated state contains an exception.

Member Function `has_value()`

```
bool has_value() const;
```

Returns: true only if is_ready() == true and the associated state contains a value.

Member Function `wait()`

```
void wait();
```

Effects: Blocks until the Nullariry function ends.

Sychronization: The completion of the call to the operator()() the Nullary function happens before wait() returns.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: is_ready() == true.

Thread safety: unsafe

Member Function `wait_until()|wait_for()`

```
bool wait_until(const system_time& abs_time);
template<typename TimeDuration>
bool wait_for(TimeDuration const& rel_time);
```

Effects: Blocks until the Nullariry function ends.

Sychronization: The completion of the call to the operator()() the Nullary function happens before wait() returns.

Returns: If the result type R is a reference, returns the stored reference. If R is void, there is no return value. Otherwise, returns an rvalue-reference to the value stored in the asynchronous result.

Throws: the stored exception, if an exception was stored and not retrieved before.

Postconditions: is_ready() == true.

Thread safety: unsafe

Member function `joinable()`

```
bool joinable() const;
```

Returns: `true` if `*this` refers to a thread of execution, `false` otherwise.

Throws: Nothing

Member function `join()`

```
void join();
```

Preconditions: `this->get_id()!=boost::this_thread::get_id()`

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete.

Postconditions: If `*this` refers to a thread of execution on entry, that thread of execution has completed. `*this` no longer refers to any thread of execution.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `join()` is one of the predefined *interruption points*.

Member function `join_until()|join_for()`

```
bool join_until(const system_time& wait_until);

template<typename TimeDuration>
bool join_for(TimeDuration const& rel_time);
```

Preconditions: `this->get_id()!=boost::this_thread::get_id()`

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete, the time `wait_until` has been reach or the specified duration `rel_time` has elapsed. If `*this` doesn't refer to a thread of execution, returns immediately.

Returns: `true` if `*this` refers to a thread of execution on entry, and that thread of execution has completed before the call times out, `false` otherwise.

Postconditions: If `*this` refers to a thread of execution on entry, and `timed_join` returns `true`, that thread of execution has completed, and `*this` no longer refers to any thread of execution. If this call to `timed_join` returns `false`, `*this` is unchanged.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `timed_join()` is one of the predefined *interruption points*.

Member function `detach()`

```
void detach();
```

Effects: If `*this` refers to a thread of execution, that thread of execution becomes detached, and no longer has an associated `boost::thread` object.

Postconditions: `*this` no longer refers to any thread of execution.

Throws: Nothing

Member function `get_id()`

```
thread::id get_id() const;
```

Returns: If `*this` refers to a thread of execution, an instance of `boost::thread::id` that represents that thread. Otherwise returns a default-constructed `boost::thread::id`.

Throws: Nothing

Member function `interrupt()`

```
void interrupt();
```

Effects: If `*this` refers to a thread of execution, request that the thread will be interrupted the next time it enters one of the predefined *interruption points* with interruption enabled, or if it is currently *blocked* in a call to one of the predefined *interruption points* with interruption enabled .

Throws: Nothing

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Non-member function `swap()`

```
void swap(shared_joiner& lhs,shared_joiner& rhs);
```

Effects: `lhs.swap(rhs)`.

Template Class `shared_threader`

```
class shared_threader {
public:
    thread::native_handle_attr_type& attributes();

    template <typename T>
    struct handle {
        typedef shared_joiner<T> type;
    };

    template <typename F>
    shared_joiner<typename result_of<F()>::type>
    fork(F f);

};
```

Member function `shared_threader::attributes()`

Reference to the thread attributes accesor.

```
thread::native_handle_attr_type& attributes();
```

Returns: A reference to the thread attributes.

Complexity: constant.

Metafunction `shared_threader::handle<>`

Metafunction that returns the result type of the fork function applied to a shared_threader and the value type.

```
template <typename T>
struct handle {
    typedef shared_joiner<T> type;
};
```

Expression: L::handle<T>::type

Return type: A shared_joiner<T>.

Complexity: constant.

Member function `shared_threader::fork`

```
template <typename F>
shared_joiner<typename result_of<F()>::type>
fork(F f);
```

Returns: A shared_joiner to the result of calling a function F.

Effects: create a thread executing the function f. The result of the function will be stored on the resulting shared_joiner.

Header `<boost/async/scheduler.hpp>`

`tp::pool` can be seen as a `AsynchronousExecutor` adding some functions and specializing some traits classes. The functions are:

- get_future
- interruption_requested

The traits are:

- asynchronous_completion_token : associating the `AsynchronousCompletionToken` `tp::task`
- partial_specialization_workaround::fork::apply: to call to submit instead of fork.

`tp::task` is an `AsynchronousCompletionToken` that models `ThreadBasedACT` and `FutureBasedACT`.

```
namespace boost { namespace async {

    template <typename C>
    class scheduler {
        explicit scheduler(
            tp::poolsize const& psize
        );
        template <typename T>
        struct handle {
            typedef tp::task<T> type;
        };
        template <typename F>
        tp::task<typename boost::result_of<F()>::type>
        fork(F f);
    };

    template <typename Channel>
    struct get_future<scheduler<Channel> > {
        template <typename T>
        struct future_type {
            typedef shared_future<T> type;
        };
        template <typename T>
        shared_future<T>& operator()(tp::task<T>& act);
    };

    template <typename Channel, typename T>
    struct asynchronous_completion_token<boost::tp::pool<Channel>,T> {
        typedef boost::tp::task<T> type;
    };

    namespace partial_specialization_workaround {
        template< typename Channel, typename F >
        struct fork<boost::tp::pool<Channel>,F> {
            static typename result_of::fork<boost::tp::pool<Channel>, F>::type
            apply( boost::tp::pool<Channel>& ae, F fn );
        };
    }
    template <typename C>
    struct get_future<tp::pool<C> > {
        template <typename T>
        shared_future<T>& operator()(tp::task<T>& act);
    };
```

```
    template <typename R>
    struct has_future_if<tp::task<R> > : mpl::true_{};

    template <typename R>
    struct has_thread_if<tp::task<R> > : mpl::true_{};
}}
```

Header `<boost/async/typeof/future.hpp>`

Include this files instead of `<boost/futures/future.hpp>` if you want TypeOf support.

```
#include <boost/futures/future.hpp>
#include <boost/typeof/typeof.hpp>

#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TEMPLATE(boost::async::unique_future, 1)
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::async::shared_future, 1)
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::async::promise, 1)
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::async::packaged_task, 1)
```

Header `<boost/async/typeof/launcher.hpp>`

Include this files instead of `<boost/async/launcher.hpp>` if you want TypeOf support.

```
#include <boost/async/launcher.hpp>
#include <boost/async/typeof/future.hpp>
#include <boost/typeof/typeof.hpp>

#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TYPE(boost::async::launcher)
BOOST_TYPEOF_REGISTER_TYPE(boost::async::shared_launcher)
```

Header `<boost/async/typeof/threader.hpp>`

Include this files instead of `<boost/async/threader.hpp>` if you want TypeOf support.

```
#include <boost/async/threader.hpp>
#include <boost/typeof/typeof.hpp>

#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TYPE(boost::async::unique_threader)
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::async::unique_joiner, 1)

BOOST_TYPEOF_REGISTER_TYPE(boost::async::shared_threader)
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::async::shared_joiner, 1)
```

Header `<boost/async/typeof/basic_threader.hpp>`

Include this files instead of `<boost/async/basic_threader.hpp>` if you want TypeOf support.

```
#include <boost/async/basic_threader.hpp>
#include <boost/typeof/typeof.hpp>

#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TYPE(boost::async::basic_threader)
```

Header `<boost/async/typeof/scheduler.hpp>`

Include this files instead of `<boost/async/scheduler.hpp>` if you want TypeOf support.

```
#include <boost/async/scheduler.hpp>
#include <boost/typeof/typeof.hpp>

#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TEMPLATE(boost::tp::task, 1)
```

Header `<boost/async/asynchronous_executor_decorator.hpp>`

```
namespace boost { namespace async {
    template <typename AE, template <class> class Decorator>
    struct asynchronous_executor_decorator : AE {
        template <typename T> struct handle {
            typedef typename AE::template handle<T>::type type;
        };

        template <typename F>
        typename AE::template handle< typename boost::result_of<F()>::type >::type
        fork( F fn );
    };

    template <typename AE, template <class> class Decorator>
    struct get_future<asynchronous_executor_decorator<AE, Decorator> > {
        template <typename T>
        struct future_type {
            typedef typename AE::template get_future<AE>::type type;
        };
        template <typename T>
        typename future_type<T>::type& operator()(typename AE::template handle<T>::type & j);
    };
}}
```

Examples

This section do includes complete examples using the library.

Parallel sort

Next follows a generic algorithm based on partitioning of a given problem in smaller problems, and compose a solution from the solution of the smaller problems.

```
template <
    typename DirectSolver,
    typename Composer,
    typename AE,
    typename Range
>
  void inplace_solve( AE & ae,
        boost::iterator_range<typename boost::range_iterator<Range>::type> range,
        unsigned cutoff );

template <
    typename DirectSolver,
    typename Composer,
    typename AE,
    typename Range
>
  void inplace_solve( AE & ae,
        boost::iterator_range<typename boost::range_iterator<Range>::type> range,
        unsigned cutoff )
  {
    unsigned size = boost::size(range);
    //std::cout << "<<par_ " << size;
    if ( size <= cutoff) DirectSolver()(range);
    else {
        partition<Range> parts(range, BOOST_PARTS);

       // wait_for_all_in_sequence(ae, &inplace_solve<DirectSolver,Composer,AE,Range>, parts);
        std::list<task_type> tasks;
        for (unsigned i=0;i < BOOST_PARTS-1; ++i) {
            task_type tmp(ae.submit(
                boost::bind(
                    &inplace_solve<DirectSolver,Composer,AE,Range>,
                    boost::ref(ae),
                    parts[i],
                    cutoff
            )));
            tasks.push_back(tmp);
        }
        inplace_solve<DirectSolver,Composer,AE,Range>(ae, parts[BOOST_PARTS-1], cutoff);
        boost::for_each(tasks, &boost::async::wait_act<task_type>);
        // wait_for_all_in_sequence

        Composer()(range);
    }
  }
```

So parallel sort could be

```
struct sort_fct {
    template<class RandomAccessRange>
    RandomAccessRange& operator()(RandomAccessRange rng) {
        return boost::sort(rng);
    }
};

struct inplace_merge_fct {
    template<class BidirectionalRange>
    BidirectionalRange&
    operator()( BidirectionalRange rng) {
        return boost::inplace_merge(rng, boost::begin(rng)+(boost::size(rng)/2));
    }
};
template <typename AE, typename Range>
void parallel_sort(AE& ae, Range& range, unsigned cutoff=10000) {
    boost::iterator_range<typename boost::range_iterator<Range>::type> rng(range);
    inplace_solve<sort_fct,inplace_merge_fct,pool_type,Range>( ae, rng, cutoff);
}
```

From a single to a multi threaded application

Appendices

Appendix A: History

Version 0.2, May 07, 2009 Adding immediate asynchronous executor + Adaptation to Boost 1.39

Features:

- Immediate asynchronous executor

Version 0.1, April 29, 2009 Extraction of the AE/ACT frameworks from Boost.Async

Features:

- An asynchronous execution framework working with `AsynchronousExecutor` and `AsynchronousCompletionToken`. It includes some generic functions and several `AsynchronousExecutor` and `AsynchronousCompletionToken`:
  - fork and fork_all to execute asynchronously functions
  - fork_after: request an `AsynchronousExecutor` to execute a function asynchronously once each one of `AsynchronousCompletionToken` in the dependency tuple parameter are ready. It is similar to the async_with_dependencies proposed Peter Dimov.
  - generic get, join, ... free functions to synchronize on an `AsynchronousCompletionToken`
  - generic get_all, join_all, ... free functions to synchronize on multiple `AsynchronousCompletionToken`
  - generic wait_for_all, wait_for_any to execute asynchronously functions and wait for the completion of all or any of them.
- Some `AsynchronousExecutor` and `AsynchronousCompletionToken` models
  - basic_threader: can be seen as a thread factory executing asynchronously a function on the returned thread.

- launchers: Launchers can be seen as a future factory executing asynchronously a function on a hidden thread.
- threader/joiner: A Threader runs a unary function in its own thread. A Threader can be seen as a Joiner factory executing asynchronously a function on a thread encapsulated on the returned Joiner. The joiner is used to synchronize with and pick up the result from a function or to manage the encapsulated thread.
- `tp::pool` and `tp::task` customization as an `AsynchronousExecutor` and an `AsynchronousCompletionToken` respectively. `tp::pool` can be seen as a `tp::task` factory executing asynchronously a function on a pool of threads.
- a generic asynchronous_executor_decorator which allows to decorate the function to be evaluated asynchronously.

Bugs

Open Bugs:

Fixed Bugs:

Appendix B: Rationale

TBC

Appendix C: Implementation Notes

TBC

Appendix D: Acknowledgments

The Threader|Joiner design has been taken from N1833 - Preliminary Threading Library Proposal for TR2 Many thanks to Kevlin Henney to make evident to me the separation between asynchronous executors, and asynchronous completion tokens. Thanks to Alan Patterson for the idea of the immediate executor.

You can help me to make this library better! Any feedback is very welcome.

Appendix E: Tests

AE/ACT

Name	Description
do_test_member_fork	Forks and get
do_test_member_fork_m	Forks and get
do_test_member_fork_bind	Forks and get
do_test_member_fork_bind_m	Forks and get
do_test_fork	Forks a nullary function and get
do_test_fork_1	Forks a unary function and get
do_test_fork_1_m	Forks a unary function and get
do_test_creation_through_functor	Forks a functor
do_test_creation_through_reference_wrapper	Forks a reference wrapper
do_test_wait	Forks and waits
do_test_wait_until	Forks and waits until a given time
do_test_wait_for	Forks and waits for a given time
do_test_get	Forks and get
do_test_wait_all	Forks several and waits all
do_test_wait_all_until	Forks several and waits all until a given time
do_test_wait_all_for	Forks several and waits all for a given time
do_test_set_all	Forks several and get all using set_all
do_test_get_all	Forks several and get all
do_test_wait_for_all	wait for all
do_test_wait_for_any	waits for any
do_test_wait_for_any_fusion_sequence	Wait for any in a fusion sequence
do_test_member_fork_detach	Forks and detach
do_test_thread_interrupts_at_interruption_point	Interrupt
do_test_join	Forks and join
do_test_join_until	Forks and joins until a given time
do_test_join_for	Forks and joins for a given time
do_test_join_all	Forks several and join all

Name	Description
do_test_join_all_until	Forks several and join all until a given time
do_test_join_all_for	Forks several and join all for a given time
do_test_fork_after_join	Fork after some dependent ACT and then join
do_test_fork_after_wait	Fork after some dependent ACT and then wait
do_test_fork_after_get	Fork after some dependent ACT and then get the value
XXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Appendix F: Tickets

Kind	Identifier	Description	Resolution	State	Tests	Version

Appendix G: Future plans

Tasks to do before review

Add an overloading for wait_for_all_in_sequence(ae, f, seq)

This will be quite useful on recursive algorithms evaluating asynchronously the same function on different parts.

```
template <
    typename DirectSolver,
    typename Composer,
    typename AsynchronousExecutor,
    typename Input>
void inplace_solve(AsynchronousExecutor& ae, Problem& input) {
  //  if (problem is small)
    if (size(range) < concurrency_threshold) {
    // directly solve problem
        DirectSolver()(input);
    } else {
        // split problem into independent parts
        BOOST_AUTO(partition, partition_view(input));
        // evaluates asynchronously inplace_solve on each element of the partition
        // using the asynchronous executor as scheduler
        wait_for_all_in_sequence(ae, inplace_solve, partition);
        // compose the result in place from subresults
        Composer()(partition);
    }
}
```

Add polymorphic act and adapters

When we need to chain `AsynchronousCompletionToken` using the fork_after the nature of the `AsynchronousCompletionToken` can change over time, an why not change also its template parameter. So at least we need to make polymorphic every function used by fork_after.

Complete the tests

Even if the current release include some test there is yet a long way before been able to review the library.

- change the test so they take less time using locks; conditions and variables.
- Complete the test for the AE/ACT framework
- Add test with functions throwing

Add more examples

Complete the reference

- ae/act framework

Change the rational and implementation sections

Use Boost.Chrono

Add C++0x move semantics on compilers supporting it and use the Boost.Move emulation otherwise

For later releases

Use C++0x variadic templates on compilers supporting it and use the preprocessor otherwise

Use C++0x Concepts on compilers supporting them and use the Boost.ConceptCheck or Boost.ConceptTraits otherwise