Executors and schedulers, revision 5

Document number: ISO/IEC JTC1 SC22 WG21 N4414
Supersedes: ISO/IEC JTC1 SC22 WG21 N4143
ISO/IEC JTC1 SC22 WG21 N3785
ISO/IEC JTC1 SC22 WG21 N3731
ISO/IEC JTC1 SC22 WG21 N3378=12-0068
ISO/IEC JTC1 SC22 WG21 N3562
Date: 2015-04-10
Authors: Chris Mysen
Reply-to: Chris Mysen <mysen@google.com>

I. Motivation

This proposal is a small set of revisions and some clarifications on the wording in N4143 presented by Chandler Carruth in Urbana. The core design principle are around creating a simple framework for task execution that can resolve the core issues with thread lifetime in `std::async` and provide a clear set of definitions about what the responsibilities and functionalities are of any executor.

As such, the proposed design here is a fairly minimal subset of executor functionality which simplifies many of the core use cases for task execution. This is not meant to be the only or final executor interface, but rather is the core interface. This document will cover some of the possibilities for extension of the interface, but does not propose those as part of the core.

It should be noted that the particular design of a given executor can have a large impact on the performance trade-offs and behaviors of the executor, for example a standard thread pool is relatively simple with reasonable sequencing of tasks due to a global queue, but a work-stealing thread pool can be high performance for small tasks and for parallel functions, at the cost of complexity and task ordering. Or a priority queue based thread pool can provide better control over task ordering, but at increased risk of priority inversions. As such, the proposal is that the core interface be generic enough to handle many of the common use cases, but proposes several executor implementations which have been found to be useful in many common contexts.

II. Design

II.1. Reference Implementations

A reference implementation for this design is WIP in the following location:

- https://github.com/ccmysen/executors_r5

There are two related reference implementations of this design. The first is the core/minimal set presented in N4143 (revision 4 of this paper), the second has some additional functionality (including timers) from the Redmond paper.

- https://github.com/ccmysen/executors_r4
- https://github.com/arturl/executors

II.2. Core Design Philosophy

There has been significant discussion around the role of an executor and the basic requirements around it. This proposal revolves around the principle that the primary role of an executor is to provide a context in which to execute tasks. This context should trivially be able to be passed between functions and objects. And this context is responsible for maintaining and cleaning up the resources associated with task execution (including the tasks themselves). The actual policies of execution are defined by the concrete executor implementations.

As such, the core executor interface only provides a mechanism for adding tasks to an executor for subsequent execution. Despite the simplicity of this interface, the abstraction is still quite powerful and several complex behaviors can be implemented on top of this interface.

Additional behaviors of the concrete executors (such as querying executor state or variations of functions to add tasks) are considered to be extensions of this core behavior and are thus out of scope of the core executor interface.

Moreover, the design approach outlined in this paper is such that an executor includes the context in which tasks are executing and is not strictly a lightweight object. This is opposed to what is proposed in N4156 in which the context is decoupled from the execution interface. The statement there is that the context *has-a* executor, whereas the statement here is that the context *is-a* executor.

II.3. Motivating Examples

There are number of simple and moderately difficult use cases which are intended to be made simple with the executor programming model. It is primarily intended to be a task concurrency model but can also be used for a number of parallelism use cases given more specialized executor models. It is important to think of executors as an important part in answering a set of questions:

1. What to execute
2. When to execute it

3. Where to execute it (the context to execute it in)
4. How to execute it (the policies/parameters to apply to execution)

The most trivial use case is the equivalent of std::async with slightly less painful blocking behavior (what and where):

```
auto fut = std::spawn(std::system_executor::get_executor(),
    std::make_package([&] { /* do some work */ return x; }));

/* do a bunch of stuff */
async_result = fut.get();
```

So this is pretty trivial, but handles the common use case.
But of course use cases are more complex than that, and many of the modifications on this core behavior either are looking for more complex sequences of events, for more efficiency, or for stronger guarantees of execution.

Let's say that you had a sequence of operations you wanted to run in a way to constrain resource usage (for example to maximize parallel usage of a database) and then wait for them all to complete. You can use thread_pools plus latches to do your thing by attaching continuations to the executor.

```
void finish_transaction() {}
std::thread_pool_executor<> db_executor(MAX_ACCESSES);
latch done(NUM_QUERY_TASKS);
for (int i = 0; i < NUM_QUERY_TASKS; ++i) {
  std::spawn(db_executor, tasks[i], [&done] { done.arrive(); }
}
done.wait();
finish_transaction();
```

Another variation on this is to do a number of parallel tasks to prepare an image for rendering on screen. In this sequence, there are 3 stage, 2 processing stages and 1 rendering stage, all running on independent executors to control various behaviors (first one is on the system_executor, the second on a bounded executor to limit parallelism for performance reasons, and the last on the gui_executor which has a specific requirement for which thread executes).

Note that this example uses a proposed in N4224 (Supplements to C++ Latches) called a flex_latch which has a notification callback which runs in the context of the last arriving thread.

```
template <typename Exec, typename Completion>
void start_processing(Exec& exec, inputs1& input, outputs1& output,
```

```
                      Completion&& comp) {
  flex_latch* l = flex_latch::create_self_deleting(
    forward<Completion>(comp), input.size());
  for (auto inp : input) {
    std::spawn(exec,
               [&inp] { /* process input */, [l] { l->arrive(); }
    );
  }
}

template <typename Exec, typename Completion>
void stage2(Exec& exec, inputs2& input, image& output,
            Completion&& comp) {
  flex_latch* l = flex_latch::create_self_deleting(
    forward<Completion>(comp), input.size());
  for (auto inp : input) {
    std::spawn(exec,
               [&inp] { /* process input */, [l] { l->arrive(); }
               );
  }
}
void render_gui(image& img) {
  /* do rendering */
}

// Hook everything up
std::thread_pool_executor<> tpe(STAGE2_PARALLEISM);
auto render_task = std::wrap(GUI::get_executor(),
    std::bind(&render_gui, out_image));
start_processing(
    std::system_executor::get_executor(),
    in1, out1,
    std::bind(&stage2, tpe, in2, out_image, render_task));
```

The above example is getting somewhat complex and for very complex chains of processing you would likely use a higher level construct to coordinate tasks, but you can see that even reasonably complex actions can be expressed in fairly simple sequences.

More motivating examples are shown inline below to explain the various concepts proposed here.

II.3. Core executor

The core executor API as proposed is composed of a single function (plus a copy and move constructor):

```
void spawn(Func&&);
```

Any class which implements this core interface could be considered an executor. This is actually simpler even than std::async which takes arguments and a launch policy. In effect it makes the statement: “launch this function in this context and according to the policy of the executor”. Depending on the executor implementation, the context and policies vary, but the visible behavior is effectively the same, which is to release the function to the executor and let the caller continue without knowledge of what happened.

This simple interface may seem trivial, but many of the interesting behaviors required of an executor can be layered on top of this and the core interface doesn’t prevent implementations from having a richer interface, but rather makes a statement that any class implementing the executor interface is capable of executing work within the context and policies defined by that class. This allows an executor which is a layer on top of a separate construct (you can, for example layer an executor on top of some form of prioritized thread pool where the executor defines which priority to run tasks at, or an executor can represent a serialization of tasks run in a GUI thread).

II.3.a. Copyability

The expectation is that an executor be copyable and moveable. This has some implications for implementers, namely, if a particular executor contains some form of shared state (e.g. queues, or a pool of threads), you would then have to make the shared state have shared ownership (with semantics similar to std::shared_ptr). Some implementations can skip the shared ownership when the executor is guaranteed to live beyond the life of any caller (e.g. with a system_executor which is guaranteed to live until destruction of the program).

The approach of shared state makes the general assumption that the underlying executor state become shared resources and requires that the executor implementations manage that shared state on behalf of all potential owners. For example serial_executor, which can be implemented with a shared queue, might maintain a reference count of the shared owners. On the other hand, `system_executor`, which has pre-defined lifetime semantics, need not keep such a reference count.

II.3.a.1. Alternative Ownership Semantics

Optimizations to allow for single-owner semantics could also be provided in cases where reference counting is undesirable for performance reasons (and where there is a clear owner). The previously proposed approach allows for more performant options as the non-copyable executor is made explicit, but this was deemed undesirable due to complexity in the interface. The Kohlhoff proposals split the context out completely and require that contexts be able to create copyable executors which is another variation on the non-copyable/copyable split. All of these proposals require references/pointers to the non-copyable state, but it is exposed in different ways. In the case where no copying is required (for example, a class with an internal thread pool), the pointer overhead is unnecessary and in which case, the only option would be to separate the copyable and non-copyable executors into separate concepts.

Another reasonable approach would be to assume all copies are pure references initially and provide a make_shared() function which creates a shared_ptr-like executor on top of the underlying executor. So in the common case, the caller making the executor would by default own it and all copies would assume the context lifetime would be guaranteed by the owner.

```
For example, you could see code like the following:
{
  thread_pool_executor tpe(NUM_THREADS);
  // run_tasks has a weak copy of the executor
  run_tasks(tpe);
  // run_something else also only has a weak copy
  run_something_else(tpe);
  // Thread pools implicitly join on pending tasks by default.
}
```

Or if the pool is a shared resource, you would do something like the following:

```
// Will release ownership when this is deleted
this.tpe = executor::make_shared(
  new thread_pool_executor(NUM_THREADS);
// Has shared ownership of tpe
run_tasks(tpe);
// Has shared ownership of tpe
run_something_else(tpe);
```

And with this every copy would take ownership of the pool and the last owner would effectively become responsible for shutting down the executor. This comes at the expense of needing to know if a particular executor has shared state, or making trivial executors with no shared state into executors with shared state (for example, a wrapper which pre-defines some parameters over an existing executor but is trivially copyable).

II.3.b. Parallelism

It should be noted that this interface does not make guarantees about the parallelism provided by the executor (and in fact the executor is allowed to run in the caller's thread if the executor policy allows it). This interface also does not make guarantees about the ordering of function execution when multiple calls are made to spawn. Parallelism and ordering are traits of the underlying context.

II.3.c. is-a vs. has-a

There is a discussion in N4242 about whether a thread_pool has-a executor or is-a executor. The discussion focuses on the Java interface where ExecutorService is-a Executor, but that in C++ it should really be modeled in a has-a relationship due to the non-reference semantics of C++.

The argument which is given here is that an executor is a context within which tasks are executed. As such, things like thread pool contexts or a thread-per-task contexts are executors, they don't have executors. This becomes more apparent when you have adapter classes which modify the behavior of underlying contexts. `serial_executor`, which enforces serial execution ordering, is a context with specific policies of execution, though it wraps another executor to do this.

The key is that these contexts also have implementation specific interfaces which do not control execution which need to be part of the context but don't necessarily need to be part of the executor interface. For example, shutdown options could be provided by many executor implementations (thread pools often have different shutdown options depending). And how to expose these in general ways without polluting the core executor interface is important.

Java does this in a rather unclean way using the ExecutorService interface, which in practice has multiple behaviors placed in the single interface (it has lifetime management, submit with futures, and invoking multiple callables). But in the case that shutdown semantics were desired, it would be trivial to extend the interface specifically for heavyweight executors to either create a handle for shutdown or to expose shutdown semantics (as is done in the thread_pool_executor here). Note though that many executors may not have a shutdown process (system_executor, loop_executor, serial_executor).

II.3.d. Executor Types

The executor types proposed here are very similar to the executors in N3785, with two notable changes. The proposed executors are as follows:

II.3.d.1. `thread_per_task_executor`

Behaves like the default behavior of std::async in which a new thread is created for each spawned task. Upon completion of the task the thread is destroyed.

II.3.d.2. `thread_pool_executor`

A simple thread pool class which constructs a pool of threads which run all tasks. Tasks are enqueued to avoid blocking on this pool of threads. Upon destruction of the executor, queued tasks are drained and the threads joined.

II.3.d.3. `loop_executor`

An executor for which queued tasks are accumulated until the execution functions are called (`loop(), run_queued_closures(), try_run_one_closure()`), at which point execution takes over the calling thread and run some or all of the queued closures (closures added after loop has started will wait until the next call to the execution functions).

II.3.d.4. `serial_executor`

An executor wrapper object which ensures that all queued functions are run sequentially where a given function cannot start until the previously queued one completes. This guarantees serial ordering of tasks but it does not guarantee that the tasks will all run on the same thread.

II.3.d.5. `system_executor`

A special executor which behaves like a pool of threads but with singleton semantics. This is intended to be the default executor for most use cases and allows for delegation to a library defined singleton executor. Provides a reasonable alternative to std::async in the general case. It is expected that the system_executor would comply with the “concurrent” execution agent concept, which means that there should be an eventual guarantee of forward progress.

It should be noted that the requirement of forward progress is intended to ensure that code will not deadlock due to insufficient resources (which can occur if you have dependencies in tasks).

It is understood that requiring a forward progress guarantee can significantly complicate the implementation of the system executor due to the need to be able to detect a lack of progress. In practice, several heavyweight thread pool implementations (including the microsoft thread

pools, as well as google internal ones) have this guarantee. It is acknowledged that on certain platforms it may be challenging to provide a system executor with a true guarantee of progress (embedded environments). It is also acknowledged that “eventual guarantee of forward progress” does not guarantee it be performant under all situations (for example, detection of lack of forward progress can take significant amount of time.

In terms of termination, the system executor is expected to shut down at some point after completion of the program, but at which point is undefined. The result is that static destructors should not rely on the continued existence of the system_executor. The system executor is not guaranteed to complete all pending tasks before shutdown nor it it required to wait for running tasks to complete.

The system_executor is not constructible and thus exports the singleton interface `system_executor::get_executor()`.

It should be noted that the inline_executor has been removed due to the fact that it changes the semantics of the spawn() function in which it effectively blocks the caller until the actual function is run, which is significantly different from the core semantic of the spawn() function.

II.3.e. `executor::work`

Because of the queueing behaviors of most executor implementations, a type erased function wrapper must be provided to executors. In the original paper, this was done with `std::function`, but this prevented move-only types from being usable with executors due to `std::function` being copyable but not moveable. This prevents, for example, a `std::packaged_task` from being used with executors.

Because of this, the library proposes a special wrapper object which can accept copyable or move-only functions and creates a single type erased function-object from it. This is implied in the templated executors, but is the type used in the type-erased executors in lieu of requiring both std::function and std::packaged_task spawn functions. That said it is somewhat duplicative and the only reason to expose it is because of the type erased interface.

One alternative here are to only support function<void()> in the type erased interface, though this means that the erased type is not compatible with the template version, which would mean that there would be some divergence in behaviors depending on which interface you chose to use.

II.3.f. Type Erasing Executor Wrapper

Because executors conform to a concept rather than to a type hierarchy, a type erasing wrapper are also provided to adapt executors to code which does not want to add a template parameter for the executor.

This executor-erasing wrapper (called `executor`) exports a similar (but not strictly the same) interface to the normal executors:

```
template <typename Exec> executor(Exec& exec);
void spawn(std::executor::work&& fn);
```

Note that this uses the `executor::work` defined above explicitly, but can construct it on the fly and basically behaves like the unerased executor.

```
void execute(std::executor& ex) {
  ex.spawn([] { func(); });
}
```

II.4. Helpers And Adapters

II.4.a. Spawn Helper Free Functions

Much like std::async, a small number of free functions is provided to extend the behavior of the default executor spawn function (which normally detaches from the caller completely). The helper functions allow for spawning with a future and the other for attaching a continuation.

Note that std::async went in a design path where the destructor of the async future blocks waiting for the async thread to complete. There are several documents (N3679, N3451, and others) which cover the problems with this from a programming model perspective. The approach taken here is to treat futures like any client-side use of future (which is that it will not block in the destructor waiting for tasks to complete). As such, it is the job of the executor to manage thread lifetime and the job of the packaged_task/future to manage their own shared state.

Some code which wants to asynchronously do some work and get the value later:

```
auto fut = std::spawn(pool,
  std::make_package([] { return /* do stuff */ }));
fut.get();
```

And some code which uses a continuation to wait for multiple tasks to complete:

```
latch l(2);
std::spawn(pool,
      [] { /* do some work */,
```

```
      [&l] { l.arrive(); });
l.wait();
```

And there is a proposal in progress which would further allow a notification on a latch (called a flex_latch), which would behave roughly as follows:

```
void finalize() {
  // finish up work
}
flex_latch l(2, &finalize);
std::spawn(pool,
      [] { /* do some work */,
      [&l] { l.arrive(); });
```

Note that `std::spawn(exec, func, continuation)` also takes ownership of the lifetime of the passed objects, so if the passed functions are move-only, this will encapsulate them and take ownership for as long as the task needs to be alive (as is the case with the native executor spawn). As such, this is not equivalent to:

```
pool.spawn( [&] { func(); continuation(); });
```

II.4.b. Executor Object Wrapper Free Functions

In addition to the spawn helper functions, there is one simple additional helper which enables wrapping objects in an executor and providing a consistent interface for querying the executor associated with this object. This also provides a special behavior for callables in which enables the wrapped object to be called (which spawns the callable on the wrapped executor). This enables an Active Object pattern in which an callable declares where it should execute. This is enabled by the fact that the executor interface has a main method for pushing work. This helper is effectively a small wrapper which packages a function in a wrapper which can either be unwrapped or can be called to post the object onto the contained executor via the `wrap` function:

```
template <typename T>
executor_wrapper<T>&& wrap(T&& obj);
```

An example of an active object is where the completion of one task causes a new task to be spawned on another executor (extending the previous latch example):

```
gui_executor ge;
void compute_image(output_image, block_id);
void draw(result) {
  … do some drawing …
```

```
}
auto draw_task = std::wrap(ge, std::bind(draw, final_image));
flex_latch completion(NUM_BLOCKS, draw_task);
for (int i = 0; i < NUM_BLOCKS; ++i) {
  std::spawn(std::system_executor::get_executor(),
    bind(compute_image, final_image, i),
    [&completion] { completion.arive(); }
  );
}
```

A variation on this could be one in which a particular function must always be run in a particular executor (for example, if there is a dedicated pool of high priority threads for handling network IO). In the following example the fact that responder must run in a different context is hidden from the caller who simply knows that there is a response callback provided to it.

```
network_executor ne = network::get_executor();
void respond(byte* response, size_t size) { /* respond */ }
auto responder = std::wrap(ne, respond);

process_data(data, responder) {
  …
  responder(response, size);  // packages args and spawns on
                              // network_executor ne
}
```

Because this has been made generic, you can also wrap objects for the behavior of being able to retrieve the executor which it is associated with:

```
thread_pool network_pool(NETWORK_POOL_SIZE);
network_interface ni(network_pool, …)
network_context = std::wrap(network_pool, ni);
void respond(byte* response, size_t size) { /* respond */ }
auto responder = std::wrap(network_context::get_executor(), respond);

process_data(data, responder) {
  ...
  responder(response, size);
}
```

This concept behaves similarly to the wrap proposal in N4242, though the invocation of the callable is somewhat different in that the version presented here invokes spawn on the wrapped executor and provides a separate means to invoke the underlying callable. This

concept is not strictly needed by the executor class, but provides a convenient mechanism for making a complete statement about execution (what to execute, where to execute it, and how to execute it).

This concept of wrapping callables in different behavior can actually be used to compose other behaviors which are desirable for executors and though not proposed here these may be used to solve a number of common issues.

For example, capturing exceptions thrown in a task without requiring packaged_task could be a useful abstraction given the exception forwarding challenge of executors.

```
template <typename Func>
class exception_catcher {
  explicit exception_catcher(Func&& func);
  exception_catcher(const exception_catcher& other);

  template<typename… Args>
  void operator()(Args… args);

  std::exception_ptr get_exception();
};
template <typename Func>
exception_catcher<Func>&& make_catcher(Func&& func);
```

Moreover, wrappers could easily be composed:

```
auto e_catcher = make_catcher(some_fun);
auto remote_callable(
  executor::wrap(system_executor::get_executor, e_catcher));
```

III. Design Comparisons

III.1. Compared to N4143 (revision 4 of this proposal)

III.1.a system_executor

In this proposal, the system executor concept has been refined somewhat to better capture the requirements around making an async replacement.

The first change was to make clearer that the system_executor must guarantee forward progress (e.g. it cannot be implemented as a bounded thread pool). This is to ensure that

workloads will not deadlock when using the system executor (unless there are particular system resource limits).

The second change is to clarify the destruction semantics of a system_executor. Namely that it will be shut down at some point before all statics are destroyed, meaning you cannot add work to system_executor during the static destruction phase of the program.

III.1.b executor_ref

The concept of an executor reference was removed from the proposal and the core executor changed to allow it to be passed by value. This has some implications for ownership semantics of executors because executors may contain shared state. Due to this, an executor implementation which contains shared state must also then have a mechanism for tracking ownership. This means that the last caller to hold the executor can become responsible for deleting it.

III.1.c spawn() free functions

The previous proposal had a free spawn() with continuation and spawn_future() to create a future from a packaged_task. These have been unified to both be called spawn() and a new bare spawn(func) has been added for consistency.

III.1.d function_wrapper

The concept of a generic move-only work object which behaves like a simplified std::function has been renamed to work and moved into the executor class, so code wishing to use it would create an object of type executor::work.

III.1.e task_wrapper

There has been a re-wording of the task_wrapper concept such that there is a generic wrap() function which can wrap either objects or callable (this also encapsulates the use case proposed by Kohlhoff). When wrapping objects, you receive a simple container which contains the underlying data as well as a get_executor call. When wrapping a callable, you get access to the callable but also receive a callable which takes the same parameters.

This is actually a semi generic callable -> callable transform that is a special active object which spawns itself in the right executor upon calling.

The returned object type has been renamed to be more generic as well.

III.3.Compared to N4242 (Executors and Asynchronous Operations, Revision 1)

There are many underlying differences between N4242 (and it's successor) and this proposal, though the two have (intentionally) converged towards a more common solution, though there are still some fundamental differences in behaviors and design goals between the proposals, particularly in the complexity of the underlying concept, some of which has clear performance or design benefits.

The most obvious of these differences are:

- system_executor semantics are substantially different. In N4242 system_executor is the executor which represents all threads in the process. In this proposal, the system executor does not need to represent all threads, but simply a suitable default executor. Many programming environments already provide such a default, but expect that not all code will want to use this executor (one could treat this as the "system default executor").
- The separation of the executor_context from the executor. This allows for a separation of the stateful execution context (which contains run queues and threads) from the stateless executor (which can be copied around trivially).
    - This creates a pattern for which there is basically a lightweight object (executor) with a reference to the heavyweight object (the context) and a very friendly connection between them. It also ends up requiring direct references between the context and the executors to ensure that the context doesn't get deleted before the executors referencing it are removed. The formalization does provide the ability couple the two so that lifetime issues don't arise.
    - In this proposal, the need for light- and heavy-weight objects is acknowledged, but hidden. The previous version of this proposal kept them under the same interface, but separated out references from concrete executors. This proposal unifies them further and requires that all executors be copyable and moves the context to be a shared state hidden behind the executor interface. This creates some interesting questions around lifetime of the shared state, but prefers a default behavior where shared state is treated as shared ownership unless the implementer knows otherwise (or there is no shared state). This reflects a basic difference in the proposals which treat the execution contexts in an is-a relationship with executor rather than a has-a relationship.
- Alternative task execution constructs.
    - N4242 provides 2 additional constructs in addition to the standard "post" to the executor, aimed at providing different degrees of cost and latency guarantees.
        - `dispatch` - which allows you to effectively ask for execution which should run quickly, even at the cost of slowing other execution. This allows for priority inversions but creates a mechanism by which you can request work be done quickly. This request is not guaranteed to be granted, though, and acts as more of a hint.

        - One could also model this in some ways as a spawn with potentially blocking semantics (like spawning a future and then blocking on its completion). Logically you would want some sort of priority mechanism to handle this though.
        - The key difference which cannot be emulated is that the executor is allowed to run this in the calling thread.
    - `defer` - this allows you to post to an executor from within an executing task, with the hint that this should generally only execute after the current task is done (a continuation). In particular, defer allows the deferred function to be placed in a thread local run queue on the same thread as the caller (if possible).
        - This approach has some nice properties w.r.t. the cost of defer. Namely the executor can avoid notifying other threads to wake up (if that is a reasonable semantic from a fairness perspective), and potentially save a mutex lock/unlock pair while executing another function.
        - In some executors, this actually delegates straight to post(), which means they are functionally equivalent. In other executors, there is no defer implementation which has significantly different semantics (e.g. serial_executors which has only one queue and no thread to notify). In other executors still (e.g. thread stealing thread pools), some of this behavior is likely to be there due to uses of thread local queues and potential optimizations to reduce notifications.

Of these differences, the most substantial is in the core API of the executor. The argument set out in this proposal is that the concepts of `dispatch` and `defer`, while potentially very useful for particular fine grained parallelism and in optimizations are not general purpose.

This is clear even in the specification in which `defer` is allowed (and likely commonly will) to be implemented as `post(spawn)`. serial_executor (strand), for example, would behave the same. Potentially other implementations which prefer thread fairness, or have separate priority mechanisms would not be able to skip the thread notify either. Generally defer is a performance optimization which you will largely only use if you are running into problems with the overhead of the core post/spawn function and you have a specific set of behaviors (namely you are posting onto your own executor from existing tasks). Moreover, it has some degree of serialization semantics when posting to the local executor, but it cannot be relied on if you are unsure of what executor you are defering to.

An alternative to forcing all implementations to create a `defer`() is actually to create executors with the preferred semantics. Work stealing thread pools solve some of this by spawning work on local queues largely at no cost to the rest of the pool. Serialization

semantics (in which a task only runs upon completion of this task) can trivially be implemented in other ways.

Similarly, the semantics of `dispatch` (when it can actually be inlined, as well as if it will block or not) are also inconsistent across implementations and calls, which makes it difficult to reason about the behavior of this function in a generic way. Sometimes you can rely on it blocking, sometimes not, and sometimes it will inline, sometimes not, depending on your context and type of executor. Moreover, the semantics of this function break the meaning of the executor, namely that you are executing work in the context of the executor (it also opens you up to priority inversions due to executing work out of order).

The behavior provided by `dipatch` is not possible to emulate with wrappers and the semantics are such that you cannot easily layer this on top of the existing `spawn`() function due to the possibility of blocking the caller or running in the caller's thread. That said explicit blocking semantics are easily buildable with futures or other mechanisms. Other functions could enable similar optimizations as well. For example, a function like spawn_if_ready, could be used to let the caller decide to directly call the function if the executor cannot. It can be argued, though, that this behavior is not general purpose and could be defined as an extension of the core executor interface available to some subset of executors where this sort of greedy-execution is desirable.

III.4. Exception Handling

Exception handling has been left out explicitly as there really is not a generic way to handle exception forwarding unless there is an explicit receiver. As such, the proposed approach for users to handle exceptions is to either handle the exception in the task directly, or to use a wrapper which forwards exceptions to future (packaged_task or the future-returning free function which allows exceptions to be attached to the future object).

In essence, the raw interface to the executor is entirely fire-and-forget, in order to get a handle to the task you must use a wrapper which creates a handle onto the task which can be used as a communication channel.

Other proposals suggest that in some specific cases, exceptions may be allowed to escape the executor (a loop executor for example could push exceptions to the caller as there is a well defined handler). As such, it is up to the specific executor to define exception handling semantics, but a reasonable behavior for a concrete executor implementation is that an unhandled exception would result in program termination.

Future work may decide that a there should be an explicit handle to tasks provided by the executor to allow for other communications beyond the basic data and exception handling of future in which case that would provide a place to place exception handlers.

IV. Outstanding questions.

IV.1. Extensions of Executors

There are a few core extensions of the core concept as proposed which are feasible ways to provide more functionality based on known use cases, each of these would modify the interface to the executor in more executor-specialized ways. A non-comprehensive list of variants which have been pulled from standard use cases and from the Google internal use cases is illustrative in that it shows some of the variety in executors which may be implemented.

- Prioritized queues (non-fair task selection)
- Prioritized thread pools (threads running at different priorities)
- Dynamically sized thread pools
- GPU thread pools (batch task operations)
- Work stealing thread pools/fork join executors
- Fiber executors (user level thread executors)
- Caching thread-per-task executors (thread-per-task but with thread re-use)
- Rate-limiting executors (prevention of starvation of threads by large numbers of a particular task type)
- Reference counted tasks (tracking when groups of tasks complete)
- Drainable executors (can accept recursively created work but not entirely new work)
- Lazy executor (executes only when results are needed - e.g. by a call to future.get())
- Executor visitor (visit upon task start or task complete - allowing behaviors like dynamic thread counts or resource tracking)
- A number of custom executors which provide application-specific behaviors or contexts (e.g. a backend API call executor which only handles a specific type of calls)

Notably these fall into 2 classes, functionally different execution behaviors (e.g. priorities, fibers, GPU, dynamically sized), and behaviors which decorate existing executors (reference counted, rate limiting, task-start notifications).

IV.2. Mechanisms for extension

There has been a lot of discussion about the Service-style extension model proposed in N4242 as a mechanism for extending the core executor framework. This follows the Extension-Object design pattern from Erich Gamma (link http://st.inf.tu-dresden.de/Lehre/WS06-07/dpf/gamma96.pdf). Conceptually this provides a nice simple framework for allowing objects to be extended without dirtying the core interface, which is nice as a general purpose mechanism for adding non-core concepts to executors with a lifetime scoped to the executor. In fact, the examples provided lie firmly in the

networking space where you have objects which change behavior over time, but in unpredictable ways or in ways which are not considered core concepts.

This approach has 4 main caveats with it as a general model for extending executors:

- You must still explicitly bake core concepts into the API whenever possible (the EO paper and other discussions state this as well), so this should not be used as the resting place for any and all non-core logic, core functionality (e.g. thread prioritization) deserves to go in the API of the executors directly rather than through a separate service model.
- Extensions are functionally still bound to the capabilities provided by the object on which they are built (they look like a visitor or decorator in this way), as a result you cannot trivially build entirely new functionality with them (priority thread pools for example need to be natively supported by the executor because they require control over the thread objects and internal queues).
- The implementation of a service is somewhat complex because it requires registering objects onto the executor directly and the lifetime of those objects being scoped to the executor. Functionally the complexity can be contained to a wrapper library which can attach services to the object without having to mess with the core API at all, which implies that this can be done as an independent library.
- The service concept makes it more challenging for the executor to natively support extensions because the extensions are decoration on top of the executor (every service is given a handle to the executor, but the reverse is not guaranteed to be true). As such an executor which is incompatible with a particular extension can create issues of mis-use of extensions (for example, a serial_executor is a lightweight concept and starting a new thread for timed operations on it adds significant overhead).

IV.3. Extending executors in this framework

The proposed design takes a significantly simpler approach to extension, with the ability to adapt existing execution contexts to the executor interface (by implementing a copyable wrapper class implementing the `spawn` function).

In practice this allows you to take custom extensions to the core interface and wrap it in the simple spawn interface fairly cheaply. One example of this is a prioritized thread pool supporting another parameter with a thread priority (e.g. `spawn(func, 10)`). You can create an executor wrapper which captures a fixed priority and adapts the prioritized executor to the standard executor interface. In this way you can trivially extend the executor concept and adapt existing code to work with it silently.

For example, a prioritized thread pool may look like the following:

```
class prioritized_thread_pool {
```

```
 public:
  template <typename Func>
  void spawn(Func&& func, int priority);
};
```

But because the standard spawn() function doesn't support priority, you can write a wrapper which handles this in a way that allows tasks to re-use.

```
template <typename Exec>
class high_priority_executor {
 public:
  high_priority_executor(Exec& exec, int priority)
      : exec_(exec), priority_(priority) {}

  template <typename Func>
  void spawn(Func&& func) {
    exec_.spawn(forward<Func>(func), priority_);
  }
}
```

Then you can use high_priority_executor everywhere you would take a normal executor and it would quietly adapt all calls to use priorities behind the scenes.

IV.4. Future Work

There are a number of possible future proposals which can follow onto this baseline, including extensions from other executor proposals which have been brought to the committee. In particular, the following interesting use cases have already been raised as future work or tabled discussions:

- Alternative dispatching - one key difference between the current proposal and N4242 (and related proposals) is in the presence of alternative dispatch approaches (namely the presence of `dispatch()` and `defer()`). Functionally these serve as different optimizations for latency or overhead.
    - `dispatch()` in particular has semantics which are very unique (in that it may inline function calls depending on the circumstances), and are difficult to emulate without native executor support.
    - `defer()` actually allows tasks to be put on a thread local queue given that they much only be dispatched once the task which spawned them completes and returns control to the executor. This has benefits in terms of not having to lock/notify the executor on each defer call because the tasks don't need to begin yet. You can approximate the behavior of waiting for task completion for spawn using a separate notification mechanism and in fact an executor with

thread local queues (which is a common performance optimization) would do similar things for all post calls.

- Timed/Deferred Execution - the concept of a deferred task has been removed from the proposal to simplify the design further, but a follow on paper will likely come up to discuss whether this is a core executor concept or something which can easily be layered on. A more detailed discussion of the design trade-offs of a deferred interface needs to be provided (in particular the downsides of not being able to natively support these in the executor for certain executor types which have native handles, as do many networking socket handling executors).
- Task cancellation - a very common pattern in task parallelism mechanisms is to start work which may not be needed right away and can be cancelled if needed. An example of this is work which is redundant, non-critical, or expensive (e.g. a database call with a timeout to prevent over-taxing the system).
- Batch spawn - This comes up in a GPU context, but can also be used to optimize highly parallel tasks as well (reducing the mutex overhead when adding tasks, which can be significant). This can also be used to create task groups which can be joined on easily without requiring the user to create additional tracking mechanisms.
- Thread local queues - this is inherent to the Kohlhoff proposal because of the presence of the defer function, but is left to future work to discuss the right design approach here and whether it's appropriate to standardize this or if this is behavior which is executor specific. Many high performance thread pools, for example, already are implemented in terms of thread local queues.
- Task and thread priorities - there are very common use cases for allowing threads to take on priorities (commonly this is to allow important tasks to get to the front of the queue or to take precedence in execution).
- There is currently a proposal outstanding which formalizes concurrent and blocking queues which are foundational to executors (http://isocpp.org/files/papers/n3533.html). In particular the performance of the queue implementation (and reducing costs of queue operations under high contention) can have a big effect on the performance of the executor.
- There was a comment at a previous WG to leave the return type of spawn() unspecified, which allows for it to return a handle in the future. This should probably be done when there is a clear meaning for the return type.
- Executor shutdown semantics - there are a number of use cases where tasks span an executor and block waiting for tasks to complete as a sort of join mechanism. There are other cases where you want to force shut down before work completes. Some variation on the API to allow different shutdown behaviors is likely needed.
- An interface like that suggested around Execution Agents in N4156 (and related papers) could provide a generic mechanism for checking the traits of an executor (whether it is concurrent, parallel, or weakly parallel, as well as the behavior of thread local storage). Discussion of whether that concept should be applied to all executors is left for a subsequent paper but some form of execution agent traits is reasonable as.

VI. Proposed Wording

VI.1. Executor Concept

The executor concept represents a single spawn function which takes a function pointer or function object and executes it according to the executor's execution policy at some point in the future.

Ownership of the passed function is taken by the executor, so a copyable type is copied and a moveable type is moved to be owned by the executor. The owned function will be deleted upon completion of execution or upon destruction of the executor.

In order to promote value semantics for executors, the executor is also copyable and moveable. Due to shared state in many concrete executor instances (e.g. queues or thread pools), this implies that executors will have to implement some form of reference counting on the internal state to track ownership of the underlying shared state.

```
executor {
 public:
  executor(const executor& other);
  executor(executor&& other);
  ~executor();

  template<class Func> void spawn(Func&& func);
};
```

executor::~executor()

Effects: Destroys the executor.

Synchronization: All closure initiations happen before the completion of the executor destructor. [Note: This means that closure initiations don't leak past the executor lifetime, and programmers can protect against data races with the destruction of the environment. There is no guarantee that all closures that have been added to the executor will execute, only that if a closure executes it will be initiated before the destructor executes. In some concrete subclasses the destructor may wait for task completion and in others the destructor may discard uninitiated tasks.]

Remark: If an executor is destroyed inside a closure running on that executor object, the behavior is undefined. [Note: one possible behavior is deadlock.]

template <class Func> void executor::spawn(Func&& func);

Effects: The specified function object shall be scheduled for execution by the executor

at some point in the future. May throw exceptions if spawn cannot complete (due to shutdown or other conditions).

Synchronization: completion of `func` on a particular thread happens before destruction of that thread's thread-duration variables. [Note: The consequence is that closures may use thread-duration variables, but in general such use is risky. In general executors don't make guarantees about which thread an individual closure executes in.]

Error conditions: The invoked `func` shall not throw an exception.

VI.1.a. thread_per_task_executor

Class `thread_per_task_executor` is a simple executor that executes each task (closure) on its own std::thread instance. Tracks spawned threads

The singleton get_executor() function makes a default executor out of the thread_per_task_executor (which is already implied in std::async).

```
class thread_per_task_executor {
 public:
  static thread_per_task_executor& get_executor();

  thread_per_task_executor();
  thread_per_task_executor(const thread_per_task_executor&);
  thread_per_task_executor(thread_per_task_executor&&);
  ~thread_per_task_executor();
  template<class Func> void spawn(Func&& func);
};
```

thread_per_task_executor::thread_per_task_executor(
const thread_per_task_executor& other)
Effects: Creates a copy of existing thread_per_task_executor by owning a handle to the internal shared state. Takes shared ownership of the internal shared state.

thread_per_task_executor::thread_per_task_executor(
thread_per_task_executor&& other)
Effects: Creates an executor from of an existing thread_per_task_executor and leaves the existing executor in an unknown state.

thread_per_task_executor::get_executor()

Effects: Gets a singleton thread_per_task_executor for use across code with the lifetime of the application.

`thread_per_task_executor::thread_per_task_executor()`

Effects: Creates an executor that runs each closure on a separate thread.

`thread_per_task_executor::~thread_per_task_executor()`

Effects: Waits for all added closures (if any) to complete, then joins and destroys the threads.

VI.1.b. thread_pool_executor

Class `thread_pool` is a simple thread pool class that creates a fixed number of threads in its constructor and that multiplexes closures onto them through some queueing mechanism.

```
class thread_pool_executor {
 public:
  // thread pools are not copyable/default constructible
  thread_pool_executor() = delete;
  thread_pool_executor(const thread_pool_executor&) = delete;

  // Construct a fixed pool of N threads and start them waiting for
  // work.
  explicit thread_pool_executor(size_t N);

  // Drain the thread pool and wait for all unfinished tasks to
  // complete.
  virtual ~thread_pool_executor();

  // optional - Force the pool to shut down without draining
  // remaining queued tasks. Waits for currently running tasks to
  // complete.
  virtual void shutdown_hard();

  // Executor interface
  template<class Func>  void spawn(Func&& func)
};
```

`thread_pool::thread_pool(int num_threads)`

Effects: Creates an executor that runs closures on `num_threads` threads.
Throws: `system_error` if the threads can't be created and started.

`thread_pool::thread_pool(const thread_pool& other)`

Effects: Creates an executor copy of an existing thread_pool by owning a handle to the internal shared state. Takes shared ownership of the internal shared state.

thread_pool::thread_pool(thread_pool&& other)

Effects: Creates an executor from of an existing thread_pool and leaves the existing pool in an unknown state.

thread_pool::~thread_pool()

Effects: Waits for all added closures (if any) to complete, then joins and destroys the threads.

thread_pool::shutdown_hard()

Effects: Waits only for actively running closures to complete, then joins and destroys the threads. Non-started functions will be destroyed.

VI.1.c. system_executor

The system executor is a system provided default for the common case scenario where a programmer just wants a reasonable place to run tasks asynchronously. Thus it provides the singleton get_executor() method to retrieve the system_executor.

This executor provides the minimal executor interface and is commonly implemented as a growable thread pool with some sort of forward progress guarantees.

```
class system_executor {
 public:
  static system_executor& get_executor();
  system_executor(const system_executor& other);
  system_executor(system_executor&& other);
  virtual ~system_executor();

 public:
  template<class Func> void spawn(Func&& func);
}
```

system_executor::system_executor(const system_executor& other)

Effects: copies an existing system executor by copying the handle to the shared executor state.

system_executor::system_executor(system_executor&&)

Effects: copies internal references of system executor to this, leaves passed system_executor in an undefined state.

`static system_executor& system_executor::get_executor()`

Effects: Gets a singleton system_executor for use across code with the lifetime of the application.

`system_executor::~system_executor()`

Effects: Waits for all added closures (if any) to complete, then joins and destroys the threads.

VI.1.d. loop_executor

Class loop_executor is a single-threaded executor that executes closures by taking control of a host thread. Closures are executed via one of three *closure-executing methods*: `loop()`, `run_queued_closures()`, and `try_run_one_closure()`. Closures are executed in FIFO order. Closure-executing methods may not be called concurrently with each other, but may be called concurrently with other member functions.

```
class loop_executor {
 public:
  loop_executor();
  loop_executor(const loop_executor& other);
  loop_executor(loop_executor&& other);
  virtual ~loop_executor();

  void loop();
  void run_queued_closures();
  void make_loop_exit();
  bool try_run_one_closure();

 // Executor interface
  template<class Func> void spawn(Func&& func);
};
```

`loop_executor::loop_executor()`

Effects: Creates a loop_executor object. Does not spawn any threads.

`loop_executor::loop_executor(const loop_executor& other)`

Effects: Creates a copy of the loop_executor object by owning a handle to the internal shared state and takes shared ownership of any underlying state.

`loop_executor::loop_executor(loop_executor&& other)`

Effects: Creates loop_executor object from other and takes ownership of any

underlying state.

loop_executor::~loop_executor()

Effects: Destroys the `loop_executor` object. Any closures that haven't been executed by a closure-executing method when the destructor runs will never be executed.

Synchronization: Must not be called concurrently with any of the closure-executing methods.

void loop_executor::loop()

Effects: Runs closures on the current thread until `make_loop_exit()` is called.

Requires: No closure-executing method is currently running.

void loop_executor::run_queued_closures()

Effects: Runs closures that were already queued for execution when this function was called, returning either when all of them have been executed or when `make_loop_exit()` is called. Does not execute any additional closures that have been added after this function is called. Invoking `make_loop_exit()` from within a closure run by run_queued_closures() does not affect the behavior of subsequent closure-executing methods. [Note: this requirement disallows an implementation like `void run_queued_closures() { add([](){make_loop_exit();}); loop(); }` because that would cause early exit from a subsequent invocation of `loop()`.]

Requires: No closure-executing method is currently running.

Remarks: This function is primarily intended for testing.

bool loop_executor::try_run_one_closure()

Effects: If at least one closure is queued, this method executes the next closure and returns.

Returns: true if a closure was run, otherwise false.

Requires: No closure-executing method is currently running.

Remarks: This function is primarily intended for testing.

void loop_executor::make_loop_exit()

Effects: Causes `loop()` or `run_queued_closures()` to finish executing closures and return as soon as the current closure has finished. There is no effect if `loop()` or `run_queued_closures()` isn't currently executing. [Note: `make_loop_exit()` is typically called from a closure. After a closure-executing method has returned, it is legal to call another closure-executing function.]

VI.1.e. serial_executor

Class `serial_executor` is an adaptor that runs its closures by scheduling them on another (not necessarily single-threaded) executor. It runs added closures inside a series of closures added to an underlying executor in such a way so that the closures execute serially. For any two closures `c1` and `c2` added to a `serial_executor e`, either the completion of `c1` happens before the execution of `c2` begins, or vice versa. If `e.spawn(c1)` happens before `e.spawn(c2)`, then `c1` is executed before `c2`.

The number of `spawn()` calls on the underlying executor is unspecified, and if the underlying executor guarantees an ordering on its closures, that ordering won't necessarily extend to closures added through a `serial_executor`.

```
template <typename Exec>
class serial_executor {
 public:
  explicit serial_executor(const Exec& underlying_executor);
  serial_executor(const serial_executor& other);
  serial_executor(serial_executor&& other);
  virtual ~serial_executor();
  Exec& underlying_executor();

  // Executor interface
  template<class Func> void spawn(Func&& func)
};
```

`serial_executor::serial_executor(const Exec& underlying_executor)`

Effects: Creates a `serial_executor` that executes closures, in an order that respects the happens-before ordering of the `serial_executor::spawn()` calls, by passing the closures to `underlying_executor`. Will make a copy of the passed executor object. [Note: several `serial_executor` objects may share a single underlying executor.]

`serial_executor::serial_executor(const serial_executor& other)`

Effects: Creates a copy of the serial_executor object by owning a handle to the internal shared state and takes shared ownership of any underlying state.

`serial_executor::serial_executor(serial_executor&& other)`

Effects: moves the shared state of the serial executor and the underlying executor to this. Executor other will be left in an undefined state.

`serial_executor::~serial_executor()`

Effects: Finishes running any currently executing closure, then destroys all remaining closures and returns.

Exec& serial_executor::underlying_executor()

Returns: The underlying executor that was passed to the constructor.

VI.2 Type Erased Executor

Class `executor` a type erasing executor object which complies to the basic executor specification with a reference to an concrete executor object. The `spawn` function behaves like the templatized executor but takes in a concrete moveable function wrapper which can erase arbitrary callable objects (including move-only objects, unlike std::function).

```
class executor {
 public:
  executor() = delete;
  executor(const executor& other);
  executor(executor&& other);

  template <typename Exec> executor(Exec& exec);

  void spawn(executors::work&& fn);
};
```

executor::executor(Exec& exec)

Requires: The lifetime of the underlying executor shall exceed that of the underlying executor.

Effects: Construct a copyable executor wrapper object from a reference to an existing executor. Maintains the reference to the executor for the lifetime of the object.

executor::executor(const executor& other)

Effects: constructs executor wrapper from other and maintains a shared reference to the contained concrete executor.

executor::executor(executor&& other)

Effect: constructs executor wrapper from other. Upon completion other will contain no reference to a concrete executor.

void executor::spawn(work&& fn)

Effects: calls spawn on the underlying executor with the passed function object.

```
namespace executors {
  class work {
   public:
```

```
    work() = delete;
    work(const work&) = delete;
    work(work&& other);
    ~work();

    template <typename T> work(T&& t);
    void operator()();
  };
}
```

class executors::work

Optional move-only function wrapper interface which extends the `std::function` concept to allow moveable objects to be be stored in the type erasing container (and is thus not a copyable object itself). This also can be used by executor implementations to store tasks.

work::work(executor::work&& other)

Effects: move-constructor to create a wrapper from an existing wrapper object.

work::~work(executor::work&& other)

Effects: destroys the callable object state contained in the work object.

template <typename T>work::work(T&& t)

Requires: the target callable takes no parameters.

Effects: creates a new work object from an existing function or function object t.

void work::operator()()

Effects: invokes the target with no parameters.

VI.3 Free Functions & Helper Objects

```
template <typename Func>
auto make_package(Func&& f) -> packaged_task<decltype(f())()>;
```

std::make_package(Func&& f)

Effects: optional helper function which returns a packaged task from the passed callable.

```
template <typename Exec, typename Func>
future<T> spawn(Exec&& exec, Func&& func);
```

future<T> std::spawn(Exec&& exec)

Effects: direct analogue to the spawn function on the executor but in free function form.

```
template <typename Exec, typename T>
future<T> spawn(Exec&& exec, packaged_task<T()>&& func);
```

future<T> std::spawn(Exec&& exec, packaged_task<T()>&& func)

Effects: helper version of spawn which spawns with a packaged task object and returns the associated future. Catches any exceptions thrown by the contained function and sets the exceptions on the returned future.

```
template <typename Exec, typename Func, typename Continuation>
void spawn(Exec&& exec, Func&& func, Continuation&& continuation)
```

void std::spawn(Exec&& exec, Func&& func, Continuation&& continuation)

Requires: neither func nor continuation will throw an exception

Effects: helper which spawns a callable which combines the calls of `func` an `continuation` serially such that `continuation` will only execute upon successful completion of `func`. This could basically be modeled as a lambda which runs the pair of `functions [ func=move(func), continuation=move(continuation) ] { func(); continuation() }`. Though this provides a convenient extension to the standard interface with less boilerplate code.

VI.4 Executor Wrapper

Convenience interface for binding objects to an executor by encapsulating both the executor and the object into a single object while providing access to both. This is actually a class of wrappers which are enabled depending on the type of the object passed in. Namely move-only types behave differently than copyable types. And both behave differently than callables. In particular, callables enable `operator()` and `run_underlying()` functions to actually execute the callable.

```
template <typename Exec, typename T, typename Enable=void>
class executor_wrapper {
 public:
  executor_wrapper(Exec exec, T&& obj);
  ~executor_wrapper();

  Exec& get_executor();
  T& get();
};
```

```
template <typename Exec, typename T>
class executor_wrapper<Exec, T,
                       typename std::enable_if<move_only<T>>::type> {
 public:
  executor_wrapper(Exec exec, T&& obj);
  ~executor_wrapper();

  Exec& get_executor();

  T&& get();
};

template <typename Exec, typename T>
class executor_wrapper<Exec, T,
                       typename std::enable_if<callable<T>>::type> {

class executor_wrapper {
 public:
  executor_wrapper(Exec exec, Func&& func);
  ~executor_wrapper();

  template <class ...Args>
  void operator()(Args... args);

  Exec& get_executor();

  // Allows optimizations where func is run on the same executor as
  // the caller and thus doesn't need to call spawn.
  template <class ...Args>
  void run_underlying(Args... args);
};
```

executor_wrapper::executor_wrapper(Exec exec, T&& obj)

Requires: the passed executor lives longer than the lifetime of the executor_wrapper object

Effects: constructs an object which contains a copy of the passed executor and contains the passed object (taking ownership of the passed object).

executor_wrapper::executor_wrapper(executor_wrapper&& other)

Requires: the executor contained in other lives longer than the executor_wrapper object

Effects: constructs the object from the other object, leaves other in undefined state.

executor_wrapper::~executor_wrapper()

Effects: deletes the stored object

Exec& executor_wrapper::get_executor()

Returns: a reference to the contained executor.

void executor_wrapper::operator()(Args… args)

Requires: the wrapper contain a callable, else this function is not enabled.

Effects: calls spawn on the contained executor with the internal func object bound to the supplied args.

template <typename... Args>
void executor_wrapper::run_underlying(Args... args)

Effects: Runs the internal callable object without spawning on the contained executor. Used for optimizations where the current context is the executor for which spawn will be called.

template <typename Exec, typename T>
executor_wrapper<Exec, typename T>&& wrap(Exec& exec, T&& obj)

Returns: an executor_wrapper object containing the passed executor and object

template <typename Exec, typename Func>
executor_wrapper<Exec, typename decay<Func>::type>&& wrap(Exec& exec, Func&& func)

Returns: an executor_wrapper object containing the passed executor and function