C++ Executors

Document number: ISO/IEC JTC1 SC22 WG21 P0008
Supersedes: ISO/IEC JTC1 SC22 WG21 N4414
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
Last Update Date: 2015-09-27
Authors: Chris Mysen
Reply-to: Chris Mysen <mysen@google.com>

This paper is a revision to N4414 from the Spring 2015 meeting in Lenexa (so is logically Revision 6 of Executors and Schedulers). The paper has been renamed as the scheduling components of executors have long been removed from the proposal, which is now focused solely on executors.

I. Design Discussion of N4414

I.1 Type Erasure

N4414 proposes the concept of a type erased executor wrapper which can be passed into classes which prefer not to take a templatized type as a parameter. There was some concern about the cost incurred by all the type hiding and virtual dispatch when it is actually unnecessary.

The proposed form of this using a simple template based inheritance would be less complex to implement and use. Actual details are discussed in the section on polymorphic executors.

I.2 Shutdown of system_executor

In N4414 there continued to be a problem with the definition of the shutdown of the system executor which did not solve the issues with std::async futures due to ownership of the shutdown/cleanup behavior of threads in the implied executor for async which spawns a new thread for each task.

In the original definition of system_executor, shutdown time for threads is not defined. The proposal here would be that system_executor actually have a well defined shutdown point immediately following the termination of main().

At this point, system_executor executes a draining shutdown at this point proposed as follows:

- It no longer accepts any new work (thus any new calls to spawn() would be an error condition at this point and may throw an exception).
- Any started threads are joined.

Note that thread_per_task_executor is also subject to the same shutdown semantics as system_executor due to the singleton nature of it (and that it is effectively the default std::async(std::launch::async, …) behavior).

Because of this behavior (which is the only one that really makes sense for the system executor),

I.3 General Shutdown Semantics of Executors

In general there is a need for there to be well defined semantics for the shutdown of executors and to allow a program to control how these semantics are applied. It is not currently proposed as to how to do this for all executors as these are executor specific (there may not be a way for the program to shut down the threads of an executor, or the resources may be controlled externally).

That said, there are a couple main considerations for shutdown, the type of shutdown behavior and the behaviors needed to initiate a shutdown sequence explicitly.

There are a number of degrees of shutdown which vary in terms of how aggressively they stop work:

1. Wait for the executor to be completely quiet (no unstarted work and no execution agents are active). This is a very weak shutdown sequence and basically states that the executor has fully drained. Because adding new work would require some sort of sequencing behavior to ensure that the executor shuts down eventually, this is not a very general shutdown behavior. On the other hand if the executor expects that tasks within the executor may continue to add work before completing their task, this could be a useful shutdown behavior (it would allow code to use the shutdown to join on completion of work signalled by drain).
2. Allow all existing work to drain, stop accepting new work. This is basically the same as 1 except that it places a strict requirement that no new work be added to execution. The benefit of this is that it creates a sort of synchronization point around shutdown in which only work prior to the shutdown initiation is allowed to complete. In practice you would

need to coordinate shutdown anyway, so this would only serve as a way to signal this to other code instead of requiring external signaling mechanisms.

3. Allow only started work to drain, delete any unstarted work. This is a harder shutdown and behaves as if all unstarted work is deleted and all active threads are joined pending completion of the current task. This mode is useful if it is known that the outcome of pending unstarted work is not needed.
4. Stop any started work and delete any unstarted work. This is as hard a shutdown as possible and makes the statement that no pending work in the executor is needed. This sort of shutdown requires that tasks be cancellable and/or execution agents be cancelable. Implementation of this is difficult in the current standard as thread cancellation is not available so in practice this would frequently behave the same as 3 without some mechanism to signal task cancellations. This also would be a best-effort shutdown.

For executors with a defined shutdown behavior (for example thread_pool_executor), option 2 is likely the least offensive default shutdown behavior. Though some executors may want to allow shutdown to be “upgraded” to a more aggressive mode when it’s known that some of the work is not required. This is not trivial to implement and it not recommended for system_executor which must handle all types of work, but specialized executors could implement such rules safely.

A potential approach to generalize shutdown for executors which support it would be to have a suite of functions related to shutdown:

- shutdown(type) - where type is an enum representing a sequence of shutdown types (as defined above). A default version of shutdown could pick a reasonable default (like option 2), though weaker or stronger variants could be used and a mechanism for increasing the aggressiveness of shutdown could be implemented as well.
- on_shutdown(callback) - callback registration to receive notifications upon shutdown initiation. This is useful if a shutdown blocks further input and a mechanism to communicate the transition to this state is desired. this is mainly useful in avoiding exceptions when attempting to add tasks to the executor.
- on_terminate(callback) - callback registration to receive notifications upon completion of the shutdown process. Primarily useful in communicating that the executor is unusable and potentially has been deleted. Can be useful as a way of tracking that an executor has become unusable and has fully drained before the point that the executor has actually been deleted. This could also take the form of a boolean shutdown future where .then could be registered or the future state read indicating that the executor has completed shutdown.

A generalized approach to shutdown is not proposed here, but the simple mechanism of allowing shutdown callbacks to be run would enable a number of simple mechanisms for tracking the current state of the executor (for example, a wrapper could register an on_shutdown callback and update internal state to ensure callers do something reasonable when attempting to spawn on a dead executor).

I.4 Ownership of Executor Objects

The proposed mechanism for executor ownership was very explicit in N4414 in that it required that executors all be internally reference counted in an attempt to deal with clear ownership semantics for shareable executors. This places additional ownership semantics directly into executors and in many cases actually cannot be satisfied. For example, a system_executor cannot be reference counted as it is not created by the program directly and thus it would not actually be able to live as long as callers required if they were to call it past its actual lifetime.

Microsoft handled this in PPL through the creation of a concept of a scheduler_ptr which abstracts the notion of a pointer to an executor and allows pointers to either a shareable or non-shareable executor. These behave in many ways like shared_ptr and weak_ptr except where weak_ptr has the shared_ptr interface with no lock() to initiate sharing.

This idea is very convenient but introduces a particular problem with shared ownership of executors: that it creates shutdown semantics which can be difficult to reason about. In particular, the last piece of code to use the executor may unexpectedly become the one to block on shutdown of the executor. This can create very unexpected behavior for code. Moreover, the semantics of a weak_ptr like object would be challenging to implement correctly without locks which could add significant overhead to code which passes executors in such a way.

Given this the proposal is to simply remove the concept of shared ownership from the executors altogether and require that code properly coordinate executor lifetime. This does not prevent code from creating shared_ptr<Exec> objects, but simply means that there is not built in support into the executor library to do so.

The disadvantage of this approach is that code needs to carefully coordinate when executors are passed around to other libraries since you would frequently be passing Executor& around to other code.

I.5 Wrapped Executors

There was some discussion in Lenexa about the notion of a wrapped executor causing potential for deadlock. This potential comes from the fact that the wrapper first has to execute in the executor which it is spawned in, which then causes the actual spawn in the target executor.

Because of this behavior operator() has some interesting behaviors in a wrapped function. First, it is non-blocking on the actual execution due to it simply calling spawn on the target executor. Second, it requires the function to execute twice, first to actually spawn the work and the second to actually execute the behavior. Solving the second behavior is relatively easy by having

executors aware of wrapped functions (by effectively bypassing executing tasks locally first and directly calling spawn on the target executor). The first issue is more difficult and can only be changed by making the wrapped function block on completion of execution of the wrapped function. This “fix” can actually create deadlocks because of priority inversion (if the original spawn blocks the execution of the actual function for example).

A simple example of this is where there is a single threaded executor with a queue of tasks to execute:

```
loop_executor ste;
ste.spawn(wrap(ste, [] () { /* some work */ }));
ste.loop();
```

In this case, wrap would create a function which behaves roughly as follows:

```
[](auto& exec, auto&& work) {
  auto fut = std::spawn(exec, work);
  fut.wait();
}
```

The net result is that the fut.wait() will block indefinitely because the spawned work will never execute. It’s trivial for someone to write code to do this themselves unintentionally. There are other variations where there are other reasons for problems (for example a prioritized thread pool where the first spawn has higher priority than the second and similar effects would occur).

Issues like this can be fixed to some degree by making all executors able to detect when the spawn is on the same executor (effectively removing the need to block in that case), though it’s not clear that this idiom is so difficult to implement that the complexity should be added to the executor interface.

So the proposal here is to drop the wrap functionality as proposed. A lighter weight wrap which simply a struct containing an executor and an arbitrary object still makes sense, but can trivially be added later as well.

I.6 Empty thread_pool_executor

Some discussion arose about a thread_pool_executor with an empty constructor. The idea would be that this executor would provide a “reasonable” default executor which would work for common use cases. There is some difficulty in specifying what a reasonable executor should do. After some discussion, this came to be better defined as an executor which could be used to maximize resource use within the application.

As such there could be a few possible proposals of how to do this:

- An empty thread pool which has some multiple of thread::hardware_concurrency. This would not make a strong guarantee of maintaining optimal throughput but does something reasonable if what you want to do is maximize CPU utilization
- A system_executor like object with thread_pool_executor semantics (which is effectively a singleton thread pool), which allows you to maximize utilization without the requirement that the executor guarantee forward progress (thus allowing you to keep the pool bounded in size).
- Something like the above (a singleton executor) but with a mechanism for better understanding the utilization of the system and dynamically adjusting threads to deal with how to maximize system performance. A weaker form of this would be a dynamic pool with a means of configuring when it spawns new threads, this would allow custom logic for maximizing network bandwidth, maximizing forward progress, utilization, memory, or other things.

That said, this concept is being left out of scope for the current proposal and can be discussed in more detail about the advantages/disadvantages of different proposals.

I.7 Parallelism Layer

There is a proposed paper which attempts to make a case for the use of the executor as the underlying concept for parallelism concepts (see N4406). There is a discussion on the mailing list about this particular concept and will be left out of this discussion, though it does open up some quite interesting concepts, namely:

- Use of executor_traits to formalize different executor behaviors and capabilities
- Use of executor_traits as a way to contain the free functions proposed in N4414 in an executor-extending object.
- Extension of executors to either be asynchronous or synchronous executions
- Extension of executors to allow for sequence based parallel executions (e.g. multiple executions of a function with different positions in a shape).

The notion that an executor can be synchronous or asynchronous is interesting, it allows for a clearer distinction between blocking and non-blocking behaviors (where the current executor definition is purely non-blocking for spawn).

By and large, the said proposal fits nicely into N4414 without significant modification (aside potentially from the naming), with some extension to allow there to be a synchronous execute function (which in most of the N4414 executors could be implemented as a spawn with a simple wait on a condition variable).

A very interesting discussion which follows from N4406 is about how to handle variations in the interface across implementations (free functions vs. core interface vs. a traits class). As defined

here, free functions remain free functions as they are wrappers around existing functionality without being executor-specific and essentially provide augmented interfaces to the executors.

Separately, the behaviors defined in N4406 allows for some functionality to be left out of the executor and for the traits class to fill in the gap (for example if an executor does not support batch execution, the traits can implement this interface on behalf of the executor). Effectively the trait class takes over as the primary interface into the executor classes and the executor is left to implement the subset which is appropriate for it. This is more than simply providing an interface to existing functionality but also allows there to be holes in the interface. On the other hand you could also provide the complete interface as part of the core executor concept as you effectively have to implement the traits object along with the executor unless a default traits object approximately does what you want.

In addition, a function rename is proposed in N4406 which can be discussed. The paper suggests execute() and async_execute() for the two proposed variations in behavior. The current proposal is effectively only proposing a single function and as such does not map perfectly, but an option would be to rename spawn() as async_execute(). Spawn has approximately the same implication of adding work to be executed at a later time. A name change is not proposed here, but is more meaningful in the light of N4406.

There is a separate discussion on the reflector which covers some of these issues and so further details can be found there.

II. Design

II.1. Reference Implementations

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

- https://github.com/ccmysen/executors_r6

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

Copyability of executors in the current proposal is not defined. Though many executors are not copyable and therefore the expectations of most code should be that an executor is not copyable unless a particular type of executor is expected.

II.3.b Ownership

A lack of copyability does not prevent ownership questions, though. As executors can and will be passed around different code, knowing the state of the executor you have can be important. This proposal assumes the C++ default answer, that it is the application's responsibility to get ownership correct.

There is an exception provided in this proposal for the system_executor, which has ownership which lasts past the end of the program lifetime (which thus requires a stage in shutdown in which the system executor is put into a shutdown state and drained). A generalization on this mechanism would allow any executor to be registered for shutdown past the end of the program so that libraries could safely take an executor and not be concerned about the lifetime of the underlying executor. Such a mechanism is not proposed here.

II.3.c. 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.d. 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 an 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.e. Executor Types

The proposed executors are as follows:

II.3.e.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.e.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.e.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.e.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.e.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.f. Polymorphic Executor Wrapper

N4414 proposed the concept of a type erased executor wrapper which can be passed into classes which prefer not to take a templatized type as a parameter. There was some concern about the cost incurred by all the type hiding and virtual dispatch when it is actually unnecessary.

The current proposed form of this using a simple template based inheritance would be less complex to implement. Such a concept would look like the following and could be provided as a default implementation for anyone who needs a polymorphic executor rather than a template based one.

```
class executor {
 public:
  virtual void spawn(executors::work&& fn) = 0;
};

template <typename Exec>
class executor_wrapper : public executor {
 public:
  executor_wrapper(Exec* exec);
  executor_wrapper(executor_impl<Exec>&& other);

  virtual void spawn(executors::work&& fn) {
    exec->spawn(std::forward<executors::work>(fn));
  }
};
```

Then calling a library which takes an executor* would be as simple as:

```
thread_pool_executor tpe;
library l(new executor_wrapper<thread_pool_executor>(&tpe));
```

II.3.g. `executor::work`

Because of the queueing behaviors of most executor implementations, a type erased function wrapper must be provided to any polymorphic interface 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 the proposal to include the polymorphic interface to executors, this 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 polymorphic interface to 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 polymorphic wrapper 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 create some divergence in behaviors depending on which interface you chose to use.

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(); });
```

III. Design Considerations

III.1. Changes since N4414

The general outline of changes is mentioned in the first section, but the summary of changes to the previous proposal are as follows:

- Wrapping executors into a callable object has been removed from the proposal
- executor_wrapper as previously defined has been has been re-used for the polymorphic executor template (which can be easily passed into code which does not desire a template based executor type definition). the previous type erasing wrapper has been dropped in favor of this one.
- Copyability has been removed as a concept for the core executor concept since it was considered more complex to implement and design than it was helpful to ownership semantics.
- Mentions of reference counting and ownership have largely been removed and now references to executors are generally taken in calls which require executors.
- Shutdown semantics of system_executor has been stated to be that the system_executor will initiate shutdown and fully drain immediately following the completion of main.

III.2. 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.

Generally executor objects are not copyable due to internal state which is generally not copyable. Individual implementations may chose to implement a copy/move interface, but these are not guaranteed on all executors.

```
executor {
 public:
  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&) = delete;
  thread_per_task_executor(thread_per_task_executor&&) = delete;

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

`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()`

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:
  system_executor(system_executor&& other) = delete;
  system_executor(const system_executor& other) = delete;

  static system_executor& get_executor();

  virtual ~system_executor();

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

 private:
  system_executor();
}
```

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) = delete;
  loop_executor(loop_executor&& other) = delete;
  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.

l`oop_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) = delete;
  serial_executor(serial_executor&& other) = delete;
  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 Executor Wrapper

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:
  virtual void spawn(executors::work&& fn) = 0;
};

template <typename Exec>
class executor_wrapper : public executor {
 public:
  executor_impl(Exec& exec);
  executor_impl(executor_impl<Exec>&& other);
```

```
  virtual void spawn(executors::work&& fn) {
    exec.spawn(std::forward<executors::work>(fn));
  }
};
```

executor_wrapper::executor_wrapper(Exec& exec)

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

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

void executor_wrapper::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>
void spawn(Exec& exec, Func&& func);
```

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

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

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

`future<std::decay<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.