Executors and schedulers, revision 4

Document number: ISO/IEC JTC1 SC22 WG21 N4143
Supersedes: 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: 2014-10-11
Authors: Chris Mysen
Reply-to: Chris Mysen <mysen@google.com>

I. Motivation

This proposal is designed as a variation on a paper presented in Redmond at the SG1 reflector which resolved several of the core concerns with the previous R3 design. The motivation behind the executor concept will not be fully restated here, but the core design principles 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 core functionality for task execution.

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, but leaves significant room for extension going forward.

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.

II. Design

II.1. Reference Implementations

There are two related reference implementations of this design. The first is the core/minimal set presented here, 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. 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_future(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).

```
template <typename Exec, typename Completion>
void start_processing(Exec& exec, inputs1& input, outputs1& output,
Completion&& comp) {
  latch l(input.size(), forward<Completion>(comp));
  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) {
  latch l(input.size(), forward<Completion>(comp));
  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 = set_executor(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: `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 seems overly simple, but much of the interesting behaviors required of an executor can be layered on top of this.

It is discouraged that a function passed into spawn would block as an arbitrary executor cannot guarantee that the thread which will un-block that thread will be run (for example, if there is a fixed pool of threads and the unblocking thread is stuck in the task queue).

II.3.1 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.3.1.1. `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.3.1.2. `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.3.1.3. `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 sum or all of the queued closures (closures added after loop has started will wait until the next call to the execution functions).

II.3.1.4. `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.

II.3.1.5. `system_executor`

A special executor which behaves like a thread pool 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. A recommendation for system_executor would be that it provide some forward progress guarantees (in which case it cannot be a completely bounded thread pool which cannot make any such guarantee).

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.3.2. `function_wrapper`

Because of the queueing behaviors of most executor implementations, a type erased function wrapper must be provided to executors. This was originally done with `std::function`, but this prevented move-only types from being usable with executors due to the requirements of `std::function` being copyable. 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.3.2. 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::function_wrapper&& fn);
```

Note that this uses the function_wrapper 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.3.2. Executor Reference Helper

Because the default executor classes are heavyweight and non-copyable, there becomes a need to pass executors around by reference instead of by value. In order to provide an abstraction for this which doesn't require unnecessary references, the library actually provides a specialized wrapper which provides the executor interface, but as a pure wrapper over an underlying executor as a reference for how to write general executor wrappers.

Let's take an example of a prioritized thread pool in which you select a thread priority at time of spawn:

```
class prioritized_thread_pool {
…
  template <typename Func> void spawn(Func&& func, Priority p);
  template <typename Func> void spawn(Func&& func);
};
```

In order to adapt this to code which expects a basic executor, you can write a wrapper which sets the spawn policy:

```
class priority_n_executor {
  priority_n_executor(prioritized_thread_pool& p, Priority p);
  priority_n_executor(const priority_n_executor& exec);

  template <typename Func> inline void spawn(Func&& func) {
    p_.spawn(forward<Func>(func), p);
  };
```

With the reference helper, you can accept a reference to an executor that detects the copyability of the underlying wrapper and avoids the extra indirection (and allows for more compiler inlining).

```
template <typename Exec>
void do_something(std::executor_ref<Exec> exec) {
  exec.spawn([] { /* some work */ });
}
priority_n_executor pn(pool, PRIORITY_HIGH);
do_something(priority_n_executor);
```

This pattern allows you to adapt some underlying class to the executor interface and pass the adaptor around as if it were any other executor without incurring additional cost from the reference.

II.4. 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_future(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, which would behave roughly like:

```
void finalize() {
  // finish up work
}
notifying_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(); });
```

In addition to the spawn helper functions, there is one simple additional helper which enables the Active Object pattern in which an object declares how it should execute. This is enabled by the fact that the executor interface has a single method for pushing work. This helper is effectively a small helper 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 set_executor function.

An example of this 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 comute_image(output_image, block_id);
void draw(result) {
  … do some drawing …
}
auto draw_task = std::set_executor(ge, bind(draw, final_image));
notifying_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(); }
  );
}
```

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).

III. Design Comparisons

III.1. Compared to - Executors Skeleton Proposal (http://tinyurl.com/pk3pc8t)

Conceptually this proposal is very similar to the proposal discussed in the SG1 reflector in Redmond, with the scheduled_executor concept removed from that proposal and some of the wording updated.

In particular, the straw polls there indicated a desire for a very simple baseline upon which other proposals could be layered. As a result of this and some implementation experience, a few changes were made:

- This included a preference to remove the scheduled_executor concept, which has been dropped here. This paper covers the same rationale & design point at the skeleton, with the only other main difference being a new formalization of the type erased wrappers which allow generic executors to be passed around code.
- The full hierarchy of type erased wrappers has been removed in favor of a single type erasing wrapper and the introduction of the concept of an `executor_ref` (a wrapper executor which is copyable).
- The free function `spawn()` remains as a way of generating futures and continuations, but the usage of variadic args (`Args&&...`) was dropped as using them can be difficult and much of the same value can be found using a lambda or bind at the call site.
- The internal function wrapper used by the executor classes to type erase the input functions has been made visible and is the primary function interface to the fully type erased executors classes (to slim down the interface to that class). This also enables setting a default function container to avoid type erasing costs on local executor objects.

III.2. Compared to - N3785 (revision 3 of this proposal)

There have been fairly major changes to the overall structure of the executors between N3785 and this proposal, mainly due to a sequence of concerns about the design choices in the original proposal. These are outlined in the proposal above, but primarily focused on the following concerns:

1. The use of an inheritance-based design instead of a concept-based one, which results in virtual function dispatches for any calls into the executor.

2. The lack of templatization of the interface and the use of std::function for tasks, disallowing the ability to pass arbitrary function types and disallowing move-only objects (particularly std::packaged_task).
3. The lack of templatization on the clock, forcing a clock to be chosen for timed tasks. The choice of including scheduled add functions was also questioned.
4. The choice to not handle exceptions and to consider this malformed and to call terminate.
5. The inclusion of an inline executor, which behaves in a way contrary to other executors in that it can cause you to disallow blocking behavior in your functions.
6. Lack of the ability to handle continuations directly in the library.
7. Concerns about the precise semantics for task destruction and synchronization relationships between tasks.

As a result, the main push was to take the previous proposal and make it work in a more template based approach and allow it to accept move-only objects.

Now some of the concerns about the costs of the virtual dispatch are somewhat unfounded because the costs of type erasure are significantly higher than the costs of virtual function dispatch, but this proposal attempts to allow even further optimizations in this regard.

Rest TBD...

III.3.Compared to - N4046 (and successor)

There are many underlying differences between N4046 (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:

- 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. Though in this proposal executors are both heavyweight and lightweight, as in, the

pattern is to create decorator objects which are copyable that layer on top of the uncopyable base executor objects. This formalizes the fact that in practice you need the lightweight object for two purposes: 1) basically something cheap, which can be as simple as a reference 2) a decorator which mutates the behavior of the underlying executor (for example, when the executor requires a priority mechanism and the reference can provide static priorities to the underlying executor object), which creates several behavioral advantages (namely that you can package behaviors into the reference object in a way which is executor-agnostic but behaves like a normal executor).

- The ability to attach a service onto an execution_context as a mechanism for extending the behavior of a core executor. This behaves somewhat analogously to the idea of an Extension Object design pattern, although in a less useful way. In the EO design pattern there is a relationship between the interface of the base class and the extensions (such that the extensions can layer on top of the interface provided by the initial object). This can be a useful way of allowing you to layer unrelated functionality onto an object which has a simpler interface without extending the concept. There are other behavioral patterns for this as well with different goals.
    - Using extension objects doesn't replace a good class design, it's intended to augment it, so the decision to move timing onto the extension objects is not entirely justified because in some cases the particular executor (or execution_context) has a native implementation of the same concept, in which case the extensions become duplicative to having a native interface on the executor (or worse, provide a less well designed implementation).
    - Extension objects also carry overhead, particularly in creating/managing the lifetime of the extensions which are attached. This is generally fine if you have a concrete lifetime in mind, but some execution contexts (system executors or gui executors).
    - The current proposal approaches extension slightly more traditionally, with the preference towards alternative implementations rather than requiring all executors to export the service. The obvious examples of this would be the concept of scheduled execution (spawn_after/spawn_at), as well as some alternatives (prioritization, thread-stealing, fork-join parallelism, batched executors).
- Alternative task execution constructs.
    - N4046 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.
            - In the proposal, dispatch unfortunately has different behaviors depending on the specific executor.
        - defer - this allows you to post to an executor from within an executing task, with the intention 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).

- In some executors, this actually delegates straight to post(), which means they are functionally equivalent (in which case defer is simply a hint

As a result of the significantly smaller API and less complex execution semantics, the implementations of this proposal are ⅕ the size of the Kohlhoff proposal. This isn't to say the simplicity is strictly better, but there is significant implementation complexity because of significant overhead imposed by the different dispatch mechanisms and the incorporation of the service concept into the different executors. Rough code size estimates are as follows:

- The core executor implementation and helper functions are ~1k lines of code, < 2k including tests.
- The Kohlhoff proposal is significantly more expansive (and includes a timer concept, and several helper functions), but if you look at the core proposal, it is still ~7k lines of code for the core concept without tests.

III.4. Exception Handling

Exception handling has been left out explicitly as there really is not a clean way to handle exception forwarding unless there is a clear 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.

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 N4046 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 a simple, but powerful way of adapting the extended executor implementations to existing code: the reference wrapper objects.

In practice this allows you to take custom extensions to the core interface and wrap it in the simple spawn interface fairly cheaply. If one looks at the priority_n_executor wrapper around a thread pool executor, in which a more extensive API is visible but not reliable across executors (e.g. `spawn(func, 10)`), in such a way that all the existing libraries continue to work. 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 N4046 is in the presence of alternative dispatch approaches (namely the presence of `post()` and `defer()`). Functionally these serve as different optimizations for latency or overhead. `dispatch()` in particular has semantics which are very unique (in that it requests low latency and potentially blocking calls), and are difficult to emulate without native executor support. Unfortunately using it correctly has proven to be difficult/
- 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 righ design approach here and whether it’s appropriate to standardize this or whether this a conceptual layer on top of the core executor.

- 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 the last 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.

VI. Proposed Wording

VI.A. Executor Concept

Concept interface:

```
class {
 public:
  typedef wrapper_type;

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

The concept interface also includes a type trait for the type of the wrapper function used to store functions internally (useful when implementing wrappers around the executor which need to also track/store functions as in the serial_executor).

VI.A.1. thread_per_task_executor

```
class thread_per_task_executor {
 public:
  thread_per_task_executor(thread_per_task_executor&) = delete;
  static thread_per_task_executor& get_executor();
  template<class Func> void spawn(Func&& func);
};
```

The thread_per_task_executor provides the core spawn interface with no queueing where each task is provided with a thread as if it were an entirely unique thread.

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

VI.A.2. thread_pool_executor

```
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();
  // 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)
};
```

VI.A.3. 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.

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

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

VI.A.4. loop_executor

The loop executor is described in previous papers. Effectively it takes over the thread of the caller who calls loop() run_queued_closures() or try_run_one_closure(), with different selections of tasks

```
class loop_executor {
 public:
  explicit loop_executor();
  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);
};
```

VI.A.5. serial_executor

Serial executor acts to serialize functions queued in it. Guarantees that functions will execute in order on the provided executor, but does not make any guarantees that the same underlying thread would be used for this purpose.

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

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

VI.B Executor Ref

```
template <typename Exec, typename CopyEnable=void>
```

```
class executor_ref {
 public:
  executor_ref() = delete;
  executor_ref(Exec& exec);
  executor_ref(const executor_ref& other);
  virtual ~executor_ref();

  template <typename Func> void spawn(Func&& func)

  Exec& get_contained_executor();
}
```

VI.C Type Erased Executor

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

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

  void spawn(function_wrapper&& fn);
};
```

VI.D Free Functions & Helper Objects

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

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

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

template <typename Exec, typename Func>
class task_wrapper {
 public:
  task_wrapper(Exec& exec, Func&& func);

  void operator()();
  Exec& get_executor();

  // Allows optimizations where func is run on the same executor as
  // the caller and thus doesn't need to call spawn.
  Func&& contained_function();
};

template <typename Exec, typename Func>
task_wrapper<Exec, Func>&& set_executor(Exec& exec, Func&& func)
```

VI.E. Function wrapper interface

```
class function_wrapper {
 public:
  function_wrapper() = delete;
  function_wrapper(const function_wrapper&) = delete;
  function_wrapper(function_wrapper&& other);
  virtual ~function_wrapper();

  template <typename T> function_wrapper(T&& t);

  void operator()();
};
```