Executors and schedulers, revision 1

Document number: ISO/IEC JTC1 SC22 WG21 N3562
Supersedes: ISO/IEC JTC1 SC22 WG21 N3378=12-0068
Date: 2013-3-15
Authors: Matt Austern, Lawrence Crowl, Chandler Carruth, Niklas Gustafsson, Chris Mysen, Jeffrey Yasskin
Reply-to: Matt Austern <austern@google.com>

This paper is a proposal for *executors*, objects that can execute units of work packaged as function objects, in the form of an abstract base class and several concrete classes that inherit from it. It is based on components that are heavily used in internal Google and Microsoft code, with changes to better match the style of the C++ standard.

This proposal discusses the design decisions behind the API and also includes a first draft of formal wording for the working paper.

I. Motivation

Multithreaded programs often involve discrete (sometimes small) units of work that are executed asynchronously. This often involves passing work units to some component that manages execution. In C++11, for example, we already have `std::async`, which potentially executes a function asynchronously and eventually returns its result in a `future`. ("As if" by launching a new thread.)

If there is a regular stream of small work items then we almost certainly don't want to launch a new thread for each, and it's likely that we want at least some control over which thread(s) execute which items. In Google's internal code, it has been convenient to represent that control as multiple *executor* objects. This allows programs to start executors when necessary, switch from one executor to another to control execution policy, and use multiple executors to prevent interference and thread exhaustion.

II. Overview and design issues

The fundamental basis of the design is the `executor` class, an abstract base class that takes closures and runs them, usually asynchronously. There are multiple implementations of that base class. Some specific design notes:

- Thread pools are a common and obvious implementation of the `executor` interface, and this proposal does indeed include thread pools, but other implementations also exist.
- The choice of which executor to use is explicit. This is important for reasons described in the *Motivation* section. In particular, consider the common case of an asynchronous operation that itself spawns asynchronous operations. If both operations ran on the same

executor, and if that executor had a bounded number of worker threads, then we could get deadlock. Programs often deal with such issues by splitting different kinds of work between different executors.

- There is a global default executor, of unspecified concrete type, that can be used when detailed control is unnecessary. There is a mechanism to change the default executor. Changing the default executor is ugly but it is sometimes useful, especially in tests.
- The interface is based on inheritance and polymorphism, rather than on templates, for two reasons. First, executors are often passed as function arguments, often to functions that have no other reason to be templates, so this makes it possible to change executor type without code restructuring. Second, a non-template design makes it possible to pass executors across a binary interface: a precompiled library can export a function one of whose parameters is an `executor*`. The cost of an additional virtual dispatch is almost certainly negligible compared to the other operations involved.
- Conceptually, an executor puts closures on a queue and at some point executes them. The queue is always unbounded, so adding a closure to an executor never blocks. (Defining "never blocks" formally is challenging, but informally we just mean that `add()` is an ordinary function that executes something and returns, rather than waiting for the completion of some potentially long running operation in another thread.)

One especially important question is just what a closure is. This proposal has a very simple answer: `std::function<void()>`. One might question this for three reasons.

First, this decision means that there is no direct provision for returning values from a work unit that's passed to an executor. That is, there is no equivalent of `std::future`. A work unit is a closure that takes no arguments and returns no value. This greatly simplifies the interface. This is indeed a limitation on user code, but in practice we haven't found it a terribly serious limitation. In practice it's often the case that when a work item finishes we're less interested in returning a value than in performing some other action. Also, since a closure can package arbitrary information, users who need to obtain results can provide a `std::packaged_task`. (Or do something similar manually.)

Second, one might wonder why this is a single concrete type, rather than (say) a template parameter that can be instantiated with an arbitrary function object of no arguments and `void` return type. One strong reason for that choice is that it's existing practice. Another is that a template parameter would complicate the interface without adding any real generality. In the end an executor class is going to need some kind of type erasure to handle all the different kinds of function objects with `void()` signature, and that's exactly what `std::function` already does. Most fundamentally, of course, `executor` is an abstract base class and `add()` is a virtual member function, and function templates can't be virtual.

Third, one might worry about performance concerns with `std::function<void()>`. Again, however, any mechanism for storing closures on an executor's queue will have to use some

form of type erasure. There's no reason to believe that a custom closure mechanism, written just for `std::executor` and used nowhere else within the standard library, would be better in that respect than `std::function`. (One theoretical advantage of a template-based interface is that the executor might sometimes decide to execute the work item inline, rather than enqueueing it for asynchronous, in which case it could avoid the expense of converting it to a closure. In practice this would be very difficult, however: the executor would somehow have to know which work items would execute quickly enough for this to be worthwhile.)

Another important questions about executors is their interaction with time: should an executor just promise to execute closures at some unspecified time, or should there be some mechanism for users to supply more specific requirements? In practice it has proven useful for users to be able to say things like "run this closure, but no sooner than 100s from now." This is useful for periodic operations in long-running systems, for example. We provide two versions of this facility: `add_after`, which runs a closure after a specified duration, and `add_at`, which runs a closure at (or, more precisely, no sooner than) a specified time point.

There are several important design decisions involving that time-based functionality. First: how do we handle executors that aren't able to provide it? The issue is that `add_at` and `add_after` involve significant implementation complexity. In Microsoft's experience it's important to allow very simple and lightweight executor classes that don't need such complicated functionality. We address this by providing two abstract base classes, `executor` and `scheduled_executor`, the latter of which inherits from the former. The time-based functionality is part of the scheduled_executor interface but not part of executor.

Second, how should we specify time? The libraries that this proposal is based on just use some integral type to specify time duration (with time measured in milliseconds or microseconds), but this proposal uses standard durations and time points. This requires some thought since `chrono::duration` and `chrono::time_point` are class templates, not classes. Some standard functionality, like `sleep_until` and `sleep_for`, is templatized to deal with arbitrary `duration` and `time_point` specializations. That's not an option for an executor library that uses virtual functions, however, since virtual member functions can't be function templates. There are a number of possible options:

1. Redesign the library to make `executor` a concept rather than an abstract base class. We believe that this would be invention rather than existing practice, and that it would make the library more complicated, and less convenient for users, for little gain.
2. Make `executor` a class template, parameterized on the clock. As discussed above, we believe that a template-based design would be less convenient than one based on inheritance and runtime polymorphism.
3. Pick a single clock and use its `duration` and `time_point`.

We chose the last of those options, largely for simplicity.

Finally, we need to think about how executors interact with exceptions. If we fail to acquire a

resource in an executor's constructor (if, for example, `thread_pool` fails to start its threads), then the obvious way to signal that error is by throwing an exception. A more interesting question is what happens if a user closure throws an exception. The exception will in general be thrown by a different thread than the one that added the closure or the thread that started the executor, and may be far separated from it in time and in code location. The decision we made is that the program will terminate if any closure passed to an executor throws an exception. We have several reasons for that decision:

- It's consistent with the behavior of the internal libraries that this proposal was based on.
- It's consistent with the way that `std::thread` behaves.
- Users who need to propagate information from closures' exceptions can wrap them and store them in data structures on the side, just as they can do with any other information that closures generate.
- Most of the use cases where this would matter could be handled by std::future.
- Such a facility has the potential to be dangerous or complicated because it would require aggregating multiple exceptions thrown by closures executing simultaneously.

In general the goal of this proposal was to standardize prior art, without any design innovation. As a consequence, this proposal mostly involves classes that have been used internally. (With mechanical changes, of course, such as changing names to match the standard's styles and using standard facilities, like `std::function<void()>`, instead of internal pre-standard equivalents.) The classes in this proposal are just a subset of what might be proposed, and the Future Directions section at the end describes some other executors that might be standardized in the future.

III. Proposed wording

This proposal includes two abstract base classes, `executor` and `scheduled_executor` (the latter of which inherits from the former); several concrete classes that inherit from `executor` or `scheduled_executor`; and several utility functions.

Executors library summary

Subclause	Header(s)
III.1 [executors.base]	`<executor>`

III.2 [executors.classes]
III.2.1 [executors.classes.loop]
III.2.2 [executors.classes.serial]
III.2.3 [executors.classes.thread_pool]
	
	`<loop_executor>`
	`<serial_executor>`
	`<thread_pool>`

III.1 Executor base classes [executors.base]

The <executor> header defines abstract base classes for executors, as well as non-member functions that operate at the level of those abstract base classes.

Header `<executor>` synopsis

```
class executor;
class scheduled_executor;

static scheduled_executor* default_executor();
static void set_default_executor(scheduled_executor* executor);

executor* singleton_inline_executor();
```

III.1.1 Class executor [executors.base.executor]

Class `executor` is an abstract base class defining an abstract interface of objects that are capable of scheduling and coordinating work submitted by clients. Work units submitted to an executor may be executed in one or more separate threads.

The initiation of a work unit is not necessarily ordered with respect to other initiations. [Note: Concrete executors may, and often do, provide stronger initiation order guarantees. Users may, for example, obtain FIFO guarantees by using the `serial_executor` wrapper.] There is no defined ordering of the execution or completion of closures added to the executor. [Note: The consequence is that closures should not wait on other closures executed by that executor. Mutual exclusion for critical sections is fine, but it can't be used for signalling between closures. Concrete executors may provide stronger execution order guarantees.]

```
class executor {
public:
  virtual ~executor();
  virtual void add(function<void()> closure) = 0;
  virtual size_t num_pending_closures() const = 0;
};
```

`executor::~executor()`

Effects: Destroys the executor.

Synchronization: All closure initiations happen before the completion of the executor

destructor. [Note: As a consequence, all closures that will ever execute will have completed before the completion of the executor destructor, and programmers can protect against data races with the destruction of the environment. Whether or not a concrete executor initiates all closures is defined by the semantics defined by that concrete executor.]

`void executor::add(std::function<void> closure);`

Effects: The specified function object shall be scheduled for execution by the executor at some point in the future.

Synchronization: completion of `closure` 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: If invoking `closure` throws an exception, the executor shall call `terminate`.

`size_t executor::num_pending_closures() ;`

Returns: the number of function objects waiting to be executed. [Note: this is intended for logging/debugging and for coarse load balancing decisions. Other uses are inherently risky because other threads may be executing or adding closures.]

III.1.2 Class `scheduled_executor` [executors.base.scheduled_executor]

Class `scheduled_executor` is an abstract base class that extends the executor interface by allowing clients to pass in work items that will be executed some time in the future.

```
class scheduled_executor : public executor {
public:
  virtual void add_at(chrono::system_clock::time_point abs_time,
                      function<void()> closure) = 0;
  virtual void add_after(chrono::system_clock::duration rel_time,
                         function<void()> closure) = 0;
};
```

`void add_at(chrono::system_clock::time_point abs_time,`
`function<void()> closure);`

Effects: The specified function object shall be scheduled for execution by the executor at some point in the future no sooner than the time represented by `abs_time`.

Synchronization: completion of `closure` on a particular thread happens before

destruction of that thread's thread-duration variables.

Error conditions: If invoking `closure` throws an exception, the executor shall call `terminate`.

`void add_after(chrono::system_clock::duration rel_time,`
`function<void()> closure);`

Effects: The specified function object shall be scheduled for execution by the executor at some point in the future no sooner than time rel_time from now.

Synchronization: completion of `closure` on a particular thread happens before destruction of that thread's thread-duration variables.

Error conditions: If invoking `closure` throws an exception, the executor shall call `terminate`.

III.1.3 Executor utility functions [executors.base.utility]

`scheduled_executor* default_executor();`

Returns: a non-null pointer to the default executor defined by the active process. If `set_default_executor` hasn't been called then the return value is a pointer to an executor of unspecified type. [Note: implementations are encouraged to ensure that separate tasks added to the initial default executor can wait on each other without deadlocks.]

`void set_default_executor(scheduled_executor* executor);`

Effect: the default executor of the active process is set to the given executor instance.
Requires: executor shall not be null.
Synchronization: Changing and using the default executor is sequentially consistent.

`executor* singleton_inline_executor();`

Returns: a non-null pointer to an executor that immediately executes any closure that is added to it using `add()`. Multiple invocations return a pointer to the same object.

III.2 Concrete executor classes [executors.classes]

This section defines executor classes that encapsulate a variety of closure-execution policies.

III.2.1 Class loop_executor [executors.classes.loop]

Header <loop_executor> synopsis

```
class 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 executor {
public:
  loop_executor();
  virtual ~loop_executor();
  void loop();
  void run_queued_closures();
  bool try_run_one_closure();
  void make_loop_exit();

  // [executor methods omitted]
};
```

loop_executor::loop_executor()

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

loop_executor::~loop_executor()

Effects: Destroys the `loop_executor` object. Any closures that haven't been executed by a closure-executing method when the destructor runs will never be executed.
Synchronization: Must not be called concurrently with any of the closure-executing methods.

void loop_executor::loop()

Effects: Runs closures on the current thread until `make_loop_exit()` is called.
Requires: No closure-executing method is currently running.

void loop_executor::run_queued_closures()

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

`loop(); }` because that would cause early exit from a subsequent invocation of `loop()`.]

Requires: No closure-executing method is currently running.

Remarks: This function is primarily intended for testing.

`bool loop_executor::try_run_one_closure()`

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

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

Requires: No closure-executing method is currently running.

Remarks: This function is primarily intended for testing.

`void loop_executor::make_loop_exit()`

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

III.2.2 Class serial_executor [executors.classes.serial]

Header `<serial_executor>` synopsis

```
class serial_executor;
```

Class `serial_executor` is an adaptor that runs its closures on a particular thread by scheduling its closures on another (not necessarily single-threaded) executor. It runs added closures in FIFO order inside a series of closures added to an underlying executor. Earlier `serial_executor` closures happen before later closures. The number of `add()` 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`. [Note: this is because `serial_executor` can batch add() calls to the underlying executor.]

```
class serial_executor : public executor {
public
  explicit serial_executor(executor* underlying_executor);
  virtual ~serial_executor();
  executor* underlying_executor();

  // [executor methods omitted]
```

```
};
```

`serial_executor::serial_executor(executor* underlying_executor)`

Requires: `underlying_executor` shall not be null.
Effects: Creates a `serial_executor` that executes closures in FIFO order by passing them to `underlying_executor`. [Note: several serial_executor objects may share a single underlying executor.]

`serial_executor::~serial_executor()`

Effects: Finishes running any currently executing closure, then destroys all remaining closures and returns. If a `serial_executor` is destroyed inside a closure running on that `serial_executor` object, the behavior is undefined. [Note: one possible behavior is deadlock.]

`executor* serial_executor::underlying_executor()`

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

III.2.3 Class thread_pool [executors.classes.thread_pool]

Header <thread_pool> synopsis

```
class thread_pool;
```

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.

```
class thread_pool : public scheduled_executor {
 public:
  explicit thread_pool(int num_threads);
  ~thread_pool();

  // [executor methods omitted]
};
```

`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 closures (if any) to complete, then joins and destroys the threads.

V. Future directions and related work

There are many other useful thread pool classes, in addition to those in this proposal. Several of them are in use within Google and Microsoft. In particular, some of the standard policy choices are:

- Creating a fixed number of threads when it's constructed and multiplexing closures on top of them. This thread pool risks deadlock if its closures wait on each other to finish.
- Starting a new thread whenever no existing thread is available to run a new closure. This risks memory exhaustion if it's presented with a burst of work.
- Allowing the number of threads to vary within a user-specified range.
- Using more advanced techniques, with the goal of running the minimum number of threads necessary to keep the system's processors busy. (One such technique is a two-level design, where a “thread manager” maintains a global variable-sized collection of threads and a set of “managed queues” implement the executor interface and feeds closures into the thread manager.)

This proposal only includes the first type of policy. Future proposals may include others, especially the last.

The Google executor library provides many options for starting threads, including thread names, priorities, and user-configurable stack sizes. Those options have proven useful, but they rely on functionality that the underlying `std::thread` class does not provide. We may submit a future proposal that includes extensions to `std::thread`.

Sometimes it’s useful to add a closure to an executor and then later remove it, before it has executed. For example, this sometimes helps with clean shutdown. The internal Google and Microsoft executor libraries both have cancellation mechanisms. We omitted those mechanisms from this proposal because they’re complicated, but we could add them to a future version if there is sufficient interest.

There is an obvious extension to executors and/or to `std::async`: something that provides essentially the semantics of async, but that also allows the user to explicitly specify which executor will be used for execution. Possibilities include an `async` member function, an overload of `std::async` that takes an executor in place of the policy, or a launch policy that explicitly uses the default executor. This is being proposed separately, in N3558.

Executor implementations differ in the order they call closures (FIFO, priority, or something else), whether they provide a happens-before relation between one closure finishing and the next closure starting, whether it's safe for one closure on a given executor to block on completion of another closure it added to the same executor, and other properties. Users sometimes want to write functions that accept only executors satisfying the properties they rely on. A future paper may invent an appropriate mechanism for such queries and constraints.

This proposal takes a simple approach to executor shutdown, generally just dropping closures that haven't started yet. Java's Executor library, on the other hand, provides a flexible mechanism for users to shut down executors with control over how closures complete after shutdown has started. C++ should consider what's appropriate in this area.

VII. Changes since the previous version

Differences between this document (R1) and ISO/IEC JTC1 SC22 WG21 N3378=12-0068 (R0):

- R0 was entirely a Google proposal, based on Google's internal executor and thread pool classes. R1 is a joint Google/Microsoft proposal and includes elements of Google's executor and Microsoft's scheduler.
- In response to feedback in and after the Portland meeting, R1 eliminates the notion of executors with finite queue length. We now document `add()` as non-blocking in all circumstances; `try_add()` has been removed. Finite-sized queues are occasionally useful, but they add complexity and they don't need to be part of the base interface; it's always possible for users to implement wrappers with finite queues.
- R0 included a `thread_manager` class, with various associated helper classes. R1 removes it. It's being removed simply to narrow the scope of this proposal and reduce the amount of work we have to do. Executors are useful even without `thread_manager`, and `thread_manager` can be added later as a separate proposal.
- In R0, `add_at` and `add_after` were part of the executor abstract base class. In this revision they have been moved to a new `scheduled_executor` abstract base class, which inherits from `executor`.
- In R0, default executor management was via static member functions of class executor. In R1 they have been changed to non-member functions. (Largely because that's more natural in light of the `executor`/`scheduled_executor` split.)
- The non-member functions `new_inline_executor` and `new_synchronized_inline_executor` that were present in R0 have been removed from R1. There are use cases for those functions, but `singleton_inline_executor`, possibly in conjunction with a `serial_executor` wrapper, is almost always good enough.
- The design discussions in R0 were in several different sections. As of version R1 they have now been consolidated.
- R0's "proposed interface" section is now a "proposed wording" section.
- R0's "synchronization" section has been removed. The guarantees in it have been moved to the proposed wording section.