Executors and schedulers, revision 3

Document number: ISO/IEC JTC1 SC22 WG21 N3785
Supersedes: ISO/IEC JTC1 SC22 WG21 N3731
ISO/IEC JTC1 SC22 WG21 N3378=12-0068
ISO/IEC JTC1 SC22 WG21 N3562
Date: 2013-10-8
Authors: Chris Mysen, Niklas Gustafsson, Matt Austern, Jeffrey Yasskin
Reply-to: Chris Mysen <mysen@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, including the ability to schedule work on GUI threads, scheduling work on a donor thread, as well as several specializations of thread pools.

- 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 strong value in having a default executor, of unspecified concrete type, that can be used when detailed control is unnecessary (which will be the common case). Changing the default executor is ugly but it is sometimes useful, especially in tests. As such there is some discussion about having such a default executor or a similar mechanism without requiring that executors be constructed and passed through to every function.
- 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.)

II.1 Use of std::function<void()>

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, but in practice the implementation of a template based approach or another approach is impractical.

First, 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. Most fundamentally, of course, `executor` is an abstract base class and `add()` is a virtual member function, and function templates can’t be virtual. Another reason 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.

Second, this decision means that there is no direct provision for returning values from a work

unit that's passed to an executor (similar to what Java does in the java ExecutorService). That is, there is no equivalent of a call which returns a `std::future`. A work unit is a closure that takes no arguments and returns no value. 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. Moreover, since a closure can package arbitrary information, users who need to obtain results can provide a `std::packaged_task`. This is a notable difference with how Java executors work, but the C++ standard uses `std::async` for future creation, so a more consistent approach here is to push this behavior into a new specification of `std::async` (covered in N3721).

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.)

II.2 Scheduled Work

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.

II.3 Exception Handling

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. As such, unhandled exceptions are considered a program error because they are difficult to signal to the caller. The decision we made is that an exception from a closure is ill-formed and the implementation must call std::terminate. 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 might matter could be handled by std::future as well.
- 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, with as little design innovation as possible. As a consequence, this proposal mostly involves classes that have been used as described. (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.

II.4 Default Executor

In the original proposal and in R1, there was a concept of a default executor in the proposed wording. The proposed wording for such a concept is shown below (modified from the original proposal to use `shared_ptr` instead of raw pointers). This has proven to be extremely useful, but there is controversy in the existence of these functions because they are effectively a single shared resource with unspecified semantics and if code heavily relies on the existence such a common executor it becomes difficult to provide explicit semantics to other libraries or portions of code. The facility to change the default executor (`set_default_executor`) exposes this to some degree but does not give fine grained control. There is also some concern that providing a default executor will cause code to rely on semantics of particular implementations unintentionally which will potentially make implementations non-portable.

Due to some of these issues, this is being removed from the proposal to let the rest make progress, but there will be a follow up proposal for this topic in particular.

> ***Executor utility functions [executors.base.utility]***
>
> **`shared_ptr<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 shared_ptr 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(shared_ptr<executor> executor);`**
>
> *Effect:* the default executor of the active process is set to the given executor instance. The previously defined executor is not destroyed until nobody has ownership of the previously used executor. The current proposal does not offer an explicit shutdown semantic, though this will minimally behave like a join on currently executing threads.
> *Requires*: executor shall not be null.
> *Synchronization*: Changing and using the default executor is sequentially consistent.

III. Executor Implementations

Several possible implementations exist of the executor class and in practice there are a number of main groups of executors which have been found to be useful in real-world code (more implementations exist, this is simply a high level classification of them). These differ along a couple main dimensions, how many execution contexts will be used, how they are selected, and how they are prioritized.

1. Thread Pools
    a. Simple unbounded thread pool, which can queue up an unbounded amount of work and maintains a dedicated set of threads (up to some maximum) which dequeue and execute work as available.
    b. Bounded thread pools, which can be implemented as a specialization of (1) with a bounded queue or semaphore, which limits the amount of queuing in an attempt to bound the time spent waiting to execute and/or limit resource utilization for work tasks which hold state which is expensive to hold. This was in prior versions of the proposal but has been since removed due to deadlock potential. It should be noted, though, that bounded pools cannot easily be implemented without an interface change as the add() functions would need to be able to fail.
    c. Thread-spawning executors, in which each task always executes in a new thread.
    d. Prioritized thread pools, which have tasks which are not equally prioritized such that work can move to the front of the execution queue if necessary. This requires a special comparator or prioritization function to allow for work ordering and normally is implemented as a blocking priority queue in front of the pool instead of a blocking queue. This has many uses but is a somewhat specialized in nature and would unnecessarily clutter the core interface.
    e. NUMA-Aware thread pools, which tend to prefer work being sent to particular threads to allow cache or location sensitive algorithms to prioritize locality. A google implementation of this is done by creating hierarchical executors which have threads which are tied to NUMA nodes and exposes the locality in the executor interface. More complex versions of this could be implemented by having locality hints and potentially some work stealing between nodes. This was left out as a fairly specialized interface which can be built on top of the existing executor interface.
    f. Work stealing thread pools, this is a specialized use case and is encapsulated in the ForkJoinPool in java, which allows lightweight work to be created by tasks in the pool and either run by the same thread for invocation efficiency or stolen by another thread without additional work. These have been left out until there is a more concrete fork-join proposal or until there is a more clear need as these can be complicated to implement.
2. Mutual exclusion executors
    a. Serial executors, which guarantee all work to be executed such that no two tasks will execute concurrently. This allows for a sequence of operations to be queued in sequence and that sequential order is maintained and work can be queued on a separate thread but with no mutual exclusion required.
    b. Loop executor, in which one thread donates itself to the executor to execute all queued work. This is related to the serial executor in that it guarantees mutual exclusion, but instead guarantees a particular thread will execute the task. These are particularly useful for testing purposes where code assumes an

executor but testing code desires control over execution.
    c. GUI thread executor, where a GUI framework can expose an executor interface to allow other threads to queue up work to be executed as part of the GUI thread. This behaves similarly to a loop executor, but must be implemented as a custom interface as part of the framework.
3. Inline executors, which execute inline to the thread which calls `add()`. This has no queuing and behaves like a normal executor, but always uses the caller's thread to execute. This allows parallel execution of tasks, though. This type of executor is often useful when there is an executor required by an interface, but when for performance reasons it's better not to queue work or switch threads. This is often very useful as an optimization for task continuations which should execute immediately or quickly and can also be useful for optimizations when an interface requires an executor but the work tasks are too small to justify the overhead of a full thread pool. There has been some discussion of whether this is important to standardize as it is more of an optimization, but has proven to be very useful in practice.

A question arises of which of these executors (or others) be included in the standard. There are use cases for these and many other executors (there are many more implemented within the Google codebase for more specialized use cases). Often it is useful to have more than one implemented executor (e.g. the thread pool) to have more precise control of where the work is executed due to the existence of a GUI thread, or for testing purposes.

In practice, Microsoft and Google have found a few core executors to be frequently useful and these have been outlined here as the core of what should be in the TS, if common use cases arise for alternative executor implementations, they can be added in the future. The core set provided here are: thread pools, serial executors, loop executors, inline executors, and thread-spawning executors.

IV. Cancellation

The current proposal does not provision for the ability to cancel work, though this is a common need of code which creates work optimistically or where work may not be required to complete. This has been omitted explicitly from the executor interface because it creates complications and can be handled externally to the executor classes. Java commonly handles this via exceptions but this approach is unnecessary and creates complication to the executor interface.

A common approach to this is to provide an external cancellation object which can be called to indicate cancellation. The concept of a cancellation token is akin to the Microsoft PPL `cancellation_token`, where a task would check the cancellation status of the token and external code would set the token state to cancelled when necessary. The common case is that code does not require cancellation and the semantics of cancellation are inconsistent. Thus, in cases where cancellation is needed, a token based approach is simple to implement.

As this is tangential to the executor interface and because this may be useful outside of the executors, the cancellation interface will not be outlined here, but is the preferred approach.

V. 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)
V.1 [executors.base]	`<executor>`

V.2 [executors.classes]
V.2.1 [executors.classes.thread_pool]
V.2.2 [executors.classes.serial]
V.2.3 [executors.classes.loop]
V.2.4 [executors.classes.inline]
V.2.5 [executors.classes.thread]
	
	`<thread_pool>`
	`<serial_executor>`
	`<loop_executor>`
	`<inline_executor>`
	`<thread_executor>`

V.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;
```

V.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. Implementations are required to avoid data races when work units are submitted concurrently.

All closures are defined to execute on some thread, but which thread is largely unspecified. As such accessing a `thread_local` variable is defined behavior, though it is unspecified which thread's `thread_local` will be accessed.

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 serial execution 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 uninitiated_task_count() const = 0;
};
```

`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.]

`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. May throw exceptions if add cannot complete (due to shutdown or other conditions).

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: The invoked `closure` shall not throw an exception.

size_t executor::uninitiated_task_count();

Returns: the approximate 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.]

V.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(const chrono::system_clock::time_point& abs_time,
                      function<void()> closure) = 0;
  virtual void add_after(const chrono::system_clock::duration& rel_time,
                         function<void()> closure) = 0;
};
```

void add_at(const 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: The invoked `closure` shall not throw an exception.

void add_after(const 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:The invoked `closure` shall not throw an exception.

V.2 Concrete executor classes [executors.classes]

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

V.2.1 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.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 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 (1.10 [intro.multithread]) the execution of `c2` begins, or vice versa. If `e.add(c1)` happens before `e.add(c2)`, then `c1` is executed before `c2`.

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

Effects: Creates a `serial_executor` that executes closures, in an order that respects the happens-before ordering of the `serial_executor::add()` calls, by passing the closures 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.

executor& serial_executor::underlying_executor()

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

V.2.3 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.

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.]

V.2.4 Class inline_executor [executors.classes.inline]

Header <inline_executor> synopsis

```
class inline_executor;
```

Class `inline_executor` is a simple executor which intrinsically only provides the add()

interface as it provides no queuing and instead immediately executes work on the calling thread. This is effectively an adapter over the executor interface but keeps everything on the caller's context.

```
class inline_executor : public executor {
public
  explicit inline_executor();
  // [executor methods omitted]
};
```

inline_executor::inline_executor()

Effects: Creates a dummy executor object which only responds to the add() call by immediately executing the provided function in the caller's thread.

V.2.5 Class thread_executor [executors.classes.thread]

Header <thread_executor> synopsis

```
class thread_executor;
```

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

```
class thread_executor : public executor {
 public:
  explicit thread_executor();
  ~thread_executor();

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

`thread_executor::thread_executor()`

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

`thread_executor::~thread_executor()`

Effects: Waits for all added 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:

- 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`.

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 uses a default executor mechanism. This is being proposed separately, in N3721.

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. It should be noted, though, that many of the common use cases for executors are such that executor lifetime matches the lifetime of the application. The use cases for controlled shutdown are more specialized and are often executor specific, so adding complex shutdown semantics is currently left to specific implementations of the executor interface.

VII. Changes

Differences between R3 and ISO/IEC JTC1 SC22 WG21 N3731 (R2):

- The name of the member function giving an approximate count of the number of

pending closures has been changed from `num_pending_closures()` to `uninitiated_task_count()`, and it has been clarified that the return value is an approximation.
- A new concrete executor class, `thread_executor`, has been added.
- A statement about thread semantics and `thread_local` has been added.
- The ordering guarantees for serial executors have been clarified.
- It is now explicit that `add()` must be thread safe.
- Clarified language related to `executor`'s destructor.
- Changed lifetime issues of `serial_executor`'s underlying executor. Changed the constructor to take `underlying_executor` by reference.

Differences between R2 and ISO/IEC JTC1 SC22 WG21 N3562 (R1):

- singleton_inline_executor was removed from the executor namespace to simplify the executor namespace and since the singleton part of such an executor is not required, instead added it as an additional executor type which can be instantiated as needed
- default_executor() and set_default_executor() now have a cleanup semantic and use the base executor instead of scheduled executor. They have also been moved out of the proposed wording and into a separate section to facilitate more discussion without blocking the base proposal.
- added justification around the use of std::function for the interface specification
- included more detailed descriptions of executor types and use cases, including some executors which have been implemented but not included in the proposal
- discussion of cancellation is added for completeness (though omitted from the formal proposal)
- additional wording around executor shutdown
- turned the time objects into references

Differences between 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.