A preliminary proposal for work executors

Document number: ISO/IEC JTC1 SC22 WG21 N3378=12-0068
Date: 2012-2-24
Authors: Matt Austern, Lawrence Crowl, Chandler Carruth, Chris Mysen, Jeffrey Yasskin
Reply-to: Jeffrey Yasskin <jyasskin@google.com>

This paper is a preliminary 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 code, with some changes to better match the style of the C++ standard. Those changes are described in Section III, *Changes from Google's internal version*.

This proposal describes the proposed API, but does not provide formal wording that could be included in the standard. Assuming there is sufficient interest, a followup proposal will include formal language and a complete open source implementation.

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, we have found it 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. Sketch of the overall design

The fundamental basis of the design is the `executor` class, an abstract base class that takes callbacks 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, largely for simplicity. 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. 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 may be finite or unbounded; different executors have different queue policies. If the queue is finite, then adding a closure to an executor may block until there is a slot in the queue. In some cases that can be a nuisance, so we also include a member function that adds a closure to an executor if and only if that can be done without blocking.

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

III. Synchronization

The abstract executor interface provides rather minimal synchronization guarantees. Specific concrete executors may provide stronger guarantees.

All operations on destructors, after construction and before destruction, are data-race-free with respect to other operations.

All closure initiations happen before the completion of the executor destructor. Completion of all initiated closures happens before the completion of the executor destructor. 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 that concrete executor.

Completion of all initiated closures on a particular thread happens before destruction of all thread-duration variables in that thread. The consequence is that closures may use thread-duration variables. Closures may execute in any thread. The consequence is that threads may not depend on using the thread-duration variables of a particular thread. Concrete executors may provide stronger association.

The initiation of a closure is not necessarily ordered with respect to other initiations. The consequence is that closures may not depend on order. Concrete executors may, and often do, provide stronger initiation order guarantees.

There is no defined ordering of the execution or completion of closures added to the executor. The consequence is that closures should not wait on other closures executed by that executor. Mutual exclusion for critical sections is fine.

Changing and using the default executor is sequentially consistent.

Destruction of the concrete executor must happen before destruction of the environment upon which the concrete executor depends. In particular, this necessary constraint implies that thread-based concrete executors must be destroyed before destruction of the standard library. They may be destroyed after main.

IV. Changes from Google's internal version

The goal of this proposal is to standardize prior art, without any design innovation. As a consequence, most of the changes from Google's internal code are purely mechanical.

- As described in the Google C++ style guide, our internal naming style isn't the same as the standard's style. This proposal changes all names to be consistent with the standard. For example, `Executor::TryAdd()` in our internal code base has been changed to `executor::try_add()` in this proposal.
- The internal Google executor library predates `std::function`, so it uses an internal class to represent a closure with no arguments and a `void` return type. This proposal changes that to `std::function<void()>`.
- Internal functions that return new'ed objects return raw pointers. This proposal changes that to unique_ptr to match C++ style.

In addition to these mechanical changes, this proposal subsets the internal Google executor library. Functionality that has turned out to be error prone or rarely used, or that depends on features that `std::thread` doesn't provide, has been dropped. Some of the dropped features are discussed in the *Possible future directions* section.

The internal Google executor library predates the standard `chrono` utilities, so it just uses `int`. (A number of milliseconds.) This proposal uses durations and time points instead. This change is slightly less mechanical than one might like, however: `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.
3. Introduce two abstract base classes instead of one: `executor`, which doesn't need to know about clocks, and `scheduled_executor<Clock>`, which does. Presumably `scheduled_executor<>` would inherit from `executor`.
4. Pick a single clock and use its duration and time_point.

We chose the last of those options, largely because it was the smallest change from the existing design.

In addition to changing `add_after` to take `system_clock::duration` instead of `int`, we also added a second overload. In the internal Google executor library there is only a single version, which takes a relative time. We added the overload in this proposal because the failure to support absolute times in the executor library is generally believed to have been a mistake.

The internal version of `thread_pool` does not start threads in the constructor, instead providing an explicit initialization function that starts the threads. This is largely because Google C++ code typically

doesn’t use exceptions, and, in the absence of exceptions, starting threads in the constructor would offer fewer options for error checking. Since that reasoning doesn’t apply to the standard, this proposal does start threads in constructors.

Google's internal thread libraries allow users to name threads, and our executor implementations take corresponding parameters to provide prefixes for threads they create. Since `std::thread` does not allow users to name threads, this proposal omits the name parameters as well.

We also wish to explicitly call out a non-change. As mentioned above, Google C++ code doesn’t use exceptions. As a consequence, the program will terminate if any closure passed to an executor throws an exception. We believe that’s still the right choice for this proposal. First, it’s consistent with the way that `std::thread` behaves. Second, 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.

V. Proposed interface

Abstract executor

```
class executor {
 public:

  // Special member functions.

  virtual ~executor();

  // Closure-scheduling methods.

  // Schedule the specified closure for execution in this executor.
  // Depending on the subclass implementation, this may block in some
  // situations.
  virtual void add(function<void()> closure) = 0;

  // Like add(), except that if the attempt to add the closure would
  // cause the caller to block in add, try_add would instead do
  // nothing and return false.
  virtual bool try_add(function<void()> closure) = 0;

  // Schedule given closure for execution in this executor no sooner
 // than time abs_time.  This call never blocks, and may violate
 // bounds on the executor's queue size.
  virtual void add_at(chrono::system_clock::time_point abs_time,
                      function<void()> closure) = 0;

  // Schedule given closure for execution in this executor no sooner
 // than time rel_time from now.  This call never blocks, and may
 // violate bounds on the executor's queue size.
  virtual void add_after(chrono::system_clock::duration rel_time,
                         function<void()> closure) = 0;

  // Query and default operations.

  // Return an estimate of the number of waiting closures.
  virtual int num_pending_closures() const = 0;

  // Return a pointer to the default executor for this process.
  static executor* default_executor();
```

```
  // Change the default executor for this process to "executor".
  static void set_default_executor(executor* other_executor);
};

// Executor factory functions.

// Creates and returns a pointer to a trivial executor, of unspecified
// dynamic type, that immediately executes the closure given to it.
unique_ptr<executor> new_inline_executor();

// Returns a pointer to a trivial executor, of unspecified dynamic
// type, that immediately executes the closure given to it. Multiple
// invocations of singleton_inline_executor return a pointer to the same
// object.
executor* singleton_inline_executor();

// Creates and returns a pointer to an executor, of unspecified
// dynamic type, that immediately executes the closure to it. If two closures
// are added to the same synchronized inline executor, the return from
// one will happen before the other is called.
unique_ptr<executor> new_synchronized_inline_executor();
```

Concrete executors

The concrete `executors loop_executor` and `serial_executor` are single-threaded executors that run their closures on particular threads. The loop_executor takes control of a single host thread, while serial_executor schedules its closures on another executor. Both execute closures in FIFO order.

```
class loop_executor : public executor {
 public:

  // Special member functions.

  loop_executor();

  // Any closures that haven't been executed by a
  // closure-executing method when the destructor runs are never
  // executed.

  // The destructor and the closure-executing methods may
  // introduce data races if they are called concurrently.
  virtual ~loop_executor();

  // Closure-executing methods. Closure-executing methods may
  // introduce data races if they are called concurrently.

  // Runs closures on the current thread until make_loop_exit()
  // is called.
  void loop();

  // Selects a set of closures C and executes them on the
  // current thread, returning when either all of them have been
  // executed or make_loop_exit() is called.
  // If 'c' has already executed or run_queued_closures()
  // happens before add(c), then c∉C. Otherwise, if add(c)
  // happens before run_queued_closures(), then c∈C. Otherwise
  // it's unspecified whether c∈C.
  void run_queued_closures();

  // If a closure is queued, this method runs it and returns
  // true. Otherwise, this method returns false immediately.
```

```
  bool try_run_one_closure();

  // Loop control:
  // When make_loop_exit() is called, loop() and
  // run_queued_closures() will finish executing any closure
  // they are currently executing (often, the closure that
  // executed make_loop_exit()) and then return.
  void make_loop_exit();

  // [executor methods omitted]
};

// Runs added closures in FIFO order inside a series of closures
// added to underlying_executor. Earlier serial_executor
// closures happen before later closures.
// The number of underlying_executor->add() calls 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. (The
// serial_executor can batch underlying_executor->add() calls.)
class serial_executor : public executor {
public:

  // Special member functions.

  explicit serial_executor(executor* underlying_executor);

  // Finishes running any executing closure; then destroys all
  // remaining closures and returns.
  //
  // If a serial_executor is destroyed inside a closure running
  // on that serial_executor, the behavior is undefined. (It
  // deadlocks inside Google.)
  virtual ~serial_executor();

  // Query methods.

  executor* underlying_executor();

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

Thread pools vary primarily in how they create new threads.

- One simple thread pool creates a fixed number of threads when it's constructed and multiplexes closures on top of them. This thread pool risks deadlock if its closures wait on each other to finish.
- Another simple thread pool starts a new thread if no existing thread is available to run a new closure. This thread pool risks memory exhaustion if it's presented with a burst of work.
- A more advanced thread pool tries to run the minimum number of threads to keep the system's processors busy.

Google has pools of the first and third kinds. Class `thread_pool` is a simple thread pool with a fixed number of threads:

```
class thread_pool : public executor {
public:

  // Special member functions.

 // Creates an executor that runs closures on num_threads threads,
 // with up to queue_length closures waiting to be executed. If
 // queue_length is negative, the queue is unbounded. Throws
```

```
 // system_error if the threads can't be started.
 explicit thread_pool(int num_threads, int queue_length = -1);

 // Waits for closures (if any) to complete, then joins and
 // destroys the threads.
 ~thread_pool();

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

Google's more advanced thread pool is a two-level structure, with a "`thread_manager`" managing a global collection of threads, and a set of "`managed_queue`s" implementing the executor interface and feeding closures into the `thread_manager`. Each `managed_queue` can be configured with limits on the number of threads its closures can occupy at once and how many closures can queue up waiting for those threads.

The `thread_manager` class arranges its threads into a set of pools to reduce lock contention, and can be configured with a policy that decides when each of those pools creates or destroys a thread. By default, the `thread_manager` tries to keep enough threads running to fill `thread::hardware_concurrency()`, even if some threads block on IO.

```
class thread_manager {
 public:
  struct options;

  explicit thread_manager(options &&options);

  // The destructor blocks until all outstanding queues have been
  // deleted and the work associated with them has completed.
  ~thread_manager();

  // Return a pointer to a work queue serviced by this
  // thread_manager with limits given by queue_options.
  unique_ptr<managed_queue> new_queue(
      const managed_queue::options &queue_options);

  // Set the num_cpus argument to default_thread_manager_policy() used
  // to create any thread_manager whose policy is left defaulted,
  // including the default thread_manager.
  // Should be called before calls to default_manager(),
  // default_queue(), and the thread_manager constructor.  If not
  // called explicitly, uses thread::hardware_concurrency.
  static void set_default_num_cpus(unsigned (*num_cpus)());
};

// A managed_queue is an executor feeding a pool of threads managed by
// thread_manager.
class managed_queue : public executor {
 public:
  struct options;

  // The destructor returns without waiting for closures submitted
  // through this queue to finish running.
  // See wait_until_complete() below.
  virtual ~managed_queue();

  options queue_options() const;   // return the queue options

  // Wait until all work currently associated with this queue
  // (including work passed to add_after()) is complete.  If new work is added
  // during the call, it is unspecified whether the call waits for the
```

```
  // new work to complete.
  void wait_until_complete();

  // [executor interface omitted]

 private:
  managed_queue([unspecified parameters]);
};

// options to thread_manager::new_queue()
struct managed_queue::options {
  // Max concurrent threads to run closures from queue
  int thread_limit = INT_MAX;
  // Max closures to be on queue before Add() blocks.
  int queue_limit = INT_MAX;
  // Max interval for a closure to run.
  // If a closure runs longer, terminate() is called.
  seconds time_limit = seconds::max();
};

struct thread_manager::options {
  // n_pools controls the number of internal pools to use.  It will be
  // rounded up to a power of two.  Each pool has its own locking, so
  // having more pools reduces lock contention; the number of pools
  // has no effect on specified queueing semantics.
  // The default setting of 0 means "pick a reasonable value based on
  // the number of CPUs available".
  int n_pools = 0;

  // A thread_manager_policy object controls thread-creation policy.
  // See below for its interface.
  // A null policy selects unspecified but sensible defaults.
  unique_ptr<class thread_manager_policy> policy;
};

  // thread_manager is configurable through a policy object, which
// decides whether to create threads given the state of the thread_manager.
class thread_manager_policy {
 public:
  struct state;
  struct action;

  virtual ~thread_manager_policy();

  // Set to true to specify that Eval should be invoked only when
  // there is no idle thread (i.e. when the thread_manager is overloaded).
  void set_invoke_policy_when_no_idle_thread(bool doevent);
  bool invoke_policy_when_no_idle_thread() const;

  // eval(state, &result) is called whenever a decision must be made
  // about whether to start a new thread on one of the thread_manager's
  // internal pools. If invoke_policy_when_no_idle_thread() is true,
  // this will be called only when there is work to do but no idle
  // thread to do it. Otherwise, this will be called unconditionally
  // after result->delay milliseconds.  It should fill in the fields of
  // *result with an indication of whether to create a thread, and
  // the amount of time to delay before any attempt will be made to
  // create another thread (eval() will not be called again on a given
  // pool until that delay has expired, though it may be called for
  // other pools within the same thread_manager).  It may be called with
  // an internal lock held; therefore, it should return quickly, it may
```

```
  // not block, and it may not call back into thread_manager or
  // managed_queue.  The state struct contains information about the
  // state of a single pool, which is not necessarily the entire
  // thread_manager.
  virtual void eval(const state &current_state, action *result) = 0;
};

// Return an instance of the default thread_manager policy.  This policy
// tries to create threads when they are needed to use available CPUs,
// and when doing so will increase the rate at which work will be done.
// The default policy needs to know the number of CPUs it should try to
// consume. The function thread::hardware_concurrency() is an acceptable
// argument.
// The returned pointer should be passed via the options argument to the
// thread_manager constructor.
unique_ptr<thread_manager_policy> default_thread_manager_policy(
  unsigned (*num_cpus)());

// Represents the state of one of a thread_manager's internal pools.
struct thread_manager_policy::state {
  system_clock::time_point time;  // time of this state
  // total closures run in pool since beginning of time
  int64 closures_run = 0;
  int queue_length = 0;    // current pool queue length

  // Count of pool's threads
  // threads = active + kill_pending + create_pending + blocked;
  // note that idle are counted as blocked
  int threads = 0;
  int active = 0;          // count of pool's active threads
  int kill_pending = 0;    // number of threads to be killed
  int create_pending = 0;  // number of threads to be created
  int idle = 0;            // count of idle threads
  // pool's threads currently blocked; i.e. waiting for IO or another
  // thread to do something (approximate)
  int blocked = 0;
  int threads_since_last_exit = 0;  // threads created for pool since
                                    // last thread was destroyed.
};

// Result from thread_manager_policy::eval, with instructions for the
// pool to create or destroy threads.
struct thread_manager_policy::action {
  // True if the pool should create a single new thread
  // must be set to false if desired_threads > 0
  bool create;
  // desired number of active threads in current pool
  // -1 specifies that any change is controlled by 'create'.
  int desired_threads;
  // delay before next creation attempt
  milliseconds delay;
};
```

VI. Possible future directions

There are many other useful thread pool classes, in addition to those in this proposal. Some of them are in

use within Google. In particular, we may submit a future proposal for explicitly-resizable thread pools.

The Google executor library provides many options for starting threads, including thread names, priorities, and user-configurable stack sizes. Those options have proven useful. We are omitting them from this proposal because 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 executor library does provide mechanisms for closure cancellation. 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.

Sometimes it's useful for code to be able to ask whether it's running in an executor, and, if so, which one. Google's executor library provides a `current_executor()` function that answers that question. We omitted it from this proposal because it requires work in all executor classes, not just in the `executor` base class — including user-written executor classes, which aren't uncommon. It's not clear that the benefit of this feature justifies that burden.

This proposal chooses the simplest possible exception policy: an uncaught exception within a closure running in an executor results in calling `std::terminate()`. One could imagine more sophisticated exception policies, such as catching exceptions and then throwing them from the thread that created the executor. We aren't proposing such policies largely because we are trying to avoid invention, and because they may be handled by `std::future`, but they could be considered for the future.

Executors can, and arguably should, be extended with an async-like facility, so as to avoid the packaged_task boilerplate. That is, provide the semantics of `std::async` but with scheduling associated with the executor. Implementation 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. An async-like facility would also permit transferring exceptions to another thread. Other options for `executor`/`async` integration also exist.

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 `try_add()` can ever return false, 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 want to write functions that accept only executors satisfying the properties they rely on, but Google's internal library only provides the single executor base class. A future paper may invent an appropriate mechanism to constrain executor parameters.

Google's internal executor library neglects 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.