Executors for C++

A Solution to the `async` Problem
A Base Concurrency Building Block

Detlef Vollmann

vollmann engineering gmbh, Luzern, Switzerland

Meeting C++, Berlin, December 2014

Overview

Introduction

Original Proposal

New Proposals

Outlook

Motivation: `async`

```
std::async([](){ std::cout << "Hello "; });
std::async([](){ std::cout << "World!\n"; });
```

- No concurrency
- No real control over execution agent
  - `launch::async` and `launch::deferred` insufficient

Motivation: Pipelines

```
pipeline::plan restaurant(
  orders
  | pipeline::parallel(chef, 3)
  | pipeline::parallel(waiter, 4)
  | end);

thread_pool pool;

pipeline::execution work(restaurant.run(&pool));
```

- Executors as building blocks for higher level abstractions

Layers

Proposal History 1

- N3378: Initial Google "Work Executors" proposal
  - Destructor sync, `try_add()`, default executor
  - Timing functions in base executor
  - `singleton_inline_executor()`
  - `thread_manager`
- Unnumbered Microsoft paper presented in Bellevue, 2012
  - `http://wiki.edg.com/twiki/pub/PublicRepository/SG1Docs2012/SG1_A_Std_Rep_of_Async_Ops.pdf`
  - Just `task_base` and `scheduler` as interface

Proposal History 2

- N3562: Joint Google/Microsoft "Executors and Schedulers" proposal
  - Removed `try_add()`, `thread_manager`
  - Separated out `scheduled_executor`
- N3731: No singleton anymore
- N3785: Current version
  - No destructor sync
  - No default executor
- Accepted into Concurrency TS
- Only about executors, (nearly) nothing about scheduling
- Status as of May 2014

Implementations

- Original Proposal
  - Boost `https://github.com/boostorg/thread`
  - Readium SDK `https://github.com/readium/readium-sdk`
  - Inhouse implementations
- Chris Kohlhoff's Proposal
  - `http://github.com/chriskohlhoff/executors`
- Chris Mysen's Proposal
  - `https://github.com/arturl/executors`
  - `https://github.com/ccmysen/executors_r4`

Original Executor Interface

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

- Pure interface so copying is generally allowed here
- For most concrete executors, copying should probably forbidden

Default Executor

```
shared_ptr<executor> default_executor();
void set_default_executor(
         shared_ptr<executor> executor);
```

- Not part of N3785
- But very likely to come
- Can serve as executor for a new launch policy

Abstract Base Class

- No template concept
- Not part of the type
  - Not really important for functions
  - Important for structures
- Can cross binary interfaces

Threads

Work units submitted to an executor may be executed in one or more separate threads.

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.*

- Pretty fuzzy
  - Will a closure stay on the same thread?
  - May a thread on which closure A runs be used for closure B before closure A has finished?
- No cancellation

add(function<void()> closure)

- Again: not a template
  - Type erasure is required anyway for storing closures
  - Executors are a low-level interface, type erasure is delegated to upper layers
  - Virtual functions cannot be templates
- No return type
  - No `std::future<>`
  - `future` is a template
  - There may be more than one future type in the future
  - Again: return types are delegated to upper layers

add()

- `add()` never blocks on other closures
  - Executors are unbounded
    - Throwing an exception if full is ok
  - Don't start any code in the closure at adding time
    - The real challenge is the `std::function<void()>`...
    - See `inline_executor`
- No exceptions from closures
  - `std::terminate()`
  - Again: exceptions are delegated to upper layers

Progress Guarantees

- Fundamental guarantees for *execution agents*
- *Weakly parallel*
  - No guaranteed concurrency, no classic mutex synchronization
- *Parallel*
  - Only concurrent after start
  - E.g. bounded thread pool
- *Concurrent*
  - All tasks make progress concurrent to each other
- The abstract interface doesn't make any specific guarantees
  - ...but concrete executors do

Destructor

- No new closure initiations after destruction
- No guarantee that started closures have completed
  - ... in the general case

uninitiated_task_count()

- (Possibly rough) number of tasks still to be started
- No interface for asking how many tasks are currently running

scheduled_executor

- Not part of any current proposal

```
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;
};
```


thread_pool

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

- No default constructor so far
- Destructor waits for all(?) closures to complete
- Strictly bounded or unbounded? I.e. *concurrent* or *parallel*?

serial_executor

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

- Runs one closure after the other (no two concurrently)
- Again no default constructor so far
- Must wrap closures before adding to underlying executor
- Destructor waits for current closure(s) to complete
- This prvides the *parallel* guarantee.

loop_executor

```
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();
};
```

- Like `serial_executor`, but runs on thread that calls `loop()`
- Runs closures in FIFO order
- No closure must be running on destruction
- `make_loop_exit()` is some kind of cancellation from inside a closure

inline_executor

- Just a function call
- This was dropped due to blocking `add()`

```
class inline_executor : public executor {
public:
  inline_executor() {}
  void add(function<void()> closure) {
    closure();
  }
  size_t uninitiated_task_count() const {
    return 0;
  }
};
```


thread_executor

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

- Like `launch::async`
- Destructor waits for all closures to complete.

async

```
async(launch::executor,
  [](){ std::cout << "Hello!\n"; });

thread_pool myPool;
async(myPool,
  [](){ std::cout << "Hello!\n"; });
```

- N3721 proposes the second version

.then

- Proposed continuation `.then` also allows for an executor:

```
auto f = std::async([](){
           std::cout << "Hello "; });
f.then(myPool,
       [](){ std::cout << "World\n"; });
```


Asynchronous Operations

- Chris Kohlhoff proposed in N3896 a framework for callback based asynchronous operation continuations. This is also a candidate for executors:
- For the original asynchronous operation

```
socket.async_receive(myPool, buffer,
                     handleReceive);
```

- For the continuation

```
socket.async_receive(myPool, buffer,
                     otherPool,
                     handleReceive);
```


Parallel Algorithms

- While the parallel algorithms (N3960) mainly target vector parallelism, many algorithms can be usefully run on executors as well.
- `std::sort(myPool, vec.begin(), vec.end());`

Generic Task Managers

- Intel's Threading Building Blocks (TBB) provide a flow graph facility that allows to model arbitrary task dependencies.
- These tasks must run on some ... executor.

Realtime Priority

- Different executors for different realtime priorities

```
class RTScheduler : public executor {
public:
  RTScheduler(int priority);

  // wait on all closures to finish
  ~RTScheduler();

  // start a new thread
  void add(function<void()> closure);
};
```

- Hmmmm...

Cyclic Realtime Tasks

```
class CyclicRTScheduler : public executor {
public:
  // start myself a single thread
  CyclicRTScheduler(int priority,
                    const chrono::steady_clock
                          ::duration& cycleTime);

  // wait on all closures to finish
  ~CyclicRTScheduler();

  void add(function<void()> closure);
};
```

- Re-runs all closures on a specified frequency

More Realtime?

- What about different closures with different cycle periods?
- Another `add()` function doesn't help.
- We don't have a way to identify a closure to the executor
  - The closure could call a member function of the executor (when?)
  - This doesn't work for multi-threaded executors

Layers

Data Concentrator

```
RTExecutor rtExec0(0);
RTExecutor rtExec80(80);

ConcentratorT dataConcentrator{
    wrap(rtExec80, ReadDev(1, in1))
    , wrap(rtExec0, ReadDev(2, in2))
    , wrap(rtExec0, StoreData(out))
};
dataConcentrator.run();
```

- Concentrator like pipeline
  - Two producer, one consumer
  - One consumer has higher priority

Overview

Proposal History 3

- Chris Kohlhoff' papers
  - Sep 2013, pre-Chicago N3747 "A Universal Model for Asynchronous Operations"
  - Jan 2014, pre-Issaquah N3896 "Library Foundations for Asynchronous Operations"
  - May 2014, pre-Rapperswil N4046 "Executors and Asynchronous Operations"
- Presented in Rapperswil, tentatively accepted as new base:
  - remove N3785 from TS: SF-F-N-A-SA 6-7-5-2-0
  - More work on N4046 for TS: 10-8-0-0-0
  - Apply N4046 to TS without significant changes: 4-2-3-5-2
- Latest version N4242

Proposal History 4

- Chris Mysen
  - Sep 2014, pre-Redmond unnumbered paper "Modified Executors Proposal (continuation of N3785)"
  - Oct 2014, pre-UC N4143 "Executors and schedulers, revision 4"
- Presented in Redmond
- Start with Chris Mysen's proposal for Concurrency TS: SF-F-N-A-SA 9-5-4-0-2
- Latest version N4143
- Big discussion in Urbana-Champaign, still no consensus

Executors and Async Ops

```
class executor {
public:
  template<class Func, class Alloc>
  void dispatch(Func&& f, const Alloc& a);

  void post(...);
  void defer(...);
};

template<class Exec, class CompletionToken>
auto dispatch(const Exec&, CompletionToken&&);

auto post(...);
auto defer(...);
```


Executors and Async Ops

- executor and execution_context
  - executor is a light-weight handle
  - execution_context actually holds the threads and tasks
  - execution_context can be used to wait on everything to shut down.
- Proposed concrete executors:
  - system_executor (like thread_executor)
  - strand (like serial_executor
  - thread_pool
  - loop_executor

Executors and Async Ops

```
template<class Executor, class T>
executor_wrapper<decay_t<T>, Executor>
  wrap(const Executor& ex, T&& t);

template<class T>
associated_executor_t<T>
  get_associated_executor(const T& t);
```

- Mechanism to tie an executor to a task.

Realtime Executor and Concentrator Example

Show source

Executors and Schedulers, R4

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

- Proposed concrete executors:
  - thread_per_task_executor
  - thread_pool_executor
  - serial_executor
  - loop_executor
  - system_executor (like default_executor)
- executor_ref
  - executor_ref is a light-weight handle
  - executor is non-copyable and holds the threads and tasks

task_wrapper

```
template <typename Ex, typename Foo>
class task_wrapper {
 public:
  task_wrapper(Ex& e, Foo&& f);

  void operator()() {
   return [] {
    get_executor().spawn(contained_function());
  }}

  Exec& get_executor();
  Func&& contained_function();
};
template <typename Ex, typename Foo>
task_wrapper<Ex, Foo>&& set_executor(Ex&, Foo&&);
```


`task_wrapper` Deadlock

```
pipeline::plan sample(
  source
  | stage1
  | set_executor(thread_per_task_executor(),
                 stage2)
  | end);

serial_executor ex;

pipeline::execution work(sample.run(&run));
```

- Deadlock, if `operator()` is used and stage1 waits for stage2.

Overview

Wrap Up

- Executors are an important building block for concurrency
- They should be low level
- The final interface is still very controversial
- Missing experience

Executors Dilemma

- Executors are a solution to the `async` desaster
  - Very urgent
- Executors are base for higher-level abstractions
  - Interface must be done right

Questions

- ??????????????????????????????????

