codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Designing a Unified Interface for Execution

Gordon Brown - Senior Software Engineer, SYCL & C++
Michael Wong - VP of Research & Development

CppCon2017 – September 2017

Acknowledgement Disclaimer

Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.
I even lifted this acknowledgement and disclaimer from some of them.
But I claim all credit for errors, and stupid mistakes. **These are mine, all mine!**

Legal Disclaimer

This work represents the view of the author and does not necessarily represent the view of Codeplay.

Other company, product, and service names may be trademarks or service marks of others.

Codeplay - Connecting AI to Silicon

Products

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR™, HSA™ and Vulkan™

Company

High-performance software solutions for custom heterogeneous systems

Enabling the toughest processor systems with tools and middleware based on open standards

Established 2002 in Scotland

~70 employees

Addressable Markets

Automotive (ISO 26262)
IoT, Smartphones & Tablets
High Performance Compute (HPC)
Medical & Industrial

Technologies: Vision Processing
Machine Learning
Artificial Intelligence
Big Data Compute

Customers

BROADCOM
RENESAS
Imagination
QUALCOMM
Movidius

Partners
intel
AMD

About me...

- Background in C++ programming models for heterogeneous systems
- Developer with Codeplay Software for 5 years
- Worked on ComputeCpp (SYCL) since it’s inception
- Contributor to the Khronos SYCL standard for 5 years
- Contributor to C++ executors and heterogeneity or 1 year

Disclaimer

The proposal describe here is a work in progress

This may not reflect the final proposal

Disclaimer

My background is in heterogeneous computing

I am not an expert on all use cases for executors

Acknowledgements

Jared Hoberock, Chris Kohlhoff, Chris Mysen, Michael Garland, Michael Wong, Carter Edwards, Hartmut Kaiser, Hans Boehm, Torvald Riegel, Lee Howes, David Hollman, Bryce Lelbach, Gor Nishanov, Thomas Heller, Geoffrey Romer and more

Agenda

Brief history

Use cases

Design

Examples

Future work

What are executors?

invoke async parallel algorithms future::then post

defer define_task_block dispatch asynchronous operations strand<>

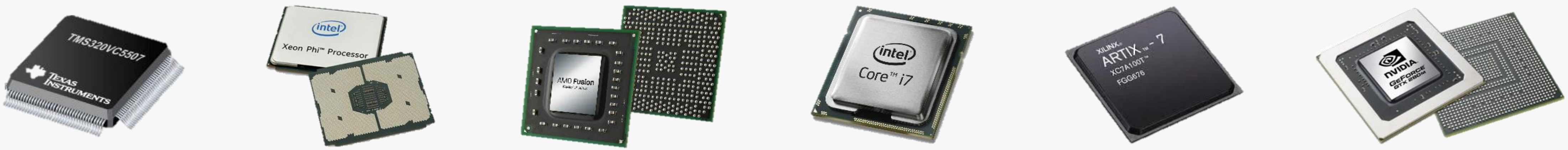
Unified interface for execution

SYCL / OpenCL / CUDA / HCC

OpenMP / MPI

C++ Thread Pool

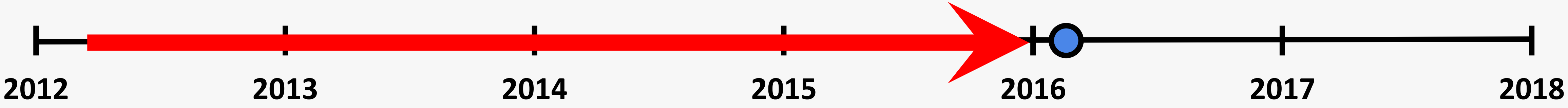
Boost.Asio / Networking TS



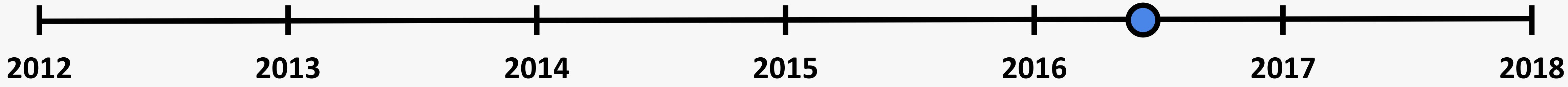
Brief history

- First executor proposal published in 2012

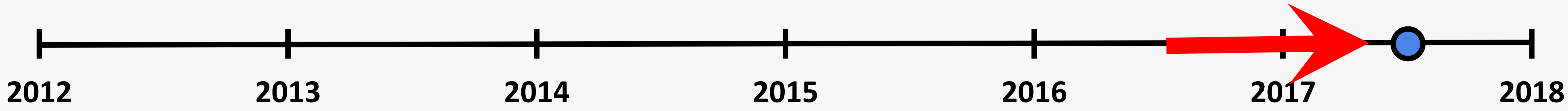
- Between 2012 and 2016 many more papers were published
- The work centred on four main proposals:
  - N4414: Executors and schedulers, revision 5
  - P0058r1: An Interface for Abstracting Execution
  - P0113r0: Executors and Asynchronous Operations, Revision 2
  - P0285r0: Using customization points to unify executors



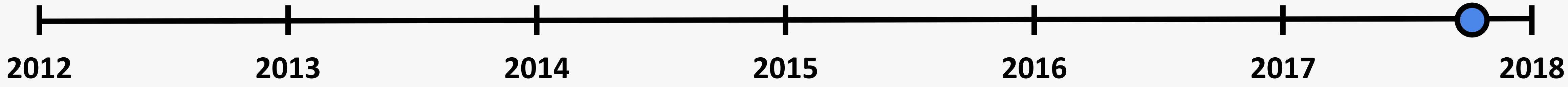
- At the Oulu 2016 meeting the executors sub group was formed
- The focus of the sub group was to bring together the various use cases of executors and form a unified proposal



- Since then the group has published multiple papers proposing a unified executors proposals:
  - P0443r2: A Unified Executors Proposal for C++
  - P0761r0: Executors Design Document

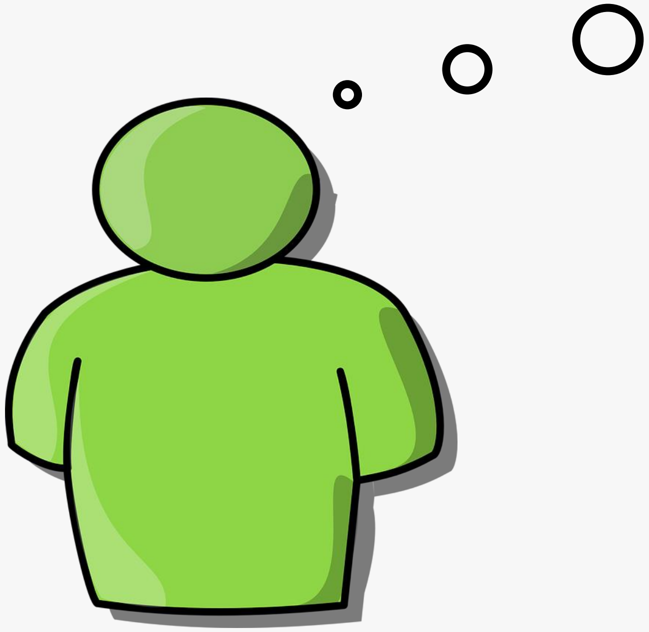


- With the feedback from SG1 the proposal is close to forming a technical specification for executors



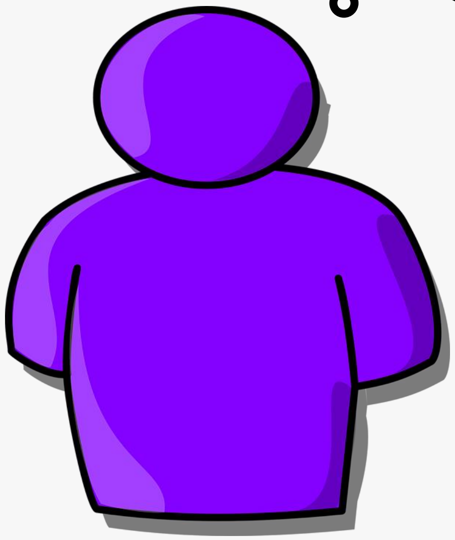
What is execution?

Multi-threaded execution



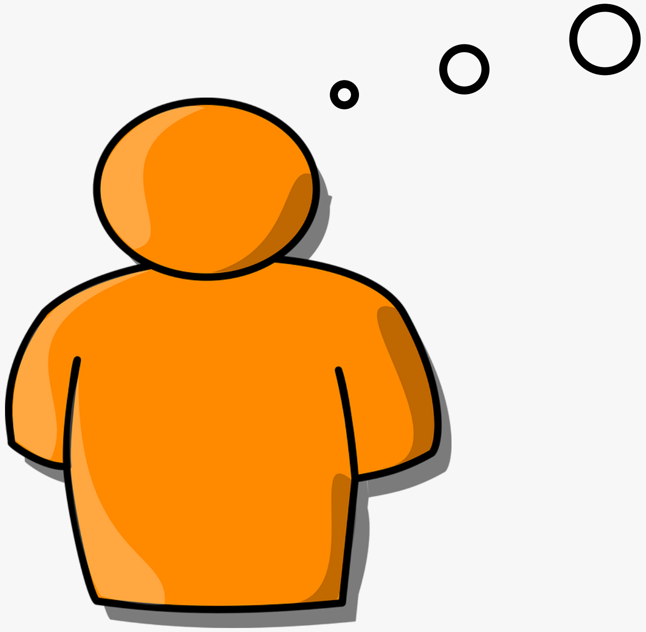
- Thread pools (fixed sized, dynamic)
- std::async()
- Launch policies
- Work that can execution as if on a std::thread

Network execution



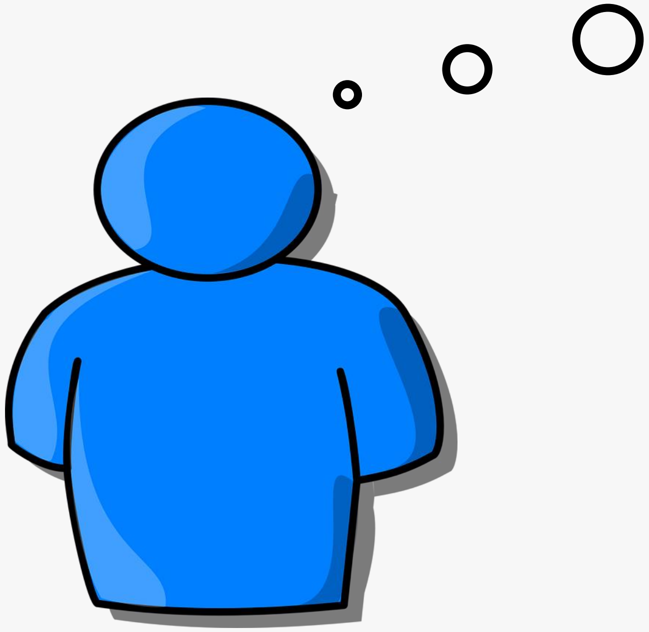
- Network devices
- Boost.asio / networking TS
- One way communication
- Work tracking

Parallel execution



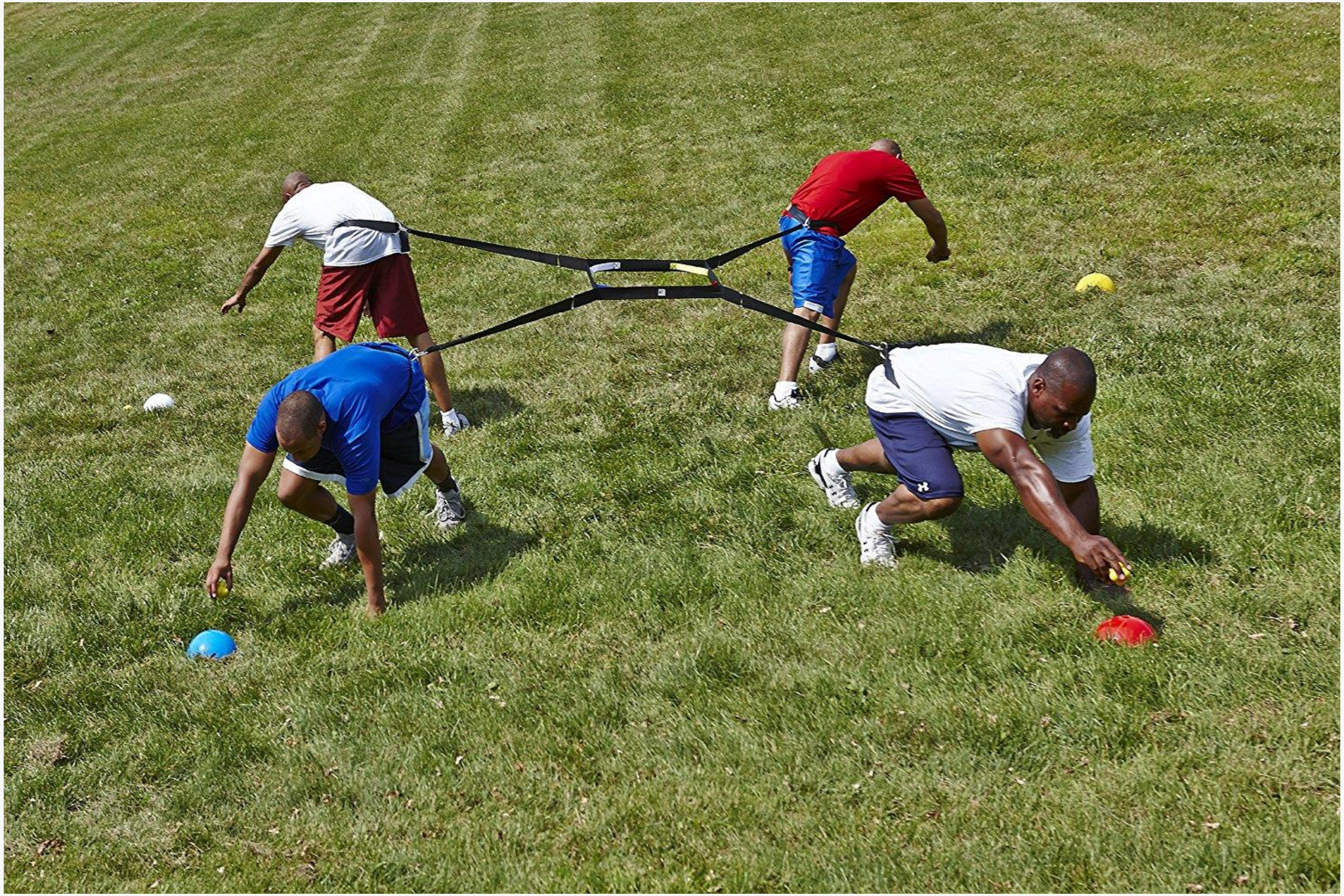
- Parallel / vectorized algorithms
- Execution policies
- Bulk execution of threads
- Channel for returning a result

Heterogeneous & distributed execution

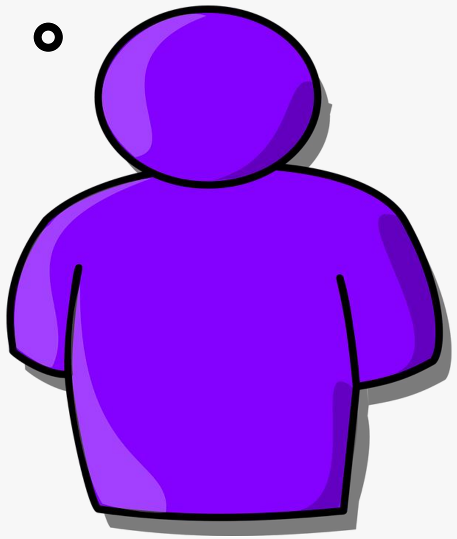


- Managed execution contexts
- Discrete non-CPU architectures
- Task graphs
- Bulk execution of threads

How do we make sure everyone gets what they want?

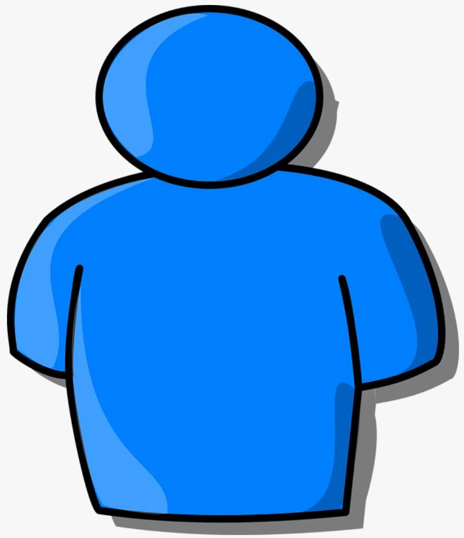


What do these all have in Common?



Work execution

- Execution of a callable object

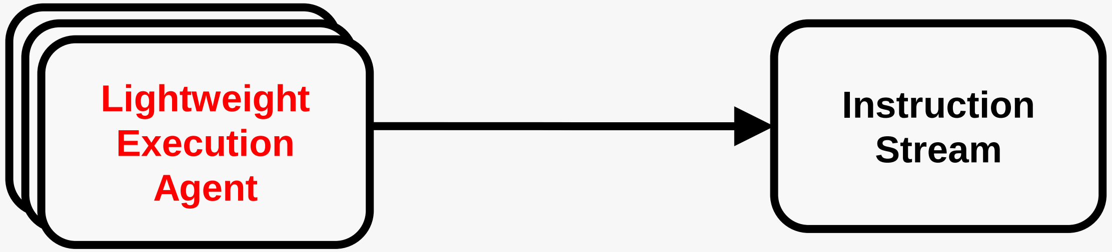


Establish a common topology of execution

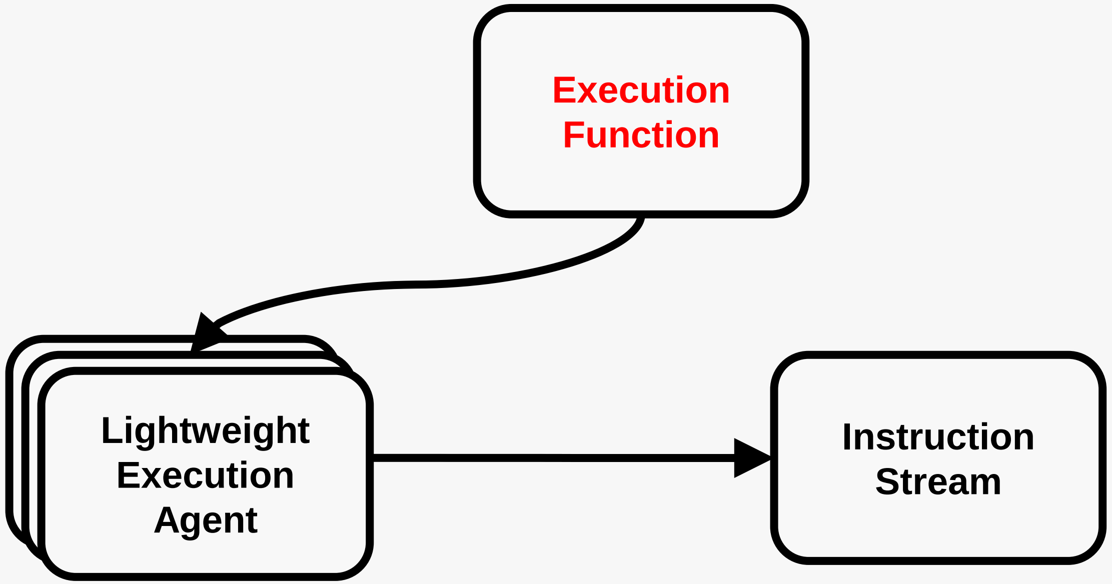
- An instruction stream is a callable object that is to be executed

Instruction Stream

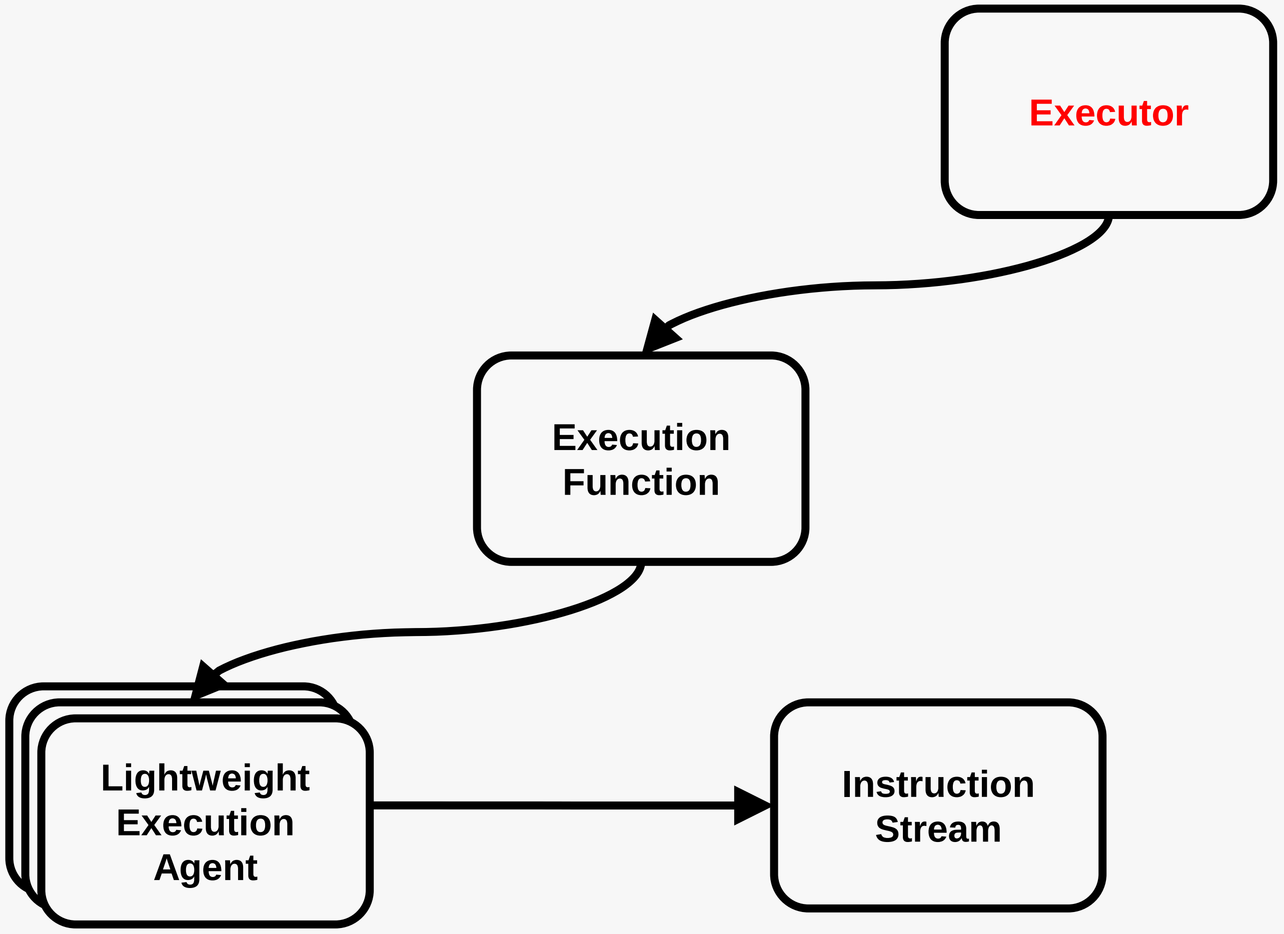
- A light-weight execution agent is a single thread of execution executing the instruction stream



- An execution function is a function which executes an instruction stream on one or more light-weight execution agents with a particular set of properties

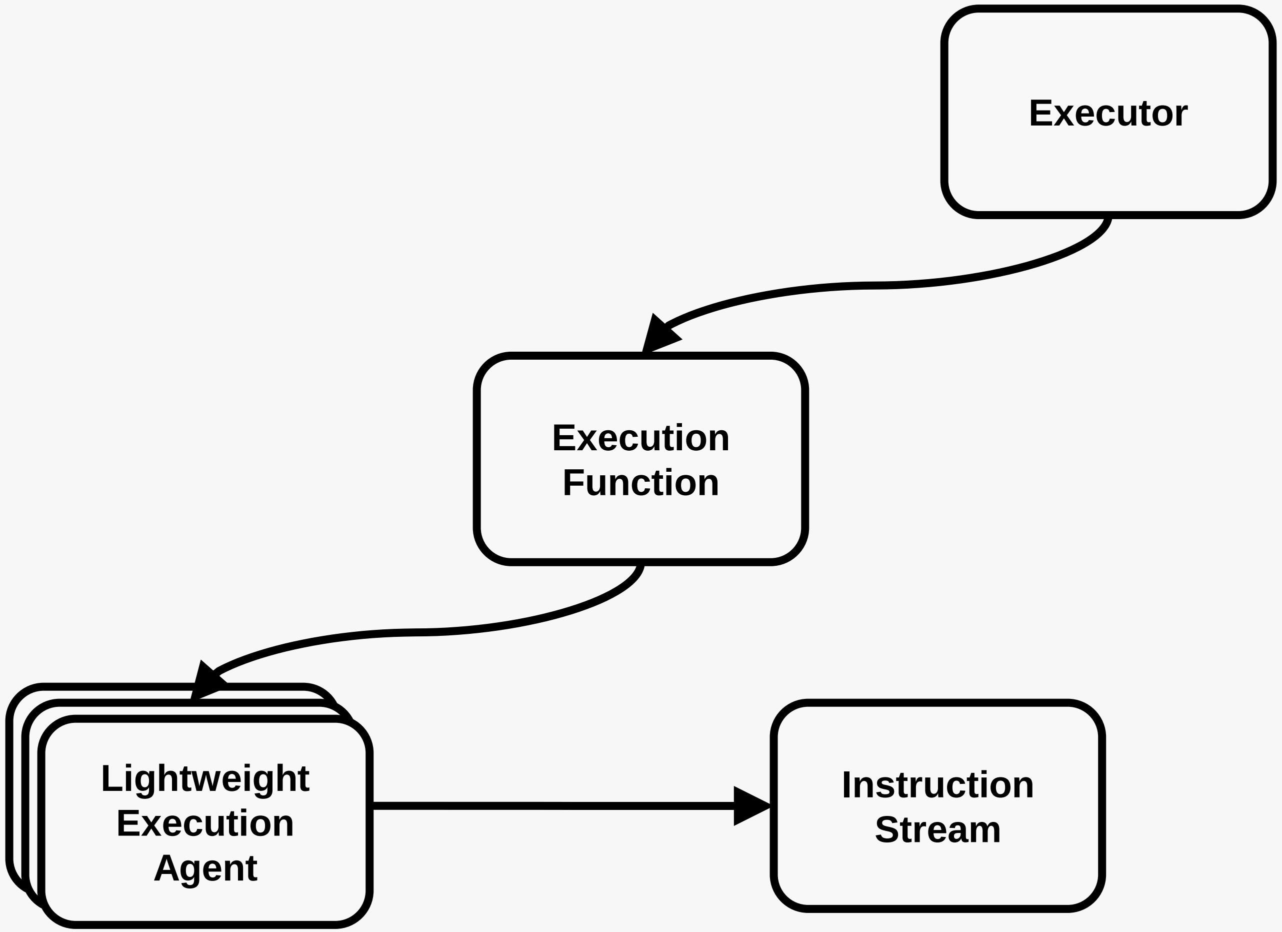


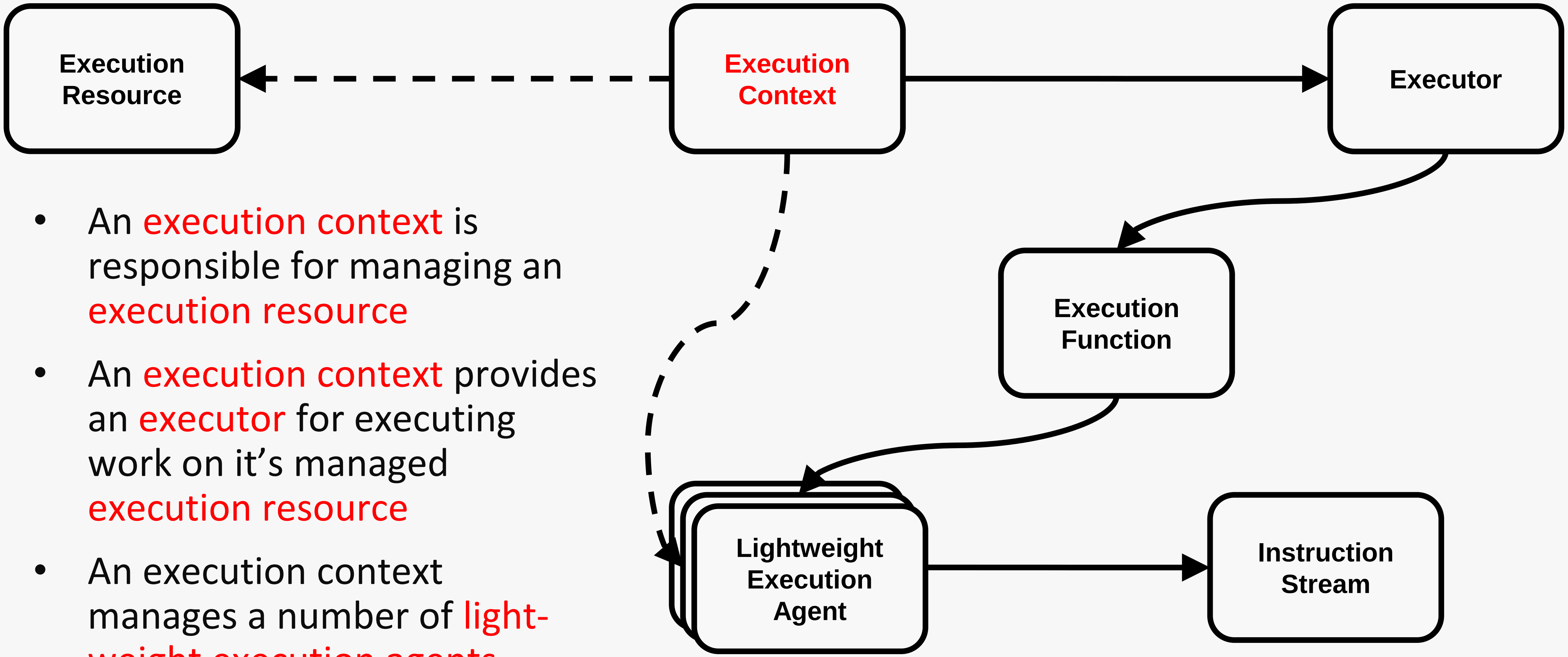
- An executor is an interface that describes where, when and how to execute work
- An executor can spawn one or more light-weight execution agents each executing the same instruction stream via execution functions



Execution Resource

- An execution resource is the hardware abstraction which is executing the work
- Examples of an execution resource are a CPU thread pool, GPU context, network device





- An execution context is responsible for managing an execution resource
- An execution context provides an executor for executing work on it's managed execution resource
- An execution context manages a number of light-weight execution agents

```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

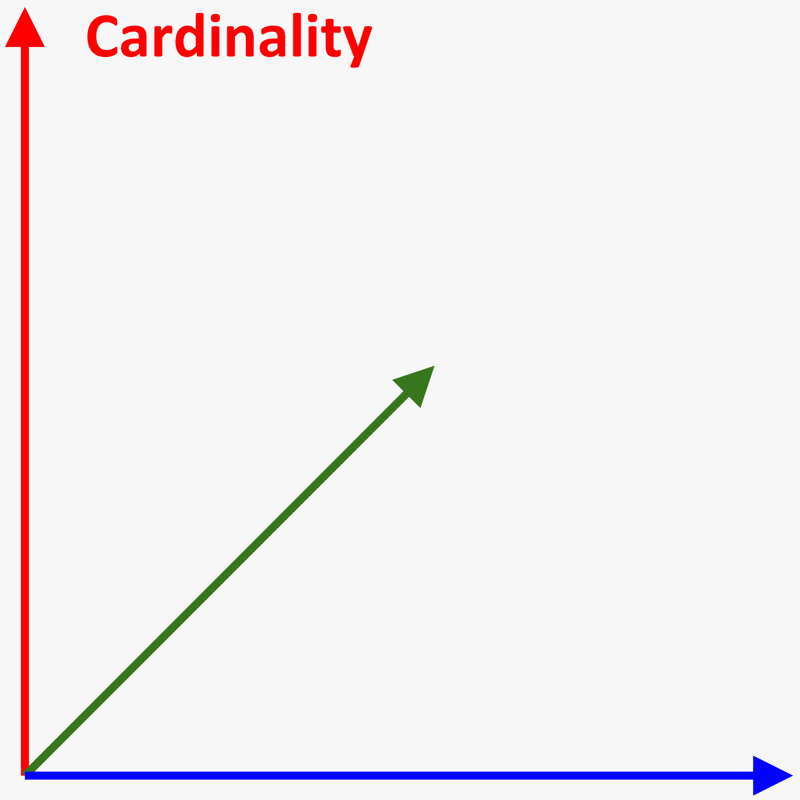
```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

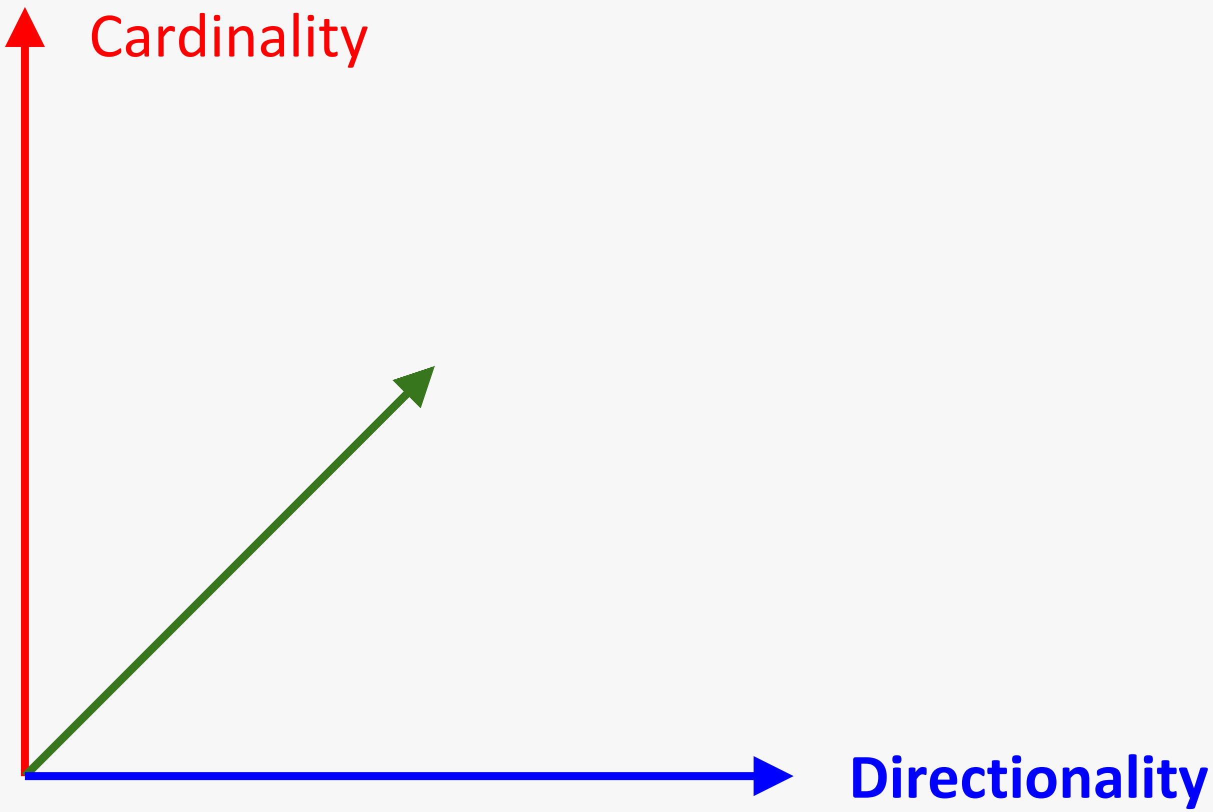
```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

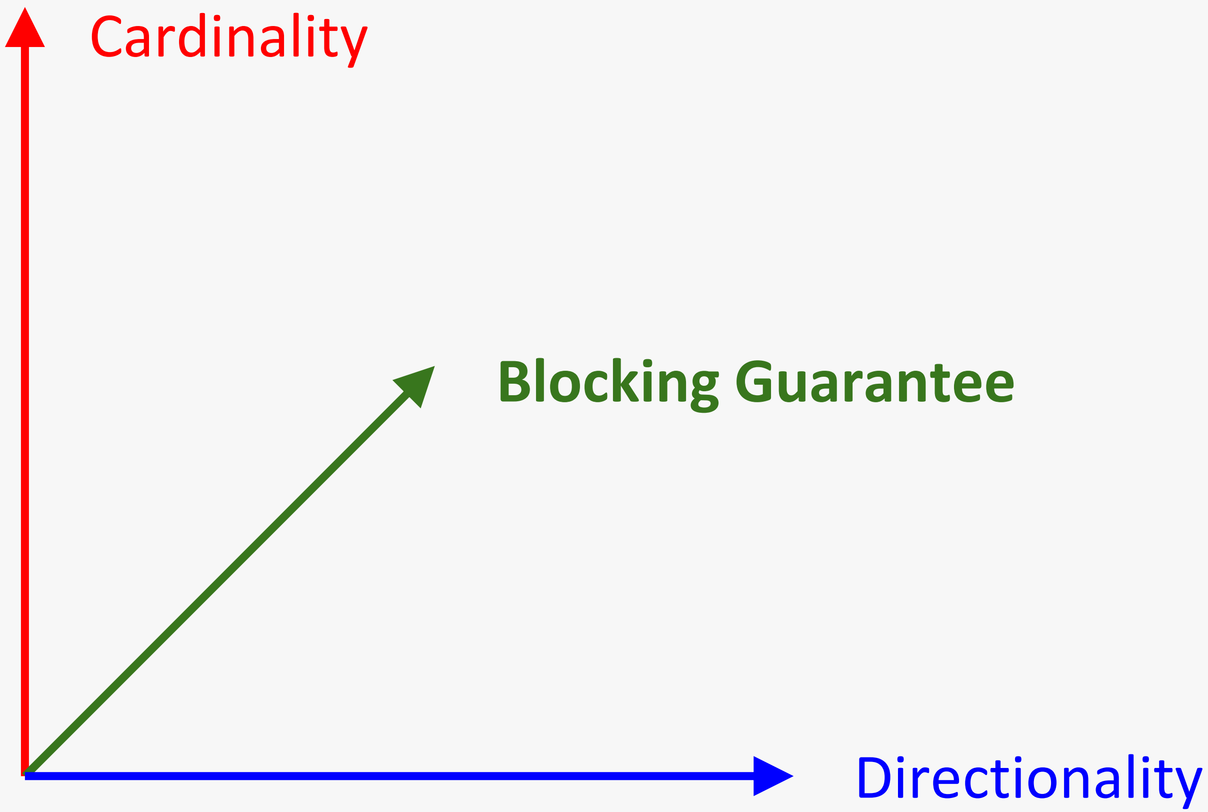
Establish the bifurcations of execution



- An executor's cardinality reflects whether an execution launches a single thread of execution or multiple threads of execution
  - Single cardinality
  - Bulk cardinality



- An executor's directionality reflects whether an execution does or does not provides a channel by which to synchronise or return a result or exception
  - One-way directionality
  - Two-way directionality



- An executor's blocking guarantee reflects whether an execution blocks or does not block the caller thread until execution is complete
  - Possibly-blocking guarantee
  - Always-blocking guarantee
  - Never-blocking guarantee

Establish the execution functions

	One-way	Two-way
Single	`execute()`	`twoway_execute()`
Bulk	`bulk_execute()`	`bulk_twoway_execute()`

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  bulk_executor exec;
  exec.bulk_execute([&](size_t index,
    auto &s){
    func(index, s);
  }, shape, sharedFactory);
}
```

Bulk One-way

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  bulk_executor exec;
  exec.bulk_execute([&](size_t index,
    auto &s){
    func(index, s);
  }, shape, sharedFactory);
}
```

Bulk One-way

```
{
  bulk_twoway_executor exec;
  auto fut = exec.bulk_twoway_execute(
    [&](size_t index, auto &r, auto &s){
    func(index, r, s);
  }, shape, resultFactory, sharedFactory);
}
```

Bulk Two-way

Establish the properties of execution

Property	Description
single	Executes an instruction stream exactly once
bulk	Executes an instruction stream a number of times
oneway	Does not return a future
twoway	Returns a future for the return value or exception and synchronisation
possibly_blocking	May or may not block the caller thread on execution completion
always_blocking	Always blocks the caller thread on execution completion
never_blocking	Never blocks the caller thread on execution completion

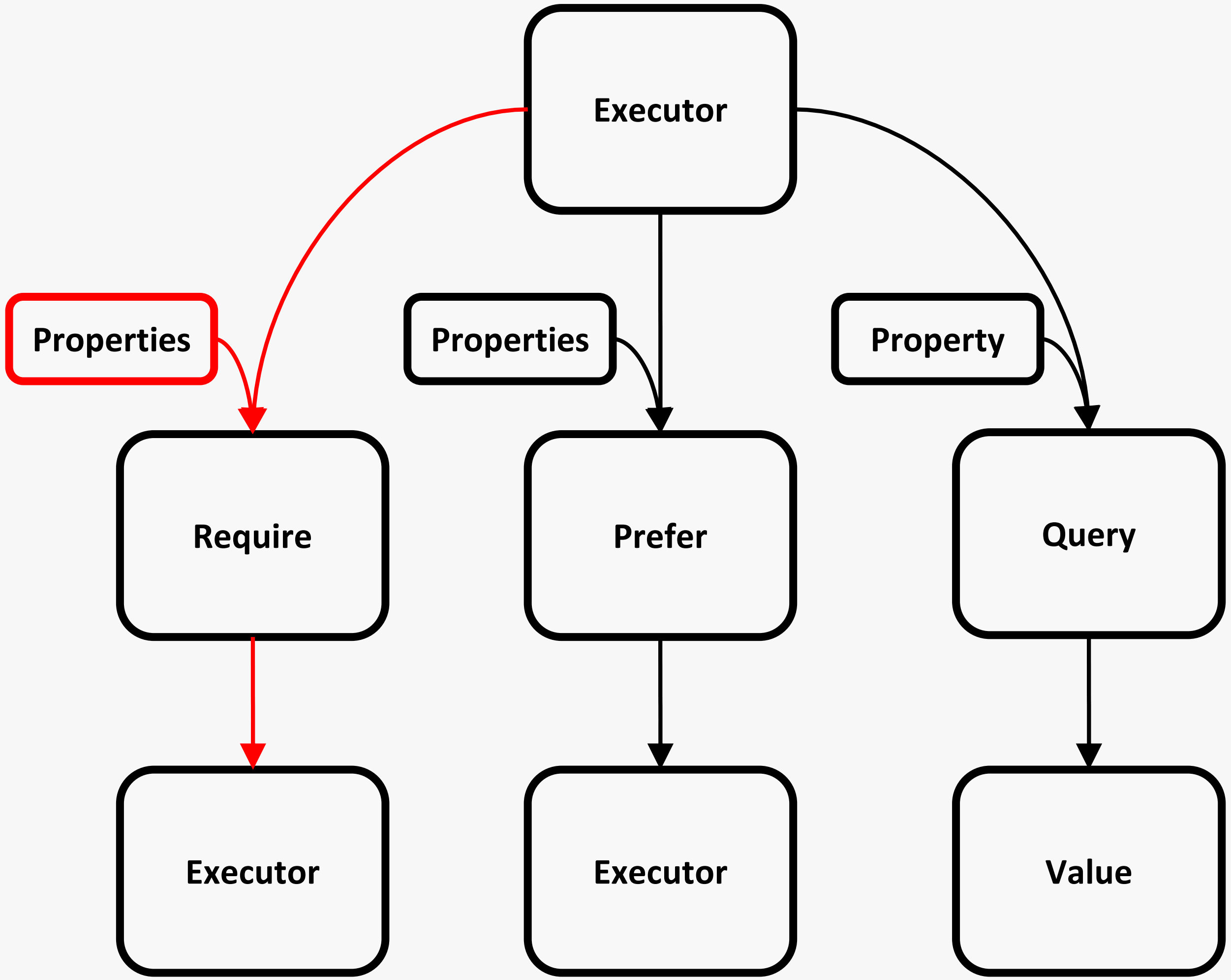
Property	Description
single	Executes an instruction stream exactly once
bulk	Executes an instruction stream a number of times
oneway	Does not return a future
twoway	Returns a future for the return value or exception and synchronisation
possibly_blocking	May or may not block the caller thread on execution completion
always_blocking	Always blocks the caller thread on execution completion
never_blocking	Never blocks the caller thread on execution completion

Properties which modify the interface

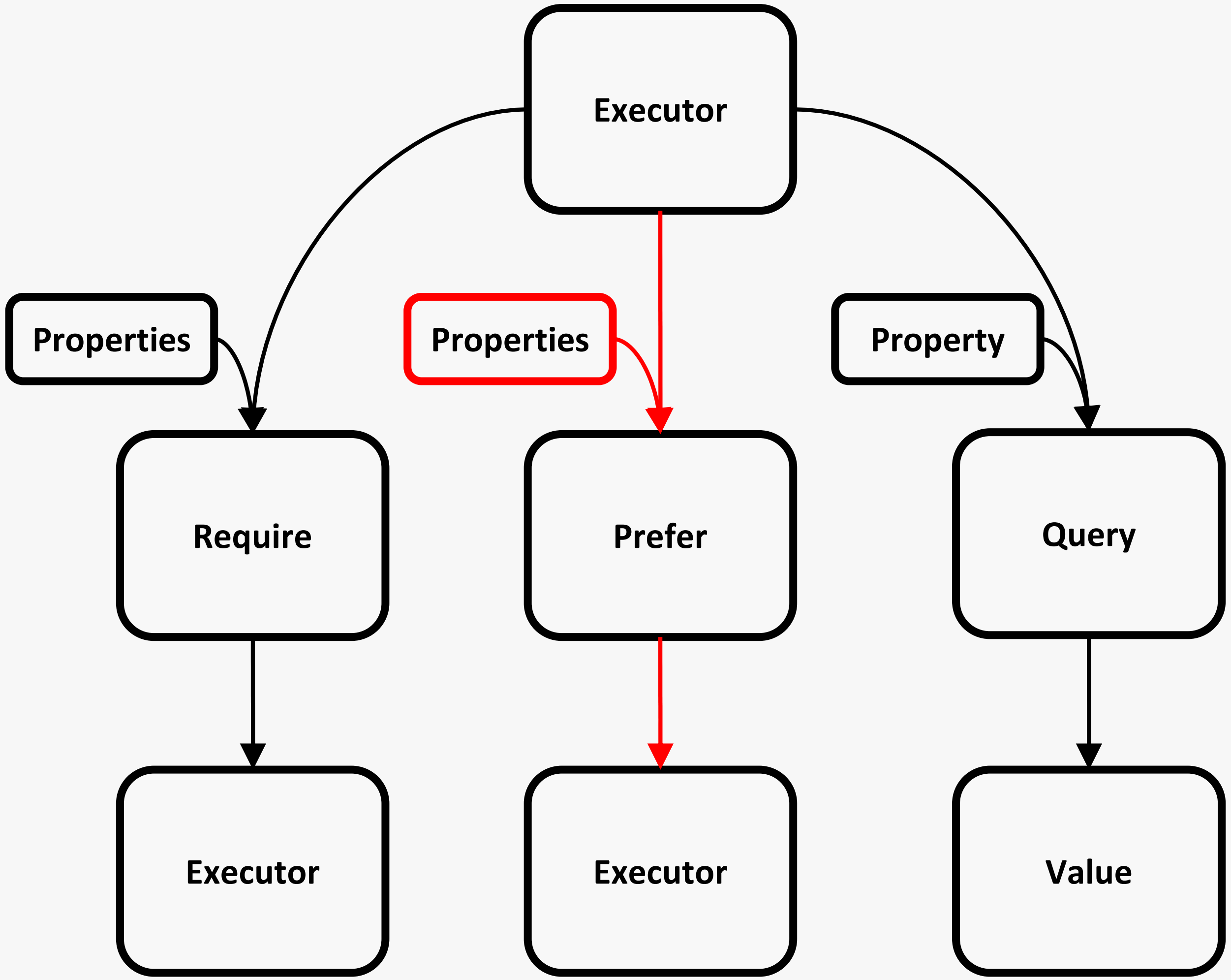
Properties	Description
Thread mapping semantics	Specifies the way in which the instruction stream is mapped to threads of execution
Bulk execution guarantees	Specifies the guarantees between threads of execution within a bulk execution
Caller forward progress guarantees	Specifies the forward progress guarantees between the threads of execution and the caller thread
Continuation	Specifies whether the instruction stream should be executed as a continuation
Future work submission	Specifies whether or not the execution context should expect future work to be submitted
Allocator	Specifies the allocator to use when allocating memory for the instruction stream

Establish how executors could be customised

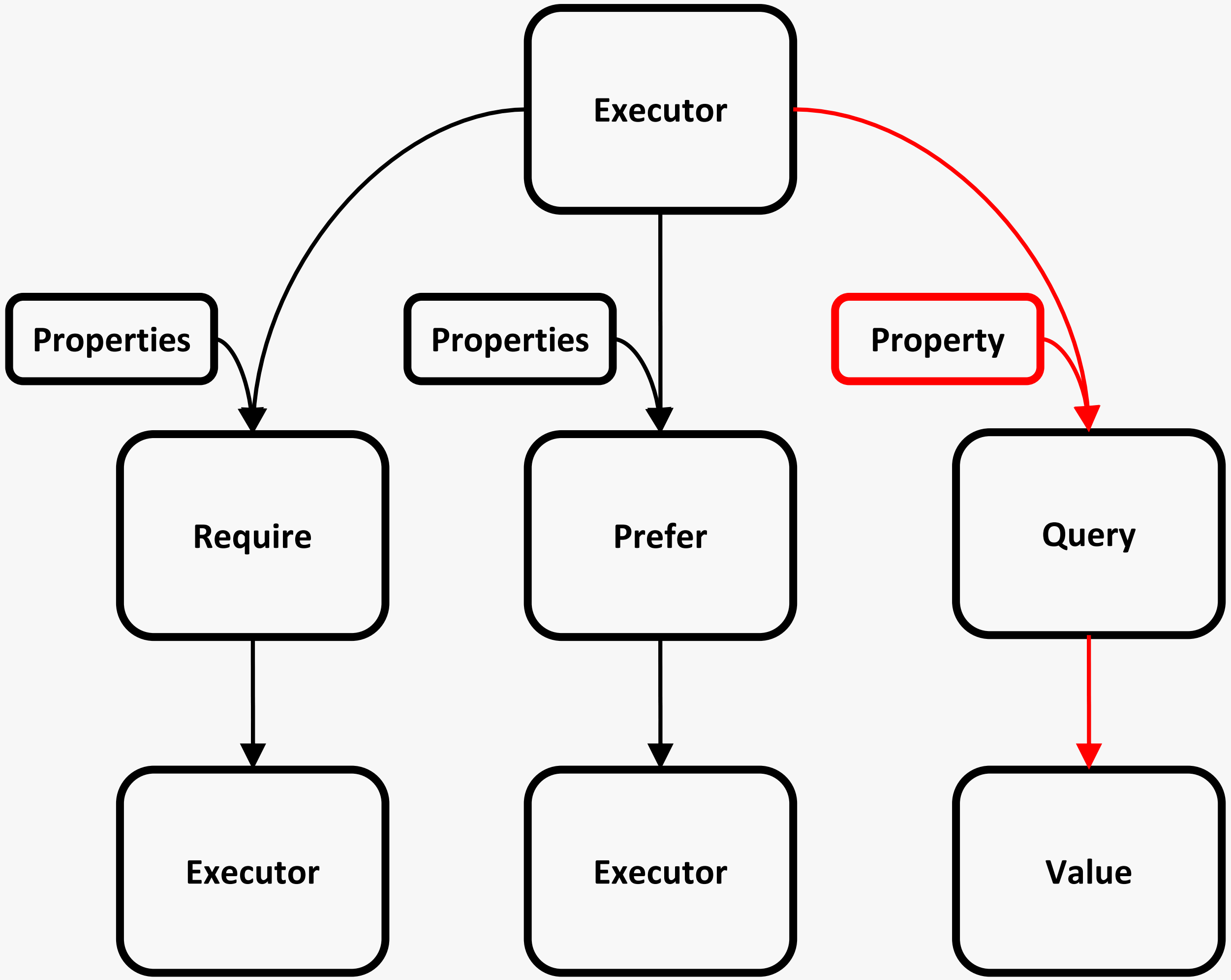
- Performing a require returns an executor that will have the requested properties
  - If the properties are already supported the original executor is returned
  - If the properties are not supported this will result in a compile-time error



- Performing a prefer returns an executor that may have the requested properties
  - If the properties are already supported the same executor is returned
  - If the properties are not supported the executor will simply return the original executor



- Performing a query returns the current value of a specific property
  - In many cases this value will be a boolean type



```
oneway_executor exec;

auto newExec = require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
oneway_executor exec;

auto newExec = require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
possibly_blocking_executor exec;

auto newExec = prefer(exec, never_blocking);

newExec.execute([&]() {
  func();
});
```

Prefer

```
oneway_executor exec;

auto newExec = require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
possibly_blocking_executor exec;

auto newExec = prefer(exec, never_blocking);

newExec.execute([&]() {
  func();
});
```

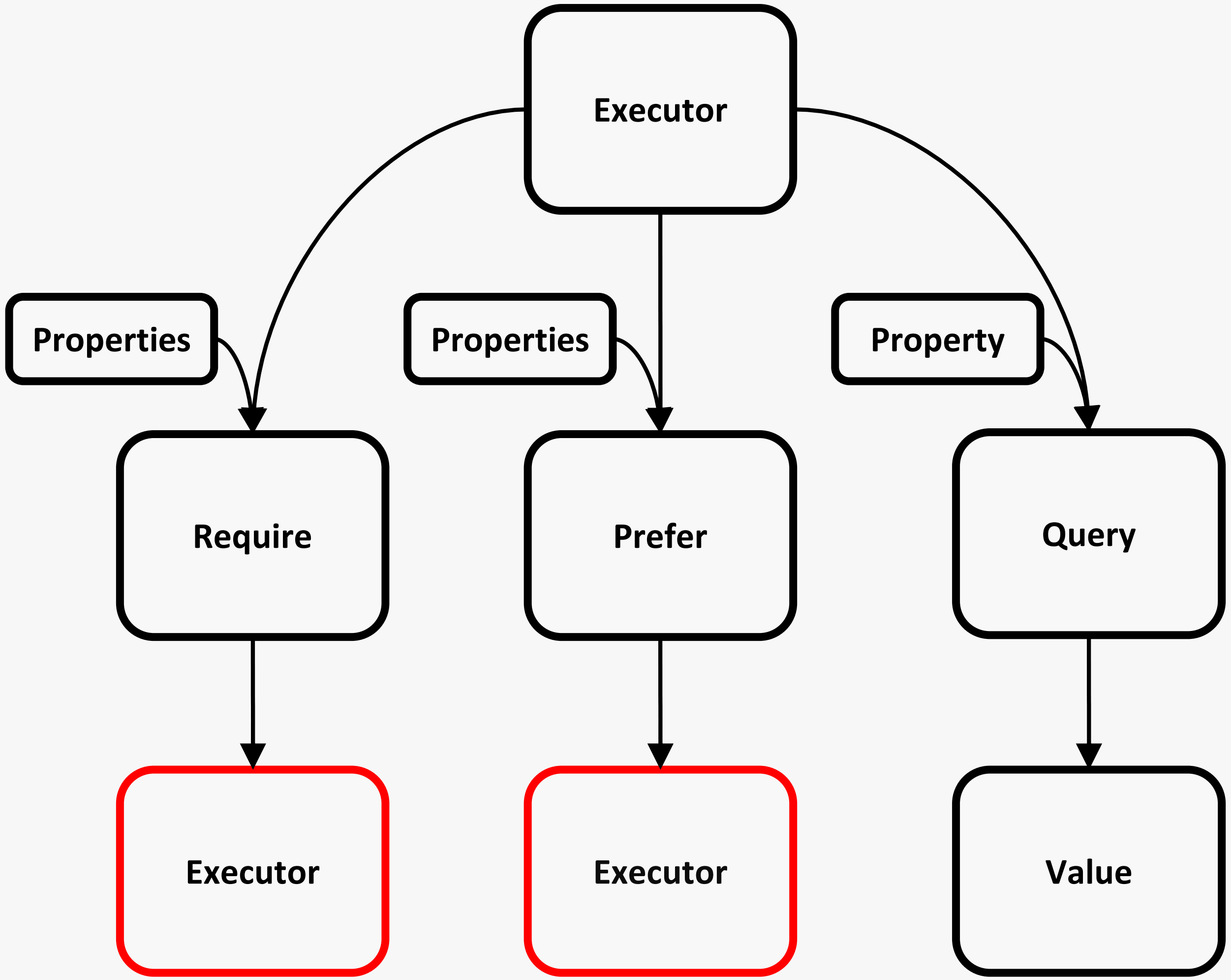
Prefer

```
possibly_blocking_executor exec;

auto newExec = prefer(exec, never_blocking);

auto isNeverBlocking = query(newExec, never_blocking);
```

Query

Polymorphic Executor

- The executor returned by require and prefer can be:
  - A static typed executor such as one_way_executor.
  - A dynamically typed executor wrapped in the polymorphic executor.



```
class my_scheduler {
public:

  twoway_executor exec;

};

{

  my_scheduler myScheduler;

  auto newExec = prefer(myScheduler.exec, never_blocking);

  auto fut = newExec.twoway_execute([&](){
    return func();
  });

}
```

Statically typed executors means executors cannot be stored generically

Statically typed executors means result of prefer must be known at compile time

```
class my_scheduler {
public:

  executor exec;

};

{

  my_scheduler myScheduler;

  myScheduler.exec = prefer(myScheduler.exec, never_blocking);

  auto fut = myScheduler.exec.twoway_execute([&](){
    return func();
  });

}
```

Implementing an executor

```
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {









  }
};
```

Naive Implementation

```
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = std::invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

  }
};
```

Naive Implementation

```
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = std::invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread([=, promise{std::move(promise)}]() mutable {





    });


  }
};
```

Naive Implementation

```
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = std::invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread([=, promise{std::move(promise)}]() mutable {
      try {


      } catch (...) {

      }
    });


  }
};
```

Naive Implementation

```cpp
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = std::invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread([=, promise{std::move(promise)}]() mutable {
      try {
        auto result = func();
        promise.set_value(result);
      } catch (...) {
        promise.set_exception(std::current_exception());
      }
    });


  }
};
```

Naive Implementation

```cpp
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = std::invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread([=, promise{std::move(promise)}]() mutable {
      try {
        auto result = func();
        promise.set_value(result);
      } catch (...) {
        promise.set_exception(std::current_exception());
      }
    }).detach();

    return fut;
  }
};
```

Naive Implementation

```
class my_executor {
public:
  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = std::invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    this->spawn_thread([=, promise{std::move(promise)}]() mutable {
      try {
        auto result = func();
        promise.set_value(result);
      } catch (...) {
        promise.set_exception(std::current_exception());
      }
    }).detach();

    return fut;
  }
};
```

Naive Implementation

Implementing std::async()

```
template <typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Function &&func, Args &&...args) {







}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {






}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

  auto fut = requiredExec.twoway_execute([&](){

  });


}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

  auto fut = requiredExec.twoway_execute([&](){
    return func(std::forward<Args>(args)...);
  });


}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

  auto fut = requiredExec.twoway_execute([&](){
    return func(std::forward<Args>(args)...);
  });

  return fut;
}
```

Using an executor

```
{

  int input = 10;

  auto fut = std::async([=](int m) {
    int factorial = 1;
    for (int i = 1; i <= m; ++i) {
      factorial *= i;
    }
    return factorial;
  }, input);

  auto result = fut.get();
}
```

```
{
  twoway_executor exec;
  int input = 10;

  auto fut = std::async(exec, [=](int m) {
    int factorial = 1;
    for (int i = 1; i <= m; ++i) {
      factorial *= i;
    }
    return factorial;
  }, input);

  auto result = fut.get();
}
```

```
{
  gpu_executor exec;
  int input = 10;

  auto fut = std::async(exec, [=](int m) {
    int factorial = 1;
    for (int i = 1; i <= m; ++i) {
      factorial *= i;
    }
    return factorial;
  }, input);

  auto result = fut.get();
}
```

Future work


UNDER
CONSTRUCTION

Execution Context / Resource

```
class static_thread_pool {
public:
  static_thread_pool_executor executor();

  void attach();

  void stop();

  void wait();

  class static_thread_pool_executor {
   public:

    template <typename Function>
    void execute(Function &&f);

    template <typename Function>
    void twoway_execute(Function &&f);

    template <typename Function, typename SharedFactory>
    std::future<std::invoke_result_t<std::decay_t<Function>>>
    bulk_execute(Function &&f, size_t n, SharedFactory &&sf);

    template <typename Function, typename ResultFactory, typename SharedFactory>
    std::future<std::invoke_result_t<std::decay_t<ResultFactory>>>
    bulk_twoway_execute(Function &&f, size_t n, ResultFactory &&rf, SharedFactory &&sf);
  };
};
```

Futures & Continuations

	One-way	Two-way
Single	`execute()`	`twoway_execute()`
Bulk	`bulk_execute()`	`bulk_twoway_execute()`

	One-way	Two-way
Single	`cut`	`way_execute()`
Bulk	`execute()`	`bulk_twoway_execute()`

LIE!

	One-way	Two-way	Then
Single	`execute()`	`twoway_execute()`	`then_execute()`
Bulk	`bulk_execute()`	`bulk_twoway_execute()`	`bulk_then_execute()`

```
{
  twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    funcA();
  });



}
```

```
{
  twoway_executor exec;
  auto fut = exec.twoway_execute([&](){
    funcA();
  }).then_execute([&]() {
    funcB();
  });
}
```

Renaming of Require / Prefer

Require	Prefer
rebind	maybe_rebind try_rebind
expect	maybe_expect try_expect
modify	maybe_modify try_modify
apply	maybe_apply try_apply
adapt	maybe_adapt try_adapt
transform	maybe_transform try_transform
transform_executor	maybe_transform_executor try_transform_executor

Back channels

```
{
  execution_context execContext([=](std::exception_ptr e){
    std::rethrow(e);
  });

  auto exec = execContext.executor();
  exec.execute([&]() {
    throw std::exception();
  });

  execContext.throw();
}
```

In summary

The unified interface for execution means…

Generic applications
Target a diverse range of resources
With a large amount of customisation

Executors are coming soon!

We're Hiring!
codeplay.com/careers/

codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you for Listening

@codeplaysoft

info@codeplay.com

codeplay.com