Contents

1 Introduction .................................................. 2

2 Motivation ..................................................... 2
  2.1 A lack of understanding ................................. 2
  2.2 Requirements that motivated the properties in [P0443R13] - executors .......... 2
  2.3 Satisfying the requirements ............................. 3
  2.4 [P1393R0] - properties contains a lot that is not required ................. 3
  2.5 [P1895R0] - tag_invoke will compose additional functionality .......... 5
  2.6 Compare the mechanisms defined in [P1393R0] - properties and [P1895R0] - tag_invoke .. 6
      2.6.1 Similarities ........................................... 6
      2.6.2 Definitions and Usage ............................... 6
  2.7 Behavioral properties are values and implement type-erasure .............. 10
  2.8 std::execution::allocator_t<> ................................ 10

3 Changes ....................................................... 11
  3.1 Naming customization points ............................ 11
  3.2 Remove dependencies on prefer .......................... 12
      3.2.1 Implementation ...................................... 12
  3.3 Remove any_executor ..................................... 13
  3.4 Move 2.4.2 Struct prefer_only to [P1393R0] - properties ...................... 14
  3.5 Modify section 2.2.11.1 Associated execution context property ............ 14
  3.6 Add section ? Behavioural tag values ........................ 15
  3.7 Add section ? Get blocking behaviour tag function ....................... 15
  3.8 Add section ? Change blocking behaviour tag function .................... 16
  3.9 Modify section 2.2.12.1 Blocking properties ....................... 16
  3.10 Modify section 2.2.12.1.1 blocking_t::always_t customization points ...... 17
  3.11 Add section ? Get relationship behaviour tag function ................. 18
  3.12 Add section ? Change relationship behaviour tag function .............. 19
  3.13 Modify section 2.2.12.3 Properties to indicate if submitted tasks represent continuations .. 19
  3.14 Add section ? Get outstanding work behaviour tag function ............. 20
  3.15 Add section ? Change outstanding work behaviour tag function .......... 20
  3.16 Modify section 2.2.12.4 Properties to indicate likely task submission in the future ........ 21
  3.17 Add section ? Get mapping behaviour tag function ...................... 22
  3.18 Add section ? Change mapping behaviour tag function ................... 22
  3.19 Modify section 2.2.12.6 Properties for mapping of execution on to threads .... 23
  3.20 Add section ? Get allocator tag function .................................. 23
  3.21 Add section ? Change allocator tag function ............................ 24
  3.22 Modify section 2.5.2 Class static_thread_pool ............................ 24
  3.23 Modify section 2.5.2.4 Scheduler creation .................................. 24
1 Introduction

This subclause describes components supporting an extensible customization mechanism, currently used most prominently by the execution support library \cite{P1393R0} - A General Property Customization Mechanism \cite{P1895R0} - A general pattern for supporting customisable functions is being considered as the mechanism that will be used to implement \cite{P1393R0} - properties. It is also intended to be used to implement basis function CPOs like execute() and algorithm CPOs like transform().

The properties mechanism was designed for use in executors and was split out of \cite{P0443R13} - executors because it would be used for far more than executors once it was in the standard. Several properties are defined for executors.

The opinion of this paper is that \cite{P1895R0} - tag_invoke provides a more idiomatic model for customization in C++ than \cite{P1393R0} - properties.

This paper will demonstrate changes to \cite{P0443R13} - executors, that would define the properties as tag_invoke functions, so that the mechanisms can be compared.

The changes in this paper are narrowly intended to explore how this type of customization should be expressed in the standard library. The semantics of the properties, as approved by SG1, are not changed.

The changes in this paper are implemented in a fork of asio github.

2 Motivation

2.1 a lack of understanding

A huge amount of committee time has been spent trying to understand what properties are and how to define a good property and how to implement a property correctly and how to use a property correctly. A lot of that discussion continues to be between authors of the papers. Some have low confidence that properties are understood.

2.2 requirements that motivated the properties in \cite{P0443R13} - executors
Table 1: requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensible</td>
<td>users are able to define their own</td>
</tr>
<tr>
<td>Survivable</td>
<td>passing objects around does not lose information</td>
</tr>
<tr>
<td>Forwardable</td>
<td>it is possible to write a catch-all that forwards to a contained object</td>
</tr>
<tr>
<td>Overridable</td>
<td>it is possible to modify or block forwarding</td>
</tr>
<tr>
<td>Defaultable and Ignorable</td>
<td>must be able to define a default when not customized and ignore when not customized</td>
</tr>
</tbody>
</table>

from: github

2.3 satisfying the requirements

Table 2: how the requirements are satisfied

<table>
<thead>
<tr>
<th>Requirement</th>
<th>tag_invoke</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensible</td>
<td>define a new tag function type in a namespace</td>
<td></td>
</tr>
<tr>
<td>Survivable</td>
<td>defined as overloads of the <code>tag_invoke()</code> function</td>
<td></td>
</tr>
<tr>
<td>Forwardable</td>
<td>a <code>tag_invoke()</code> function that does not constrain the namespaced tag function</td>
<td></td>
</tr>
<tr>
<td>Overridable</td>
<td>a <code>tag_invoke()</code> function constrained to a specific namespaced tag function</td>
<td></td>
</tr>
<tr>
<td>Defaultable and Ignorable</td>
<td>the namespaced tag function type has an overload of <code>operator()</code> constrained with <code>tag_invocable&lt;..&gt; == false</code></td>
<td>define a new traits struct in a new namespace</td>
</tr>
<tr>
<td></td>
<td></td>
<td>defined as overloads of <code>require()</code>, <code>require_concept()</code>, <code>prefer()</code>, and <code>query()</code> functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>require()</code>, <code>require_concept()</code>, <code>prefer()</code>, and <code>query()</code> functions that do not constrain the namespaced tag struct argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>require()</code>, <code>require_concept()</code>, <code>prefer()</code>, and <code>query()</code> functions constrained to a specific namespaced tag struct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the namespaced tag struct overloads <code>require()</code>, <code>require_concept()</code>, <code>prefer()</code>, and <code>query()</code> hidden friend functions with the target unconstrained and constrained with <code>can_require&lt;..&gt;</code>, <code>can_require_concept&lt;..&gt;</code>, <code>can_prefer&lt;..&gt;</code>, and <code>can_query&lt;..&gt; == false</code></td>
</tr>
</tbody>
</table>

2.4 [P1393R0] - properties contains a lot that is not required

Properties has features that were not in the motivating requirements.

**Meaning:** A fixed set of functions are defined that operate on properties and each of these functions has a fixed meaning across all properties. The search for meaning, when pairing each property with each function in the fixed set, consumes a lot of committee time.

**Polymorphism:** A property defines what return type to use when type-erasing the fixed set of functions that
operate on properties. In practice, the author of a property is unlikely to choose a polymorphic type that all users of that property will find satisfying. Also, each concept is expected to author a polymorphic type that implements that concept and supports forwarding of a specified set of properties. Of course, property authors may not be the authors of the concepts in which they traffic. Finally, when a property supports multiple concepts, like all the concepts in [P0443R13] - *executors*, it is unreasonable to provide a polymorphic type that satisfies all the concepts.

**Traits:** A property is a struct of traits members. A lot of those members control the **Meaning** and **Polymorphism** features.

### 2.4.0.1 Meaning

In [P1393R0] - *properties* functions are defined to have the same meaning across all properties. Each property selects the meanings that are valid.

NOTE: these meanings cannot be enforced across all properties.

The relevant section in [P1393R0] - *properties*:

.. each property imposes certain requirements on that object’s behavior or exposes some attribute of that object. As well as modifying the behavior of an object, properties can be applied to an object to enforce the presence of an interface, potentially resulting in an object of a new type that satisfies some concept associated with that property.

Authors of [P1393R0] - *properties* have verbally stated various contracts implied by properties for each of require/prefer/query. I do not see any description or enforcement of those contracts in the current paper.

In particular, the excerpt above states that require/prefer/query/require_concept are allowed to mutate the object. This does not match my understanding of the intent of the authors and seems particularly egregious for query. I would expect for all require/prefer/require_concept expressions to only allow const l-value or non-const r-value target objects. query would only allow const l-value target objects. Perhaps this is an oversight and will be remedied in a subsequent revision.

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
<th>trait members</th>
</tr>
</thead>
<tbody>
<tr>
<td>require</td>
<td>make a new instance of the target instance that satisfies the supplied property <strong>iff</strong> the target instance does not satisfy the supplied property, otherwise return the original target instance unchanged</td>
<td>is_requirable, polymorphic_wrapper_type</td>
</tr>
<tr>
<td>prefer</td>
<td>make a new instance of the target instance that satisfies the supplied property <strong>iff</strong> the supplied property is supported by the target and the target instance does not satisfy the supplied property, otherwise return the original target instance unchanged</td>
<td>is_preferable, polymorphic_wrapper_type</td>
</tr>
</tbody>
</table>
function | meaning | trait members
--- | --- | ---
require_concept | make a new instance of a type that satisfies a concept determined by the supplied property \(\text{iff}\) the target instance does not satisfy the concept determined by the supplied property, otherwise return the original target instance unchanged | is_requirable_concept, polymorphic_wrapper_type
query | return the value of the supplied property that this target instance satisfies | polymorphic_query_result_type, static_query_v, value()

### 2.4.0.2 Traits

Table 4: traits to define polymorphism, applicability, and the set of functions (meanings) supported for each property

<table>
<thead>
<tr>
<th>trait member</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_applicable_property_v&lt;T&gt;</td>
<td>(\text{true \ if \ } T\ \text{is allowed to support this property. In [P0443R13] - executors this is always derived from concepts})</td>
</tr>
<tr>
<td>is_requirable_concept</td>
<td>(\text{true \ if \ } \text{this property can be used with require_concept()})</td>
</tr>
<tr>
<td>is_requirable</td>
<td>(\text{true \ if \ } \text{this property can be used with require()})</td>
</tr>
<tr>
<td>is_preferable</td>
<td>(\text{true \ if \ } \text{this property can be used with prefer()})</td>
</tr>
<tr>
<td>static_query_v&lt;T&gt;</td>
<td>accesses the value that would be returned from a call to query() that is targeted on (T\ \text{iff the value has been made available at compile-time.})</td>
</tr>
<tr>
<td>value()</td>
<td>provides a value that will be compared to static_query_v&lt;T&gt; to “determine whether invoking require or require_concept would result in an identity transformation.” - [P1393R0] - properties</td>
</tr>
<tr>
<td>polymorphic_wrapper_type&lt;SupportableProperties...&gt;</td>
<td>defines the polymorphic type for a specific concept. The concept-specific expressions are built-in (like execute() for any_executor&lt;&gt;)</td>
</tr>
<tr>
<td>polymorphic_query_result_type</td>
<td>defines the result type for query when used by any polymorphic_wrapper_type&lt;SupportableProperties...&gt; that has the property in SupportableProperties... A wrapper, similar to prefer_only&lt;&gt; must be used to modify the property to override polymorphic_query_result_type</td>
</tr>
</tbody>
</table>

In addition to the traits, prefer_only<> is a wrapper that is used to modify a property so that only prefer is allowed. This allows a polymorphic_wrapper_type<SupportableProperties...>, that has the modified property in SupportableProperties..., to allow a target that does not support that property to be type-erased.

### 2.5 [P1895R0] - tag_invoke will compose additional functionality

tag_invoke is a mechanism to simplify the creation of new CPOs and to satisfy the requirements that motivated the properties mechanism.

**Meaning:** When a new CPO is defined using tag_invoke, the meaning is declared in the name of the CPO,
just like any other function. names can be built with all of the prefixes and suffixes already in use get\_, is\_, make\_, \_if, etc.. or without empty\(), clear\(), etc.

**Polymorphism:** When a new CPO is defined using `tag_invoke`, there is no need to define or implement the polymorphic types. polymorphism is a separate concern with a separate mechanism that is in terms of `tag_invoke()`\). This ability to build additional functionality, with composition and layering and without changes to the core spec, is considered an advantage.

polymorphism-per-function allows any CPO that supports `tag_invoke()`\) to be added to a vtable, including functions like `execute()`\) that participate in a concept

polymorphism-per-function allows any CPO that supports `tag_invoke()`\) to be added to a vtable with a user-specified result type (`variant<..>\) or `MyTypeErasedResult`\).

polymorphism-per-function-overload allows any CPO that supports `tag_invoke()`\) to be added to a vtable multiple times with differing argument types.

**Traits:** When a new CPO is defined using `tag_invoke`, there is no need to define traits to assign meaning or define polymorphism or applicability. std::invocable<CPO, target, Args\...) is the test for applicability. The name assigns meaning. Polymorphism is a separate concern.

### 2.6 Compare the mechanisms defined in [P1393R0] - properties and [P1895R0] - `tag_invoke`  

#### 2.6.1 Similarities

<table>
<thead>
<tr>
<th>topic</th>
<th>[P1895R0] - <code>tag_invoke</code></th>
<th>[P1393R0] - properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite list of globally reserved names</td>
<td><code>tag_invoke</code></td>
<td>require, prefer, query, static_query, and require_concept</td>
</tr>
<tr>
<td>defined as a namespaced function object</td>
<td>function object</td>
<td>type</td>
</tr>
<tr>
<td>find customizations by ADL</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>isolate namespaced definitions from ADL</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>transparent forwarding</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

NOTE: even with the ADL protections provided by these mechanisms, we have found that each customization defined for these mechanisms must carefully constrain the contexts for which the customization is considered.

*see twitter “… build times for the linked expression from 10min to 10s …”*

#### 2.6.2 Definitions and Usage
Table 6: definition mechanisms of tag_invoke functions and properties traits

<table>
<thead>
<tr>
<th>property</th>
<th>tag_invoke functions</th>
</tr>
</thead>
</table>
| struct my_property
{  
  template<class... Ps>
  static constexpr bool
    is_applicable_property_v;
  
  /* optional */
  static constexpr bool
    is_requirable_concept = /* ... */;

  /* optional */
  static constexpr bool
    is_requirable = /* ... */;

  /* optional */
  static constexpr bool
    is_preferable = /* ... */;

  /* optional */
  template<class... Ps>
    class polymorphic_wrapper_type;

  /* optional */
  using
    polymorphic_query_result_type = /* ... */;

  /* optional */
  template<class T>
    static constexpr /* ... */
      static_query_v = /* ... */;

  /* optional */
  static constexpr /* ... */
    value() const { return /* ... */; }
} |
| inline constexpr struct my_function_fn
{  
  template<
    typename... Tn>
  auto operator() (Tn&&... tn) const
    noexcept(noexcept(
        tbd::tag_invoke(*this, (Tn&&)tn...)))
    -> decltype(
        tbd::tag_invoke(*this, (Tn&&)tn...))
    { return 
        tbd::tag_invoke(*this, (Tn&&)tn...);
    }
} my_function{}; |
Table 7: usage of existing properties vs. tag_invoke (example from [P0443R13] - executors)

<table>
<thead>
<tr>
<th>properties</th>
<th>tag_invoke functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>// obtain an executor</td>
<td>executor auto ex = ...;</td>
</tr>
<tr>
<td>executor auto blocking_ex = std::require(</td>
<td>executor auto ex = ...;</td>
</tr>
<tr>
<td>ex, execution::blocking.always);</td>
<td>require the execute operation to block</td>
</tr>
<tr>
<td>// require the execute operation to block</td>
<td>executor auto blocking_ex =</td>
</tr>
<tr>
<td>executor auto blocking_ex = std::prefer(</td>
<td>execution::make_with_blocking(</td>
</tr>
<tr>
<td>blocking_ex, execution::priority(p));</td>
<td>ex, execution::always_blocking);</td>
</tr>
<tr>
<td>// prefer to execute with a particular priority p</td>
<td>executor auto blocking_ex_with_priority = tbd::prefer(</td>
</tr>
<tr>
<td>executor auto blocking_ex_with_priority =</td>
<td>execution::make_with_priority,</td>
</tr>
<tr>
<td>std::prefer(</td>
<td>blocking_ex, p);</td>
</tr>
<tr>
<td>blocking_ex, execution::priority(p));</td>
<td>// prefer to execute with a particular priority p</td>
</tr>
<tr>
<td>// execute my blocking, possibly prioritized</td>
<td>executor auto blocking_ex_with_priority =</td>
</tr>
<tr>
<td>// work</td>
<td>execution::execute(</td>
</tr>
<tr>
<td>execution::execute(</td>
<td>blocking_ex_with_priority, work);</td>
</tr>
<tr>
<td>blocking_ex_with_priority, work);</td>
<td>// execute my blocking, possibly prioritized</td>
</tr>
<tr>
<td></td>
<td>// work</td>
</tr>
</tbody>
</table>

8
Table 8: forwarding of arbitrary properties vs. tag_invoke (example from [P0443R13] - executors)

<table>
<thead>
<tr>
<th>properties</th>
<th>tag_invoke functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>template &lt;typename... SupportedProperties&gt;</code></td>
<td><code>template &lt;typename... Cpos&gt;</code></td>
</tr>
<tr>
<td><code>struct Any {</code></td>
<td><code>struct Any {</code></td>
</tr>
<tr>
<td><code>  template &lt;class Property&gt; Property::polymorphic_wrapper_type&lt;</code></td>
<td><code>  template &lt;typename Tag, typename Tn...&gt; friend auto tag_invoke(</code></td>
</tr>
<tr>
<td><code>     SupportedProperties...&gt;</code></td>
<td><code>    Tag tag,</code></td>
</tr>
<tr>
<td><code>     require_concept(Property p) const {</code></td>
<td><code>    const Any&amp; self,</code></td>
</tr>
<tr>
<td><code>       return</code></td>
<td><code>    Tn&amp;&amp;... tn) {</code></td>
</tr>
<tr>
<td><code>       Property::polymorphic_wrapper_type&lt;</code></td>
<td><code>    return tbd::tag_invoke(</code></td>
</tr>
<tr>
<td><code>       SupportedProperties...&gt;{</code></td>
<td><code>    tag, self.t_, (Tn&amp;&amp;)tn...);</code></td>
</tr>
<tr>
<td><code>       std::require(t_, p)</code></td>
<td><code>    }</code></td>
</tr>
<tr>
<td><code>     }</code></td>
<td><code> }</code></td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>template &lt;class Property&gt;</code></td>
<td><code>template &lt;typename Tag, typename Tn...&gt;</code></td>
</tr>
<tr>
<td><code>Any require(Property p) const {</code></td>
<td><code>friend auto tag_invoke(</code></td>
</tr>
<tr>
<td><code>    return Any{std::require(t_, p)};</code></td>
<td><code>    Tag tag,</code></td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>    const Any&amp; self,</code></td>
</tr>
<tr>
<td><code>template &lt;class Property&gt;</code></td>
<td><code>Tn&amp;&amp;... tn) {</code></td>
</tr>
<tr>
<td><code>Any prefer(Property p) const {</code></td>
<td><code>    return tbd::tag_invoke(</code></td>
</tr>
<tr>
<td><code>    return Any{std::require(t_, p)};</code></td>
<td><code>    tag, self.t_, (Tn&amp;&amp;)tn...);</code></td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>    }</code></td>
</tr>
<tr>
<td><code>template &lt;class Property&gt;</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>typename</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>Property::polymorphic_query_result_type</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>query(Property p) const {</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>    return std::query(t_, p);</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>};</code></td>
<td><code>};</code></td>
</tr>
</tbody>
</table>
Table 9: customizing known properties vs. tag_invoke (example from [P0443R13] - executors)

<table>
<thead>
<tr>
<th>properties</th>
<th>tag_invoke functions</th>
</tr>
</thead>
</table>
| struct C {
  auto require(
    const execution::allocator_t<void>& a
  ) const;
  template<class PAlloc>
  auto require(
    const execution::allocator_t<PAlloc>& a
  ) const;
  auto query(execution::context_t) const noexcept;
  auto query(execution::allocator_t<void>) const noexcept;
  template<class ProtoAllocator>
  auto query(
    execution::allocator_t<ProtoAllocator>
  ) const noexcept;
}; | struct C {
  template<allocator A>
  friend auto tag_invoke(
    tbd::make_with_allocator tag,
    const C& self,
    A&& a) { /*...*/}
  friend auto tag_invoke(
    execution::get_context tag,
    const C& self) noexcept { /*...*/}
  friend auto tag_invoke(
    tbd::get_allocator tag,
    const C& self) noexcept { /*...*/}
}; |

2.7 Behavioral properties are values and implement type-erasure

[P0443R13] - executors defines several properties that model behavioural-properties. blocking, mapping, relationship, outstanding work all model behavioural-properties.

[P0443R13] - 2.2.12 Behavioral properties defines a pattern of using behaviour properties nested in a group property type. All of the properties are value types that model equality-comparable. The group property is additionally a type-eraser that can be constructed from any of the nested behaviour properties.

The implementation in asio uses an integer to represent which nested behaviour property value was used to construct a particular group property value.

This is a novel pattern. It is unclear whether it would be possible to add new nested properties to the standard over time. It is unclear how a user or library would be able to add additional nested properties without changing the standard. It is unclear what the ABI impact of a new nested property would have on the type-erasing group properties.

2.8 std::execution::allocator_t<> questions arise out of the allocator property.

What does it mean to require(t, std::execution::allocator(my_allocator))?

options:
1. mutate t and force it to reallocate the contents with the specified allocator
2. copy t and allocate the contents of the copy with the specified allocator
3. construct a new ‘default’ T so that it will use the specified allocator
4. T is a reference type and constructs a new reference that uses the specified allocator locally. The target of the reference is not changed or notified.

Only case 4 has been explored. An executor is a reference to an execution context. Case 4 would not be appropriate for a `vector<>` and this use `require(vector<>, allocator)`, has not been explored.

What does it mean to `query(myvector, std::execution::allocator)`?

1. compile error because `std::execution::allocator::is_applicable_property<T>` requires `executor<T>` to be true?
2. call `T::get_allocator()` if it exists?

It is unclear that such a general property as `allocator_t` should be namespaced in `std::execution` and restricted to only work when `T` is an executor.

3 Changes

3.1 Naming customization points

Changing the existing properties in [P0443R13] - `executors` into `tag_invoke` functions will involve renaming and refactoring them.

`tag_invoke` CPO’s are designed and named as any other function is named. properties have a different naming model in that the name must compose with the fixed set of functions defined in [P1393R0] - `properties`.

Note: I did not find a way in [P1393R0] - `properties` to disable `query` for a property. Thus the two-property options listed below define `query` for both properties and it is sometimes unclear what `query` would mean for one of the properties.

To convey the difference, here are two familiar patterns from the standard library and some options for defining them as properties.

**Pattern A**: the `c.empty()` and `c.clear()` methods on a container, like `vector`. To be customizable, these might be defined as the `tag_invoke` functions `is_empty(c)` and `clear(c)`. Two options for how these functions could be expressed as properties are shown. The single property version is how the existing properties are structured.

<table>
<thead>
<tr>
<th>function</th>
<th>query</th>
<th>requires</th>
<th>prefer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_empty(t) &amp; clear(t)</code></td>
<td><code>query(t, empty)</code></td>
<td><code>requires(t, empty)</code></td>
<td><code>prefer(t, empty)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>query</th>
<th>requires</th>
<th>prefer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_empty(t)</code></td>
<td><code>query(t, is_empty)</code></td>
<td><code>x</code></td>
<td><code>x</code></td>
</tr>
<tr>
<td><code>clear(t)</code></td>
<td><code>query(t, clear)</code></td>
<td><code>requires(t, clear)</code></td>
<td><code>prefer(t, clear)</code></td>
</tr>
</tbody>
</table>

**Pattern B**: the `c.get_allocator()` method and `C(allocator)` constructor on a container, like `vector<>`. To be customizable, these might be defined as the `tag_invoke` functions `get_allocator(c)` and `make_with_allocator(c, a)`. Two options for how this function and constructor could be expressed as
properties are shown. The single property version is how the existing properties are structured.

Table 12: naming get_allocator() and make_with_allocator() functions vs. a single property named allocator

<table>
<thead>
<tr>
<th>function</th>
<th>query</th>
<th>requires</th>
<th>prefer</th>
</tr>
</thead>
</table>
| get_allocator(t) & make_with_allocator(t, a) | vs query(t, requires(t, prefer(t, allocator(a)))

Table 13: naming get_allocator() and make_with_allocator() functions vs. get_allocator and with_allocator properties

<table>
<thead>
<tr>
<th>function</th>
<th>query</th>
<th>requires</th>
<th>prefer</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_allocator(t)</td>
<td>vs query(t, get_allocator)</td>
<td>requires(t, prefer(t, with_allocator(a)))</td>
<td>x</td>
</tr>
<tr>
<td>make_with_allocator(t, a)</td>
<td>vs query(t, requires(t, prefer(t, with_allocator(a))))</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>

3.2 remove dependencies on prefer

prefer is intended to simplify the boilerplate needed to apply a function only if it is supported by the target. This support does not need to be included in tag_invoke. It can be implemented in terms of tag_invoke. This ability to build additional functionality, with composition and layering and without changes to the core spec, is considered an advantage.

Table 14: prefer usage comparison

<table>
<thead>
<tr>
<th>properties</th>
<th>tag_invoke functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>executor auto blocking_ex_with_priority = std::prefer(blocking_ex, execution::priority(p));</td>
<td>executor auto blocking_ex_with_priority = tbd::prefer(execution::make_with_priority, blocking_ex, p);</td>
</tr>
</tbody>
</table>

3.2.1 Implementation

Here is an implementation of the prefer CPO in terms of tag_invoke.

```cpp
inlineconstexpr struct prefer_fn {
    // allow target to customize prefer
    template<typename Tag, typename... Tn>
    auto operator()(Tag&& tag, Tn&&... tn) const
    noexcept(noexcept(tbd::tag_invoke(*this, tag, (Tn&&)tn...)))
    -> decltype(tbd::tag_invoke(*this, tag, (Tn&&)tn...)) {
        return
    }
```
3.3 remove any_executor

The any_executor type is a polymorphic wrapper for an executor. any_executor is tightly integrated with properties. This integration allows a user-specified list of properties to be forwarded through to the wrapped executor. Each property defines a polymorphic type for the result of query and a polymorphic wrapper type for the result of require.

In [P1895R0] - tag_invoke, type-erasure is described such that multiple overloads of any tag_invoke function can be forwarded to a wrapped object. There is no distinction between queries and algorithms and concept basis functions, all use the same mechanism to build a vtable on a generic implementation. This allows a single type any_unique, any_ref, user-defined-type-eraser, etc. to represent any concept that can be defined in terms of tag_invoke expressions.

Given the any_unique implementation referenced in [P1895R0] - tag_invoke (see godbolt), the following definition of any_executor_unique would provide an implementation that, with the other changes in this paper, would enable polymorphic wrapping for the executor concept and any other tag_invoke function overloads added to the list.

```
template <typename Function>
using execute_o = std::tag_t<tbd::overload<void(this_&&, Function)>
  (execution::execute)>;

template <typename... OverloadN>
using any_executor_unique = any_unique<
  execute_o<std::function>,
  execute_o<void(*)(*)>,
  OverloadN...>;
```

There is also an implementation of any_ref in a fork of asio (see github). This branch allows any_executor_ref
Some codebases have their own polymorphic function wrapper (such as `folly::Function`). This approach allows that wrapper to be punched through without converting it to a `std::function` by creating an overload of `execute` for `folly::Function` that has a separate vtable entry.

```cpp
template <typename... OverloadN>
using folly_any_executor_ref = any_executor_ref<
    execution::get_blocking_o_t<folly::Function>,
    OverloadN...>;
```

overloading of the tag_invoke functions that this paper proposes, to replace the properties defined in [P0443R13] - `executors`, is supported the same way as the basis function overloads.

```cpp
template <typename... OverloadN>
using my_any_executor_ref = any_executor_ref<
    execution::get_blocking_o_t<>,
    execution::make_with_blocking_o_t<
        any_executor_ref<
            execution::get_blocking_o_t<>>,
        decltype(execution::always_blocking)>,
    execution::make_with_blocking_o_t<
        any_executor_ref<
            execution::get_blocking_o_t<>>,
        decltype(execution::never_blocking)>,
    OverloadN...>;
```

This paper will not try to define the `any_unique`, `any_ref`, or `any_executor` types in wording. Instead this paper will remove `any_executor` so that it can be defined in terms of `any_unique/any_ref/etc..` when available and by user defined type-erasure until general type-erasure ships in the standard.

### 3.4 move 2.4.2 Struct prefer_only to [P1393R0] - `properties`

`prefer_only` is part of the polymorphic support for properties in general. There is nothing executor specific about this type.

### 3.5 Modify section 2.2.11.1 Associated execution context property

Associated execution context `property_tag_function`

```cpp
- struct context_t
- {
-     template <class T>
-         static constexpr bool is_applicable_property_v = executor<T>;
-     static constexpr bool is_requirable = false;
-     static constexpr bool is_preferable = false;
- ```
- using polymorphic_query_result_type = any;
- 
- template<
class Executor>
- static constexpr decltype(auto) static_query_v
- = Executor::query(context_t());
- 
+ inline constexpr struct get_context_t {
+ template<typename T>
+ auto constexpr operator()(const T& t) const
+ noexcept(noexcept(
+ tbd::tag_invoke(*this, t)))
+ -> decltype(
+ tbd::tag_invoke(*this, t)) {
+ return
+ tbd::tag_invoke(*this, t);
+ }
+ } get_context{};

The context_t property can be used only with query, which get_context(e) function returns the execution context associated with the executor executor e.

The value returned from std::query(e, context_t) get_context(e), where e is an executor, shall not change between invocations unless the executor is assigned another executor with a different context.

3.6 Add section ? Behavioural tag values

? Behavioural tag values

Behavioural tag values define a set of mutually-exclusive values that describe a behaviour.

Unless otherwise specified, behavioural tag values conform to the following specification.

inline constexpr struct behaviour_group_t {
 friend constexpr bool operator==(const behaviour_group_t& t1, const behaviour_group_t& t2) {
 return true;
 }
 friend constexpr bool operator!=(const behaviour_group_t& t1, const behaviour_group_t& t2) {
 return false;
 }
} behaviour_group {};

The result of get_group for the value of an object’s behavioral tag value shall not change between invocations unless the object is assigned another object with a different value of that behavioral tag value.

3.7 Add section ? Get blocking behaviour tag function

? Get blocking behaviour tag function
inline constexpr struct get_blocking_t {
    template<typename T>
    auto constexpr operator()(const T& t) const
        noexcept(noexcept(
            tbd::tag_invoke(*this, t)))
        -> decltype(
            tbd::tag_invoke(*this, t)) {
        return
            tbd::tag_invoke(*this, t);
    }
} get_blocking{};

The get_blocking(e) function returns the blocking tag value that describes the behaviour of the execution functions on object e.

The value returned from get_blocking(e), where e is an object, shall not change between invocations unless e is assigned another object with a different blocking tag value.

For some subexpression e, let E be a type such that decltype((e)) is E. The expression execution::get_blocking(e) is expression-equivalent to:

— tbd::tag_invoke(get_blocking, e), if that expression is valid.
— Otherwise, execution::get_blocking(e) is ill-formed.

### 3.8 Add section ? Change blocking behaviour tag function

? Change blocking behaviour tag function

inline constexpr struct make_with_blocking_t {
    template<typename T, typename B>
    auto constexpr operator()(const T& t, B&& b) const
        noexcept(noexcept(
            tbd::tag_invoke(*this, t, (B&&)b)))
        -> decltype(
            tbd::tag_invoke(*this, t, (B&&)b)) {
        return
            tbd::tag_invoke(*this, t, (B&&)b);
    }
} make_with_blocking{};

For some subexpressions e and b, let E be a type such that decltype((e)) is E, and let B be a type such that decltype((b)) is B. The expression execution::make_with_blocking(e, b) is expression-equivalent to:

— tbd::tag_invoke(make_with_blocking, e, b), if that expression is valid.
— Otherwise, execution::make_with_blocking(e, b) is ill-formed.

**Returns**: A value t1 of type T1 that holds a copy of t. T1 provides an overload of tag_invoke such that make_with_blocking(t1, b) returns a copy of t1, an overload of tag_invoke such that get_blocking(t1) returns b, and execution functions derived from T1 shall block the calling thread until the submitted functions have finished execution.

### 3.9 Modify section 2.2.12.1 Blocking properties

? Blocking properties behavioural tag values
The `blocking_t` property following tag values describes what guarantees executors provide about the blocking behavior of their execution functions. `blocking_t` provides nested property types and objects as described below. Behavioural tag value types and objects are provided as described below.

<table>
<thead>
<tr>
<th>Nested Property Type</th>
<th>Tag Value</th>
<th>Nested Property Object Name</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>blocking_t::possibly_t</code></td>
<td>possibly_blocking_t</td>
<td><code>possibly_blocking</code></td>
<td>Invocation of an executor’s execution function may block pending completion of one or more invocations of the submitted function object.</td>
</tr>
<tr>
<td><code>blocking_t::always_t</code></td>
<td>always_blocking_t</td>
<td><code>always_blocking</code></td>
<td>Invocation of an executor’s execution function shall block until completion of all invocations of submitted function object.</td>
</tr>
<tr>
<td><code>blocking_t::never_t</code></td>
<td>never_blocking</td>
<td></td>
<td>Invocation of an executor’s execution function shall not block pending completion of the invocations of the submitted function object.</td>
</tr>
</tbody>
</table>

3.10 Modify section 2.2.12.1.1 `blocking_t::always_t` customization points

In addition to conforming to the above specification, the `blocking_t::always_t` property `always_blocking` tag value provides the following customization:

```cpp
-struct always_t
-{
  - static constexpr bool is_required = true;
  - static constexpr bool is_preferable = false;
  -
  - template <class T>
  -  static constexpr bool is_applicable_property_v = executor<T>;
  -
  - template<class Executor>
  -   friend see-below require(Executor ex, blocking_t::always_t);
-};
+inline constexpr struct always_blocking_t {
  + friend constexpr bool operator==(const always_blocking_t&, const always_blocking_t&)
  + {
  +    return true;
  +  }
  + friend constexpr bool operator!=(const always_blocking_t&, const always_blocking_t&)
  + {
  +    return false;
  +  }
  + template<class Executor>
  +   friend see-below constexpr tag_invoke(
```
If the executor has the blocking_adaptation_t::allowed_t property, this customization uses an adapter to implement the blocking_t::always_t property always_blocking_t tag.

-template<class Executor>
 friend see-below require(Executor ex, blocking_t::always_t);
+template<class Executor>
 friend see-below tag_invoke(
   execution::make_with_blocking_t,
   const Executor& ex,
   always_blocking_t);

Returns: A value e1 of type E1 that holds a copy of ex. E1 provides an overload of require_tag_invoke such that e1.require_make_with_blocking(blocking_always_e1, always_blocking) returns a copy of e1, an overload of query_tag_invoke such that query_get_blocking(e1, blocking) returns blocking.always.always_blocking, and functions execute and bulk_execute execution functions derived from T1 shall block the calling thread until the submitted functions have finished execution. e1 has the same executor properties as ex, except for the addition of the blocking_t::always_t property, and removal of blocking_t::never_t and blocking_t::possibly_t properties if present.

Remarks: This function shall not participate in overload resolution unless blocking_adaptation_t::static_query_v is blocking_adaptation.allowed.

3.11 Add section ? Get relationship behaviour tag function

Get relationship behaviour tag function

inline constexpr struct get_relationship_t {
  template<typename T>
  auto constexpr operator()(const T& t) const
  noexcept(noexcept(tbd::tag_invoke(*this, t)))
  -> decltype(tbd::tag_invoke(*this, t)) {
    tbd::tag_invoke(*this, t);
  }
} get_relationship{};

The get_relationship(e) function returns the relationship tag value that describes the behaviour of the execution functions on object e.

The value returned from get_relationship(e), where e is an object, shall not change between invocations unless e is assigned another object with a different relationship tag value.

For some subexpression e, let E be a type such that decltype((e)) is E. The expression execution::get_relationship(e) is expression-equivalent to:

— tbd::tag_invoke(get_relationship, e), if that expression is valid.
— Otherwise, execution::get_relationship(e) is ill-formed.
3.12 Add section ? Change relationship behaviour tag function

? Change relationship behaviour tag function

```cpp
inline constexpr struct make_with_relationship_t {
    template<typename T, typename R>
    auto constexpr operator()(const T& t, R&& r) const
        noexcept(noexcept(tbd::tag_invoke(*this, t, (R&&)r)))
        -> decltype(tbd::tag_invoke(*this, t, (R&&)r)) {
        return tbd::tag_invoke(*this, t, (R&&)r);
    }
} make_with_relationship{};
```

For some subexpressions e and r, let E be a type such that decltype((e)) is E, and let R be a type such that decltype((r)) is R. The expression execution::make_with_relationship(e, r) is expression-equivalent to:

— tbd::tag_invoke(make_with_relationship, e, r), if that expression is valid.
— Otherwise, execution::make_with_relationship(e, r) is ill-formed.

Returns: A value t1 of type T1 that holds a copy of t. T1 provides an overload of tag_invoke such that make_with_relationship(t1, r) returns a copy of t1, an overload of tag_invoke such that get_relationship(t1) returns r.

3.13 Modify section 2.2.12.3 Properties to indicate if submitted tasks represent continuations

? Properties Tag values to indicate if submitted tasks represent continuations

The relationship_t property following tag values allows users of executors to indicate that submitted tasks represent continuations.

relationship_t provides nested property types and objects as indicated below. Behavioural tag value types and objects are provided as described below.

<table>
<thead>
<tr>
<th>Nested Property Type</th>
<th>Nested Property Tag Value</th>
<th>Object Name</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship_t::fork_t</td>
<td>relationship.fork</td>
<td>fork_relationship</td>
<td>Function objects Tasks submitted through the executor do not represent continuations of the earlier running task. This is a hint to order newly submitted tasks independently of the running task.</td>
</tr>
<tr>
<td>fork_relationship_t</td>
<td>fork_relationship</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


### 3.14 Add section ? *Get outstanding work behaviour tag function*

Get outstanding work behaviour tag function

```cpp
inline constexpr struct get_outstanding_work_t {
  template<typename T>
  auto constexpr operator()(const T& t) const
    noexcept(noexcept(tbd::tag_invoke(*this, t)))
    -> decltype(tbd::tag_invoke(*this, t)) {
    return tbd::tag_invoke(*this, t);
  }
} get_outstanding_work{};
```

The `get_outstanding_work(e)` function returns the outstanding work tag value that describes the behaviour of the execution functions on object `e`.

The value returned from `get_outstanding_work(e)`, where `e` is an object, shall not change between invocations unless `e` is assigned another object with a different outstanding work tag value.

For some subexpression `e`, let `E` be a type such that `decltype((e))` is `E`. The expression `execution::get_outstanding_work(e)` is expression-equivalent to:

- `tbd::tag_invoke(get_outstanding_work, e)`, if that expression is valid.
- Otherwise, `execution::get_outstanding_work(e)` is ill-formed.

### 3.15 Add section ? *Change outstanding work behaviour tag function*

Change outstanding work behaviour tag function

```cpp
inline constexpr struct make_with_outstanding_work_t {
  template<typename T, typename O>
  auto constexpr operator()(const T& t, O&& o) const
    noexcept(noexcept(tbd::tag_invoke(*this, t, (O&&)o)))
    -> decltype(tbd::tag_invoke(*this, t, (O&&)o)) {
    return tbd::tag_invoke(*this, t, (O&&)o);
  }
} make_with_outstanding_work{};
```
```cpp
tbd::tag_invoke(*this, t, (O&&)o);
}
} make_with_outstanding_work{};
```

For some subexpressions `e` and `o`, let `E` be a type such that `decltype((e))` is `E`, and let `O` be a type such that `decltype((o))` is `O`. The expression `execution::make_with_outstanding_work(e, o)` is expression-equivalent to:

- `tbd::tag_invoke(make_with_outstanding_work, e, o)`, if that expression is valid.
- Otherwise, `execution::make_with_outstanding_work(e, o)` is ill-formed.

**Returns:** A value `t1` of type `T1` that holds a copy of `t`. `T1` provides an overload of `tag_invoke` such that `make_with_outstanding_work(t1, o)` returns a copy of `t1`, an overload of `tag_invoke` such that `get_outstanding_work(t1)` returns `o`.

### 3.16 Modify section 2.2.12.4 Properties to indicate likely task submission in the future

Properties Tag values to indicate likely task submission in the future

The `outstanding_work_t` property following tag values allows users of executors to indicate that task submission is likely in the future.

`outstanding_work_t` provides nested property types and objects as indicated below. Behavioural tag value types and objects are provided as described below.

<table>
<thead>
<tr>
<th>Nested Property Type</th>
<th>Tag Value</th>
<th>Object Name</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>outstanding_work_t::untracked_t</td>
<td>untracked_outstanding_work_t</td>
<td>outstanding_work.untracked</td>
<td>The existence of the executor object does not indicate any likely future submission of a function object task. The existence of the executor object represents an indication of likely future submission of a function object task. The executor object or its associated execution context may choose to maintain execution resources in anticipation of this submission.</td>
</tr>
<tr>
<td>outstanding_work_t::tracked_t</td>
<td>tracked_outstanding_work_t</td>
<td>tracked_outstanding_work</td>
<td></td>
</tr>
</tbody>
</table>

[Note: This behaviour is equivalent to selecting whether an object acts as a `shared_ptr<>` or as a `weak_ptr<>` to some internal resources — end note]

[Note: The outstanding_work_t::tracked_t and outstanding_work_t::untracked_t properties are used to communicate to the associated execution context intended future work submission on the executor object. The untracked_outstanding_work_t and tracked_outstanding_work_t tag value types are used to communicate to the object any intended future work submission on the object. The intended effect of the properties tag values is to change the behavior of execution context’s the object’s facilities for awaiting outstanding work; specifically whether it considers the existence of the executor object with the outstanding_work_t::tracked_t property enabled tracked_outstanding_work_t tag value type selected, outstanding work when deciding what to wait on. However this will be largely defined by the execution context implementation of the object. It is intended that the execution context object will define its wait facilities and on-destruction behaviour and provide an interface for querying this. An initial work towards this is included in [P0737R0]. — end note]
### 3.17 Add section ? Get mapping behaviour tag function

Get mapping behaviour tag function

```cpp
inline constexpr struct get_mapping_t {
    template<typename T>
    auto constexpr operator()(const T& t) const
        -> decltype(tbd::tag_invoke(*this, t)) {
        return tbd::tag_invoke(*this, t);
    }
} get_mapping{};
```

The `get_mapping(e)` function returns the mapping tag value that describes the behaviour of the execution functions on object `e`.

The value returned from `get_mapping(e)`, where `e` is an object, shall not change between invocations unless `e` is assigned another object with a different relationship tag value.

For some subexpression `e`, let `E` be a type such that `decltype((e))` is `E`. The expression `execution::get_mapping(e)` is expression-equivalent to:

- `tbd::tag_invoke(get_mapping, e)`, if that expression is valid.
- Otherwise, `execution::get_mapping(e)` is ill-formed.

### 3.18 Add section ? Change mapping behaviour tag function

Change mapping behaviour tag function

```cpp
inline constexpr struct make_with_mapping_t {
    template<typename T, typename M>
    auto constexpr operator()(const T& t, M&& m) const
        -> decltype(tbd::tag_invoke(*this, t, (M&&)m)) {
        return tbd::tag_invoke(*this, t, (M&&)m);
    }
} make_with_mapping{};
```

For some subexpressions `e` and `m`, let `E` be a type such that `decltype((e))` is `E`, and let `M` be a type such that `decltype((m))` is `M`. The expression `execution::make_with_mapping(e, m)` is expression-equivalent to:

- `tbd::tag_invoke(make_with_mapping, e, m)`, if that expression is valid.
- Otherwise, `execution::make_with_mapping(e, m)` is ill-formed.

**Returns:** A value `t1` of type `T1` that holds a copy of `t`. `T1` provides an overload of `tag_invoke` such that `make_with_mapping(t1, m)` returns a copy of `t1`, an overload of `tag_invoke` such that `get_mapping(t1)` returns `m`. 

22
3.19 Modify section 2.2.12.6 Properties for mapping of execution on to threads

Properties Tag values for mapping of execution on to threads

The `mapping_t` property following tag values describes what guarantees an object provides about the mapping of execution agents onto threads of execution.

`mapping_t` provides nested property types and objects as indicated below. Behavioural tag value types and objects are provided as described below.

<table>
<thead>
<tr>
<th>Nested Property Tag</th>
<th>Value</th>
<th>Nested Property Tag</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mapping_t::thread_t</code></td>
<td>thread_mapping_t</td>
<td><code>mapping_t::new_thread_t</code></td>
<td>new_thread_mapping_t</td>
<td>Execution agents are mapped onto threads of execution.</td>
</tr>
<tr>
<td><code>mapping_t::new_thread_t</code></td>
<td>new_thread_mapping_t</td>
<td><code>mapping_t::other_t</code></td>
<td>other_mapping_t</td>
<td>Each execution agent is mapped onto a new thread of execution.</td>
</tr>
<tr>
<td><code>mapping_t::other_t</code></td>
<td>other_mapping_t</td>
<td></td>
<td></td>
<td>Mapping of each execution agent is implementation-defined.</td>
</tr>
</tbody>
</table>

[ Note: A mapping of an execution agent onto a thread of execution implies the execution agent runs as-if on a `std::thread`. Therefore, the facilities provided by `std::thread`, such as thread-local storage, are available. `mapping_t::new_thread_t` provides stronger guarantees, in particular that thread-local storage will not be shared between execution agents. — end note ]

3.20 Add section ? Get allocator tag function

Get allocator tag function

```cpp
inline constexpr struct get_allocator_t {
    template<typename T>
    auto constexpr operator()(const T& t) const
        noexcept(noexcept(tbd::tag_invoke(*this, t)))
        -> decltype(tbd::tag_invoke(*this, t)) {
        tbd::tag_invoke(*this, t) {
            return tbd::tag_invoke(*this, t);
        }
    }
} get_allocator{};
```

The `get_allocator(e)` function returns the allocator in use by object `e`.

The value returned from `get_allocator(e)`, where `e` is an object, shall not change between invocations unless `e` is assigned another object with a different relationship tag value.

For some subexpression `e`, let `E` be a type such that `decltype((e))` is `E`. The expression `execution::get_allocator(e)` is expression-equivalent to:

- `tbd::tag_invoke(get_allocator, e)`, if that expression is valid.
- Otherwise, `execution::get_allocator(e)` is ill-formed.
3.21 Add section Change allocator tag function

* Change allocator tag function

```cpp
inline constexpr struct make_with_allocator_t {
    template<typename T, typename A>
    auto constexpr operator()(const T& t, A&& a) const
        noexcept(noexcept(tbd::tag_invoke(*this, t, (A&&)a)))
        -> decltype(tbd::tag_invoke(*this, t, (A&&)a)) {
        return tbd::tag_invoke(*this, t, (A&&)a);
    }
} make_with_allocator{};
```

For some subexpressions e and a, let E be a type such that `decltype((e))` is E, and let A be a type such that `decltype((a))` is A. The expression `execution::make_with_allocator(e, a)` is expression-equivalent to:

- `tbd::tag_invoke(make_with_allocator, e, a)`, if that expression is valid.
- Otherwise, `execution::make_with_allocator(e, a)` is ill-formed.

Returns: A value t1 of type T1 that holds a copy of t. T1 provides an overload of `tag_invoke` such that `make_with_allocator(t1, a)` returns a copy of t1, an overload of `tag_invoke` such that `get_allocator(t1)` returns a, and functions derived from T1 shall use a when allocating.

3.22 Modify section 2.5.2 Class static_thread_pool

* Class `static_thread_pool`

- the number of existing executor objects associated with the `static_thread_pool` for which the `execution::outstanding_work.tracked` property `execution::tracked_outstanding_work` is established;

3.23 Modify section 2.5.2.4 Scheduler creation

```cpp
scheduler_type scheduler() noexcept;
```

Returns: A scheduler that may be used to create sender objects that may be used to submit receiver objects to the thread pool. The returned scheduler has the following properties already established supports the following `tag_invoke` functions:

- `execution::allocator`
- `execution::allocator(std::allocator<void>())`
- `tbd::get_allocator() -> see-below`

3.24 Modify section 2.5.2.5 Executor creation

```cpp
executor_type executor() noexcept;
```

Returns: An executor that may be used to submit function objects to the thread pool. The returned executor has the following properties already established supports the following `tag_invoke` functions:
3.25 Modify section 2.5.3 static_thread_pool scheduler types

All scheduler types accessible through static_thread_pool::scheduler(), and subsequent invocations of the member function require, conform to the following specification.

```cpp
class C
{
public:

    // types:
    using sender_type = see-below;

    // construct / copy / destroy:
    C(const C& other) noexcept;
    C(C&& other) noexcept;
    C& operator=(const C& other) noexcept;
    C& operator=(C&& other) noexcept;

    // scheduler operations:
    - see-below require(const execution::allocator_t<void>& a) const;
    - template<class ProtoAllocator>
      see-below require(const execution::allocator_t<ProtoAllocator>& a) const;
    - see-below query(execution::context_t) const noexcept;
    - see-below query(execution::allocator_t<void>) const noexcept;
    - template<class ProtoAllocator>
      see-below query(execution::allocator_t<ProtoAllocator>) const noexcept;
      + template<class ProtoAllocator>
        friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);
        + friend see-below tag_invoke(execution::get_context_t, const C&) noexcept;
        + friend see-below tag_invoke(execution::get_allocator_t, const C&) noexcept;

    bool running_in_this_thread() const noexcept;
};

bool operator==(const C& a, const C& b) noexcept;
bool operator!=(const C& a, const C& b) noexcept;
```
3.26 Modify section 2.5.3.3 Operations

```cpp
[[see-below]]
require(const execution::allocator_t<void>& a) const;

Returns: `require(execution::allocator(x))`, where `x` is an implementation-defined default allocator.
```

```cpp
template<class ProtoAllocator>
- see-below
require(const execution::allocator_t<ProtoAllocator>& a) const;
+ friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator a);

Returns: An scheduler object of an unspecified type conforming to these specifications, associated with the same thread pool as `*this`, with the `execution::allocator_t<ProtoAllocator>` property established. Function `execution::get_allocator() -> ProtoAllocator`, such that allocation and deallocation associated with function submission will be performed using a copy of `a.alloc`. All other `tag_invoke` functions of the returned scheduler object are identical to those of `*this`.
```

```cpp
- static_thread_pool& query(execution::context_t) const noexcept;
- see-below
query(execution::allocator_t<void>) const noexcept;
- see-below
query(execution::allocator_t<ProtoAllocator>) const noexcept;
+ friend see-below tag_invoke(execution::get_allocator_t, const C&) noexcept;

Returns: A reference to the associated `static_thread_pool` object.
```

```cpp
Returns: The allocator object associated with the executor, with type and value as either previously established by the `execution::allocator_t<ProtoAllocator>` property specified in a previous call to `execution::make_with_allocator` or the implementation defined default allocator established by the `execution::allocator_t<void>` property.
```

3.27 Modify section 2.5.3.4 Comparisons

```cpp
bool operator==(const C& a, const C& b) noexcept;
```

Returns: `true` if `&a.query(execution::context) == &b.query(execution::context)` and `a` and `b` have identical properties, otherwise `false`.

Returns: `true` if

- `execution::get_context(a) == execution::get_context(b)` and,
- `execution::get_allocator(a) == execution::get_allocator(b)`,
- otherwise `false`.

3.28 Modify section 2.5.3.6 Sender creation

```cpp
sender_type schedule() noexcept;
```

Returns: A sender that may be used to submit function objects to the thread pool. The returned sender supports the following `tag_invoke` functions:

- `execution::blocking.possibly`
- `execution::relationship.fork`
- `execution::outstanding_work.untracked`
- `execution::allocator`
- `execution::allocator(std::allocator<void>())`
--- execution::get_blocking() -> execution::possibly_blocking_t
--- execution::get_relationship() -> execution::fork_relationship_t
--- execution::get_outstanding_work() -> execution::untracked_outstanding_work_t
--- tbd::get_allocator() -> see-below

### 3.29 Modify section 2.5.4 static_thread_pool sender types

All sender types accessible through `static_thread_pool::scheduler().schedule()`, and subsequent invocations of the member function `require`, conform to the following specification.

```cpp
class C {
    public:
        // construct / copy / destroy:
        C(const C& other) noexcept;
        C(C&& other) noexcept;
        C& operator=(const C& other) noexcept;
        C& operator=(C&& other) noexcept;
        // sender operations:
        - see-below require(execution::blocking_t::never_t) const;
        - see-below require(execution::blocking_t::possibly_t) const;
        - see-below require(execution::blocking_t::always_t) const;
        - see-below require(execution::relationship_t::continuation_t) const;
        - see-below require(execution::relationship_t::fork_t) const;
        - see-below require(execution::outstanding_work_t::tracked_t) const;
        - see-below require(execution::outstanding_work_t::untracked_t) const;
        - see-below require(const execution::allocator_t<void>& a) const;
        - template<class ProtoAllocator>
        - see-below require(const execution::allocator_t<ProtoAllocator>& a) const;
        - static constexpr execution::bulk_guarantee_t query(execution::bulk_guarantee_t) const;
        - static constexpr execution::mapping_t query(execution::mapping_t) const;
        - execution::blocking_t query(execution::blocking_t) const;
        - execution::relationship_t query(execution::relationship_t) const;
        - execution::outstanding_work_t query(execution::outstanding_work_t) const;
        - see-below query(execution::context_t) const noexcept;
        - see-below query(execution::allocator_t<void>) const noexcept;
        - template<class ProtoAllocator>
        - see-below query(execution::allocator_t<ProtoAllocator>& a) const noexcept;
        + friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::never_blocking_t);
        + friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::possibly_blocking_t);
        + friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::always_blocking_t);
        + friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::continuation_relationship_t);
        + friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::fork_relationship_t);
        + friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::tracked_outstanding_work_t);
        + friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::untracked_outstanding_work_t);
        + template<class ProtoAllocator>
        + friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);
        +
```
friend constexpr execution::thread_mapping_t tag_invoke(execution::get_mapping_t, tbd::any_instance_of<C>) noexcept;
friend see-below tag_invoke(execution::get_blocking_t, const C&) noexcept;
friend see-below tag_invoke(execution::get_relationship_t, const C&) noexcept;
friend see-below tag_invoke(execution::get_outstanding_work_t, const C&) noexcept;
friend see-below tag_invoke(execution::get_context_t, const C&) noexcept;
friend see-below tag_invoke(execution::get_allocator_t, const C&) noexcept;

bool running_in_this_thread() const noexcept;

bool operator==(const C& a, const C& b) noexcept;
bool operator!=(const C& a, const C& b) noexcept;

3.30 Modify section 2.5.4.3 Operations

- see-below require(execution::blocking_t::never_t) const;
- see-below require(execution::blocking_t::possibly_t) const;
- see-below require(execution::blocking_t::always_t) const;
- see-below require(execution::relationship_t::continuation_t) const;
- see-below require(execution::relationship_t::fork_t) const;
- see-below require(execution::outstanding_work_t::tracked_t) const;
- see-below require(execution::outstanding_work_t::untracked_t) const;
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::never_blocking_t);
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::possibly_blocking_t);
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::always_blocking_t);
+ friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::continuation_relationship_t);
+ friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::fork_relationship_t);
+ friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::tracked_outstanding_work_t);
+ friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::untracked_outstanding_work_t);
+ friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);

Returns: An sender object of an unspecified type conforming to these specifications, associated with the same thread pool as this, and having the requested property established behaviour. When the requested property is part of a group that is defined as a mutually exclusive set, any other properties in the group are removed from the returned sender object. All other properties of the returned sender object, tag_invoke functions on the returned sender object, which are not modified by the requested behaviour, are identical to those of this.

see-below require(const execution::allocator_t<void>& a) const;

Returns: require(execution::allocator(x), where x is an implementation-defined default allocator.

template<class ProtoAllocator>
- see-below require(const execution::allocator_t<ProtoAllocator>& a) const;
+ friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);

Returns: An sender object of an unspecified type conforming to these specifications, associated with the same thread pool as this, with the execution::allocator_t<ProtoAllocator> property established function execution::get_allocator() -> ProtoAllocator, such that allocation and deallocation associated with function submission will be performed using a copy of a.alloc. All other properties tag_invoke functions of the returned scheduler object are identical to those of this.

static constexpr execution::bulk_guarantee_t query(execution::bulk_guarantee_t) const;

Returns: execution::bulk_guarantee.parallel
Returns: execution::mapping.thread

-Returns: The type and value of the given property behaviour of *this.

-Returns: A reference to the associated static_thread_pool object.

3.31 Modify section 2.5.4.4 Comparisons

bool operator==(const C& a, const C& b) noexcept;

Returns: true if &a.query(execution::context) == &b.query(execution::context) and a and b have identical properties, otherwise false.

Returns: true if

— a.query(execution::bulk_guarantee) == b.query(execution::bulk_guarantee) and,
— execution::get_mapping(a) == execution::get_mapping(b) and,
— execution::get_blocking(a) == execution::get_blocking(b) and,
— execution::get_relationship(a) == execution::get_relationship(b) and,
— execution::get_outstanding_work(a) == execution::get_outstanding_work(b) and,
— &execution::get_context(a) == &execution::get_context(b) and,
— execution::get_allocator(a) == execution::get_allocator(b),
— otherwise false.

3.32 Modify section 2.5.5 static_thread_pool executor types

All executor types accessible through static_thread_pool::executor(), and subsequent invocations of the member function require, conform to the following specification.
C(const C& other) noexcept;
C(C&& other) noexcept;

C& operator=(const C& other) noexcept;
C& operator=(C&& other) noexcept;

// sender operations:
- see-below require(execution::blocking_t::never_t) const;
- see-below require(execution::blocking_t::possibly_t) const;
- see-below require(execution::blocking_t::always_t) const;
- see-below require(execution::relationship_t::continuation_t) const;
- see-below require(execution::relationship_t::fork_t) const;
- see-below require(execution::outstanding_work_t::tracked_t) const;
- see-below require(execution::outstanding_work_t::untracked_t) const;
- see-below require(const execution::allocator_t<void>& a) const;
- template<class ProtoAllocator>
  - see-below require(const execution::allocator_t<ProtoAllocator>& a) const;
- static constexpr execution::bulk_guarantee_t query(execution::bulk_guarantee_t) const;
- static constexpr execution::mapping_t query(execution::mapping_t) const;
- execution::blocking_t query(execution::blocking_t) const;
- execution::relationship_t query(execution::relationship_t) const;
- execution::outstanding_work_t query(execution::outstanding_work_t) const;
- see-below query(execution::context_t) const noexcept;
- see-below query(execution::allocator_t<void>) const noexcept;
- template<class ProtoAllocator>
  - see-below query(execution::allocator_t<ProtoAllocator>) const noexcept;
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::never_blocking_t);
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::possibly_blocking_t);
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::always_blocking_t);
+ friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::continuation_relationship_t);
+ friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::fork_relationship_t);
+ friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::tracked_outstanding_work_t);
+ friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::untracked_outstanding_work_t);
+ template<class ProtoAllocator>
  - friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);
+ friend constexpr execution::thread_mapping_t tag_invoke(execution::get_mapping_t, tbd::any_instance_of<C>) noexcept;
+ friend see-below tag_invoke(execution::get_blocking_t, const C&) noexcept;
+ friend see-below tag_invoke(execution::get_relationship_t, const C&) noexcept;
+ friend see-below tag_invoke(execution::get_outstanding_work_t, const C&) noexcept;
+ friend see-below tag_invoke(execution::get_context_t, const C&) noexcept;
+ friend see-below tag_invoke(execution::get_allocator_t, const C&) noexcept;

  bool running_in_this_thread() const noexcept;
};

bool operator==(const C& a, const C& b) noexcept;
bool operator!=(const C& a, const C& b) noexcept;
3.33 Modify section 2.5.5.3 Operations

- see-below require(execution::blocking_t::never_t) const;
- see-below require(execution::blocking_t::possibly_t) const;
- see-below require(execution::blocking_t::always_t) const;
- see-below require(execution::relationship_t::continuation_t) const;
- see-below require(execution::relationship_t::fork_t) const;
- see-below require(execution::outstanding_work_t::tracked_t) const;
- see-below require(execution::outstanding_work_t::untracked_t) const;

+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::never_blocking_t);
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::possibly_blocking_t);
+ friend see-below tag_invoke(execution::make_with_blocking_t, const C&, execution::always_blocking_t);
+ friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::continuation_relationship_t);
+ friend see-below tag_invoke(execution::make_with_relationship_t, const C&, execution::fork_relationship_t);
+ friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::tracked_outstanding_work_t);
+ friend see-below tag_invoke(execution::make_with_outstanding_work_t, const C&, execution::untracked_outstanding_work_t);
+ template<class ProtoAllocator>
  + friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);

Returns: An executor object of an unspecified type conforming to these specifications, associated with the same thread pool as *this, and having the requested property established behaviour. When the requested property is part of a group that is defined as a mutually exclusive set, any other properties in the group are removed from the returned executor object. All other properties of the returned executor object tag_invoke functions on the returned executor object, which are not modified by the requested behaviour, are identical to those of *this.

see-below require(const execution::allocator_t<void>& a) const;

Returns: require(execution::allocator(x)), where x is an implementation-defined default allocator.

template<class ProtoAllocator>
  - see-below require(const execution::allocator_t<ProtoAllocator>& a) const;
  + friend see-below tag_invoke(execution::make_with_allocator_t, const C&, ProtoAllocator);

Returns: An executor object of an unspecified type conforming to these specifications, associated with the same thread pool as *this, with the execution::allocator_t<ProtoAllocator> property established function execution::get_allocator() -> ProtoAllocator, such that allocation and deallocation associated with function submission will be performed using a copy of a.alloc. All other properties tag_invoke functions of the returned executor object are identical to those of *this.

static constexpr execution::bulk_guarantee_t query(execution::bulk_guarantee_t) const;

Returns: execution::bulk_guarantee.parallel

- static constexpr execution::mapping_t query(execution::mapping_t) const;
  + friend constexpr execution::thread_mapping_t tag_invoke(execution::get_mapping_t, tbd::any_instance_of<C>) noexcept;

Returns: execution::mapping.thread execution::thread_mapping_t.

- execution::blocking_t query(execution::blocking_t) const;
- execution::relationship_t query(execution::relationship_t) const;
- execution::outstanding_work_t query(execution::outstanding_work_t) const;
  + friend see-below tag_invoke(execution::get_blocking_t, const C&) noexcept;
  + friend see-below tag_invoke(execution::get_relationship_t, const C&) noexcept;
  + friend see-below tag_invoke(execution::get_outstanding_work_t, const C&) noexcept;

Returns: The type and value of the given property behaviour of *this.
-static_thread_pool& query(execution::context_t) const noexcept;
+friend static_thread_pool& tag_invoke(execution::get_context_t, const C&) noexcept;

Returns: A reference to the associated static_thread_pool object.

-see-below query(execution::allocator_t<void>) const noexcept;
-see-below query(execution::allocator_t<ProtoAllocator>) const noexcept;
+friend see-below tag_invoke(execution::get_allocator_t, const C&) noexcept;

Returns: The allocator object associated with the executor, with type and value as either previously established by the execution::allocator_t<ProtoAllocator> property specified in a previous call to execution::make_with_allocator or the implementation defined default allocator established by the execution::allocator_t<void> property.

3.34 Modify section 2.5.5.4 Comparisons

bool operator==(const C& a, const C& b) noexcept;

Returns: true if a.query(execution::context) == b.query(execution::context) and a and b have identical properties, otherwise false.

Returns: true if

- a.query(execution::bulk_guarantee) == b.query(execution::bulk_guarantee) and,
- execution::get_mapping(a) == execution::get_mapping(b) and,
- execution::get_blocking(a) == execution::get_blocking(b) and,
- execution::get_relationship(a) == execution::get_relationship(b) and,
- execution::get_outstanding_work(a) == execution::get_outstanding_work(b) and,
- &execution::get_context(a) == &execution::get_context(b) and,
- execution::get_allocator(a) == execution::get_allocator(b),
- otherwise false.

3.35 Modify section 1.2 Usage Example

-execute(std::require(ex, blocking.always), foo);
+execute(
    execution::make_with_blocking(
        ex, execution::always_blocking),
    foo);

3.36 Modify section 1.3 Executors Execute Work

Executors avoid this unscalable situation by defining cpos in terms of tag_invoke from [P1895R0] adopting [P1393R0]'s properties design based on require and prefer:

// require the execute operation to block
-exector auto blocking_ex = std::require(ex, execution::blocking.always);
+executor auto blocking_ex = execution::make_with_blocking(
+    ex, execution::always_blocking);

// prefer to execute with a particular priority p
-exector auto blocking_ex_with_priority = std::prefer(blocking_ex, execution::priority(p));

32
3.37 Modify section 2.1.2 Header synopsis

```cpp
+ executor auto blocking_ex_with_priority = tbd::prefer(
+    execution::make_with_blocking, blocking_ex, execution::always_blocking);
```

```cpp
// execute the nonblocking work on the given executor
- execute(require(ex, blocking.never), move(work));
+ execute(make_with_blocking(ex, never_blocking), move(work));
```

```cpp
3.37 Modify section 2.1.2 Header synopsis

- // Associated execution context property:
+ // Associated execution context functions:

- struct context_t;
- constexpr context_t context;
+ inline constexpr get_context_t get_context = get_context_t{};

- // Blocking properties:
+ // Blocking behaviour:

- struct blocking_t;
- constexpr blocking_t blocking;
+ // Blocking values
+ inline constexpr always_blocking_t always_blocking = always_blocking_t{};
+ inline constexpr never_blocking_t never_blocking = never_blocking_t{};
+ inline constexpr possibly_blocking_t possibly_blocking = possibly_blocking_t{};
+ // Blocking functions
+ inline constexpr make_with_context_t make_with_context = make_with_context_t{};

// Properties to allow adaptation of blocking and directionality:

struct blocking_adaptation_t;
constexpr blocking_adaptation_t blocking_adaptation;

- // Properties to indicate if submitted tasks represent continuations:
+ // Behaviour that indicates if submitted tasks represent continuations:

- struct relationship_t;
- constexpr relationship_t relationship;
+ // Relationship values
```
inline constexpr fork_relationship_t fork_relationship = fork_relationship_t{};
inline constexpr continuation_relationship_t continuation_relationship = continuation_relationship_t{};

// Relationship functions
inline constexpr get_relationship_t get_relationship = get_relationship_t{};
inline constexpr make_with_relationship_t make_with_relationship = make_with_relationship_t{};

// Properties to indicate likely task submission in the future:
// Behaviour that indicates likely task submission in the future:

// OutstandingWork_t
constexpr outstanding_work_t outstanding_work;

// OutstandingWork values
inline constexpr tracked_outstanding_work_t tracked_outstanding_work = tracked_outstanding_work_t{};
inline constexpr untracked_outstanding_work_t untracked_outstanding_work = untracked_outstanding_work_t{};

// OutstandingWork functions
inline constexpr get_outstanding_work_t get_outstanding_work = get_outstanding_work_t{};
inline constexpr make_with_outstanding_work_t make_with_outstanding_work = make_with_outstanding_work_t{};

// Properties for bulk execution guarantees:
struct bulk_guarantee_t;
constexpr bulk_guarantee_t bulk_guarantee;

// Properties for mapping of execution on to threads:
// Behaviour for mapping of execution on to threads:

// Mapping_t
constexpr mapping_t mapping;

// Mapping values
inline constexpr thread_mapping_t thread_mapping = thread_mapping_t{};
inline constexpr new_thread_mapping_t new_thread_mapping = new_thread_mapping_t{};
inline constexpr other_mapping_t other_mapping = other_mapping_t{};

// Mapping functions
inline constexpr get_mapping_t get_mapping = get_mapping_t{};
inline constexpr make_with_mapping_t make_with_mapping = make_with_mapping_t{};
### References

[https://wg21.link/p0443r13](https://wg21.link/p0443r13)

[https://wg21.link/p0737r0](https://wg21.link/p0737r0)

[https://wg21.link/p1393r0](https://wg21.link/p1393r0)

[https://wg21.link/p1895r0](https://wg21.link/p1895r0)