1 Requirements and Concepts

Throughout this clause, the names of template parameters are used to express type requirements, and the concepts are designed to support type checking at compile time. In order to make the concepts more concise, some constraints related to the **`Ranges TS`** are not listed, such as the concept template **`CopyConstructible`** and the concept template **`MoveConstructible`**.

1.1 Binary Semaphores

1.1.1 Intention

This concept is an abstraction for the Ad-hoc synchronizations required in the “Sync Concurrent Invoke” model. Typical implementations may have one or more of the following mechanisms:

- simply use “std::promise<void>” to implement, as mentioned earlier, or
- use the “Spinlock” if executions are likely to be blocked for only short periods, or
- use the Mutexes together with the Condition Variables to implement, or
- use the primitives supported by specific platforms, such as the “Futex” in modern Linux, the “Semaphore” defined in the POSIX standard and the “Event” in Windows, or
- have “work-stealing” strategy that may execute other unrelated tasks while waiting.

1.1.2 BinarySemaphore requirements

A type **`BS`** meets the **`BinarySemaphore`** requirements if the following expressions are well-formed and have the specified semantics (**`bs`** denotes a value of type **`BS`**).

`bs.wait()`

Effects: Blocks the calling thread until the permit is released.
Return type: **`void`**
Synchronization: Prior **`release()`** operations shall synchronize with this operation.

`bs.release()`

Effects: Release the permit.
Return type: **`void`**
Synchronization: This operation synchronizes with subsequent **`wait()`** operations.

1.1.3 Concept template BinarySemaphore

```
namespace requirements {
```

```
template <class T>
concept bool BinarySemaphore() {
  return requires(T semaphore) {
    { semaphore.wait() };
    { semaphore.release() };
  };
}

}
```

1.2 Atomic Counters

1.2.1 Intention

This concept is an abstraction for the "many-to-one" synchronizations required for the execution structures. Typical implementations may have one or more of the following mechanisms:

- use an integer to maintain the count and use a mutex to prevent concurrent reading or writing, or
- manage an atomic integer maintaining the count with lock-free operations, or
- adopt the "Tree Atomic Counter" strategy, as mentioned earlier.

In order to implement it with the C++ programming language, the requirements for the "Atomic Counter" is divided into 3 parts: the **`LinearBuffer`** requirements, the **`AtomicCounterModifier`** requirements and the **`AtomicCounterInitializer`** requirements, which illustrates the requirements for the return types, for the modifications and for the initializations, respectively.

1.2.2 Requirements

1.2.2.1 LinearBuffer requirements

A type **`LB`** meets the **`LinearBuffer`** requirements if the following expressions are well-formed and have the specified semantics (**`lb`** denotes a value of type **`LB`**).

`lb.fetch()`

Requires: The number of times that this function has been invoked shall be less than the predetermined.

Effects: Acquires an entity.

Return type: undefined

Returns: The acquired entity

1.2.2.2 AtomicCounterModifier requirements

A type `ACM` meets the `AtomicCounterModifier` requirements if the following expressions are well-formed and have the specified semantics (`acm` denotes a value of type `ACM`).

`acm.increase(s)`

Requires: `s` shall be convertible to type `std::size_t`.
Effect: Increase the Atomic Counter entity corresponding to `acm` by `s`.
Return type: Any type that meets the `LinearBuffer` requirements
Returns: A value whose type meets the `LinearBuffer` requirements, each of the first `(s + 1)` times of `fetch()` operations to which shall acquire a value whose type meets the `AtomicCounterModifier` requirements, and that corresponds to the Atomic Counter entity as `acm` does.
Post condition: `acm` no longer corresponds to an Atomic Counter entity.

`acm.decrement()`

Effect: If the state of the Atomic Counter entity corresponding to `acm` is positive, decrease the state of the entity by one.
Return type: `bool`
Returns: `true` if the state of the entity is positive before the operation, `false` otherwise.
Post condition: `acm` no longer corresponds to an Atomic Counter entity.
Synchronization: If this operation returns true, it synchronizes with subsequent `decrement()` operations that return `false` on any entity meets the `AtomicCounterModifier` requirements and that corresponds to the same Atomic Counter entity as `acm` does; otherwise, prior `decrement()` operations that return `true` on any entity whose type meets the `AtomicCounterModifier` requirements, and that corresponds to the same Atomic Counter entity as `acm` does shall synchronize with this operation.

1.2.2.3 AtomicCounterInitializer requirements

A type `ACI` meets the `AtomicCounterInitializer` requirements if the following expressions are well-formed and have the specified semantics (`aci` denotes a value of type `ACI`).

`aci(s)`

Requires: `s` shall be convertible to type `std::size_t`.
Effect: Initialize an Atomic Counter entity whose initial count shall be equals to s.
Return type: Any type that meets the `LinearBuffer` requirements
Returns: A value whose type meets the `LinearBuffer` requirements, each of the first `(s + 1)` times of `fetch()` operations to which shall acquire a value whose type meets the `AtomicCounterModifier` requirements, and corresponds to the initialized Atomic Counter entity.

1.2.3 Concepts

1.2.3.1 Concept template LinearBuffer

```
namespace requirements {

template <class T, class U>
concept bool LinearBuffer() {
  return requires(T buffer) {
    { buffer.fetch() } -> U;
  };
}

}
```

1.2.3.2 Concept template AtomicCounterModifier

```
namespace requirements {

template <class T>
concept bool AtomicCounterModifier() {
  return requires(T modifier) {
    { modifier.decrement() } -> bool;
  } && (requires(T modifier) {
    { modifier.increase(0u) } -> LinearBuffer<T>;
  } || requires(T modifier) {
    { modifier.increase(0u).fetch() } -> AtomicCounterModifier;
  });
}

}
```

1.2.3.3 Concept template AtomicCounterInitializer

```
namespace requirements {

template <class T>
concept bool AtomicCounterInitializer() {
  return requires(T initializer) {
    { initializer(0u).fetch() } -> AtomicCounterModifier;
  };
```

```
}

}
```

1.3 Runnable and Callable Types

The **`Callable`** types are defined in the C++ programming language with specified input types and return type. The **`Runnable`** types are those **`Callable`** types which have no input and unspecified return type. The **`Callable`** types are required to be **`CopyConstructible`**, but the **`Runnable`** types are only required to be **`MoveConstructible`**.

1.3.1 Concept template Runnable

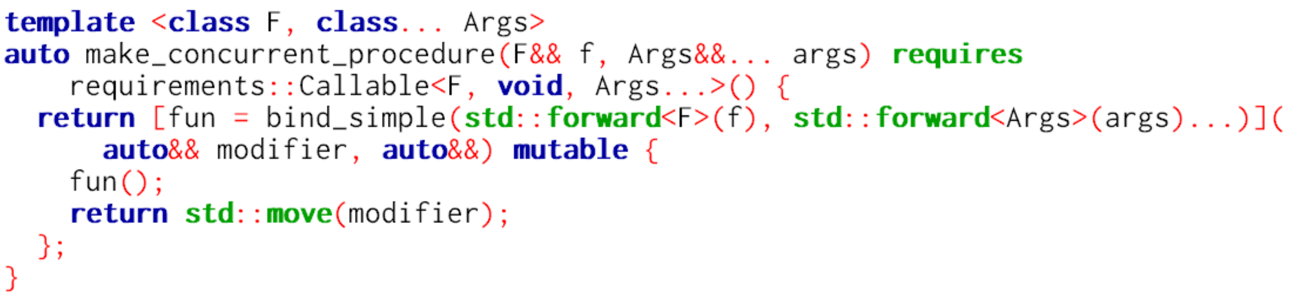
```
template <class F>
concept bool Runnable() {
  return requires(F f) {
    { f() };
  };
}
```

1.3.2 Concept template Callable

```
template <class F, class R, class... Args>
concept bool Callable() {
  return requires(F f, Args&&... args) {
    { f(std::forward<Args>(args)...) } -> R;
  };
}
```

1.4 Concurrent Procedures

1.4.1 Intention



```
template <class F, class... Args>
auto make_concurrent_procedure(F&& f, Args&&... args) requires
    requirements::Callable<F, void, Args...>() {
  return [fun = bind_simple(std::forward<F>(f), std::forward<Args>(args)...)](
      auto&& modifier, auto&&) mutable {
    fun();
    return std::move(modifier);
  };
}
```

Figure 1

```
auto proc = [](auto&& modifier, auto&& callback) {
  do_something();
  modifier = concurrent_fork(std::move(modifier),
                             callback,
                             /* Some Concurrent Callers */);
  do_something_else();
  modifier = concurrent_fork(std::move(modifier),
                             callback,
                             /* Some Concurrent Callers */);
  do_something_else();
  return std::move(modifier);
};
```

Figure 2

```
class ConcurrentProcedureTemplate {
 public:
  template <class Modifier, class Callback>
  auto operator()(Modifier&& modifier, Callback&& callback) {
    modifier_ = std::forward<Modifier>(modifier);
    callback_ = std::forward<Callback>(callback);
    this->run();
    return std::move(modifier_);
  }

 protected:
  template <class... ConcurrentInvokers>
  void fork(ConcurrentInvokers&&... invokers) {
    modifier_ = concurrent_fork(std::move(modifier_), callback_, invokers...);
  }

  virtual void run() = 0;

 private:
  abstraction::AtomicCounterModifier modifier_;
  abstraction::Callable<void()> callback_;
};
```

Figure 3

The "Concurrent Callable" is a Callable type defined in the C++ programming language. This concept is an abstraction for the smallest concurrent task fragment required in the execution structures. Typical implementations may have one or more of the following mechanisms:

- be wrapped from a Callable type (in other words, gives up the chance to call the function template **concurrent_fork**), as is shown in Figure 1 (*note that* ***std::bind(std::forward<F>(f)), std::forward<Args>(args)...)()*** *will perform* ***F(Args&...);*** *with the helper function template* ***bind_simple*** *the implementation will perform* ***F(Args&&...)***), or
- be implemented manually, and may call the function template **concurrent_fork**, as is shown in Figure 2, or
- be implemented with a "Template" with runtime abstraction by inheriting from an abstract class, as is shown in Figure 3 (*note that abstraction::AtomicCounterModifier and abstraction::Callable are wrappers for Atomic Counter Modifiers and Callables, respectively; their principles are the same as* ***std::function***).

1.4.2 ConcurrentProcedure requirements

A type **`CP`** meets the **`ConcurrentProcedure`** requirements if the following expressions are well-formed and have the specified semantics (**`cp`** denotes a value of type **`CP`**).

`cp(acm, c)`

Requires: The original types of **`acm`** and **`c`** shall meet the **`AtomicCounterModifier`** requirements and the **`Callable<void>`** requirements, respectively.

Effects: Execute the user-defined concurrent procedure synchronously.

Return type: Any type that meets the **`AtomicCounterModifier`** requirements

Note: It is allowed to invoke the function template **`concurrent_fork`** within this scope.

1.4.3 Concept template ConcurrentProcedure

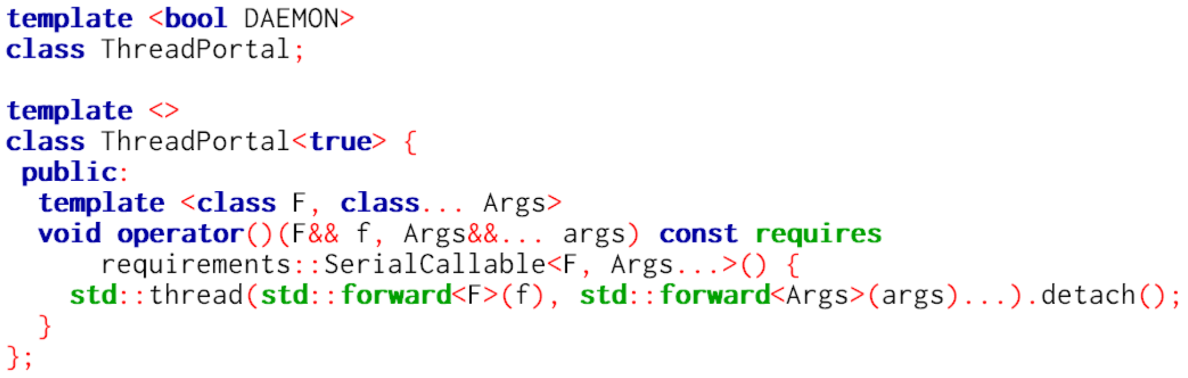
```
namespace requirements {

template <class T, class U, class V>
concept bool ConcurrentProcedure() {
  return requires(T procedure, U&& modifier, V&& callback) {
    { procedure(std::forward<U>(modifier), std::forward<V>(callback)) }
        -> AtomicCounterModifier;
  };
}

}
```

1.5Execution Agent Portals

1.5.1 Intention



```
template <bool DAEMON>
class ThreadPortal;

template <>
class ThreadPortal<true> {
 public:
  template <class F, class... Args>
  void operator()(F&& f, Args&&... args) const requires
      requirements::SerialCallable<F, Args...>() {
    std::thread(std::forward<F>(f), std::forward<Args>(args)...).detach();
  }
};
```

Figure 4

```
template <>
class ThreadPortal<false> {
 public:
  template <class F, class... Args>
  void operator()(F&& f, Args&&... args) const requires
      requirements::SerialCallable<F, Args...>() {
    ThreadManager::instance().emplace(
        std::thread(std::forward<F>(f), std::forward<Args>(args)...));
  }
};
```

Figure 5

Large-scale concurrent programming usually requires load balancing for every part of the program. Although there are many libraries provide us with quite a few APIs for concurrent algorithms, they are usually harmful in load balancing, especially when there are other works to be done that attach to higher priorities.

Currently in C++, we have the term “Execution Agent”, which is *“an entity such as a thread that may perform work in parallel with other execution agents”*. An “Execution Agent Portal” is an abstraction for the method required for the execution structures, that to submit callable units to concrete Execution Agents. Typical implementations may have one or more of the following mechanisms:

- submit the input callable unit to the current Execution Agent and sequentially execute it, or
- submit the input callable unit to a new daemon thread (*not able to join it at all; the exit of all non-daemon threads may kill all daemon threads*), as is shown in Figure 4, or
- submit the input callable unit to a new non-daemon thread so that it can run even if the “main” function has exit, as is shown in Figure 5 (*note that the class ThreadManager is a singleton type that manages the thread objects*), or
- submit the input callable unit to some remote executor, or
- submit the input callable unit to a threadpool entity.

1.5.2 ExecutionAgentPortal requirements

A type **`EAP`** meets the **`ExecutionAgentPortal`** requirements if the following expressions are well-formed and have the specified semantics (**`eap`** denotes a value of type **`EAP`**).

`eap(f, args...)`

Requires: The original types of **`f`** and each parameter in **`args`** shall satisfy the **`MoveConstructible`** requirements. **`INVOKE (std::move(f), std::move(args)...)`** shall be a valid expression.

Effects: Submit the parameters to a concrete Execution Agent which executes **`INVOKE (std::move(f), std::move(args)...)`** asynchronously. Any return value from this invocation is ignored.

1.6 Concurrent Callables

1.6.1 Intention

```
template <class Portal = abstraction::ConcurrentCallablePortal,
          class ConcurrentProcedure = abstraction::ConcurrentProcedure>
class SinglePhaseConcurrentCallable {
 private:
  class Callable {
   public:
    explicit Callable(ConcurrentProcedure&& procedure)
        : procedure_(std::forward<ConcurrentProcedure>(procedure)) {}

    template <class AtomicCounterModifier, class SerialCallable>
    void operator()(AtomicCounterModifier&& modifier, SerialCallable&& callback) {
      concurrent_join(procedure_(std::move(modifier),
                                 copy_construct(callback)), callback);
    }

   private:
    ConcurrentProcedure procedure_;
  };

 public:
  template <class T, class U>
  explicit SinglePhaseConcurrentCallable(T&& portal, U&& procedure)
      : portal_(std::forward<T>(portal)),
        callable_(std::forward<U>(procedure)) {}

  template <class AtomicCounterModifier, class SerialCallable>
  void operator()(AtomicCounterModifier&& modifier,
                  SerialCallable&& callback) requires
      requirements::ConcurrentProcedure<
          ConcurrentProcedure, AtomicCounterModifier, SerialCallable>() &&
      requirements::SerialCallable<
          Portal, Callable, AtomicCounterModifier, SerialCallable>() {
    portal_(std::move(callable_),
            std::forward<AtomicCounterModifier>(modifier),
            std::forward<SerialCallable>(callback));
  }

 private:
  Portal portal_;
  Callable callable_;
};
```

Figure 6

This concept is an abstraction for async tasks required for the execution structures. Typical implementations may have one or more of the following mechanisms:

- combine an Execution Agent Portal entity and a Concurrent Procedure entity, repack the Concurrent Procedure entity into another callable unit that will execute the function template concurrent_join as the Concurrent Procedure is executed, submit the callable unit with the Execution Agent Portal entity, as is shown in Figure 6.
- combine multiple Execution Agent Portal entities and their corresponding Concurrent Procedure entities, execute the Concurrent Procedure entities sequentially with different Execution Agent Portal entities.

1.6.2 ConcurrentCallable requirements

A type `CC` meets the **`ConcurrentCallable`** requirements if the following expressions are well-formed and have the specified semantics (**`cc`** denotes a value of type **`CC`**).

`cc(acm, c)`

Requires: The original types of **`acm`** and **`c`** shall meet the **`AtomicCounterModifier`** requirements and the **`Callable`** requirements, respectively.

Effects: Execute the user-defined concurrent callable unit asynchronously.

Return type: **`void`**

Note: It is allowed to invoke the function template **`concurrent_fork`** within this scope.

1.7 Concurrent Callers

1.7.1 Intention

This concept is an abstraction for task launching strategies required for the execution structures. Typical implementations may have one or more of the following mechanisms:

- abstract the tasks into one or multiple entities that meet the **`ConcurrentCallable`** requirements, or
- sequentially launch the tasks, or
- concurrently launch the tasks when there is a large number of them, or
- recursively split the large launching work into several small ones (optimally, 3) and execute them concurrently when adequate execution resources are provided, as mentioned earlier.

1.7.2 ConcurrentCaller requirements

A type **`CC`** meets the **`ConcurrentCaller`** requirements if the following expressions are well-formed and have the specified semantics (**`cc`** denotes a value of type **`CC`**).

`cc.size()`

Return type: **`std::size_t`**

Returns: The number of times that **`cc.call(lb, ccb)`** shall perform the **`lb.fetch()`** operation.

`cc.call(lb, c)`

Requires: The original types of **`lb`** and **`c`** shall meet the **`LinearBuffer`** requirements and the **`Callable<void>`** requirements, respectively; each of the first **`size()`** times of the **`lb.fetch()`** operation shall acquire a value whose type meets the **`AtomicCounterModifier`** requirements, and that corresponds to a same Atomic Counter entity.

Effects: Perform **`size()`** times of the **`lb.fetch()`** operation synchronously, and invoke **`size()`** times of the function template **`concurrent_join`** asynchronously.

Return type: **`void`**

1.7.3 Concept template ConcurrentCaller

```
namespace requirements {
```

```
template <class T, class U, class V>
concept bool ConcurrentCaller() {
  return requires(const T c_caller, T caller, U& buffer, const V& callback) {
    { c_caller.size() } -> size_t;
    { caller.call(buffer, callback) };
  };
}

template <class T, class U, class V>
constexpr bool concurrent_caller_all(T&, const U&, V&) {
  return ConcurrentCaller<V, T, U>();
}

template <class T, class U, class V, class... W>
constexpr bool concurrent_caller_all(T& buffer, const U& callback, V& caller, W&...
callers) {
  return concurrent_caller_all(buffer, callback, caller) &&
      concurrent_caller_all(buffer, callback, callers...);
}

// true if every Vi satisfies ConcurrentCaller<Vi, T, U>()
template <class T, class U, class... V>
concept bool ConcurrentCallerAll() {
  return requires(T& buffer, const U& callback, V&... callers) {
    requires concurrent_caller_all(buffer, callback, callers...);
  };
}

}
```

2 Function Templates

2.1 Function template async_concurrent_invoke

```
template <class Callback,
         class... ConcurrentCallers>
void async_concurrent_invoke(const Callback& callback,
                             ConcurrentCallers&&... callers) {
  async_concurrent_invoke_explicit(DefaultAtomicCounterInitializer(),
                                   callback,
                                   callers...);
```

```
}
```

Function template **`async_concurrent_invoke`** is a wrapper for function template **`async_concurrent_invoke_explicit`** with default “many-to-one” synchronization strategy.

2.2Function template async_concurrent_invoke_explicit

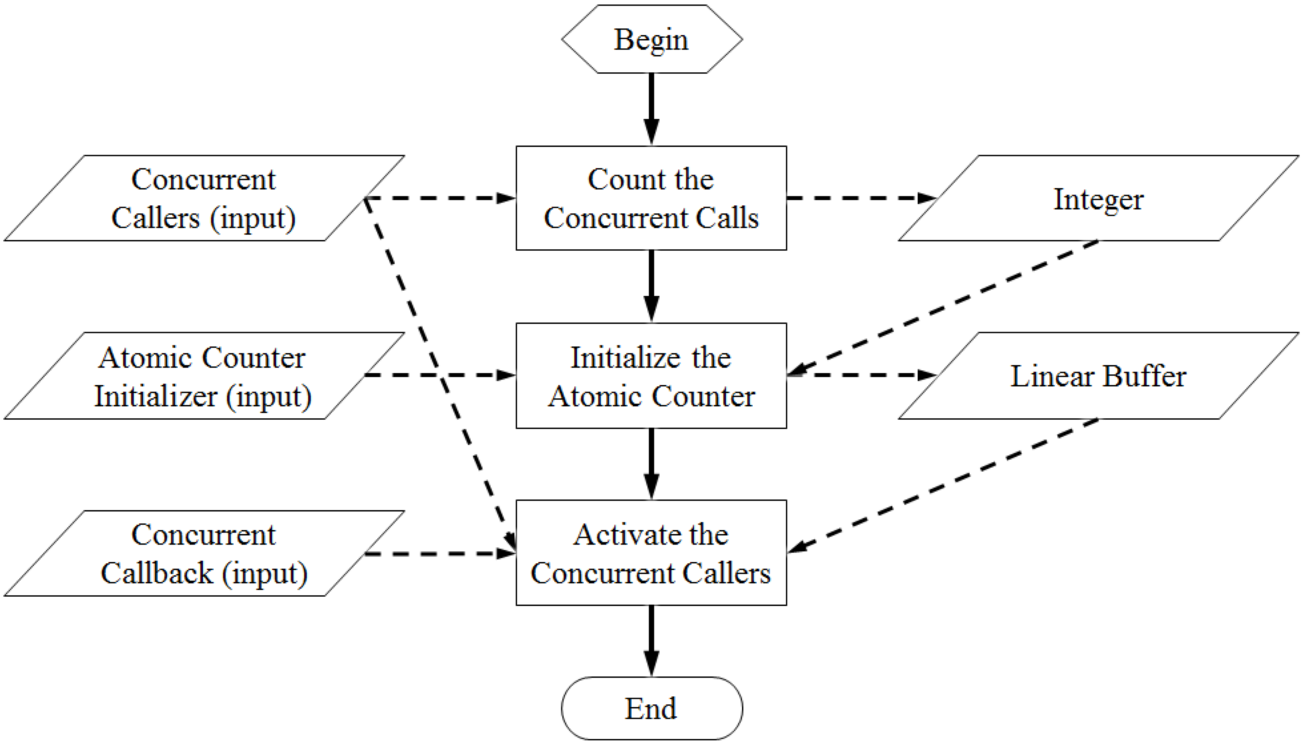


Figure 7

```
template <class AtomicCounterInitializer,
          class Callback,
          class... ConcurrentCallers>
void async_concurrent_invoke_explicit(AtomicCounterInitializer&& initializer,
                                      const Callback& callback,
                                      ConcurrentCallers&&... callers) requires
    requirements::AtomicCounterInitializer<AtomicCounterInitializer>() &&
    requirements::Callable<Callback, void>() &&
    requirements::ConcurrentCallerAll<
        decltype(initializer(0u)),
        Callback,
        ConcurrentCallers...>();
```

Requires: The types **`AtomicCounterInitializer`**, **`Callable`** and each type in **`ConcurrentCallers`** pack shall meet the **`AtomicCounterInitializer`** requirements, the **`Callable`** requirements and the **`ConcurrentCaller`** requirements, respectively.

Effects: Execute the “Async Concurrent Invoke” model, whose flow chart is shown in Figure 7.

Return type: **`void`**

2.3Function template sync_concurrent_invoke

```
template <class Runnable, class... ConcurrentCallers>
auto sync_concurrent_invoke(Runnable&& runnable,
                            ConcurrentCallers&&... callers) {
  return sync_concurrent_invoke_explicit(DefaultAtomicCounterInitializer(),
                                         DefaultBinarySemaphore(),
                                         runnable,
                                         callers...);
}
```

Function template **`sync_concurrent_invoke`** is a wrapper for function template **`sync_concurrent_invoke_explicit`** with default “many-to-one” synchronization and default blocking strategy.

2.4Function template sync_concurrent_invoke_explicit

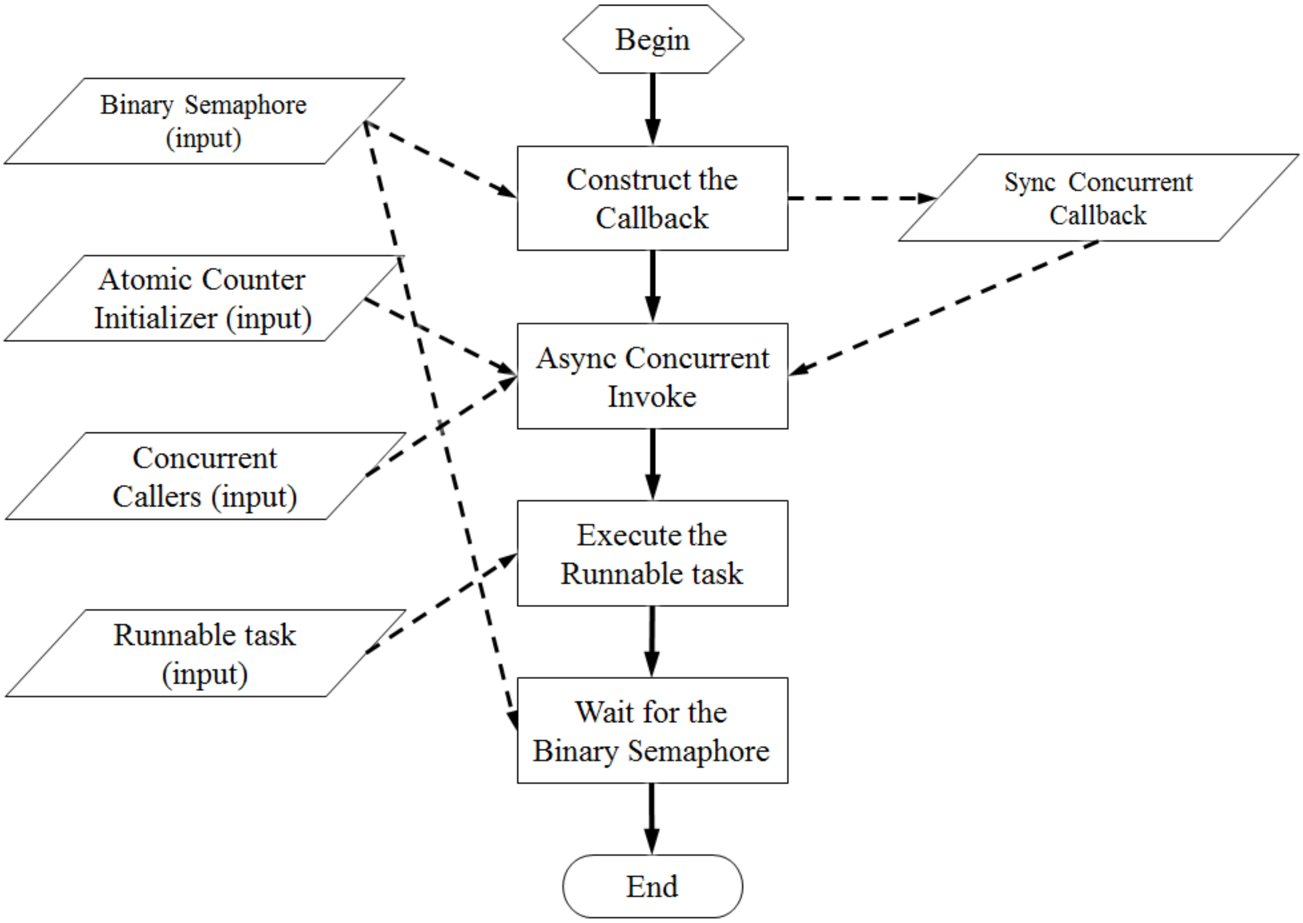


Figure 8

```
template <class BinarySemaphore>
class SyncInvokeHelper {
 public:
  explicit SyncInvokeHelper(BinarySemaphore& semaphore) : semaphore_(semaphore) {}

  ~SyncInvokeHelper() { semaphore_.wait(); }

 private:
  BinarySemaphore& semaphore_;
};

template <class AtomicCounterInitializer,
          class BinarySemaphore,
          class Runnable,
          class... ConcurrentCallers>
auto sync_concurrent_invoke_explicit(AtomicCounterInitializer&& initializer,
                                     BinarySemaphore&& semaphore,
                                     Runnable&& runnable,
                                     ConcurrentCallers&&... callers) requires
    requirements::AtomicCounterInitializer<AtomicCounterInitializer>() &&
    requirements::BinarySemaphore<BinarySemaphore>() &&
    requirements::Runnable<Runnable>() &&
    requirements::ConcurrentCallerAll<
        decltype(initializer(0u)),
        SyncConcurrentCallback<std::remove_reference_t<BinarySemaphore>>,
        ConcurrentCallers...>();
```

Requires: The types **`AtomicCounterInitializer`**, **`BinarySemaphore`**, **`SerialCallable`** and each type in **`ConcurrentCallers`** pack shall meet the **`AtomicCounterInitializer`** requirements, the **`BinarySemaphore`** requirements, the **`SerialCallable`** requirements and the **`ConcurrentCaller`** requirements, respectively.

Effects: Execute the “Sync Concurrent Invoke” model, whose flow chart is shown in Figure 8.

Return type: **`std::result_of_t<SerialCallable()>`**

Returns: anything that **`callable()`** returns

2.5 Function template concurrent_fork

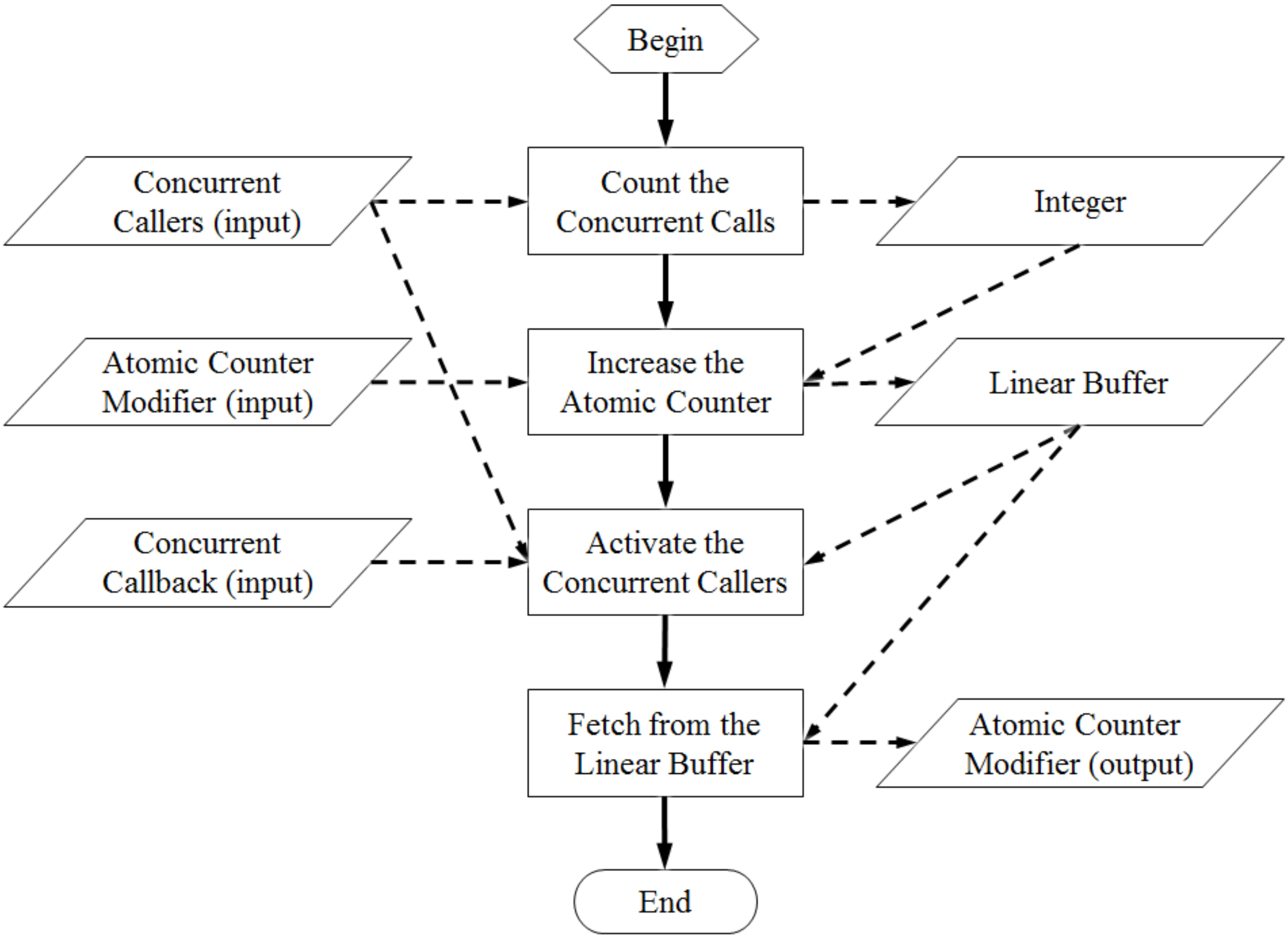


Figure 9

```
template <class AtomicCounterModifier,
          class Callback,
          class... ConcurrentCallers>
auto concurrent_fork(AtomicCounterModifier&& modifier,
                     const Callback& callback,
                     ConcurrentCallers&&... callers) requires
    requirements::AtomicCounterModifier<AtomicCounterModifier>() &&
    requirements::Callable<Callback, void>() &&
    requirements::ConcurrentCallerAll<
        decltype(modifier.increase(0u)),
        Callback,
        ConcurrentCallers...>();
```

Requires: The types **`AtomicCounterModifier`**, **`SerialCallable`** and each type in **`ConcurrentCallers`** pack shall meet the **`AtomicCounterModifier`** requirements, the **`SerialCallable`** requirements and the **`ConcurrentCaller`** requirements, respectively.

Effects: Execute the “Concurrent Fork” model, whose flow chart is shown in Figure 9.

Return type: **`decltype(modifier.increase(0u).fetch())`**

Returns: An Atomic Counter Modifier entity corresponds to an Atomic Counter entity.

2.6 Function template concurrent_join

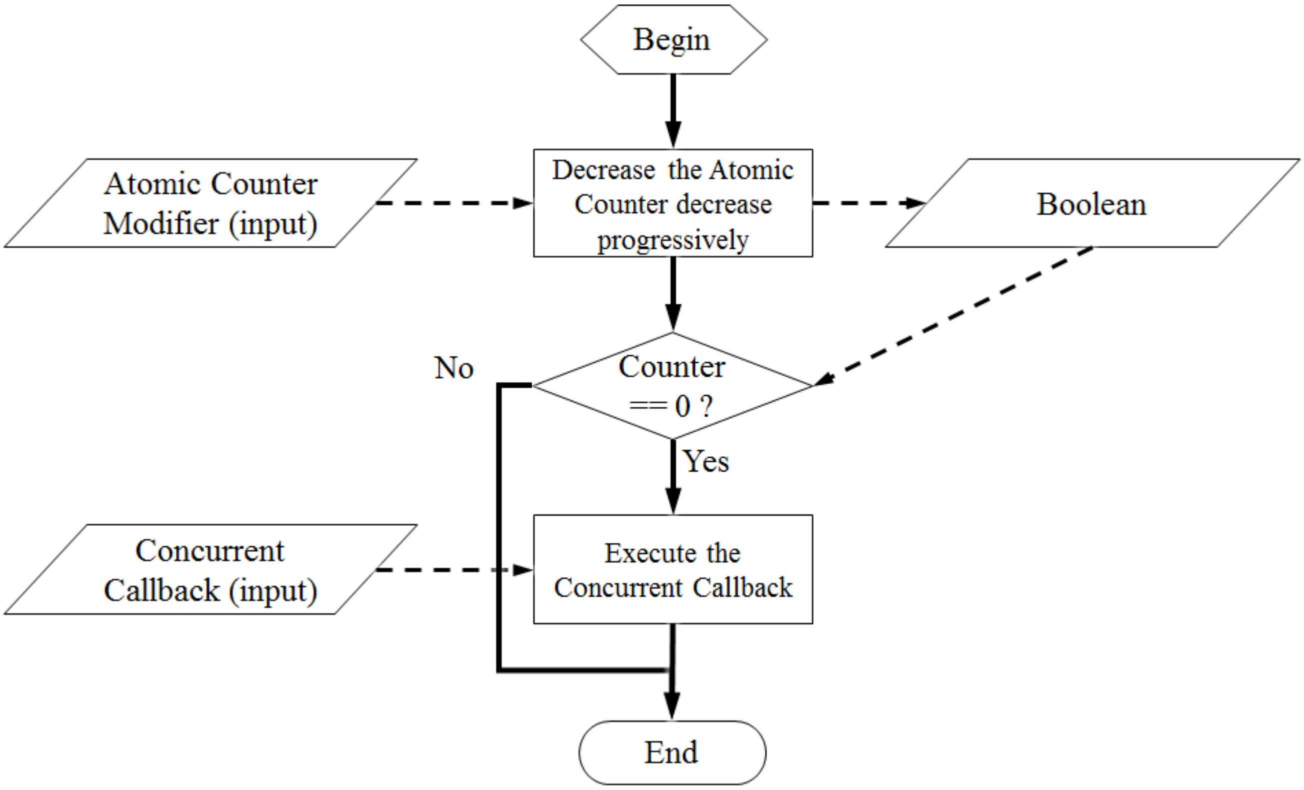


Figure 10

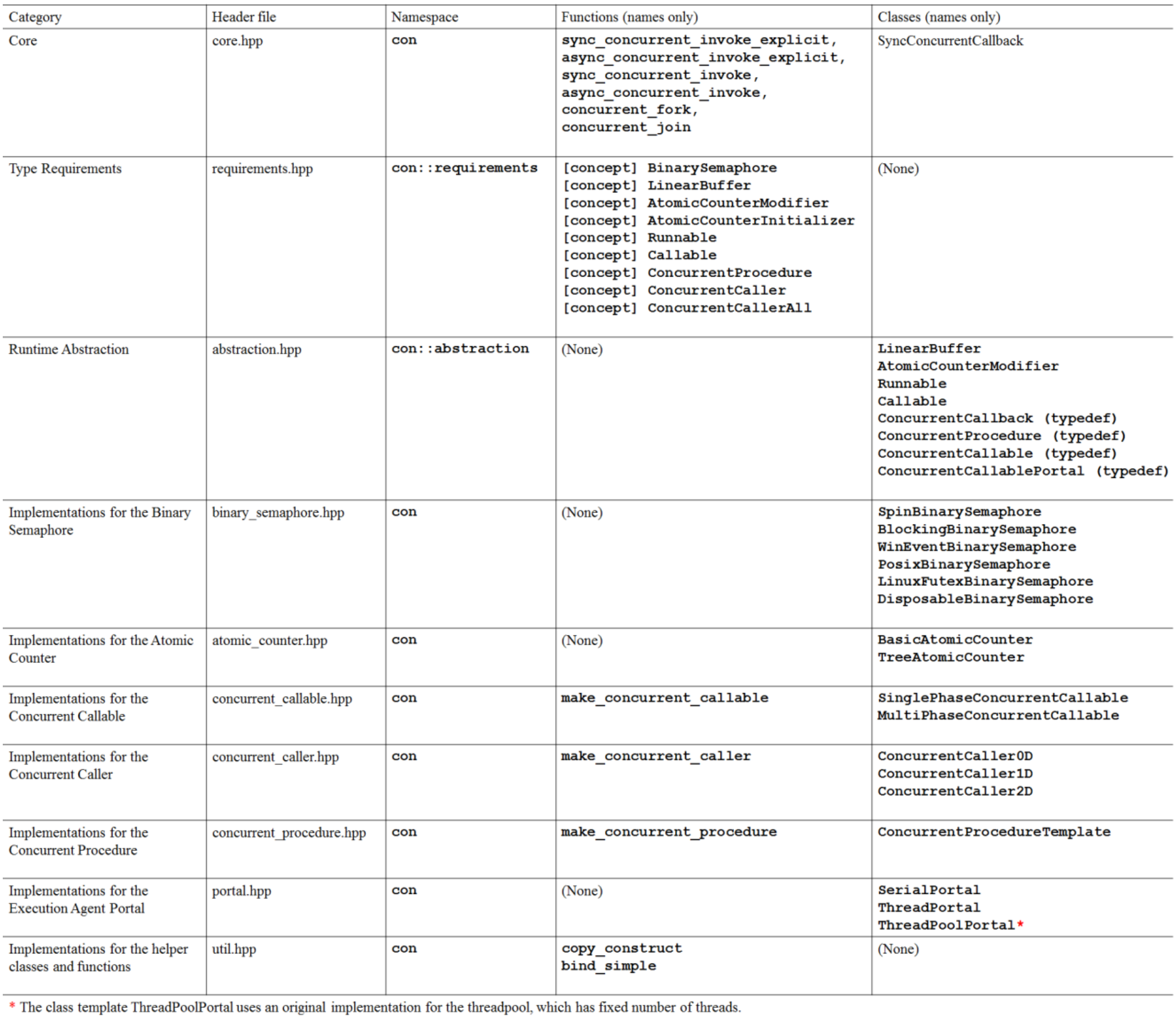
```
template <class AtomicCounterModifier,
          class Callback>
void concurrent_join(AtomicCounterModifier&& modifier,
                     Callback& callback) requires
    requirements::AtomicCounterModifier<AtomicCounterModifier>() &&
    requirements::Callable<Callback, void>();
```

Requires: The types **`AtomicCounterModifier`** and **`Callable`** shall meet the **`AtomicCounterModifier`** requirements and the **`Callable`** requirements, respectively.

Effects: Perform **`modifier.decrement()`**, if the returned value is false, execute **`callback()`**, whose flow chart is shown in Figure 10.

Return type: **`void`**

3 Implementation



Category	Header file	Namespace	Functions (names only)	Classes (names only)

Core	core.hpp	`con`	`sync_concurrent_invoke_explicit,`
			`async_concurrent_invoke_explicit,`
			`sync_concurrent_invoke,`
			`async_concurrent_invoke,`
			`concurrent_fork,`
			`concurrent_join`
				SyncConcurrentCallback

Type Requirements	requirements.hpp	`con::requirements`	`[concept] BinarySemaphore`
			`[concept] LinearBuffer`
			`[concept] AtomicCounterModifier`
			`[concept] AtomicCounterInitializer`
			`[concept] Runnable`
			`[concept] Callable`
			`[concept] ConcurrentProcedure`
			`[concept] ConcurrentCaller`
			`[concept] ConcurrentCallerAll`
				(None)

Runtime Abstraction	abstraction.hpp	`con::abstraction`	(None)	`LinearBuffer`
				`AtomicCounterModifier`
				`Runnable`
				`Callable`
				`ConcurrentCallback (typedef)`
				`ConcurrentProcedure (typedef)`
				`ConcurrentCallable (typedef)`
				`ConcurrentCallablePortal (typedef)`

Implementations for the Binary Semaphore	binary_semaphore.hpp	`con`	(None)	`SpinBinarySemaphore`
				`BlockingBinarySemaphore`
				`WinEventBinarySemaphore`
				`PosixBinarySemaphore`
				`LinuxFutexBinarySemaphore`
				`DisposableBinarySemaphore`

Implementations for the Atomic Counter	atomic_counter.hpp	`con`	(None)	`BasicAtomicCounter` `TreeAtomicCounter`
Implementations for the Concurrent Callable	concurrent_callable.hpp	`con`	`make_concurrent_callable`	`SinglePhaseConcurrentCallable` `MultiPhaseConcurrentCallable`

Implementations for the Concurrent Caller	concurrent_caller.hpp	`con`	`make_concurrent_caller`	`ConcurrentCaller0D`
				`ConcurrentCaller1D`
				`ConcurrentCaller2D`

Implementations for the Concurrent Procedure	concurrent_procedure.hpp	`con`	`make_concurrent_procedure`	`ConcurrentProcedureTemplate`

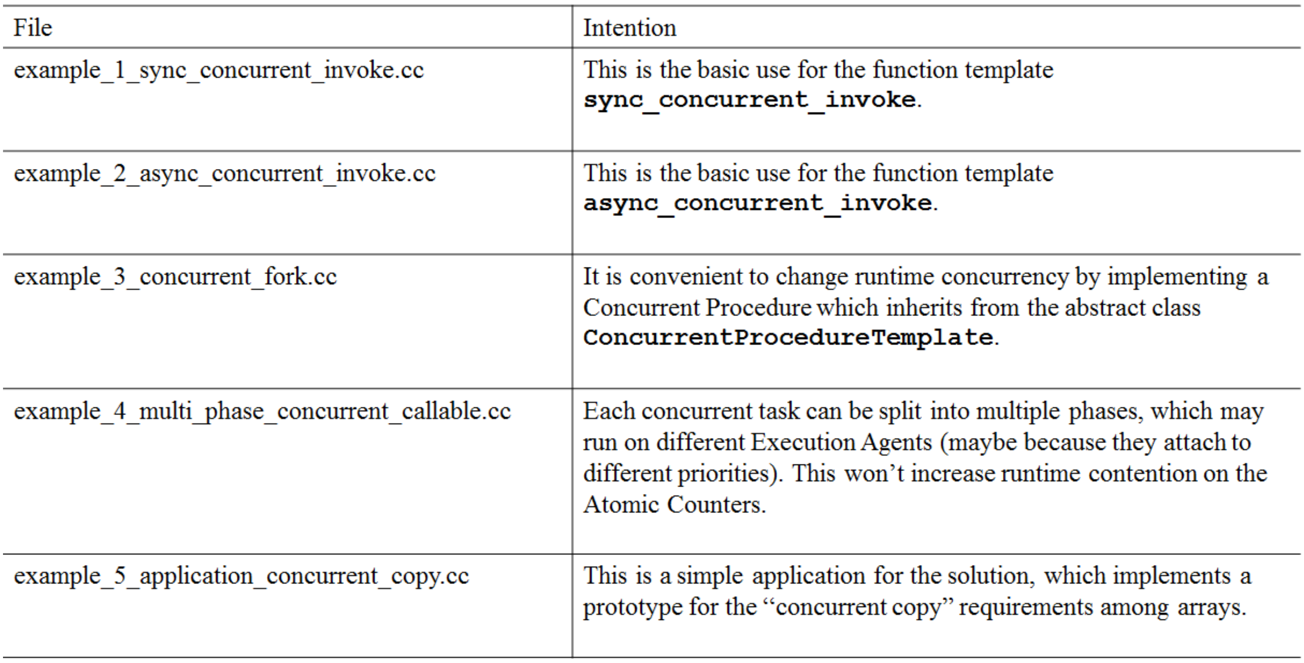
Implementations for the Execution Agent Portal	portal.hpp	`con`	(None)	`SerialPortal`
				`ThreadPortal`
				`ThreadPoolPortal`*

Implementations for the helper classes and functions	util.hpp	`con`	`copy_construct` `bind_simple`	(None)

* The class template ThreadPoolPortal uses an original implementation for the threadpool, which has fixed number of threads.

Figure 11

Although some details are still to be considered to make this solution standardized, I've already implemented a prototype for the entire solution in C++ (with **C++14 (minimum supported)** and the **Concept TS**). The header file **"concurrent.h"** (which includes other 10 header files) enables users to use anything in the library. Every type and function in the solution is defined in the namespace `con`. The overview of the library is shown in Figure 11.



File	Intention
example_1_sync_concurrent_invoke.cc	This is the basic use for the function template **sync_concurrent_invoke**.
example_2_async_concurrent_invoke.cc	This is the basic use for the function template **async_concurrent_invoke**.
example_3_concurrent_fork.cc	It is convenient to change runtime concurrency by implementing a Concurrent Procedure which inherits from the abstract class **ConcurrentProcedureTemplate**.
example_4_multi_phase_concurrent_callable.cc	Each concurrent task can be split into multiple phases, which may run on different Execution Agents (maybe because they attach to different priorities). This won't increase runtime contention on the Atomic Counters.
example_5_application_concurrent_copy.cc	This is a simple application for the solution, which implements a prototype for the "concurrent copy" requirements among arrays.

Figure 12

For a better understanding for the implementation, 5 examples is attached, as is shown Figure 12.