Structural Support for C++ Concurrency

Document number: P0642R0
Date: 2017-05-26
Project: Programming Language C++
Audience: SG1, SG8, EWG, LEWG

Authors: Mingxin Wang (College of Software Engineering, Jilin University, China),
Wei Chen (College of Computer Science and Technology, Key Laboratory for Software Engineering, Jilin University, China)
Reply-to: Mingxin Wang <wmx16835vv@163.com>

Table of Contents

1 Introduction 2
2 Motivation and Scope 2
3 Impact On the Standard 3
4 Design Decisions 5
  4.1 Execution Structures 5
  4.2 Synchronizations 9
    4.2.1 One-to-one 9
    4.2.2 One-to-many 10
    4.2.3 Many-to-one 11
5 Technical Specifications 14
  5.1 Requirements and Concepts 14
    5.1.1 Binary Semaphores 14
    5.1.2 Atomic Counters 15
    5.1.3 Runnable and Callable Types 18
    5.1.4 Concurrent Procedures 19
    5.1.5 Execution Agent Portals 21
    5.1.6 Concurrent Callables 22
    5.1.7 Concurrent Callers 23
  5.2 Function Templates 25
    5.2.1 Function template async_concurrent_invoke 25
    5.2.2 Function template async_concurrent_invoke_explicit 25
    5.2.3 Function template sync_concurrent_invoke 26
    5.2.4 Function template sync_concurrent_invoke_explicit 27
    5.2.5 Function template concurrent_fork 28
    5.2.6 Function template concurrent_join 29
  5.3 Implementation 30

1 Introduction

Currently, there is little structural support in concurrent programming in C++. Based on the requirements in concurrent programming, this proposal intends to add structural support in concurrent programming in C++, enabling implementations to have robust architecture and flexible synchronization algorithm.

The issue is concluded into two major problems: the first is about the architecture in concurrent programming, and the second is about the algorithms for synchronizations.

The problems can be solved with a novel architecture and dedicated synchronization algorithms. With the support of the solution, not only are users able to structure concurrent programs like serial ones as flexible as function calls, but also to choose algorithms for synchronizations based on platform or performance considerations.

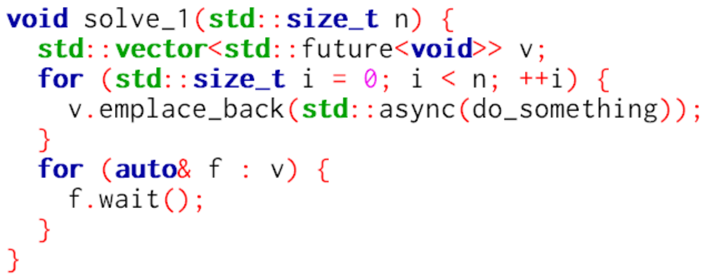
2 Motivation and Scope

This paper intends to solve 2 major problems in concurrent programming:

1. **The architecture: how to structure concurrent programs like serial ones as flexible as function calls**

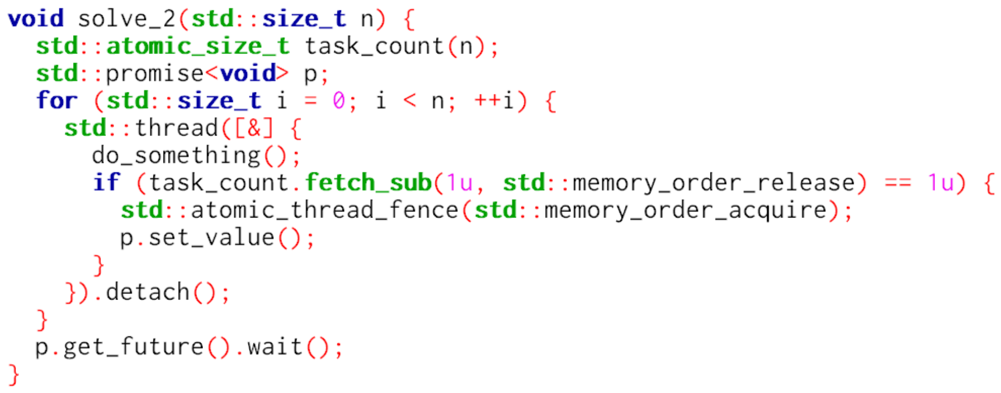
“Function” and “Invoke” are the basic concepts of programming, enabling users to wrap their logic into units and decoupling every parts from the whole program. Based on the novel solution, these concepts can be naturally generalized in concurrent programming.

2. **The algorithm: how to implement synchronization requirements to adapt to different runtime environment and performance requirements**



```cpp
void solve_1(std::size_t n) {
  std::vector<std::future<void>> v;
  for (std::size_t i = 0; i < n; ++i) {
    v.emplace_back(std::async(do_something));
  }
  for (auto& f : v) {
    f.wait();
  }
}
```

Figure 1



```cpp
void solve_2(std::size_t n) {
  std::atomic_size_t task_count(n);
  std::promise<void> p;
  for (std::size_t i = 0; i < n; ++i) {
    std::thread([&] {
      do_something();
      if (task_count.fetch_sub(1u, std::memory_order_release) == 1u) {
        std::atomic_thread_fence(std::memory_order_acquire);
        p.set_value();
      }
    }).detach();
  }
  p.get_future().wait();
}
```

Figure 2

Suppose there’s a common requirement to implement a model that launch **n** async tasks and wait for their completion. Implementations

- may maintain **n** "promises" for each task and perform several Ad-hoc synchronization operations, as is shown in Figure 1, or
- may manage an atomic integer maintaining the unfinished number of tasks (initially, **n**) with lock-free operations, and let the first **(n - 1)** finished tasks synchronize with the last finished one, then let the last finished task perform an Ad-hoc synchronization operation to unblock the model (some other advanced solutions such as the “Latch” or the “Barrier” work on the same principle), as is shown in Figure 2.

It is true that “**sharing is the root of all contention**”. The first implementation may introduce more context switching overhead, but the contention is never greater than 2. Although the second implementation has better performance with low concurrency, when concurrency is high, the contention may vary from 1 to n, and may prevent progress. For some performance sensitive requirements, **a "compromise" of the two solutions is probably more optimal**.

Thanks to the Concepts TS, I was able to implement the entire solution in C++ gracefully, making it easier for users to debug their code implemented with this solution. I hope this solution is positive for improving the C++ programming language.

3 Impact On the Standard

Requirements	**Utilities**	**Interfaces**	**Structural Supports**
One-to-one synchronizations	Threads Fibers Futures (Promises)	Executors Coroutines	Async
Many-to-one synchronizations	Latches Barriers Atomics	The Missing	
One-to-many synchronizations	Latches Barriers Condition Variables		

Figure 3

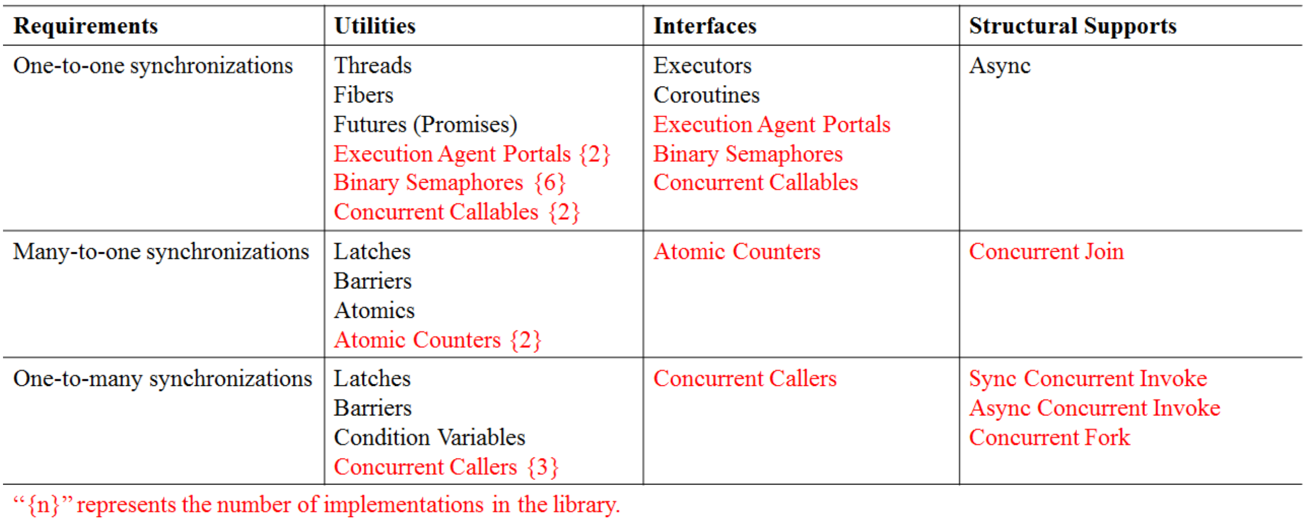
"One-to-one", "many-to-one" and "one-to-many" are the basic synchronization requirements in concurrent programming. What we have for the requirements in C++ is shown in Figure 3.

Note that,

- **Utilities**: the code we actually have (classes, functions...), and
- **Interfaces**: the code required for structuring (semantics, requirements, concepts...), and
- **Structural supports**: how to build a program (methods, patterns, language features...).

For example, for the basic function calls:

- The utilities are the functions in the standard library, and
- The interface is "function" itself, users are required to write code with parameters and return values (and maybe exceptions, etc.), and
- The structural support is "invocation", and users are able to "invoke" functions with specific syntax.

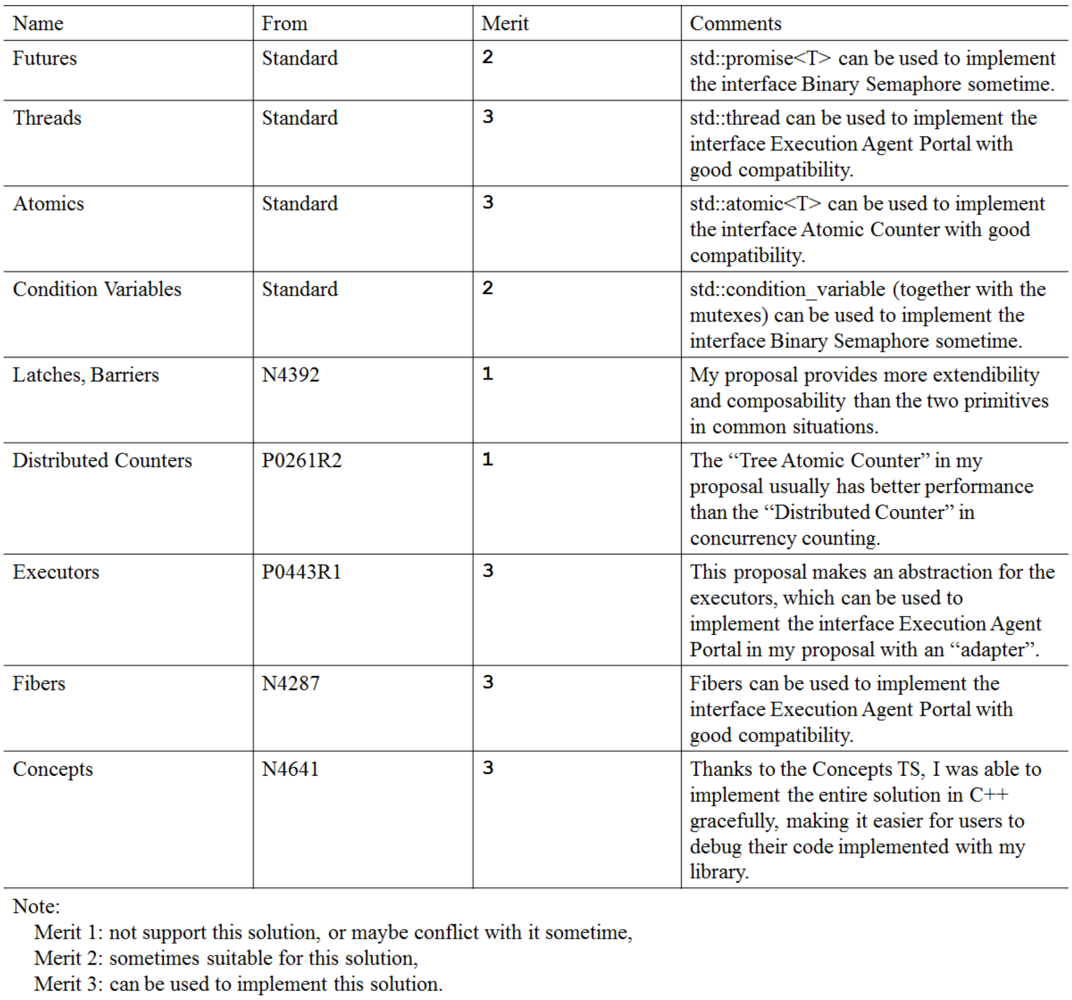


Requirements	**Utilities**	**Interfaces**	**Structural Supports**
One-to-one synchronizations	Threads Fibers Futures (Promises) Execution Agent Portals {2} Binary Semaphores {6} Concurrent Callables {2}	Executors Coroutines Execution Agent Portals Binary Semaphores Concurrent Callables	Async
Many-to-one synchronizations	Latches Barriers Atomics Atomic Counters {2}	Atomic Counters	Concurrent Join
One-to-many synchronizations	Latches Barriers Condition Variables Concurrent Callers {3}	Concurrent Callers	Sync Concurrent Invoke Async Concurrent Invoke Concurrent Fork

“{n}” represents the number of implementations in the library.

Figure 4

This solution is particularly for the missing parts, as is shown in Figure 4.



Name	From	Merit	Comments
Futures	Standard	2	std::promise<T> can be used to implement the interface Binary Semaphore sometime.
Threads	Standard	3	std::thread can be used to implement the interface Execution Agent Portal with good compatibility.
Atomics	Standard	3	std::atomic<T> can be used to implement the interface Atomic Counter with good compatibility.
Condition Variables	Standard	2	std::condition_variable (together with the mutexes) can be used to implement the interface Binary Semaphore sometime.
Latches, Barriers	N4392	1	My proposal provides more extendibility and composability than the two primitives in common situations.
Distributed Counters	P0261R2	1	The “Tree Atomic Counter” in my proposal usually has better performance than the “Distributed Counter” in concurrency counting.
Executors	P0443R1	3	This proposal makes an abstraction for the executors, which can be used to implement the interface Execution Agent Portal in my proposal with an “adapter”.
Fibers	N4287	3	Fibers can be used to implement the interface Execution Agent Portal with good compatibility.
Concepts	N4641	3	Thanks to the Concepts TS, I was able to implement the entire solution in C++ gracefully, making it easier for users to debug their code implemented with my library.

Note:
Merit 1: not support this solution, or maybe conflict with it sometime,
Merit 2: sometimes suitable for this solution,
Merit 3: can be used to implement this solution.

Figure 5

The related proposals are shown in Figure 5.

4 Design Decisions

4.1Execution Structures

In concurrent programs, executions of tasks always depend on one another, thus the developers are required to control the synchronizations among the executions; **these synchronization requirements can be divided into 3 basic categories: “one-to-one”, “one-to-many”, “many-to-one”**. Besides, there are “many-to-many” synchronization requirements; since they are usually not “one-shot”, and often be implemented as a “many-to-one” stage and a “one-to-many” stage, they are not fundamental ones.

“Function” and “Invoke” are the basic concepts of programming, enabling users to wrap their logic into units and decoupling every parts from the entire program. Are these concepts able to be generalized in concurrent programming? The answer is YES, and that’s what this paper revolves around.

When producing a “Function”, only the requirements (pre-condition), input, output, effects, synchronizations, exceptions, etc. for calling this function shall be considered; who or when to “Invoke” a “Function” is not to be concerned about. When it comes to concurrent programming, there shall be a beginning and an ending for each “Invoke”; in other words, a “Concurrent Invoke” shall begin from “one” and end up to “one”, which forms a “one-to-many-to-one” synchronization.

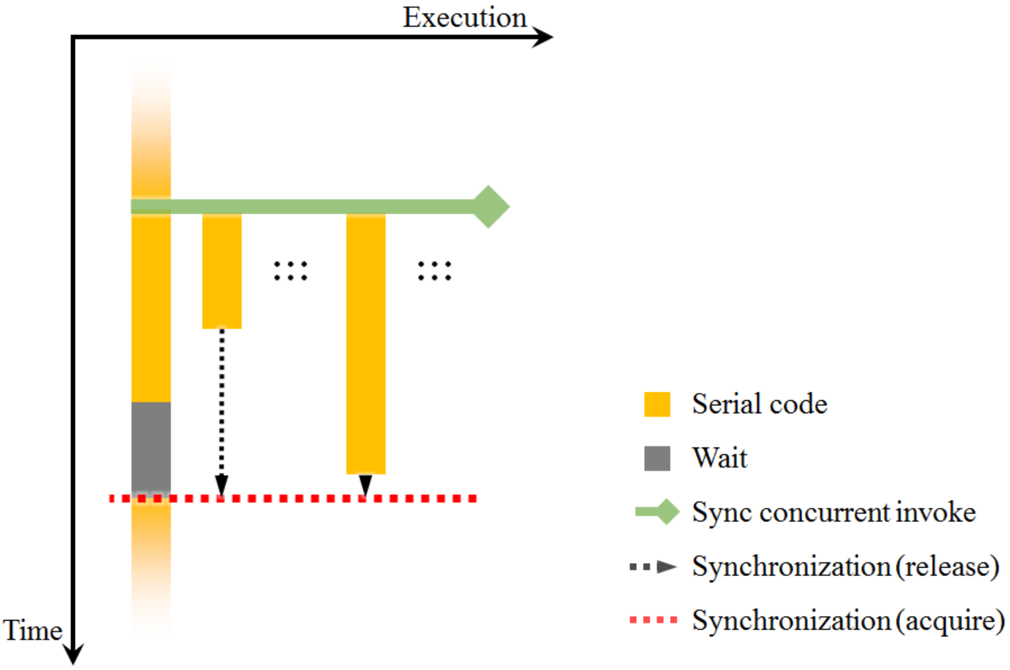
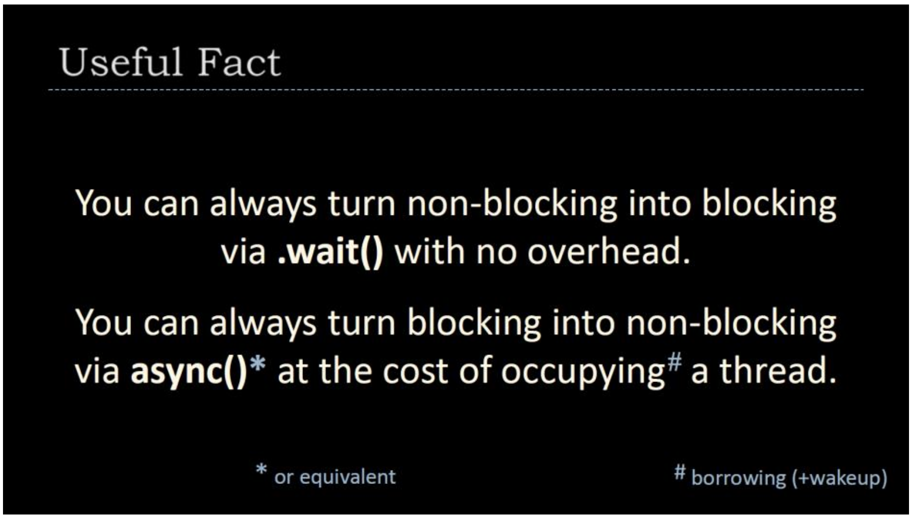


Figure 6

The most common concurrent model is starting several independent tasks and waiting for their completion. This model is defined as “**Sync Concurrent Invoke**”. A typical scenario for the “Sync Concurrent Invoke” model is shown in Figure 6.



Useful Fact

You can always turn non-blocking into blocking via **.wait()** with no overhead.

You can always turn blocking into non-blocking via **async()*** at the cost of occupying# a thread.

* or equivalent

borrowing (+wakeup)

Figure 7

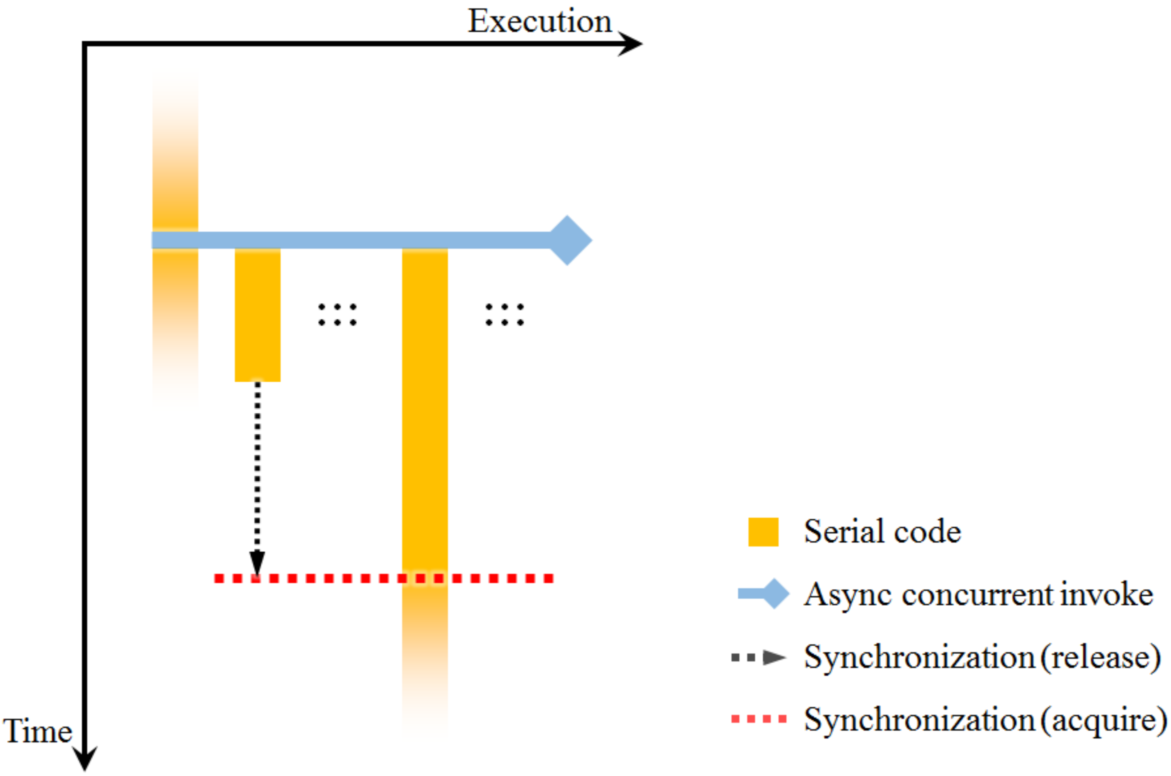


Figure 8

Figure 7 is one slide from your previous talk: "*blocking is harmful*", and "*you can always turn blocking into non-blocking at a cost of occupying a thread*". Nonetheless, is there a "more elegant" way to avoid blocking? Yes, there is, just let the execution agent that executes the last finished task in a "Sync Concurrent Invoke" to do the rest of the works (the concept "execution agent" is defined in C++ ISO standard 30.2.5.1: *An execution agent is an entity such as a thread that may perform work in parallel with other execution agents*). This model is defined as "**Async Concurrent Invoke**". A typical scenario for the "Async Concurrent Invoke" model is shown in Figure 8.

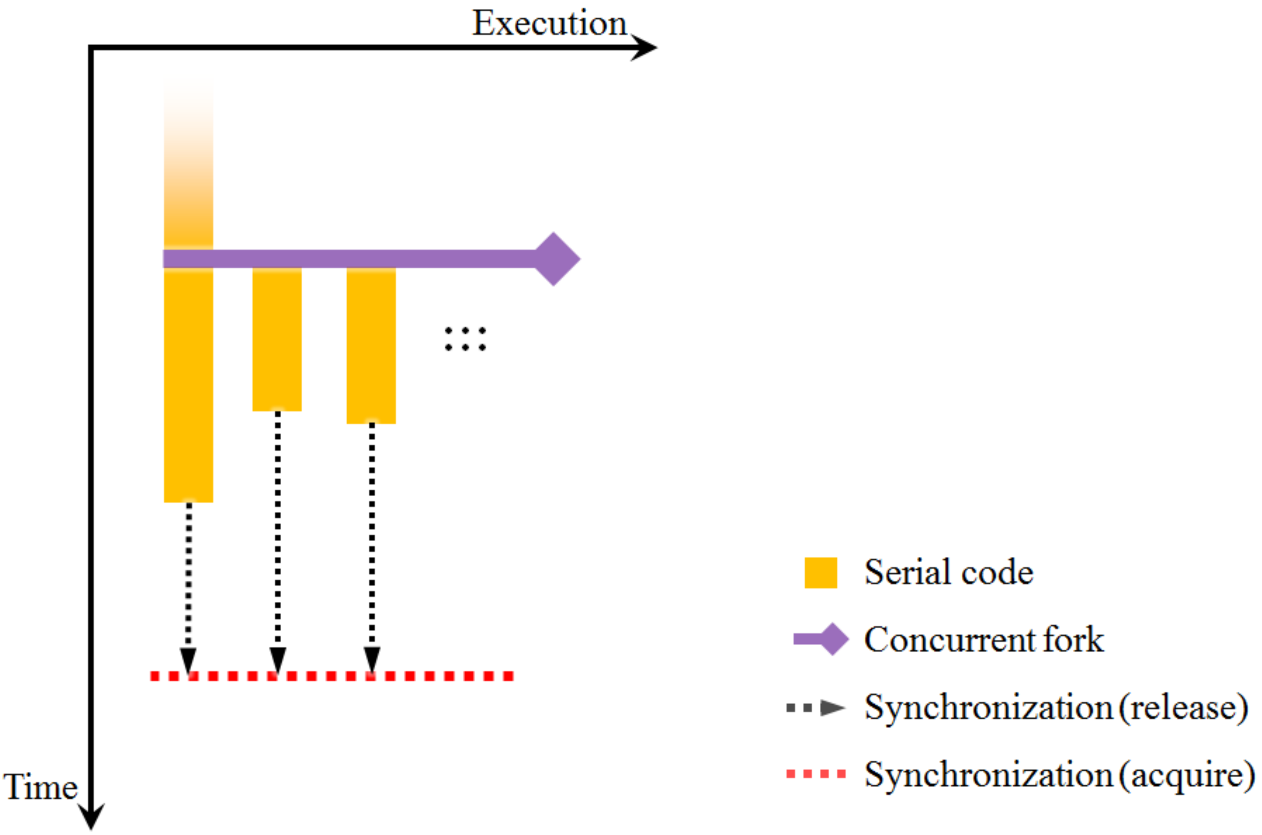


Figure 9

The “Sync Concurrent Invoke” and the “Async Concurrent Invoke” models are the static execution structures for concurrent programming, but not enough for runtime extensions. For example, when implementing a concurrent quick-sort algorithm, it is hard to predict how many subtasks will be generated. So we need a more powerful execution structure that can expand a concurrent invocation, which means, to add other tasks executed concurrently with the current tasks in a same concurrent invocation at runtime. This model is defined as “**Concurrent Fork**”. A typical scenario for the “Concurrent Fork” model is shown in Figure 9.

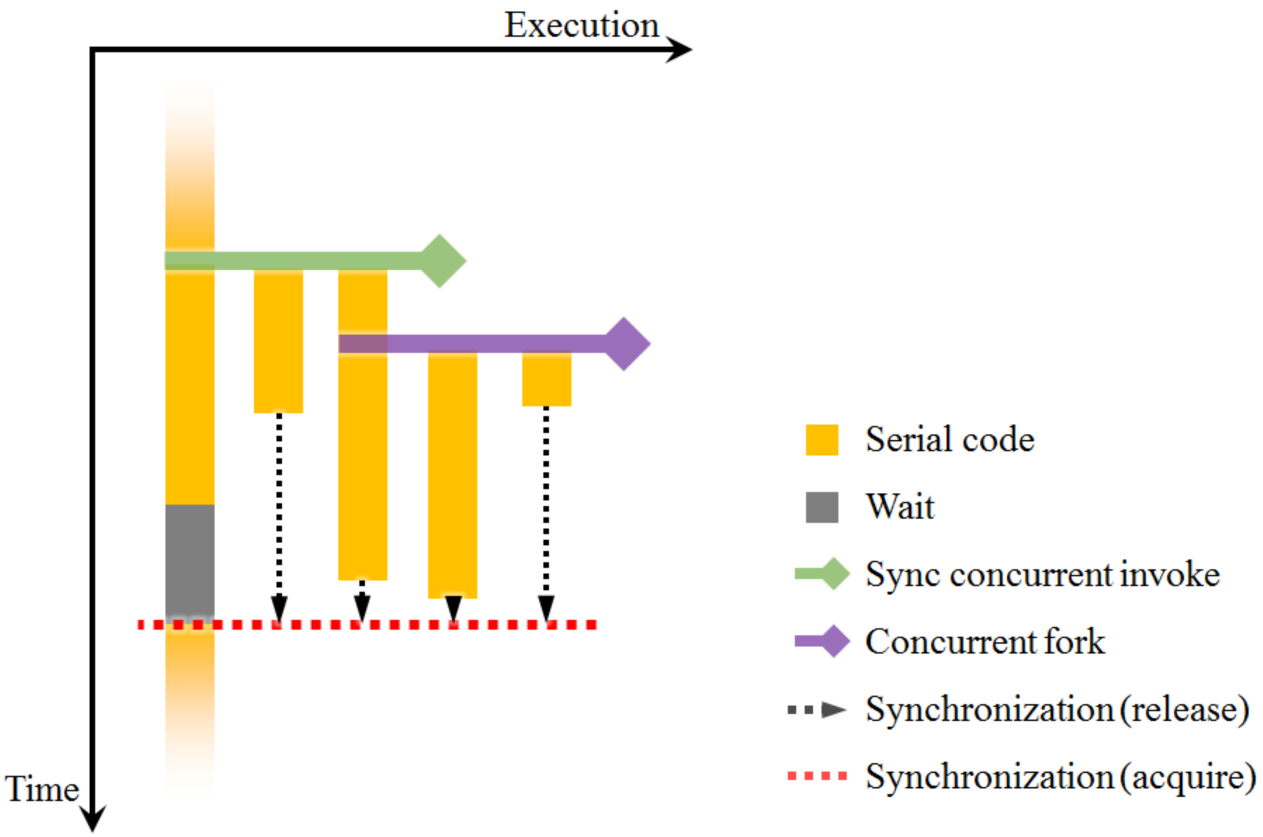


Figure 10

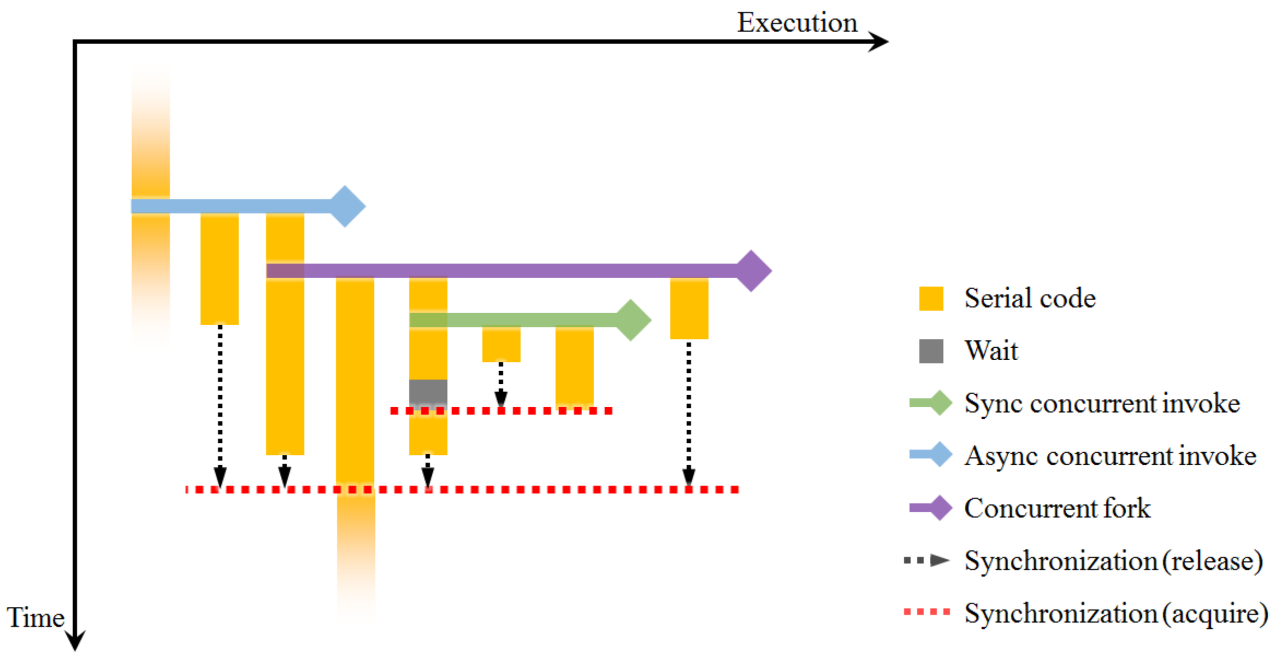


Figure 11

With the concept of the “Sync Concurrent Invoke”, the “Async Concurrent Invoke” and the “Concurrent Fork” models, we can easily build concurrent programs with complex dependencies among the executions, meanwhile, stay the concurrent logic clear. Figure 10 shows a typical scenario for a composition of the “Sync Concurrent Invoke” and the “Concurrent Fork” models; Figure 11 shows a more complicated scenario.

From the “Sync Concurrent Invoke”, the “Async Concurrent Invoke” and the “Concurrent Fork” models, we can tell that:

- the same as serial invocations, the "Sync Concurrent Invoke" and the "Async Concurrent Invoke" models can be applied recursively, and
- by changing "Sync Concurrent Invoke" into "Async Concurrent Invoke", we can always turn blocking into non-blocking at a cost of managing the execution agents, because it's hard to predict which task is the last finished; users are responsible for the **load balance** in the entire program when applying the "Async Concurrent Invoke" model, and
- applying the “Concurrent Fork” model requires one concurrent invocation to expand, no matter the invocation is a “Sync” one or an “Async” one.

4.2 Synchronizations

4.2.1 One-to-one

The “one-to-one” synchronization requirements are much easier to implement than the other two. Providing there are two tasks named A and B, and B depends on the completion of A, we can simply make a serial implementation (sequentially execute A and B) with no extra context switching overhead. If task A and B are supposed to be executed on different execution agents (maybe because they attach to different priorities), extra overhead for the synchronization is inevitable, we may let another execution agent to execute B after A is completed. Usually we can implement such “one-to-one” synchronization requirements by starting a thread or submitting a task to a threadpool, etc.; besides, many patterns and frameworks provide us with more options, for example, the well-known “Future” pattern.

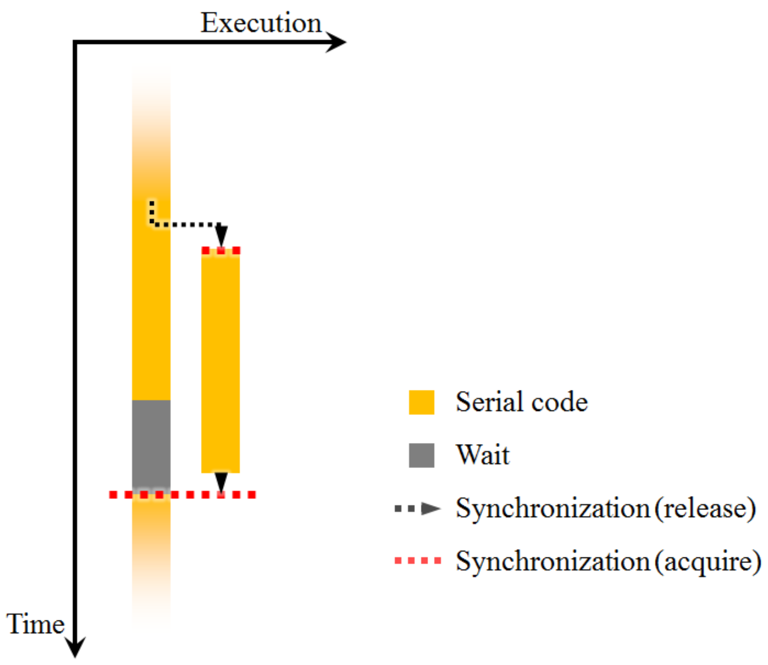


Figure 12

Currently in C++, the “Future” plays an important role in concurrent programming, which provides Ad-hoc synchronization from one thread to another, as is shown in Figure 12. It is apparent that “one-to-many” and “many-to-one” synchronization requirements can be converted into multiple “one-to-one” ones. Thus the “Future” can be used to deal with any synchronization situation with nice composability.

Besides, there are many primitives supported by various platforms, such as the “Futex” in modern Linux, the “Semaphore” defined in the POSIX standard and the “Event” in Windows. The “work-stealing” strategy is sometimes used for the “one-to-one” synchronization requirements, such as the Click programming language, the “Fork/Join Framework” in the Java programming language and the “TLP” in the .NET framework.

4.2.2 One-to-many

The “one-to-many” synchronization requirements are just as easy as the “one-to-one” ones most of the time. However, **when a “one-to-many” synchronization requirement is broken down into too many “one-to-one” ones, it will introduce too much extra overhead**.

One solution to this problem is to perform these “one-to-one” synchronization operations concurrently. Suppose we have 10000 tasks to launch asynchronously, we can divide the launching work into 2 phases,

- launch other 100 tasks,
- each task launched in the previous step launches 100 tasks respectively.

So that we can finish launching within 200 (instead of 10000) units of “one-to-one” synchronization time, and that’s the bottle neck for 2 phases. Similarly, if we divide the launching work into 4 phases, we can finish launching within 40 (instead of 200) units of “one-to-one” synchronization time.

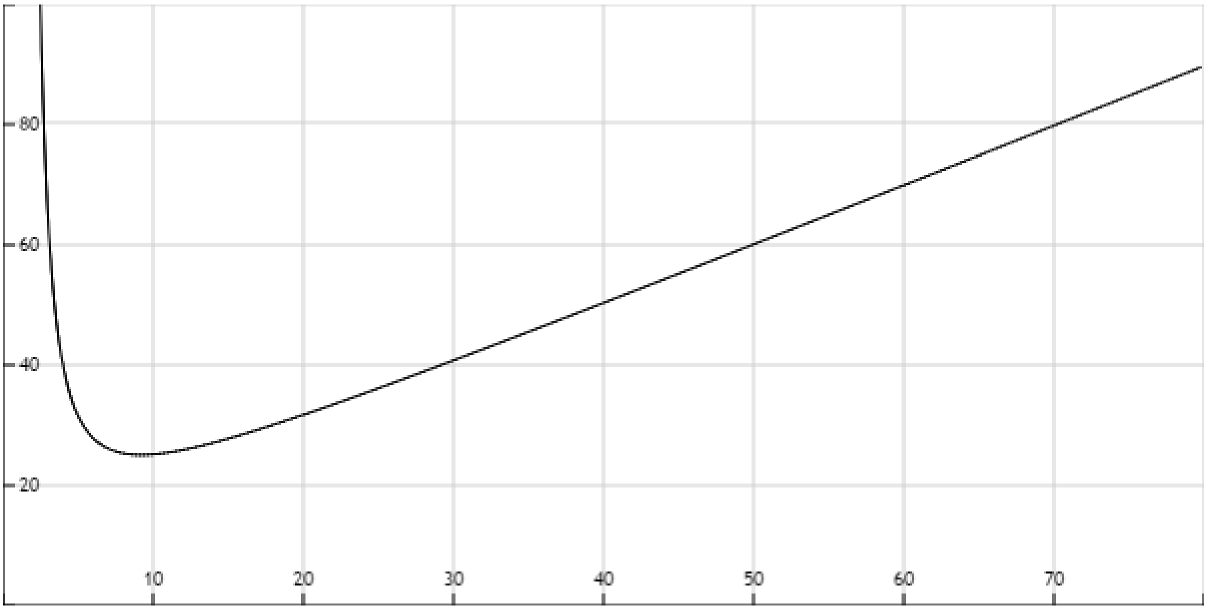
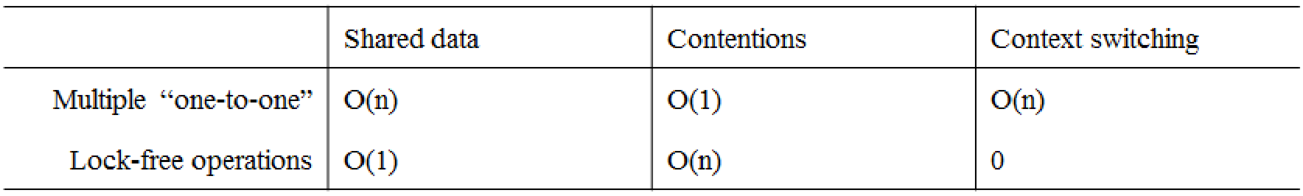


Figure 13 – The Graph for $f(x) = x\sqrt[x]{n}$, n = 10000

Generally, if we divide a launching work of n tasks into x phases, the minimum unit of time required to finish it is a function of x, $f(x) = x\sqrt[x]{n}$, whose graph is similar with Figure 13. It is apparent that $f(0^+) \to +\infty$ and

$f(+\infty) \rightarrow +\infty$. Let $f'(x) = 0$ and we obtain $x = \ln(n)$, where $f(x)$ is minimum. When $x = \ln(n)$, the ideal maximum number of tasks that each task may split into is $\sqrt[\ln(n)]{n} \equiv e$, which is a constant number greater than 2 and less than 3. When the maximum number of tasks that each task may split into is x, the minimum unit of time required to finish the work is a function of n and x, $g(n, x) = x \log_x^n$, it is apparent that $g(n,2) > g(n,3)$ holds when x is greater than 1, thus when a launching work of n tasks is divided into some phases, and adequate execution resources are provided, **it is theoretically optimal to divide each work into 3 smaller ones**.

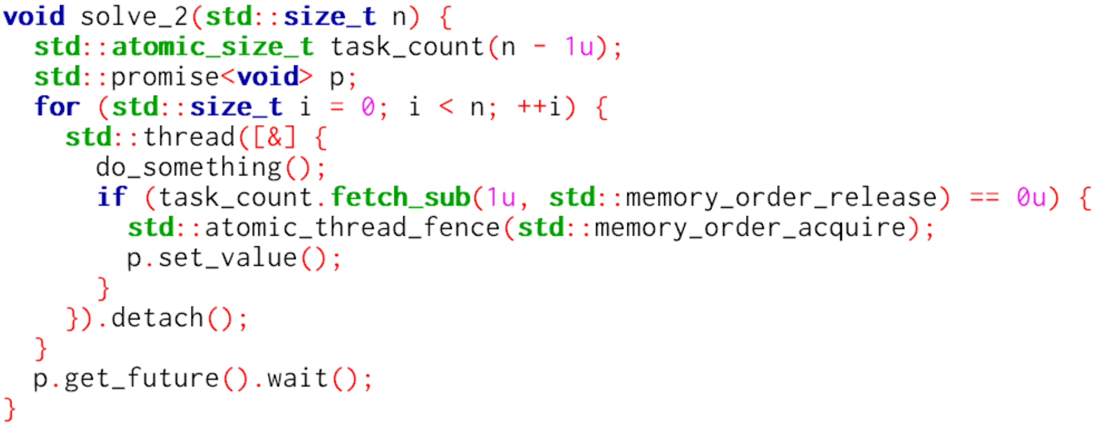
4.2.3 Many-to-one



	Shared data	Contentions	Context switching
Multiple "one-to-one"	O(n)	O(1)	O(n)
Lock-free operations	O(1)	O(n)	0

Figure 14

As mentioned earlier, the most common methods to implement the "many-to-one" synchronization requirements are to break it down into multiple "one-to-one" synchronizations, and to use an atomic integer maintaining with lock-free operations to let the first finished tasks synchronize with the last finished one. The advantages and disadvantages of the two methods complement each other, as is shown in Figure 14.



```
void solve_2(std::size_t n) {
  std::atomic_size_t task_count(n - 1u);
  std::promise<void> p;
  for (std::size_t i = 0; i < n; ++i) {
    std::thread([&] {
      do_something();
      if (task_count.fetch_sub(1u, std::memory_order_release) == 0u) {
        std::atomic_thread_fence(std::memory_order_acquire);
        p.set_value();
      }
    }).detach();
  }
  p.get_future().wait();
}
```

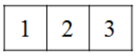
Figure 15

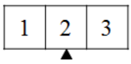
```
void solve_2(std::size_t n) {
  std::atomic_size_t task_count(n - 1u);
  std::promise<void> p;
  for (std::size_t i = 0; i < n; ++i) {
    std::thread([&] {
      do_something();
      std::atomic_thread_fence(std::memory_order_release);
      std::size_t cur = task_count.load(std::memory_order_relaxed);
      do {
        if (cur == 0u) {
          std::atomic_thread_fence(std::memory_order_acquire);
          p.set_value();
          break;
        }
      } while (!task_count.compare_exchange_weak(cur,
                                                  cur - 1u,
                                                  std::memory_order_relaxed));
    }).detach();
  }
  p.get_future().wait();
}
```

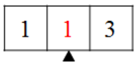
Figure 16

Note that the state of "0" for the atomic integers is never utilized, when using atomic integers to maintain the number of unfinished tasks. The number of unfinished tasks is able to map to range **[0, n)**, so that the implementation shown in Figure 2 can be reconstructed, as is shown in Figure 15 and Figure 16.

Inspired from the class "LongAdder" in the Java programming language, I found it helpful to split one atomic integer into many "cells" to decrease contention and increase concurrency in operations on the integers. Unfortunately, although this method can reduce the complexity of writing, it increases the complexity of reading, e.g. if we are to check whether an integer becomes 0 and subtract 1 from it (this is exactly the requirement in the preceding paper), the operations that a "LongAdder" will likely perform are as follows:

- sum up the cells 

 and we get 6,
- check whether 6 equals to 0, and we get [FALSE],
- choose a cell at random 

,
- subtract 1 from the chosen cell 

, and the total complexity is O (*number of cells*).

Other algorithms and data structures such as the "distributed counter" models, the "combining tree" models and the "counting network" models also introduce diverse extra overhead while used to solve this challenge.

	Shared data	Contentions	Context switching
Multiple "one-to-one"	O(n)	O(1)	O(n)
Lock-free operations	O(1)	O(n)	0
Tree Atomic Counter*	O(n / m)	O(m)	0

* The expression "m" represents the maximum allowed contention defined by users.

Figure 17

Then I learnt from the class “Phaser” in the Java programming language that instead of splitting an integer into independent “cells”, it is better to “**tiering them up to form a tree**”. Since the class “Phaser” has more functions than just “tiering”, and it is not convenient to control contentions on each node, I improved the design based on “single responsibility principle” that enables users to set the upper limit of the contention as they expect. I define the new design as “**Tree Atomic Counter**”, whose property is shown in Figure 17.

As the accurate value of the counter is not required at runtime here, the "Tree Atomic Counter" is more optimal because not only does it reduce the contention, but also remains the complexity of reading always to be O(1).

Similar with other widely used tree-shaped data structures, a “Tree Atomic Counter” is composed of several nodes. Each node holds an atomic integer and a reference to its parent. Each “Tree Atomic Counter” associates with a constant integer MAX_N, which represents the maximum count of its each node, in other words, the count of each node belongs to the interval [0, MAX_N]. This property guarantees that the contention on each node never exceeds (MAX_N + 1). When tiering one node to another, the count of the parent node is increased by 1.

The structure of the “Tree Atomic Counter” has the following properties:

- the count represented by a “Tree Atomic Counter” equals to the sum of the count held by all its nodes,
- the root node holds an invalid reference (e.g. a null pointer).

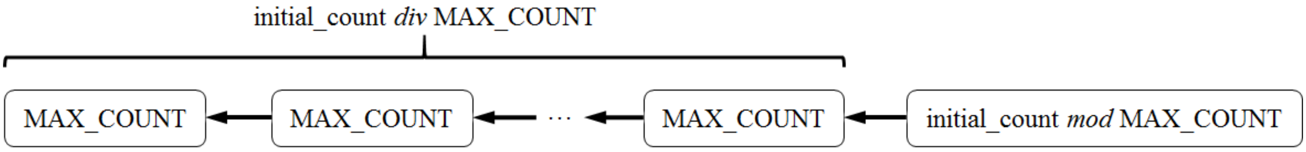


Figure 18

Unlike other widely used tree-shaped data structures such as the “Red-black Tree” or the “AVL Tree”, the “Tree Atomic Counter” doesn’t require to be balanced. Moreover, the shape of a “Tree Atomic Counter” has no effects on its use or performance. For the initialization of a “Tree Atomic Counter”, we can make it either balanced or completely unbalanced, as is shown in Figure 18, just like a forward list!

Figure 19

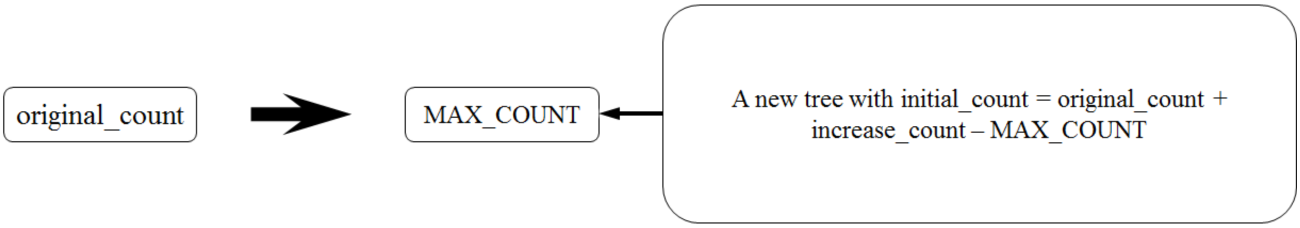


Figure 20

When increasing a count on a node, there are two strategies:

- directly add the count to the node if the total count on this node won’t exceed MAX_COUNT, as is shown in

Figure 19, or

- attach a new tree on the node otherwise, as is shown in Figure 20.

If we are to check whether a “Tree Atomic Counter” becomes 0 and subtract 1 from it with a node, there are three strategies:

- decrease 1 from the node if the count of the node is not 0, or
- recursively perform this operation to the parent node if the parent node is valid and the count of the current node equals to 0, or
- do nothing otherwise.

5 Technical Specifications

5.1 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`**.

5.1.1 Binary Semaphores

5.1.1.1 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.

5.1.1.2 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.

5.1.1.3 Concept template BinarySemaphore

```
namespace requirements {

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

}
```

5.1.2 Atomic Counters

5.1.2.1 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 concurrently 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.

5.1.2.2 Requirements

5.1.2.2.1 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

5.1.2.2.2 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.

5.1.2.2.3 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.

5.1.2.3 Concepts

5.1.2.3.1 Concept template LinearBuffer

```
namespace requirements {

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

}
```

5.1.2.3.2 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;
  });
}

}
```

5.1.2.3.3 Concept template AtomicCounterInitializer

```
namespace requirements {

template <class T>
```

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

}
```

5.1.3 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`**.

5.1.3.1 Concept template Runnable

```
namespace requirements {

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

}
```

5.1.3.2 Concept template Callable

```
namespace requirements {

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

}
```

5.1.4 Concurrent Procedures

5.1.4.1 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 21

```
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 22

```
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 23

The “Concurrent Procedure” 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 21 (*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 22, or
- be implemented with a "Template" with runtime abstraction by inheriting from an abstract class, as is shown in Figure 23 (*note that abstraction::AtomicCounterModifier and abstraction::Callable are wrappers for Atomic Counter Modifiers and Callables, respectively; their principles are the same as* ***`std::function`***).

5.1.4.2 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.

5.1.4.3 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;
  };
}

}
```

5.1.5 Execution Agent Portals

5.1.5.1 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 24

```
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 25

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 24, 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 25 (*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.

5.1.5.2 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.

5.1.6 Concurrent Callables

5.1.6.1 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 26

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 26.
- combine multiple Execution Agent Portal entities and their corresponding Concurrent Procedure entities, execute the Concurrent Procedure entities sequentially with different Execution Agent Portal entities.

5.1.6.2 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.

5.1.7 Concurrent Callers

5.1.7.1 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.

5.1.7.2 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`**

5.1.7.3 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...);
  };
}

}
```

5.2Function Templates

5.2.1 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.

5.2.2 Function template async_concurrent_invoke_explicit

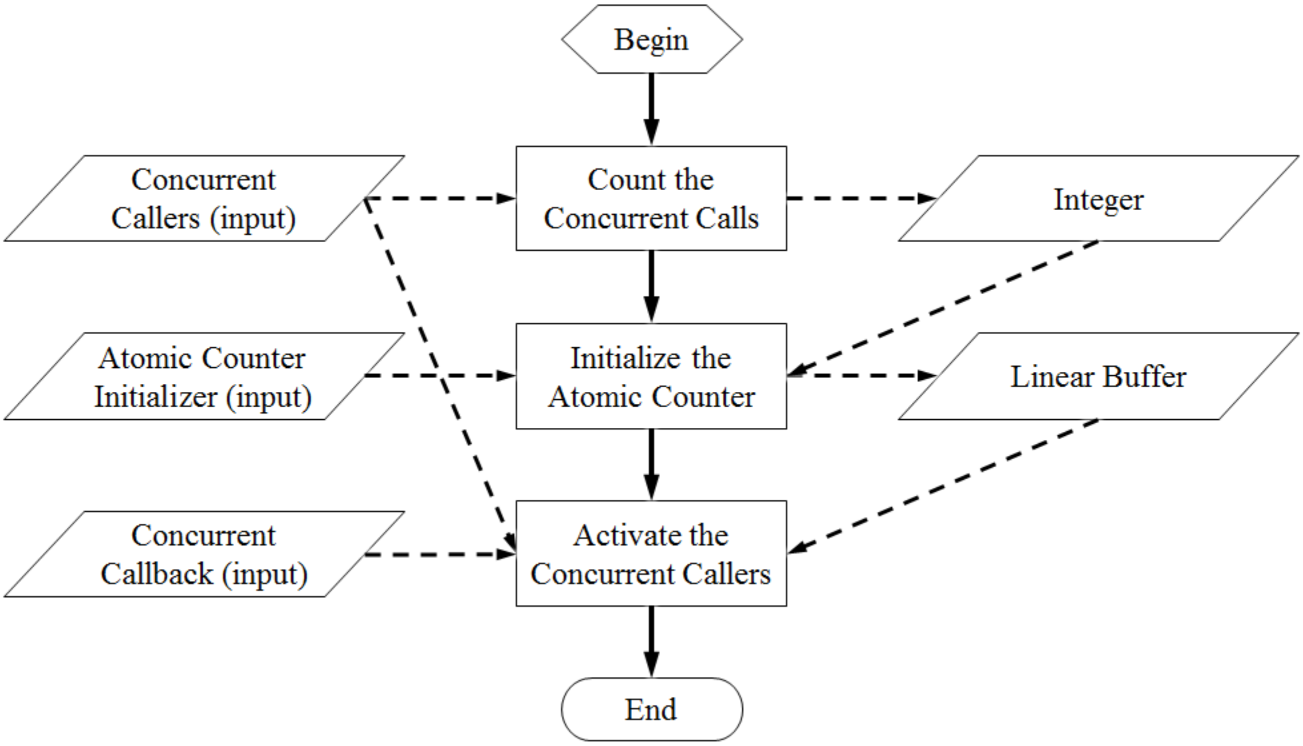


Figure 27

```
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 27.
Return type: **`void`**

5.2.3 Function 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.

5.2.4 Function template sync_concurrent_invoke_explicit

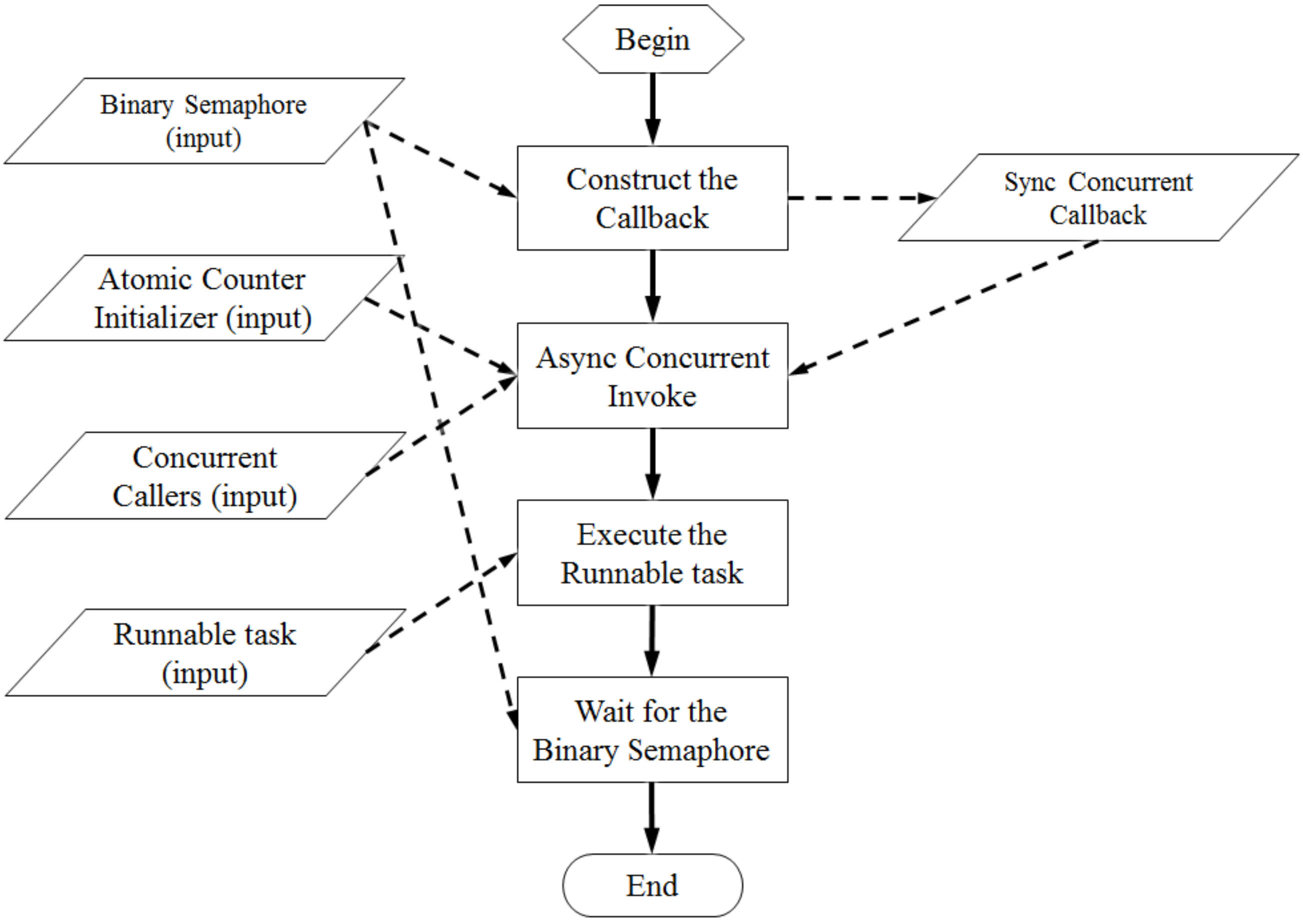


Figure 28

```
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 28.
Return type: **`std::result_of_t<SerialCallable()>`**
Returns: anything that **`callable()`** returns

5.2.5 Function template concurrent_fork

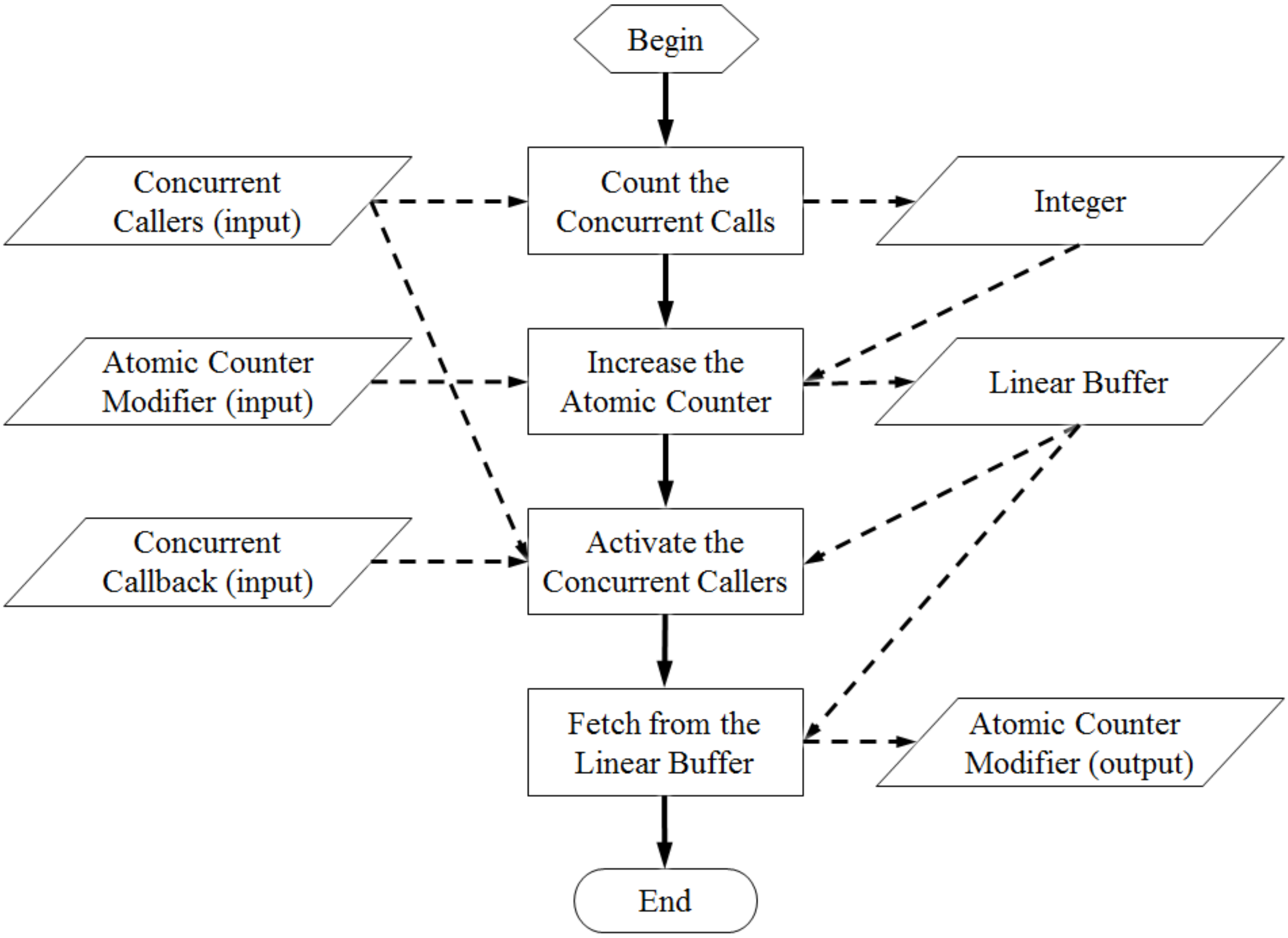


Figure 29

```
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 29.

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

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

5.2.6 Function template concurrent_join

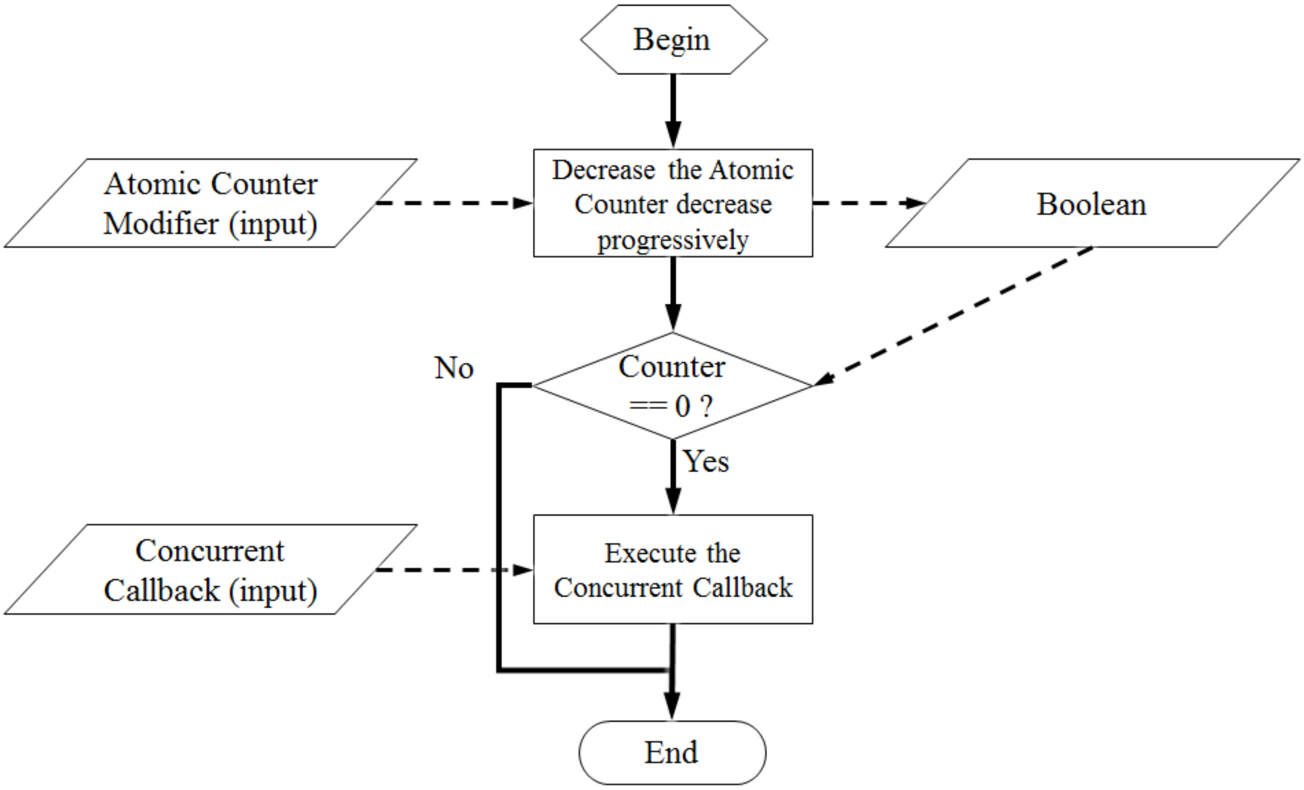
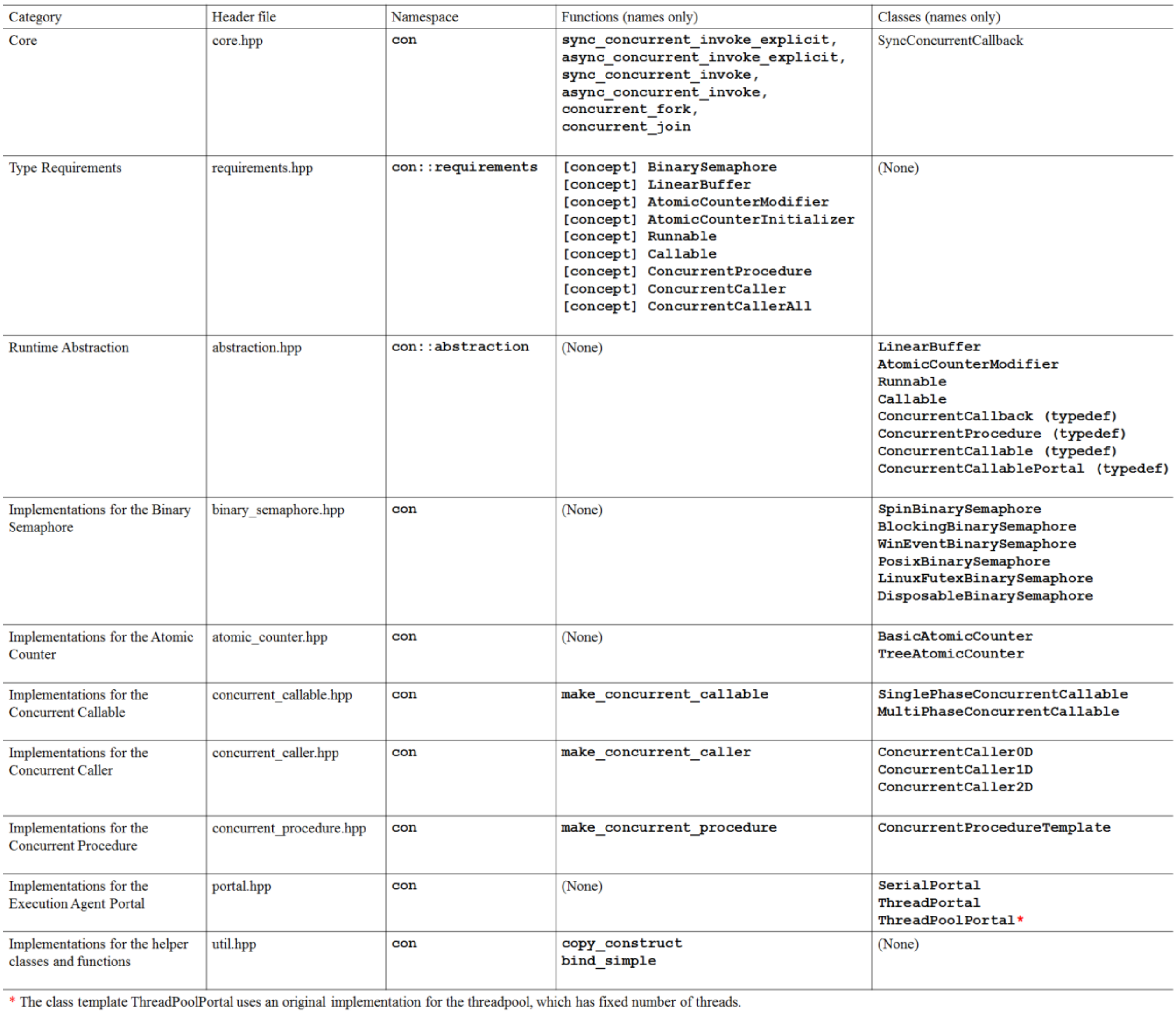


Figure 30

```
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 30.
Return type: **`void`**

5.3 Implementation



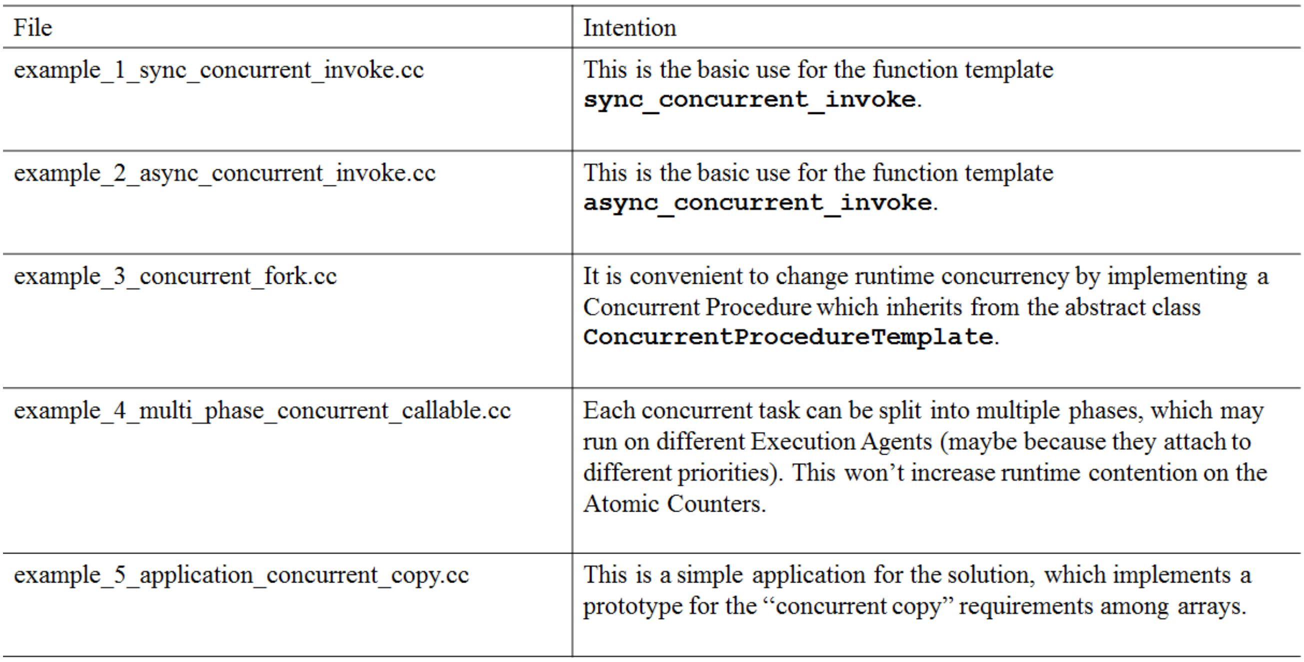
Category	Header file	Namespace	Functions (names only)	Classes (names only)
Core	core.hpp	con	sync_concurrent_invoke_explicit, async_concurrent_invoke_explicit, sync_concurrent_invoke, async_concurrent_invoke, concurrent_fork, concurrent_join	SyncConcurrentCallback
Type Requirements	requirements.hpp	con::requirements	[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 31

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**, available at https://github.com/wmx16835/structural-support-for-cxx-concurrency). 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 31.

The Binary Semaphores and the Atomic Counters, etc., are required in the core function templates. Some implementation for the requirements are provided with .h files, which are not completely documented. As is mentioned earlier, these implementations may have various mechanisms, including some primitives out of the C++ standard (for example, the "Futex"). These implementations are only recommended, but not all of them are available on every platform.



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 32

For a better understanding for the implementation, 5 examples is attached, as is shown Figure 32. Examples are prepared to demonstrate basic usage for the library, and the last example named "**example_5_application_concurrent_copy.cc**" shows an application for concurrent copy algorithm. In order to make the examples easy to understand, more complex applications are not provided, but that does not mean those applications are not implementable with this solution. For example, concurrent sort algorithms are much easier to implement with "sync_concurrent_invoke" and "concurrent_fork" function templates.