The Concurrent Invocation Library

Document number: P0642R2
Date: 2019-06-16
Project: Programming Language C++
Audience: SG1, LEWG, LWG
Authors: Mingxin Wang (Microsoft (China) Co., Ltd.),
Wei Chen (College of Computer Science and Technology, Key Laboratory for Software Engineering, Jilin University, China)
Reply-to: Mingxin Wang <mingxwa@microsoft.com>

Table of Contents

1 History

1.1 Changes from P0642R1

- Change the title of the paper from "Structure Support for C++ Concurrency" into "The Concurrent Invocation Library";
- Change the motivating example into a more generic one;
- Change function templates **`sync_concurrent_invoke`** and **`async_concurrent_invoke`** into **`concurrent_invoke`**.
- Remove the concepts of "Atomic Counter", "Atomic Counter Initializer", "Atomic Counter Modifier", "Linear Buffer", which become implementation-defined details;
- Add class **`bad_concurrent_invocation_context_access`** and class templates **`concurrent_token`** and **`concurrent_finalizer`**;
- Remove the concept of "Execution Agent Portal", which could be replaced by the Executors [P0443R10];
- Add two executor extensions: **`in_place_executor`** and **`thread_executor`**;
- Change requirements for runtime polymorphism into compile-time overload resolution.

1.2 Changes from P0642R0

- Redefine the **`AtomicCounterModifier`** requirements: change the number of times of "each of the first **`fetch()`** operations to the returned value of **`acm.increase(s)`**" from **`(s + 1)`** to **`s`**;
- Redefine the **`ConcurrentProcedure`** requirements: change the return type of **`cp(acm, c)`** from "Any type that meets the **`AtomicCounterModifier`** requirements" to **`void`**, update the corresponding sample code;
- Redefine the signature of function template **`concurrent_fork`**: change the return type from "Any type that meets the **`AtomicCounterModifier`** requirements" to **`void`**, update the corresponding sample flow chart.

2 Introduction

Currently, there is little structural support to invoke multiple procedures concurrently in C++. Although we could use multiple call to **`std::async`** or use other facilities such as **`std::experimental::latch`** to control runtime concurrency and synchronization, there are certain limitations in usability, extendibility and performance. Based on the requirements in concurrent invocation, this proposal intends to add structural support in concurrent invocation.

With the support of the proposed library, not only are users able to structure concurrent programs like serial ones as flexible as function calls, but also to customize execution structure based on platform and performance considerations. The implementation for the library is available here.

3 Motivation and Scope

This section includes a typical concurrent programming scenario, leading to 6 aspects of limitations when designing concurrent programs with the facilities in the standard.

3.1 Limitations

Suppose it is required to make several (two or more; let's take "two" as an example) different library calls and save their return values for subsequent operations. The library APIs are defined as follows:

```
ResultTypeA call_library_a();
ResultTypeB call_library_b();
```

In order to increase performance, we may make the two function calls concurrently. With the utilities defined in the standard, we could use "**`std::thread`**", "**`std::async`**" or "**`std::latch`**" (Concurrency TS). For example, if "**`std::async`**" is used, the following code may be produced:

```
std::future<ResultTypeA> fa = std::async(call_library_a);
std::future<ResultTypeB> fb = std::async(call_library_b);
ResultTypeA ra = fa.get();
ResultTypeB rb = fb.get();
// Subsequent operations
```

In the code above, there could be performance limitations in different execution contexts. Concretely, there could be 6 aspects of performance limitations in blocking, execution resource management, synchronization, memory management, runtime extension and supporting async libraries.

3.1.1 Blocking

The sample code tries to obtain the result of the asynchronous calls via **`std::future::get()`**. However, this will

also block the calling thread and may reduce throughput of a system.

Additionally, we may turn to **std::experimental::when_all** and **std::experimental::future::then** to avoid blocking:

```
std::experimental::when_all(std::move(fa), std::move(fb)).then(
    [](auto&& f) {
      ResultTypeA ra = std::get<0u>(f.get()).get();
      ResultTypeB rb = std::get<1u>(f.get()).get();
      // Subsequent operations
    });
```

However, it requires more code, much runtime overhead (except for blocking), and potentially more difficulty in managing execution resources since the thread executing the continuation is *unspecified*.

Even if blocking caused by **std::future::get()** is rather acceptable than use a callback, there are many blocking synchronization primitives that may have better performance supported by various platforms, such as the "Futex" in modern Linux, the "Semaphore" defined in the POSIX standard and the "Event" in Windows. Besides, the "work-stealing" strategy is sometimes used in large-scale systems, such as the Click programming language, the "Fork/Join Framework" in the Java programming language and the "TLP" in the .NET framework.

3.1.2 Execution Resource Management

In the sample code for scenario 1, the two tasks are launched with **std::async** using default policy **std::launch::async | std::launch::deferred**. Behind the function, two concrete threads are created for the two tasks and will be destroyed when the tasks are completed.

This solution is more efficient than sequential calls of the two functions if there are abundant execution resources (e.g., CPU load is low) and the overhead in calling the functions is less than the that in creating new threads. However, in a high-concurrency system, "threads" are relatively "sensitive" resources because

1. creating and destroying threads usually involve system calls, which may block other system calls and cost much CPU time, and
2. too many running threads may increase management costs in an operating system and reduce throughput.

A solution to this issue is to use a more generic "Execution Agent" (e.g., thread pool) to control the total number of threads, as well as to avoid overhead in creating and destroying concrete threads. However, if it is required to use another execution agent other than creating a new thread to increase performance, **std::async** won't help and we may need to write similar code from scratch.

3.1.3 Synchronization

If **n** libraries are concurrently called with **std::async**, there will be a total number of **n** times of **std::future::get()**, introducing **n** times of "acquire-release" synchronization overhead. However, since the concurrent calls are only required to happen before the subsequent process, one "acquire" synchronization operation would be enough.

In the standard, there are four utilities that could efficiently perform such "many-to-one" synchronization: **std::experimental::latch**, **std::experimental::barrier**, **std::condition_varable** and

std::atomic, where the semantics of the former three ones are coupled with "blocking", not being able to be optimized.

3.1.4 Memory Management

With **std::async**, we are not able to specify the algorithm for memory allocation (e.g., with a customized allocator using a memory pool). This is usually not a major issue in performance, but it would be more flexible to allow such customization. In addition, the potential number of allocations in the sample code also has the potential to be reduced.

3.1.5 Runtime Extension

If one of the libraries in the motivating example may fork independent tasks at runtime that shall share a same synchronization point, the library is required to perform proper synchronization and return when all the subtasks are completed.

For example, when implementing a library for parallel quick-sort algorithm, we may not able to know the expected concurrency at the beginning of the algorithm, because the number is related to the order of the input data. Therefore, we may seek for more flexible facilities for concurrency control that support runtime extension. For example, the "Phaser" in the Java programming language [java.util.concurrent.Phaser] provides such mechanism, but similar facilities are missing in C++.

3.1.6 Supporting Async Libraries

Although one procedure occupies one execution agent in many cases, there are certain requirements where some procedures may cross multiple execution agents, and **std::async** will not work. For example, when a procedure involving async IO calls with a library like:

```
template <class ResponseHandler>
void async_socket_io(const socket_request& request, ResponseHandler&& handler)
```

Requires: **is_invocable_v<extending_t<ResponseHandler>, socket_response>** is **true**.

Effects: Execute the socket request and invoke the handler with the response data on an unspecified thread when the data is available.

Note: **socket_request** and **socket_response** represents the request and response data, respectively. **extending_t** is introduced in the "Extending Argument" library [P1648R0].

std::async will not work either, and more code is required to control synchronizations among execution agents.

3.2 The Solution

To implement with the proposed library for the same requirement in the previous section, that is to make different library calls and save their return values for subsequent operations, the following code could be acceptable:

```
// #1: Construct a value of thread_executor
```

```
thread_executor e;

// #2: Construct the Concurrent Invocation Unit
auto ciu = std::make_tuple(
  make_concurrent_callable(e, [](const contextual_data& cd) {
    cd.result_of_library_a = call_library_a();
  }),
  make_concurrent_callable(e, [](const contextual_data& cd) {
    cd.result_of_library_b = call_library_b();
  }));

// #3: Make invocation
std::future<contextual_data> f = concurrent_invoke(
    std::move(ciu), std::in_place_type<contextual_data>);

// #4: Block and get the stable context
contextual_data cd = f.get();

// Subsequent operations
```

The type **`contextual_data`** is defined as follows:

```
struct contextual_data {
  mutable ResultTypeA result_of_library_a;
  mutable ResultTypeB result_of_library_b;
};
```

The fields in the type are marked as **`mutable`** because it is guaranteed that each of them will not be accessed concurrently, even if the two fields may be accessed at the same time.

On step **`#1`**, a value of **`thread_executor`** was constructed, which is a proposed Oneway Executor, providing asynchronization.

With the executor, we could construct a "Concurrent Invocation Unit" (CIU) on step **`#2`**. A CIU could either be an asynchronous callable type or an aggregation (container or tuple) of CIU. The function template **`make_concurrent_callable`** is a helper function in the proposed library that constructs an asynchronous callable object with a Oneway executor and a callable object.

On step **`#3`**, the concurrent invocation is performed with a proposed function template **`concurrent_invoke`**. Note that the second argument **`std::in_place_type<contextual_data>`** is an "Extending Argument" [P1648R0], and is equivalent to **`contextual_data{}`** but will only construct the value before its first usage and do not require the type to be copy/move constructible.

3.2.1 Avoiding Blocking

Blocking is usually harmful to throughput in performance critical scenarios. To avoid blocking with the proposed library, we could add a third argument to the function template **`concurrent_invoke`** indicating a callback function.

For example:

```
auto cb = make_concurrent_callback(
    thread_executor{}, [](contextual_data&& data) { /*Subsequent operations*/ });

concurrent_invoke(std::move(ciu), std::in_place_type<contextual_data>,
    std::move(cb));
```

In the code above, **`make_concurrent_callback`** is a helper function template proposed in the library to construct callback for concurrent invocation. A new thread will be created for the continuation. Note that the third argument **`concurrent_invoke`** is also an extending argument like the second one.

3.2.2 Managing Execution Resources

In the sample code with the proposed library, **`thread_executor`** was used for asynchronization. However, thread could be expensive execution resources in high performance service, and frequently creating and destroying threads may increase system overhead. Therefore, the "thread pool" was invented to reuse execution resources in different context.

To apply different management strategy for execution resource for various needs, we could construct different Oneway Executors other than **`thread_executor`**. For example, the class **`static_thread_pool::executor_type`** proposed in the "Executors" library [P0443R10].

3.2.3 Exploring Synchronization

Too many synchronization operations is harmful for performance. If **`n`** libraries are called with **`std::async`**, there will be a total number of **`n`** times of full acquire-release synchronizations, whereas **`n`** times of release synchronization and only one acquire synchronization operations are required for **`concurrent_invocation`**. Therefore, the synchronization overhead for the proposed library, for a same concurrency requirement, could be no higher than **`std::experimental::latch`**, while providing non-blocking mechanism.

3.2.4 Customizing Memory Management

The function template overloads **`concurrent_invoke`** accepts one to four arguments, representing the CIU, the context, the callback and the memory allocator, respectively. The second, third and fourth argument are all extending argument. **`global_memory_allocator{}`** [P1172R1] is the default fourth argument that allocates global memory, and could be replaced with any type that meets specific requirements.

3.2.5 Exploring Runtime Extension

If runtime extension is required for some procedures, we could change the parameter type of the CIU from the "contextual data type" into a **`token`**. For example, the expression in second step of the sample implementation:

```
make_concurrent_callable(e, [](const contextual_data& cd) {
```

```
  cd.result_of_library_b = call_library_b();
}
```

is equivalent to:

```
make_concurrent_callable(e, [](auto&& token) {
  token.context().result_of_library_b = call_library_b();
}
```

Note that the proposed library will try to invoke the input callable object with the **`token`**; if it is not invocable with a **`token`**, the library will try to invoke with **`token.context()`**; if it is also not invocable with the result of **`token.context()`**, the library will try to invoke with no arguments; if it is still not invocable, the expression is ill-formed.

With the **`token`**, we will be able to do more things than performing operations on the context. One of the coolest things is to "fork" the current procedure with another CIU, and the CIU will share a same concurrent invocation with the current procedure as if it were a part of the original CIU for the initial concurrent invocation. This technique is useful when the concurrency is only known at runtime.

3.2.6 Supporting Async Libraries

When working with asynchronous libraries, it usually requires more engineering effort to control concurrency and synchronization. There are little facilities in C++ that we could use directly for such requirements. In the Java programming language, method **`thenCompose(Function<? super T,? extends CompletionStage<U>> fn)`** in the interface **`java.util.concurrent.CompletionStage<T>`** provides such mechanism. However, not only could it fragment the program, reducing readability, but also tightly couples to the **`Future`** mechanism, reducing performance.

Async libraries are easily supported with the proposed library in a concurrent invocation, because the end of a procedure is not defined as the last line of the callable code, but the destruction of the **`token`**. For example, if we need to call the async library mentioned in the "Limitations in Supporting Async Libraries":

```
template <class ResponseHandler>
void async_socket_io(const socket_request& request, ResponseHandler&& handler);
```

we could include the **`token`** as a part of the response handler:

```
[](auto&& token) {
  // ...
  async_socket_io(
      /* some request */,
      [token = std::move(token)](socket_response) { /* ... */ });
  // ...
};
```

This feature also provides convenience to perform asynchronous and recursive concurrent invocation.

4 Impact on the Standard

Utility	From	Comments
Oneway Executors	P0443R10	It is recommended to use Oneway Executors to build a "Concurrent Invocation".
Extending Argument	P1648R0	To simplify the API of the library and do not decrease usability, the semantics of several arguments in the library are designed to be extending argument.
Applicable Template	P1649R0	As there is compile-time routing in the library, the "Applicable Template" library could help in implementation.
Memory Allocator	P1172R1	This solution support customization in allocation but does not support fancy pointers.

Figure 1 – Related Work

Figure 1 includes the related work in the standard and the TSes.

5 Design Decisions

5.1 Execution 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 "Invocation" are the basic concepts of programming, enabling users to wrap their logic into units and decoupling every part from the entire program. This solution generalizes these concepts in concurrent programming.

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 "Invocation"; in other words, a "Concurrent Invocation" shall begin from "one" and end up to "one", which forms a "one-to-many-to-one" synchronization.

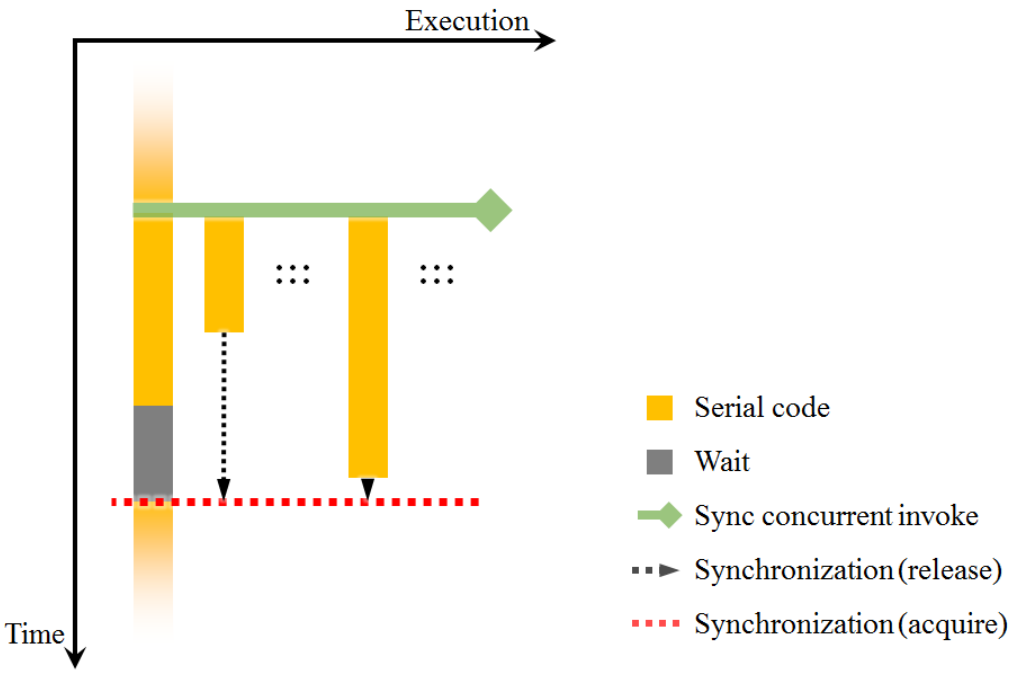

Figure 2

The most common concurrent model is starting several independent tasks and waiting for their completion. This is the basic model for "Concurrent Invoke", and typical scenario is shown in Figure 2.

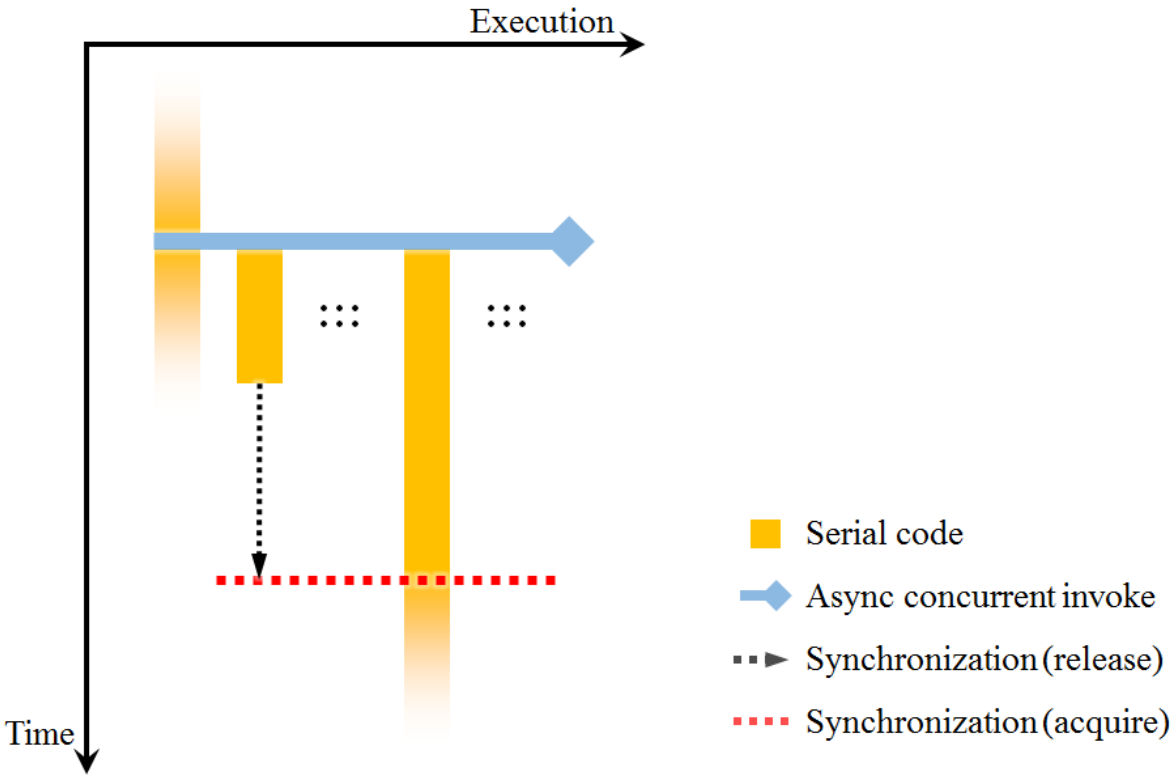

Figure 3

Turning blocking program into non-blocking ones is a common way to break bottleneck in throughput. We could let the execution agent that executes the last finished task in a concurrent invocation 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*). A typical scenario is shown in Figure 3.

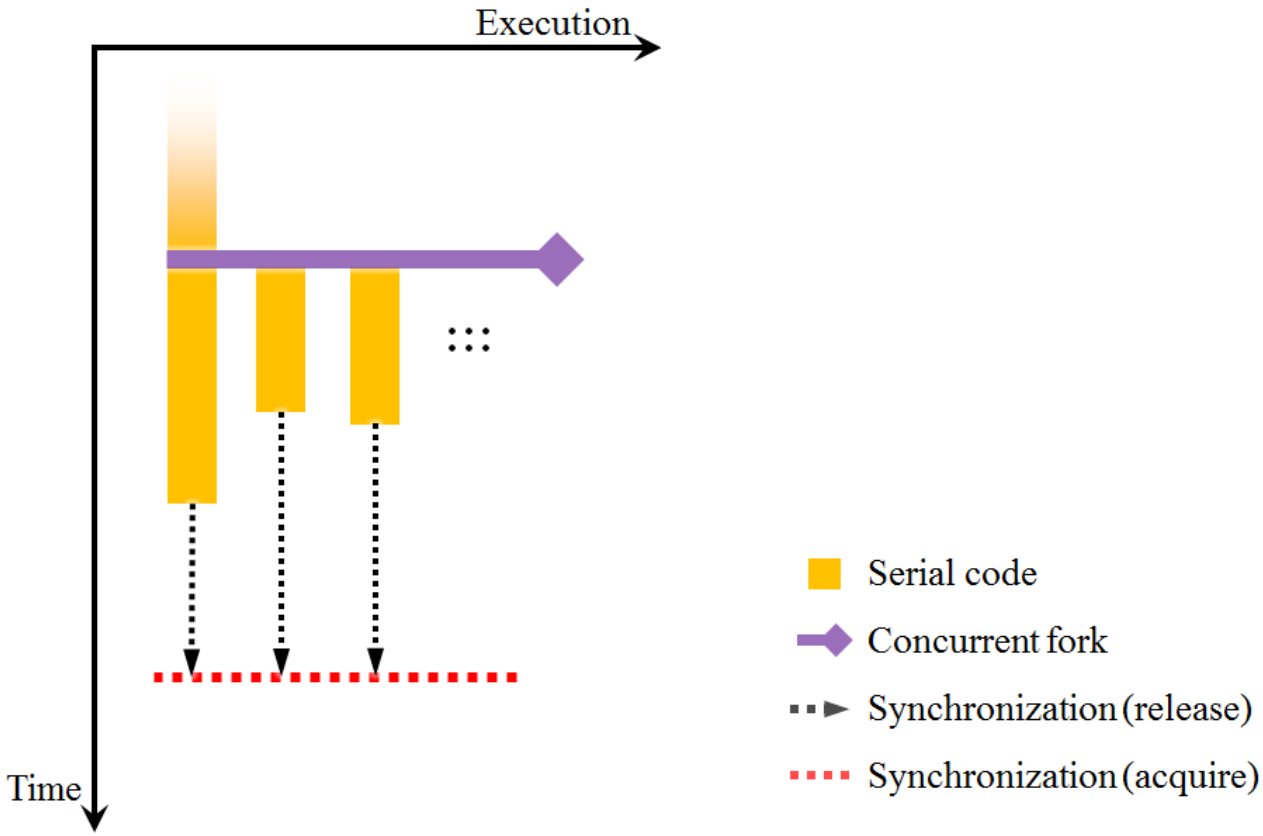

Figure 4

The "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. Therefore, 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 4.

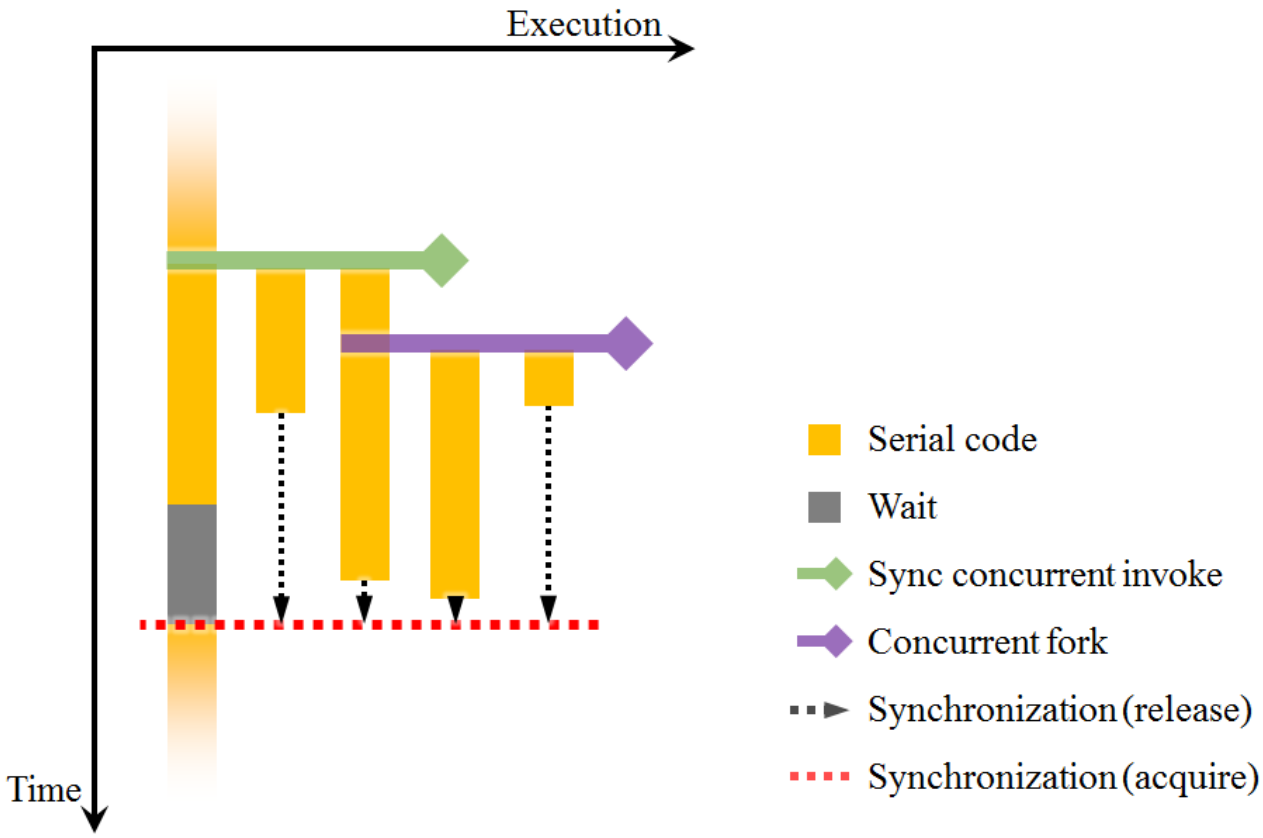

Figure 5

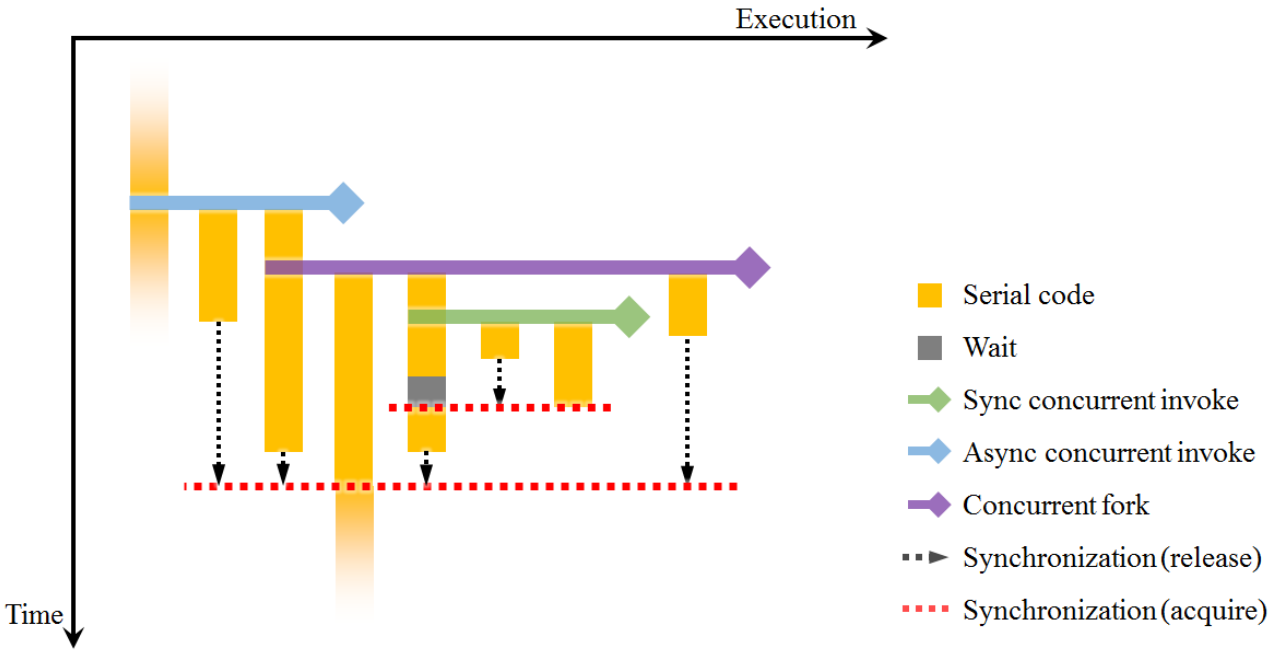

Figure 6

With the concept of the "Concurrent Invoke" the "Concurrent Fork" models, we can easily build concurrent programs with complex dependencies among the executions, meanwhile, stay the concurrent logic clear. Figure 5 shows a typical

scenario for a composition of the " Concurrent Invoke" and the "Concurrent Fork" models; Figure 6 shows a more complicated scenario.

From the "Concurrent Invoke" and the "Concurrent Fork" models, we can tell that:

- the same as serial invocations, the "Concurrent Invoke" models can be applied recursively, and
- applying the "Concurrent Fork" model requires one existing concurrent invocation to expand.

5.2 Binary Semaphores VS Futures

In previous revisions of this proposal, the concept "Binary Semaphore" was introduced as an abstraction for the Ad-hoc synchronizations required in the "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.

However, I found there could be more requirements in blocking:

- sometimes blocking could be tolerated to reduce engineering cost, but we may also need timing mechanism to ensure the stability of the entire system, which will make it the concept more complicated and the lifetime of the context shall be extended until every procedures in the concurrent invocation has finished, and
- If we perform blocking as we invoke the library, the CIU will not be destroyed until blocking is released automatically or due to timeouts, etc. On the one hand, it may be good for performance because async procedures may reuse the resources on the call stack of the calling thread without copying them. However, on the other hand, if the CIU could be destroyed in time, not only could the resources be released, but we will be able to submit tasks to some execution agents in batch, when the executor is destroyed, to reduce the number of critical section and increase performance. After all, it is convenient to manage all the contextual resources in the "concurrent context" if necessary.

Therefore, the concept of "Binary Semaphore" was removed from this revision based on the considerations above, and only provide a "future" version even if it does not have potentially the highest performance. Meanwhile, the proposed library retained the extendibility to customize any blocking mechanism with a "callback" such as platform-specific primitives or the "work-stealing" strategy, etc. if necessary.

5.3 Polymorphism VS Compile-time Routing

I have tried many ways to design the API for the concurrent invocation library, and I once thought that polymorphism could be the best solution for engineering experience, and that was my original motivation for the PFA [P0957R2]. However, after exploring more in metaprogramming, I found proper compile-time routing has more usability and zero runtime overhead comparing to polymorphism. Therefore, the PFA was separated from this paper from revision 1.

Here is a part of a deprecated design for concurrent invocation:

```
template <class F = /* A polymorphic wrapper */, class C = std::vector<F>>
class concurrent_invoker {
```

```
 public:
  template <class _F>
  void attach(_F&& f);

  template <class T>
  void invoke(T&& context);
};
```

The class template **concurrent_invoker** holds a container for the procedures to be invoked concurrently. However, it may introduce extra runtime overhead since **F** is a (default or customized) polymorphic wrapper, and an extra variable-size container will be constructed even if the concurrency could be determined at compile-time.

In order not to introduce extra runtime overhead and retain usability, the concept of "CIU" (Concurrent Invocation Unit) was proposed with recursive semantics, and I designed another library for compile-time routing [P1649R0] as an infrastructure for the proposed library.

5.4 Variable Parameter VS Single Parameter

There are many variable parameter function templates in the standard providing the mechanism for in-place construction. Actually, I found in-place construction is indispensable in concurrent programming, especially for the contexts including concurrent data structure such as concurrent queue or map, which are usually not move constructible at all.

However, since there could be many parameters for a function template with different semantics, it becomes difficult to let all of them have the potential for in-place construction. Therefore, a separate "Extending Argument" library [P1648R0] is designed as a generic solution for lifetime extension and in-place construction, making it possible to in-place-construct any argument in a function template without variable parameter and retain clear semantics.

5.5 Allocator VS Memory Allocator

Allocator is a widely used facility in C++ as an abstraction for memory management. However, it does not work well in self-referencing or type-erased context. For example, it is not feasible to allocate a value of a type template specified by the allocator:

```
template <class T, class Alloc>
class foo;
```

We are not able to instantiate the class template **std::allocator** with **foo** specified by the allocator itself because infinite template recursion is not supported (e.g., **std::allocator<foo<T, std::allocator</* infinite recursion */>>>**). Although the issue could be mitigated if infinite template recursion could be supported with another proposal of mine, recursive template instantiation [P1230R0], it makes the semantics complicated so that have limitations in extendibility. Therefore, the concept of "Memory Allocator" [P1172R1] was proposed not only for recursive scenarios, but also for type-erasure [P0957R2].

6 Technical Specifications

6.1 Header <concurrent_invocation> synopsis

```
namespace std {

template <class CTX, class E_CB, class MA>
class concurrent_breakpoint;

class invalid_concurrent_breakpoint;

template <class CTX, class E_CB, class MA = global_memory_allocator>
class concurrent_token;

template <class CTX, class E_CB, class MA = global_memory_allocator>
class concurrent_finalizer;

class thread_executor;

template <class CTX>
class promise_callback;

template <>
class promise_callback<void>;

template <class E_F, class CTX, class E_CB, class MA>
class contextual_concurrent_callable;

template <class E_E, class E_F>
class concurrent_callable;

template <class E_CT, class CTX, class E_CB, class MA>
class contextual_concurrent_callback;

template <class E_E, class E_CT>
class concurrent_callback;

template <class E_E, class E_F>
auto make_concurrent_callable(E_E&& e, E_F&& f);

template <class E_E, class E_CT>
auto make_concurrent_callback(E_E&& e, E_CT&& ct);
```

```
template <class CIU, class E_CTX, class E_CB,
    class E_MA = global_memory_allocator>
void concurrent_invoke(CIU&&, E_CTX&&, E_CB&&, E_MA&& = E_MA{});

template <class CIU, class E_CTX = in_place_type_t<void>>
decltype(auto) concurrent_invoke(CIU&& ciu, E_CTX&& ctx = E_CTX{});

}
```

6.2 `ConcurrentInvocationUnit` requirements

A type meets the ConcurrentInvocationUnit requirements of specific types **`CTX`**, **`E_CB`**, **`MA`** if

- it is invocable with **`concurrent_token<CTX, E_CB, MA>`**, or
- it is a "Generic Tuple" or a "Generic Container" of types meeting the **`ConcurrentInvocationUnit`** requirements of **`CTX`**, **`E_CB`**, **`MA`**.

A "Generic Tuple" is an instantiation of **`std::tuple`**, **`std::pair`** or **`std::array`**. A "Generic Container" is the type of any range expression that is iterable with a range-based **`for`** statement.

6.3 Core Types

There are three core class templates and one core class in this library.

1. Class template **`concurrent_breakpoint`** is the data structure for concurrent invocation and has unspecified constructors.
2. Class **`invalid_concurrent_breakpoint`** inherits from **`std::logic_error`** and is the exception thrown when **`concurrent_token`** or **`concurrent_finalizer`** is not associated to a valid **`concurrent_breakpoint`**.
3. Class template **`concurrent_token`** is the facility for CIUs to fork/join the concurrent invocation.
4. Class template **`concurrent_finalizer`** is the facility for Concurrent Callback to access the context and clean up.

6.3.1 Class template `concurrent_token`

Any well-formed instantiation for **`concurrent_token`** is default-constructible, move-constructible and move-assignable. The default constructor will construct a value of **`concurrent_token`** associated with no concurrent invocation.

```
template <class CTX, class E_CB, class MA>
class concurrent_token {
 public:
  concurrent_token();
  concurrent_token(concurrent_token&&);
```

```
  ~concurrent_token();
  concurrent_token& operator=(concurrent_token&&);

  template <class CIU> void fork(CIU&& ciu) const;
  decltype(auto) context() const;
};
```

`~concurrent_token();`

Effects: Join the current procedure to the concurrent invocation if the `*this` associates to a valid value of `concurrent_breakpoint<CTX, E_CB, MA>`, and destroy `*this`. The last join operation will trigger the execution of the callback.

```
template <class CIU>
void fork(CIU&& ciu) const;
```

Requires: `*this` is associated with a concurrent invocation, and `CIU` shall meet the `ConcurrentInvocationUnit` requirements of `CTX`, `E_CB` and `MA`.

Effects: Fork the concurrent invocation associated to `*this` with the provided `ciu`.

```
decltype(auto) context() const;
```

Requires: `*this` is associated with a concurrent invocation.

Return Type: `const CTX&` if `is_void_v<CTX>` is `false`, otherwise, `void`.

Returns: A const reference of the context associated to the concurrent invocation if `is_void_v<CTX>` is `false`, otherwise, nothing.

6.3.2 Class template `concurrent_finalizer`

Any well-formed instantiation for `concurrent_finalizer` is default-constructible, move-constructible and move-assignable. The default constructor will construct a value of `concurrent_finalizer` associated with no concurrent invocation.

```
template <class CTX, class E_CB, class MA>
class concurrent_finalizer {
 public:
  concurrent_finalizer();
  concurrent_finalizer(concurrent_finalizer&&);
  ~concurrent_finalizer();
  concurrent_finalizer& operator=(concurrent_finalizer&&);

  decltype(auto) context() const;
};
```

`~concurrent_finalizer();`

Effects: Destroy corresponding the data structure of the concurrent invocation if the `*this` associates to a valid value of `concurrent_breakpoint<CTX, E_CB, MA>`, and destroy `*this`.

`decltype(auto) context() const;`

Requires: `*this` is associated with a concurrent invocation.

Return Type: `CTX&` if `is_void_v<CTX>` is `false`, otherwise, `void`.

Returns: A reference of the context associated to the concurrent invocation if `is_void_v<CTX>` is `false`, otherwise, nothing.

6.4 Helper Utilities

Helper utilities are not required for every usage for this library but has the potential for improving engineering experience with concurrent invocation.

6.4.1 Class thread_executor

The class `thread_executor` meets the `OneWayExecutor` requirements [P0443R10], and is default-constructible and copy-constructible.

```
class thread_executor {
 public:
  template <class F>
  void execute(F&& f) const;
};

template <class F>
void execute(F&& f) const;
```

Requires: `F` shall be a callable type with no arguments.

Effects: Create a new thread to execute `f`. The program shall not exit before completion of the execution.

6.4.2 Class template `promise_callback`

The class template `promise_callback` is a helper for the function templates overloads `concurrent_invoke` with one or two parameters. Any well-formed instantiation for `promise_callback` has unspecified constructors.

```
template <class CTX>
class promise_callback {
 public:
  template <class E_CB, class MA>
  void operator()(concurrent_finalizer<CTX, E_CB, MA>&& finalizer) &&;
};

template <>
class promise_callback<void> {
```

```
 public:
  template <class CTX, class E_CB, class MA>
  void operator()(concurrent_finalizer<CTX, E_CB, MA>&&) &&;
};
```

6.4.3 Helper for CIU

There are two class templates and a function template that helps creating asynchronous CIUs with a Oneway Executor and a callable value.

```
template <class E_F, class CTX, class E_CB, class MA>
class contextual_concurrent_callable {
 public:
  contextual_concurrent_callable(contextual_concurrent_callable&&);
  contextual_concurrent_callable& operator=(contextual_concurrent_callable&&);
  void operator()() &&;
};
```

Any well-formed instantiation for **contextual_concurrent_callable** is move-constructible and move-assignable. It may associate with a callable value **f** of type **E_F** and a value **token** of type **concurrent_token<CTX, E_CB, MA>** associating with a concurrent invocation.

Invoking an rvalue reference of **contextual_concurrent_callable<E_F, CTX, E_CB, MA>** will perform **make_extended_view(std::forward<E_F>(f))**, and invoke the return value **nf** of type **F**:

- If **std::is_invocable_v<F, concurrent_token<CTX, E_CB, MA>>** is **true**, perform **std::invoke(nf, std::move(token))**, or
- If **std::is_invocable_v<F, std::add_lvalue_reference_t<const CTX>>** is **true**, perform **std::invoke(nf, token.context())** and destroy **token**, or
- If **std::is_invocable_v<F>** is **true**, perform **std::invoke(nf)** and destroy **token**, or
- Otherwise, the expression is ill-formed.

```
template <class E_E, class E_F>
class concurrent_callable {
 public:
  template <class _E_E, class _E_F>
  explicit concurrent_callable(_E_E&& e, _E_F&& f);
  concurrent_callable(const concurrent_callable&);
  concurrent_callable(concurrent_callable&&);
  concurrent_callable& operator=(const concurrent_callable&);
  concurrent_callable& operator=(concurrent_callable&&);

  template <class CTX, class E_CB, class MA>
  void operator()(concurrent_token<CTX, E_CB, MA>&& token) &&;
};
```

Any well-formed instantiation for **concurrent_callable** is copy-constructible, copy-assignable, move-constructible and move-assignable. It associates with a value **e** of type **E_E** and a value **f** of type **E_F**. The type **extending_t<E_E>** shall meet the **OneWayExecutor** requirements.

Invoking an rvalue reference of **concurrent_callable<E_E, E_F>** will perform **make_extended_view(std::forward<E_E>(e)).execute(contextual_callable)**, where **contextual_callable** is a value of type **contextual_concurrent_callable<E_F, CTX, E_CB, MA>**.

```
template <class E_E, class E_F>
auto make_concurrent_callable(E_E&& e, E_F&& f);
```

Effects: Construct a value of type **concurrent_callable<decay_t<E_E>, decay_t<E_F>>**.

Returns: **concurrent_callable<decay_t<E_E>, decay_t<E_F>>(forward<E_E>(e), forward<E_F>(f))**.

6.4.4 Helper for Concurrent Callback

Like the CIU, there are two class templates and a function template that helps creating asynchronous callback with a Oneway Executor and a callable continuation.

```
template <class E_CT, class CTX, class E_CB, class MA>
class contextual_concurrent_callback {
 public:
  contextual_concurrent_callback(contextual_concurrent_callback&&);
  contextual_concurrent_callback& operator=(contextual_concurrent_callback&&);
  void operator()() &&;
};
```

Any well-formed instantiation for **contextual_concurrent_callback** is move-constructible and move-assignable. It may associate with a callable value **ct** of type **E_CT** and a value **finalizer** of type **concurrent_finalizer<CTX, E_CB, MA>** associating with a concurrent invocation.

Invoking an rvalue reference of **contextual_concurrent_callback<E_CT, CTX, E_CB, MA>** will perform **make_extended_view(std::forward<E_CT>(ct))**, and invoke the return value **nct** of type **CT**:

- If **std::is_invocable_v<CT, concurrent_finalizer<CTX, E_CB, MA>>** is **true**, perform **std::invoke(nct, std::move(finalizer))**, or
- If **std::is_invocable_v<CT, CTX>** is **true**, perform **std::invoke(nct, std::forward<CTX>(finalizer.context()))** and destroy **finalizer**, or
- If **std::is_invocable_v<CT>** is **true**, perform **std::invoke(nct)** and destroy **finalizer**, or
- Otherwise, the expression is ill-formed.

```
template <class E_E, class E_CT>
class concurrent_callback {
 public:
  template <class _E_E, class _E_CT>
  explicit concurrent_callback(_E_E&& e, _E_CT&& ct);
  concurrent_callback(const concurrent_callback&);
```

```
  concurrent_callback(concurrent_callback&&);
  concurrent_callback& operator=(const concurrent_callback&);
  concurrent_callback& operator=(concurrent_callback&&);

  template <class CTX, class E_CB, class MA>
  void operator()(concurrent_finalizer<CTX, E_CB, MA>&& finalizer) &&;
};
```

Any well-formed instantiation for **concurrent_callback** is copy-constructible, copy-assignable, move-constructible and move-assignable. It associates with a value **e** of type **E_E** and a value **ct** of type **E_CT**. The type **extending_t<E_E>** shall meet the **OneWayExecutor** requirements.

Invoking an rvalue reference of **concurrent_callback<E_E, E_CT>** will perform **make_extended_view(std::forward<E_E>(e)).execute(contextual_callback)**, where **contextual_callback** is a value of type **contextual_concurrent_callback<E_CT, CTX, E_CB, MA>**.

```
template <class E_E, class E_CT>
auto make_concurrent_callback(E_E&& e, E_CT&& ct);
```

Effects: Construct a value of type **concurrent_callback<decay_t<E_E>, decay_t<E_CT>>**.

Returns: **concurrent_callback<decay_t<E_E>, decay_t<E_CT>>(forward<E_E>(e), forward<E_CT>(ct))**.

6.5 Function templates concurrent_invoke

```
template <class CIU, class E_CTX, class E_CB, class E_MA>
void concurrent_invoke(CIU&& ciu, E_CTX&& ctx, E_CB&& cb, E_MA&& ma);
```

Requires: Type **CIU** meets the **ConcurrentInvocationUnit** requirements of **extending_t<E_CTX>**, **E_CB** and **extending_t<MA>**. Type **extending<E_CB>** shall be a callable type with **concurrent_finalizer<extending_t<E_CTX>, E_CB, MA>**. Type **extending<MA>** shall meet the **MemeoryAllocator** requirements [P1172R1] of **sizeof(concurrent_breakpoint<extending_t<E_CTX>, E_CB, extending_t<MA>>)** and **alignof(concurrent_breakpoint<extending_t<E_CTX>, E_CB, extending_t<MA>>)**.

Effects: Construct a value of **concurrent_breakpoint<extending_t<E_CTX>, E_CB, extending_t<MA>>** and perform concurrent invocation with **ciu** on the breakpoint.

```
template <class CIU, class E_CTX = in_place_type_t<void>>
decltype(auto) concurrent_invoke(CIU&& ciu, E_CTX&& ctx = E_CTX{});
```

Effects: Equivalent to

```
    using R = conditional_t<
       is_move_constructible_v<extending_t<E_CTX>>, extending_t<E_CTX>, void>;
    promise<R> p;
    future<R> result = p.get_future();
    concurrent_invoke(forward<CIU>(ciu), forward<E_CTX>(ctx),
       promise_callback<R>{move(p)});
    return result;
```

Return type: **`std::future<extending_t<CTX>>`** if **`extending_t<CTX>`** is move-constructible, or **`std::future<void>`** otherwise.

Returns: A value of **`std::future`** that will be available after the concurrent invocation has finished.