The Concurrent Invocation Library

Document number: P0642R3
Date: 2019-10-06
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 P0642R2

- Add exception support for **`concurrent_invoke`**;
- Remove **`const`** qualifier when accessing the contextual data concurrently;
- Rename the "**`ConcurrentInvocationUnit`**" requirements to "**`ConcurrentSessionAggregation`**";
- Add the class template **`concurrent_invocation_error`**;
- Add the class template **`concurrent_breakpoint`** as the exposed data structure for concurrent invocation;
- Rename the member function **`fork`** of **`concurrent_token`** to **`spawn`**;
- Move member functions **`spawn`** and **`context`** from **`concurrent_token`** to **`concurrent_breakpoint`**;
- Remove class template **`concurrent_finalizer`**;
- Rename the class template **`concurrent_callable`** to **`serial_concurrent_session`**;
- Rename the class template **`contextual_concurrent_callable`** to **`concurrent_callable`**;
- Add support for non-moveable but reducible context type;
- Add error channel for the class template **`async_concurrent_continuation`**;
- Make **`concurrent_invoke`** internally blocking rather than returning a value of **`std::future`**;
- Temporarily remove the class **`thread_executor`** because it is currently unimplementable as a pure library with complete semantics on any platform I know, thanks to Billy O'Neal.

1.2 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.

1.3 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::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 is intended 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 limitations in different execution contexts. Concretely, there could be 6 aspects of limitations in blocking, execution resource management, exception handling, runtime extension, synchronization 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 Exception Handling

It usually requires a lot of effort to handle exceptions appropriately across concurrent contexts, because C++ only support automatically propagating exception in serial code. Although **`std::exception_ptr`** allows us to store an exception and replay it across contexts, a standard way to handle exceptions across concurrent context would largely simplify the implementation for concurrent algorithms.

3.1.4 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++.

3.1.5 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.

3.1.6 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<sink_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. **sink_t** is introduced in the "Sinking Argument" library [P1648R2].

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 an executor
thread_executor e;

// #2: Construct the Concurrent Session Aggregation
auto csa = std::tuple{
  serial_concurrent_session{e, [](contextual_data& cd) {
    cd.result_of_library_a = call_library_a();
  }},
  serial_concurrent_session{e, [](contextual_data& cd) {
    cd.result_of_library_b = call_library_b();
  }}};

// #3: Make invocation and block
contextual_data result = concurrent_invoke(
    std::move(csa), std::in_place_type<contextual_data>);

// Subsequent operations
```

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

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

On step **#1**, a value of **thread_executor** was constructed, which is some implementation of the Executor, providing asynchronization.

With the executor, we could construct a "Concurrent Session Aggregation" (CSA) on step **#2**. A CSA could either be a type meeting the **ConcurrentSession** requirements (defined in the technical specifications) or an aggregation (container or tuple) of CSA. The class template **serial_concurrent_session** is a helper class in the proposed library that constructs a concurrent session with an 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 "Sink Argument" [P1648R2], 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 continuation. For example:

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

concurrent_invoke(std::move(csa), std::in_place_type<contextual_data>,
    std::move(ct));
```

In the code above, **async_concurrent_continuation** is a helper class template proposed in the library to construct continuation for concurrent invocation. A new thread is expected to be created for the continuation in the sample code.

3.2.2 Managing Execution Resources

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. For example, the class **static_thread_pool::executor_type** proposed in the "Executors" library [P0443R10].

3.2.3 Exception Handling

The proposed library provides a comprehensive mechanism for exception handling during concurrent invocation, no matter invoking synchronously or asynchronously. For instance, the concurrent invocation will collect the exceptions propagated from each concurrent session and propagate all the exceptions out of the context when the invocation is done.

Synchronous concurrent invocation propagates the caught exception by throwing a nested exception, which could be caught with a try-catch block:

```
try {
  contextual_data result = concurrent_invoke(
      std::move(csa), std::in_place_type<contextual_data>);
} catch (const concurrent_invocation_error<>& ex) {
  for (auto& ep : ex.get_nested()) {
    // ...
  }
}
```

Asynchronous concurrent invocation propagates the caught exceptions with the error channel in the continuation, which is required by design and optional for the class template **`async_concurrent_continuation`**. For example:

```
auto continuation = async_concurrent_continuation{
    thread_executor{},
    []() { /* "Normal control flow... */ }, [](auto&& exceptions) {
  for (auto& ep : exceptions) {
    // ...
  }
}};
```

3.2.4 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.

3.2.5 Exploring Runtime Extension

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

```
serial_concurrent_session{e, [](const contextual_data& cd) {
  cd.result_of_library_b = call_library_b();
}
```

is equivalent to:

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

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

With the **`breakpoint`**, we will be able to do more things than performing operations on the context. One of the coolest things is to "spawn" the current session with another CSA, and the CSA will share a same concurrent invocation with the current session as if it were a part of the original CSA 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

This design is a pure library extension, depends on another ongoing proposal for sink argument [P1648R2]. It also has the potential to be used to implement the parallel algorithms in Parallel TS with expected performance.

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.

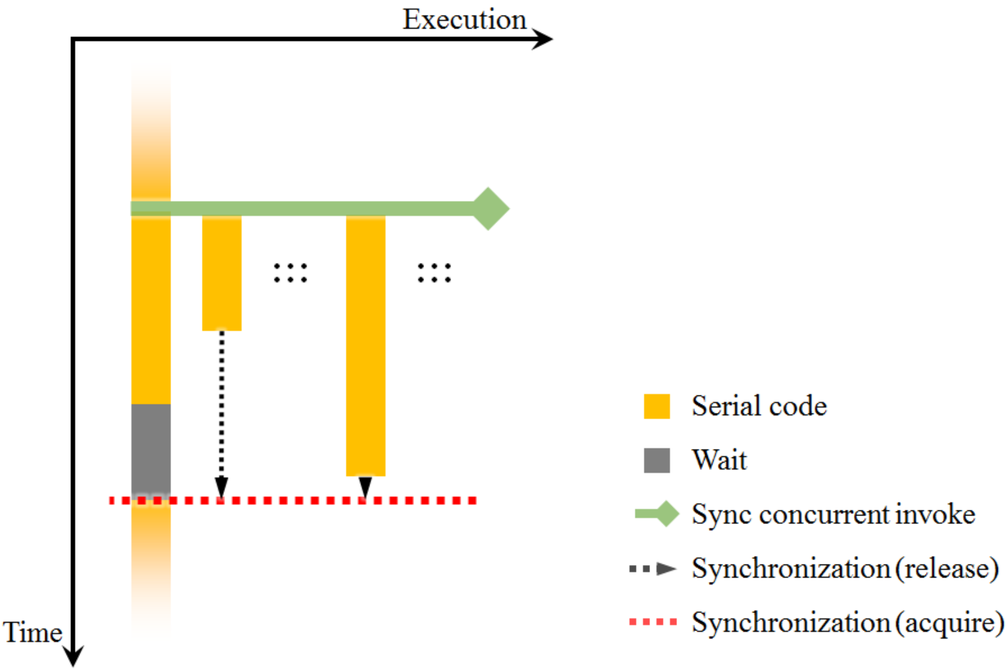

Figure 1

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 1.

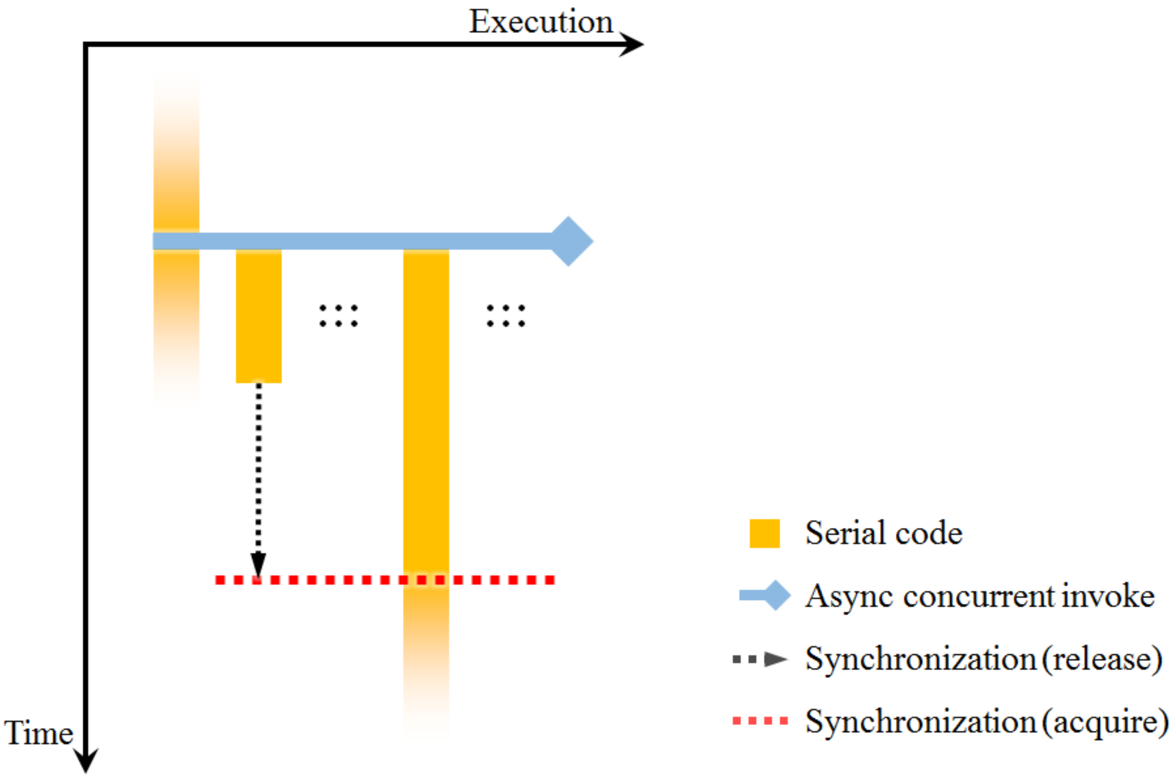

Figure 2

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 2.

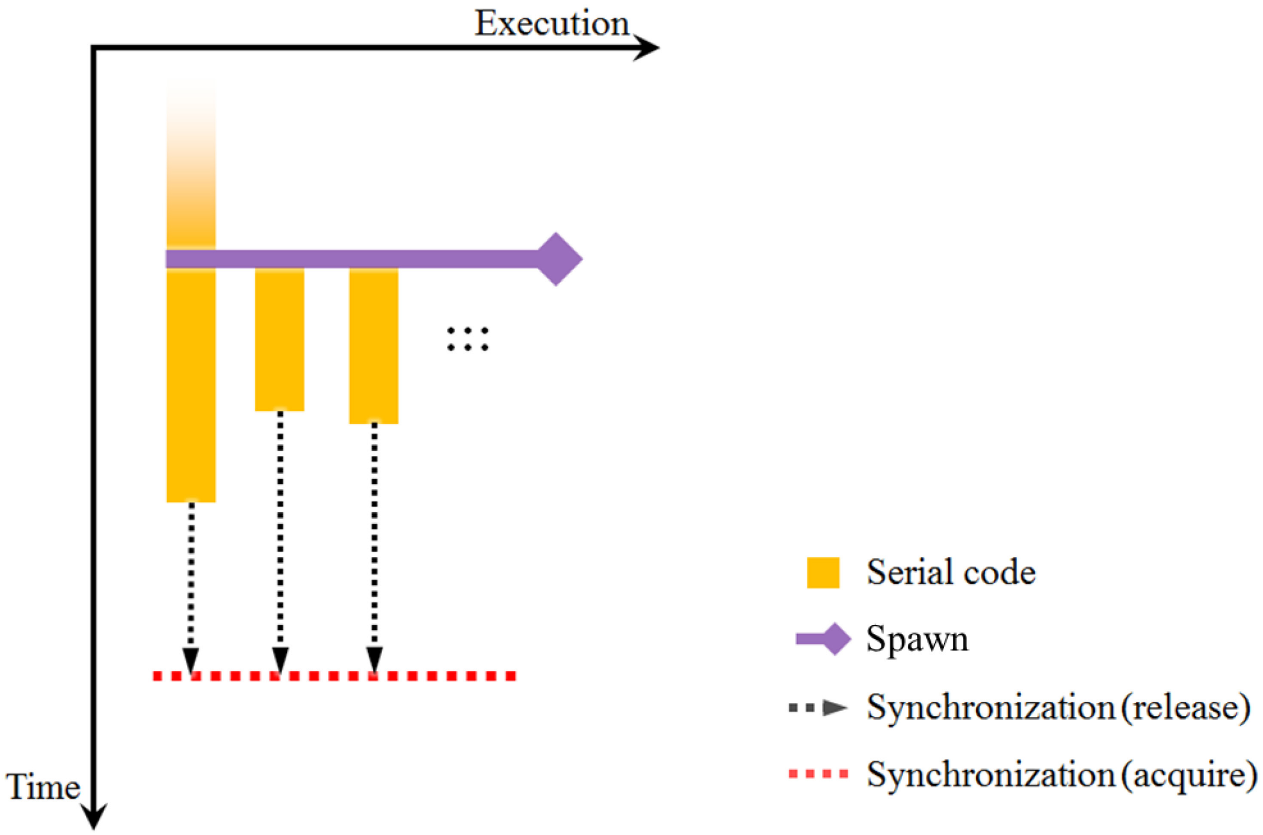

Figure 3

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 "**Spawn**" (previously "Concurrent Fork"). A typical scenario for the "Spawn" model is shown in Figure 3.

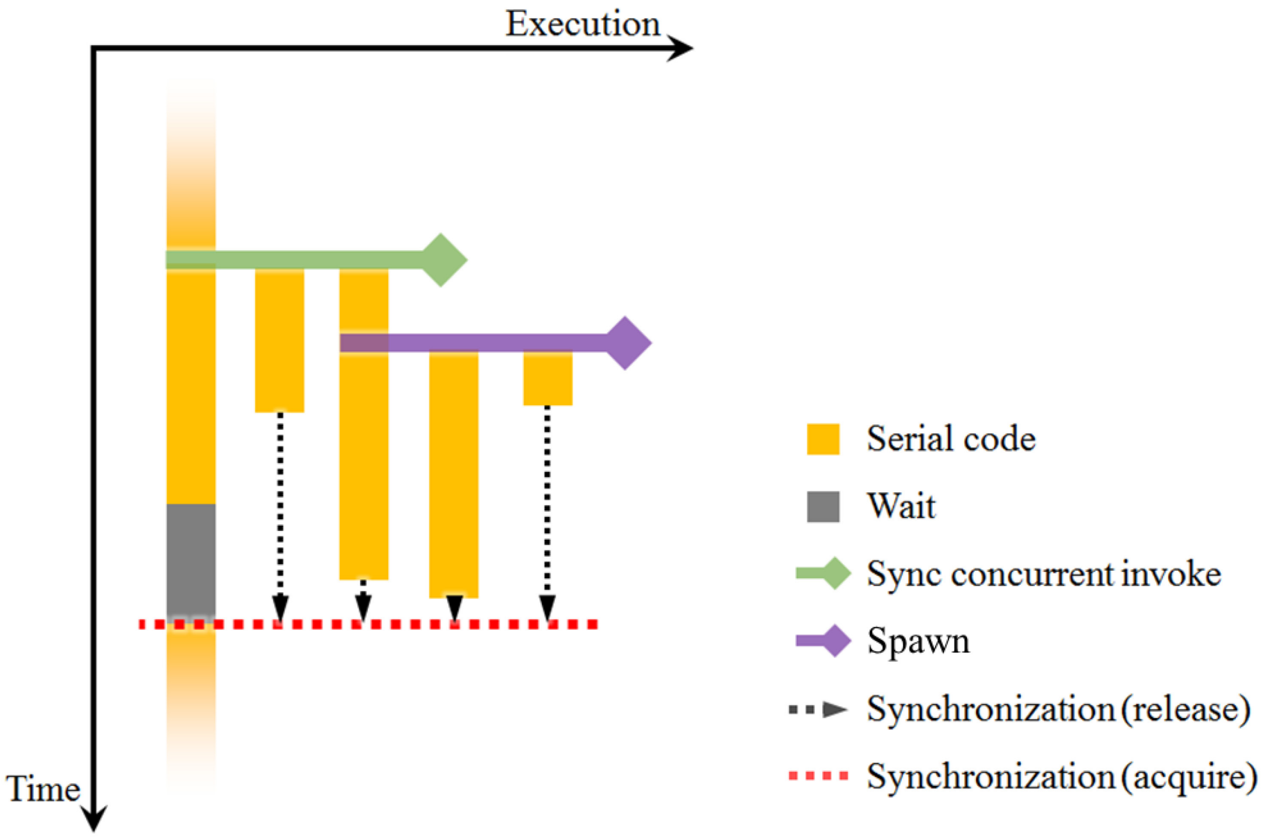

Figure 4

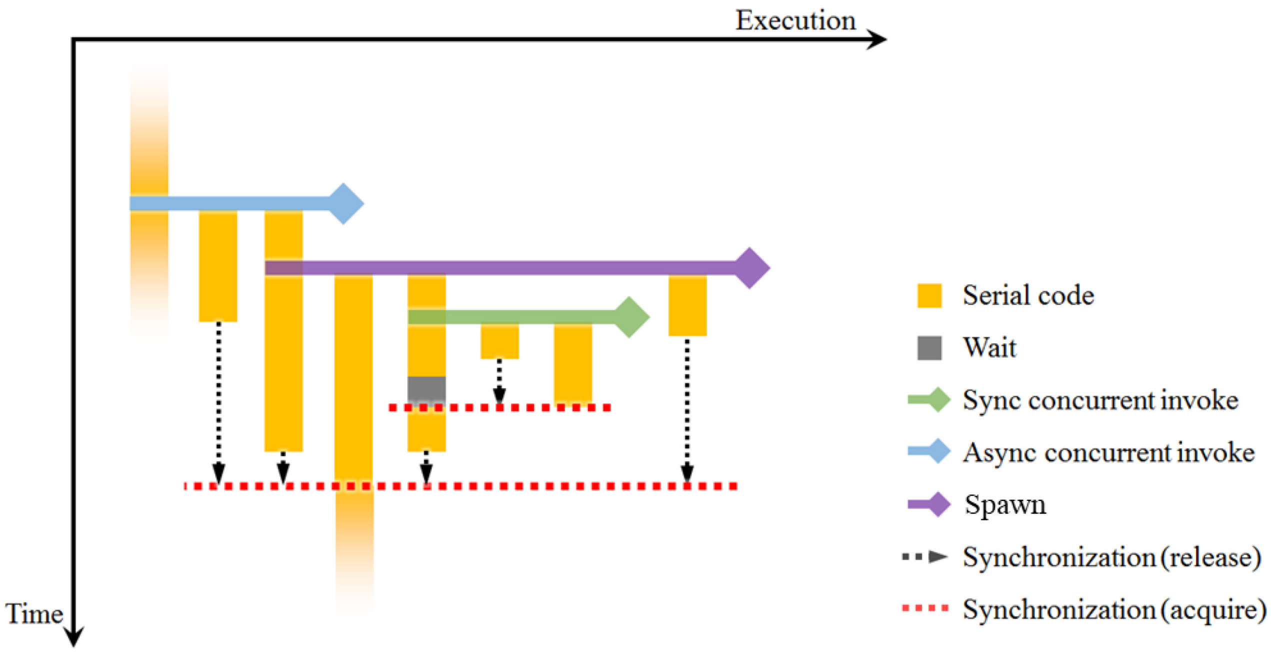

Figure 5

With the concept of the "Concurrent Invoke" the "Spawn" models, we can easily build concurrent programs with complex dependencies among the executions, meanwhile, stay the concurrent logic clear. Figure 4 shows a typical scenario for a composition of the " Concurrent Invoke" and the "Spawn" models; Figure 5 shows a more complicated scenario.

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

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

5.2 Comparing with the Sender/Receiver Model

During the discussion for P0642 in SG1 in Cologne, 2019, I was encouraged to read [P1660R0] to see if the “Senders & Receivers” model could be compatible with the proposed “Concurrent Invocation Library”. After reading P1660 and some related papers, I found there are many similar “concepts” trying to abstract asynchronous execution.

The following content is copied from my reply in the email thread (CIU (aka. Concurrent Invocation Unit) was renamed to CSA (aka. Concurrent Session Aggregation) in this revision):

As a result, *I think the abstraction for “sender & receiver” may be overdesigned and unnecessary, because we would be able to design more concrete APIs (like the way P0642 does) with the same extendibility/performance, and “more concrete APIs” promises less learning cost and more usability*.

Thanks to Kirk Shoop tried to implement P0642 with the Sender/Receiver model. Looking into the sample code, I have the following opinions:

5.2.1 About Execution Closure

I noticed that a new class template **`ensure_callback`** (line 1446~1464) was added as a decorator of tasks being submitted to an EA, providing default implementation for **`done`** and **`error`** functions. There is also a facade template (PFA, P0957) **`Callback`** (line 516~524) providing corresponding polymorphic support. They are used in the class **`thread_executor`** (line 795) and class template **`static_thread_pool`** (line 1538, 1588). However, I neither see a single call to the two functions **`done`** and **`error`** in the sample code, nor how users could interact with the two functions, since the lifetime of the object, at least in the two use cases, are managed by the underlying infrastructures.

I think the two facilities are unreasonably over-designed for **`thread_executor`** and **`static_thread_pool`**, and I do not find enough motivation to introduce the complexity, at least for the sample code.

5.2.2 About spawn and sync_wait

If I understand correctly, the class template **`ConcurrentInvokeSender`** (line 1195~1217), aka. **`concurrent_invoke_sender_t`** (line 1225, 1226), should be the "Sender" model for concurrent invocation. It is also the return type of **`concurrent_invoke`** (line 1219~1223), where users could perform **`sync_wait`** (line 1434~1440) or **`spawn`** (line 1514~1519).

Per usability, I do not see increment comparing to P0642; on the contrary, it is additionally requested for users to explicitly wrap the returned "Sender" with a **`sync_wait`** or **`spawn`**. I am looking for more concrete use cases where **`sync_wait`** and **`spawn`** could be useful. Analyzing the memory usage for **`spawn`**, it turned out to be less efficient than the implementation for P0642, because it allocates larger memory on the heap (line 1516), including:

1. The VTABLE introduced by the **`virtual`** keyword (line 1474), and

2. The decay-copied context parameter (line 1201), and
3. The decay-copied CIU (line 1202), and
4. The decay-copied continuation (line 1203), and
5. The additional bit required for **`std::optional`** (line 1204), and
6. The pointer to **`spawned_op_base`** (line 1481).

   In the list above, 1~4 seem to be inevitable with "Sender & Receiver", because a "Sender" needs the information to build a **`breakpoint`** (line 1207~1210). 5 is required to distinguish the state of an **`Op`** on its destruction. 6 is needed for destroying the **`Op`** (line 1486). In the implementation for P0642, it is also needed to destroy the breakpoint with the invocation has finished, that is why **`this`** pointer was provided to the "Concurrent Callback" (line 954, 971). However, the "Sender & Receiver" model does not support such customization and need to store the pointer, even if the offset of the pointer to the **`breakpoint`** is always a compile-time constant.

As a conclusion, I think “Sender & Receiver” seems NOT a zero-overhead abstraction to implement concurrent invocation.

5.2.3 About Exception

After testing the code, I found **`sync_wait`** on a **`ConcurrentInvokeSender`** throws **`std::vector<std::exception_ptr>`** when there is an error in the invocation, while **`std::p0642::concurrent_invocation_error`** is supposed to be thrown. To clarify the design of P0642, the error type passed to the continuation in asynchronous concurrent invocation is **`std::vector<std::exception_ptr>`**, while the exception type thrown in synchronous concurrent invocation is **`std::p0642::concurrent_invocation_error`**.

The reason why the types are designed to be different is that the mechanism of passing an error as an argument and throwing exception are different in C++. When passing an error in value, the type of the error could be resolved at compile-time, and users expect the type to have as clear semantics as possible to code with. On the other hand, when throwing an error, the type of the error could only be resolved at runtime in a **`catch`** block, that is the reason why we have the exception inheritance hierarchy to help us resolving the type of an error.

I think “Sender & Receiver” may lack of consideration in the difference between the two ways of error handling.

5.2.4 About the “done” Channel

I noticed that the **`done`** function is added both in the class template **`async_concurrent_continuation`** (line 1247) and **`sync_wait_promise`** (line 1393~1396). However, I do not see any reference to the functions, or how it could be used. For instance, if **`done`** should be called in a concurrent invocation, there must be a place to store the cancelation information to determine whether to call it at runtime (e.g., with an **`std::atomic_bool`**). However, it should not be the right decision to go, because:

1. Cancelation is only one possible collaboration in a concurrent invocation, it has nothing special from other collaborations, and
2. A single “bit” (**`std::atomic_bool`**) is usually not enough for a cancelation. For example, when implementing a thread pool, a cancelation signal should always be sent to the managed threads. If the thread pool is simply implemented with a traditional monitor (one **`std::mutex`** + one **`std::condition_variable`**), the cancelation signal should always be sent when the calling thread holds a lock on the mutex.

I believe cancelation is a useful pattern, but I still think it should not be in the same level with the concurrent invocation.

5.3 Blocking Algorithms

In previous revisions of this proposal, the concept "Binary Semaphore" was introduced as an abstraction and **`std::future`** was used 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 CSA 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 CSA 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 based on the considerations above, leaving the blocking algorithm to be implementation-defined.

5.4 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 [P0957R3]. 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 "CSA" (Concurrent Session Aggregation) was proposed with recursive semantics.

5.5 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 "Sink Argument" library [P1648R2] is designed as a generic solution for lifetime extension and in-place construction for sink arguments, making it possible to in-place-construct any argument in a function template without variable parameter and retain clear semantics.

6 Technical Specifications

6.1 Header <concurrent_invocation> synopsis

```
namespace std {

class invalid_concurrent_invocation_context;
template <class CTX = void> class concurrent_invocation_error;
template <class CTX, class CB> class concurrent_breakpoint;
template <class CTX, class CB> class concurrent_token;
struct sync_concurrent_callback;
template <class CT> class async_concurrent_callback;
template <class CSA, class S_CTX, class CT>
void concurrent_invoke(CSA&& csa, S_CTX&& ctx, CT&& ct);
template <class CSA, class S_CTX = in_place_type_t<void>>
auto concurrent_invoke(CSA&& csa, S_CTX&& ctx = S_CTX{});

class unexecuted_concurrent_callable;
template <class F, class CTX, class CB> class concurrent_callable;
template <class E, class F> class serial_concurrent_session;
template <class E, class F, class EH>
class async_concurrent_continuation;
struct throwing_concurrent_error_handler;

}
```

6.2 Type Requirements

6.2.1 `ConcurrentSession` requirements

A type **`CS`** meets the **`ConcurrentSession`** requirements of specific types **`CTX`**, **`CB`** if the following expressions are well-formed and have specific semantics (**`cs`** denotes a value of **`CS`**; **`t`** denotes a value of **`concurrent_token<CTX, CB>`**) .

cs.start(t)

Effects: Start a concurrent session with the given token and is encouraged to return immediately. The session will be alive until the given token becomes invalid.

6.2.2 `ConcurrentSessionAggregation` requirements

A type **`CSA`** meets the **`ConcurrentSessionAggregation`** requirements of specific types **`CTX`**, **`CB`** if

- it meets the **`ConcurrentSession`** requirements of **`CTX`**, **`CB`**, or
- it is a "Generic Tuple" or a "Generic Container" of types meeting the **`ConcurrentSessionAggregation`** requirements of **`CTX`**, **`CB`**.

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.

6.2.3 `ConcurrentContinuation` requirements

A type **`CT`** meets the **`ConcurrentContinuation`** requirements of specific type **`CTX`** if the following expressions are well-formed and have specific semantics (**`ct`** denotes a value of **`CT`**; **`ctx`** denotes a value of **`CTX`** if **`!is_void_v<CTX>`**; **`ex`** denotes a value of **`std::vector<std::exception_ptr>`**).

When **`is_void_v<CTX>`** is **`true`**:

`ct()`

Effects: Executing normal control flow.

`ct.error(ex)`

Effects: Executing error control flow.

Otherwise:

`ct(ctx)`

Effects: Executing normal control flow.

`ct.error(ex, ctx)`

Effects: Executing error control flow.

6.3 Core Types

There are four 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`** is in invalid state but used to access the corresponding **`concurrent_breakpoint`**.
3. Class template **`concurrent_invocation_error`** is thrown during blocking concurrent invocation when any concurrent session propagates any exception.
4. Class template **`concurrent_token`** is the facility for **`CSA`** to collaborate in the concurrent invocation.
5. Class template **`async_concurrent_callback`** and class **`sync_concurrent_callback`** are the callback types for non-blocking and blocking concurrent invocation.

6.3.1 Class template `concurrent_breakpoint`

Any value of **`concurrent_breakpoint`** shall associate with a concurrent invocation. The constructors and destructor of any instantiation of the class template **`concurrent_breakpoint`** is undefined.

```
template <class CTX, class CB>
class concurrent_breakpoint {
 public:
  template <class CSA> void spawn(CSA&& csa);
  add_lvalue_reference_t<CTX> context();
};
```

`template <class CSA> void spawn(CSA&& csa);`

Effects: Starting each of them with a token associated to **`*this`** as if they are part of the initiating concurrent invocation.

`add_lvalue_reference_t<CTX> context();`

Returns: An lvalue reference of the context if **`is_void_v<CTX>`** is **`false`**.

6.3.2 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 CB>
class concurrent_token {
 public:
  concurrent_token();
  concurrent_token(concurrent_token&&);
  ~concurrent_token();
  concurrent_token& operator=(concurrent_token&&);

  bool is_valid() const noexcept;
  void reset() noexcept;
  concurrent_breakpoint<CTX, CB>& get() const;
  void set_exception(exception_ptr&& p);
};
```

`~concurrent_token();`

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

`bool is_valid() const noexcept;`

> *Returns*: **`true`** if and only if **`*this`** associates with a concurrent invocation.

`void reset() noexcept;`

> *Effects*: If **`*this`** associates with a concurrent invocation, detach from the invocation; otherwise no effect.

`concurrent_breakpoint<CTX, CB>& get() const;`

> *Requires*: **`*this`** associates with a concurrent invocation.
> *Returns*: An lvalue reference of the associated breakpoint.
> *Throws*: **`invalid_concurrent_breakpoint`** if **`!is_valid()`**.

`void set_exception(exception_ptr&& p);`

> *Requires*: **`*this`** associates with a concurrent invocation.
> *Effects*: Propagate an exception to the associated concurrent invocation and detach from it.
> *Throws*: **`invalid_concurrent_breakpoint`** if **`!is_valid()`**.

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 Helper for CSA

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

```
template <class F, class CTX, class CB>
class concurrent_callable {
 public:
  explicit concurrent_callable(F&& f, concurrent_token<CTX, CB>&& token);
  concurrent_callable(concurrent_callable&&);
  concurrent_callable& operator=(concurrent_callable&&);
  void operator()() noexcept;
};
```

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

Invoking a value of type **concurrent_callable<E_F, CTX, CB>** will invoke **f** of type **F** and destroy **token**:

- If **std::is_invocable_v<F, concurrent_breakpoint<CTX, CB>>** is **true**, perform **std::invoke(f, token.get())**, or
- If **std::is_invocable_v<F, std::add_lvalue_reference_t<CTX>>** is **true**, perform **std::invoke(f, token.get().context())**, or
- If **std::is_invocable_v<F>** is **true**, perform **std::invoke(f)**, or
- Otherwise, the expression is ill-formed.

If a value of **concurrent_callable** associating with a valid value of **concurrent_token** is destroyed, an exception of type **unexecuted_concurrent_callable** will be attached to the concurrent invocation.

```
template <class E, class F>
class serial_concurrent_session {
 public:
  template <class _E, class _F>
  explicit serial_concurrent_session(_E&& e, _F&& f);
  serial_concurrent_session(serial_concurrent_session&&);
  serial_concurrent_session(const serial_concurrent_session&);
  serial_concurrent_session& operator=(serial_concurrent_session&&);
  serial_concurrent_session& operator=(const serial_concurrent_session&);

  template <class CTX, class CB>
  void start(concurrent_token<CTX, CB>&& token);
};

template <class _E, class _F>
serial_concurrent_session(_E&&, _F&&)
    -> serial_concurrent_session<decay_t<_E>, decay_t<_F>>;
```

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** and a value **f** of type **F**. The type **E** shall meet the **Executor** requirements.

```
template <class CTX, class CB>
void start(concurrent_token<CTX, CB>&& token);
```

Effects: Equivalent to **move(e_).execute(concurrent_callable<F, CTX, CB>{move(f_), move(token)})**.

6.4.2 Helper for Concurrent Continuation

The class template **async_concurrent_continuation** is a default async implementation for concurrent continuation required in non-blocking concurrent invocations. It meets the **ConcurrentContinuation** requirements of any potential type. A value of an instantiation of **async_concurrent_continuation** associates with an **Executor**, a callable value indicating normal control flow channel, and an exception handler indicating error control flow channel. The default implementation for error control flow channel throws **concurrent_invocation_error** to the host executor.

```
template <class E, class F, class EH>
class async_concurrent_continuation {
 public:
  template <class _E, class _F, class _EH = EH>
  explicit async_concurrent_continuation(_E&& e, _F&& f, _EH&& eh = EH{});

  template <class CTX>
  void operator()(CTX&& ctx);
  void operator()();

  template <class CTX>
  void error(vector<exception_ptr>&& ex, CTX&& ctx);
  void error(vector<exception_ptr>&& ex);
};

struct throwing_concurrent_error_handler {
  template <class CTX>
  void operator()(vector<exception_ptr>&& ex, CTX&& ctx) const;
  void operator()(vector<exception_ptr>&& ex) const;
};

template <class _E, class _F, class _EH>
async_concurrent_continuation(_E&&, _F&&, _EH&&)
    -> async_concurrent_continuation<decay_t<_E>, decay_t<_F>, decay_t<_EH>>;
template <class _E, class _F>
```

```
async_concurrent_continuation(_E&&, _F&&)
    -> async_concurrent_continuation<decay_t<_E>, decay_t<_F>,
        throwing_concurrent_error_handler>;
```

6.5Function templates `concurrent_invoke`

The "reduced value" of a value **`ctx`** of type **`CTX`** is defined as:

- If **`ctx.reduce()`** is a well-formed expression, the reduced value is the return value of the expression, or
- If **`CTX`** is move-constructible, the reduced value is **`ctx`**, or
- Otherwise, there is no reduced value of **`ctx`**.

The "reduced type" of **`ctx`** is the type of the reduced value if exist, or **`void`** otherwise.

```
template <class CSA, class S_CTX, class CT>
void concurrent_invoke(CSA&& csa, S_CTX&& ctx, CT&& ct);
```

Requires: Type **`CSA`** meets the **`ConcurrentSessionAggregation`** requirements of **`sunk_t<S_CTX>`**, **`async_concurrent_callback<decat_t<CT>>`**. Type **`decay_t<CT>`** shall meet the **`ConcurrentContinuation`** requirements of the reduced type of **`sunk_t<S_CTX>`**.

Effects: Construct a value of **`concurrent_breakpoint<sunk_t<S_CTX>, async_concurrent_callback<decat_t<CT>>>`** and perform non-blocking concurrent invocation with **`csa`** on the breakpoint.

```
template <class CSA, class S_CTX = in_place_type_t<void>>
decltype(auto) concurrent_invoke(CSA&& csa, S_CTX&& ctx = S_CTX{});
```

Requires: Type **`CSA`** meets the **`ConcurrentSessionAggregation`** requirements of **`sunk_t<S_CTX>`**, **`sync_concurrent_callback`**.

Effects: Construct a value of **`concurrent_breakpoint<sunk_t<S_CTX>, sync_concurrent_callback>`** and perform blocking concurrent invocation with **`csa`** on the breakpoint.

Return type: The reduced type of **`sunk_t<S_CTX>`**.

Returns: The reduced value of **`sink(ctx)`** if the return type is not **`void`**.

Throws: **`concurrent_invocation_error`** if any associated session propagates an exception.