Document number: P2074R0
Date: 2020-01-13 (pre-Prague mailing)
Reply To: Dmitry Duka (dduka@nvidia.com)
Contributors: Marco Foco (mfoco@nvidia.com)
Audience: WG21, Tooling Study Group SG15

Asynchronous callstacks & coroutines

Abstract

This paper discusses “asynchronous” callstacks. A chain of asynchronous calls expressed with coroutines differs to a chain of synchronous calls expressed with regular functions. While debugging, the former are not supported in the tooling, while the latter is. We present a possible solution allowing to mitigate the absence of tooling support for asynchronous callstacks.

Discussion

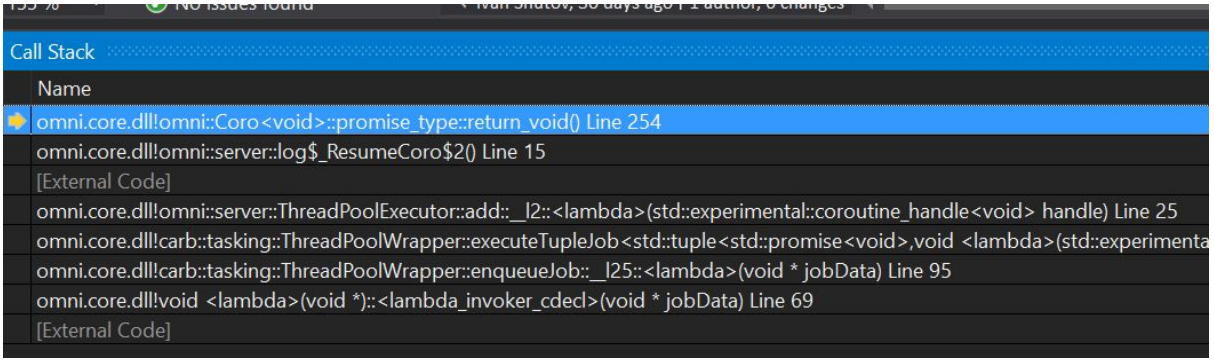
Regular functions being called in a particular way organize callstacks. To such callstacks we will refer as “synchronous”. Coroutines on the other hand organize both “synchronous” and “asynchronous” callstacks. A coroutine calling a regular function is not different from a regular function calling another regular function. A coroutine calling another coroutine though creates parent-child relationship of a different sort, because coroutines may execute asynchronously. Thus we call such callstacks “asynchronous”. To put it shortly, in general, setting a breakpoint inside a coroutine will not yield a callstack showing a parent coroutine. Instead it’ll show an execution context resuming the child coroutine. This largely depends on the particular implementation of the Awaiter/promise_type interfaces for C++ coroutines. Programmers debugging asynchronous code based on coroutines will likely need asynchronous callstacks far more frequently than synchronous callstacks. Because of the asynchronous nature of C++ coroutines, it is important to understand which coroutine is the parent of a given running coroutine. Without such ability debugging coroutines is going to be quite tedious and unreliable exercise.

Problem

The code below illustrates 3 coroutines **op, add** and **log**. **op** calls **add**, which in turn calls **log**.

```
12  omni::Coro<void> log(uint64_t x)
13  {
14      std::cout << x << std::endl;
15      co_return;
16  }
17
18  omni::Coro<uint64_t> add(uint64_t x, uint64_t y)
19  {
20      co_await log(x);
21      co_await log(y);
22      co_return x + y;
23  }
24
25  omni::Coro<uint64_t> op(uint64_t x, uint64_t y)
26  {
27      co_return co_await add(x, y);
28  }
```

In practice coroutines may live in different compile units and/or namespaces, so relationships between them may not be as clear. Now, if we want to debug **log**, we can set a breakpoint and this may yield the following callstack in the debugger:



In this example the coroutine support library uses thread pool to execute coroutines. It is implemented such that a coroutine is always started, suspended and resumed from a thread from the thread pool. As shown in the image, the synchronous callstack just shows that the **log** coroutine has been called from an execution context, a thread pool in our case. The information about which coroutine or function scheduled the execution of this coroutine is not available. This is expected and such synchronous callstack of course has value in their own right. However, we don't have information available to understand which coroutine or regular function scheduled the execution of the **log** coroutine in the first place. This information is even more valuable in asynchronous environment compared to a synchronous one.

Other languages

Python

Callstacks generated in Python 3 using coroutines (as provided by asyncio library and the language itself) do yield expected results. Python coroutines are stackful. Callstacks include both top-level synchronous portions (**loop.run_until_complete(coro())**) as well as asynchronous functions which called the current coroutine.

JavaScript

Node.js and different browsers implement asynchronous callstacks differently. Chrome browser for instance does present asynchronous callstacks correctly even if the breakpoint is after a suspend point of a coroutine. Node.js on the other hand only present asynchronous callstack if a coroutine was not suspended yet. Firefox browser behaves similarly.

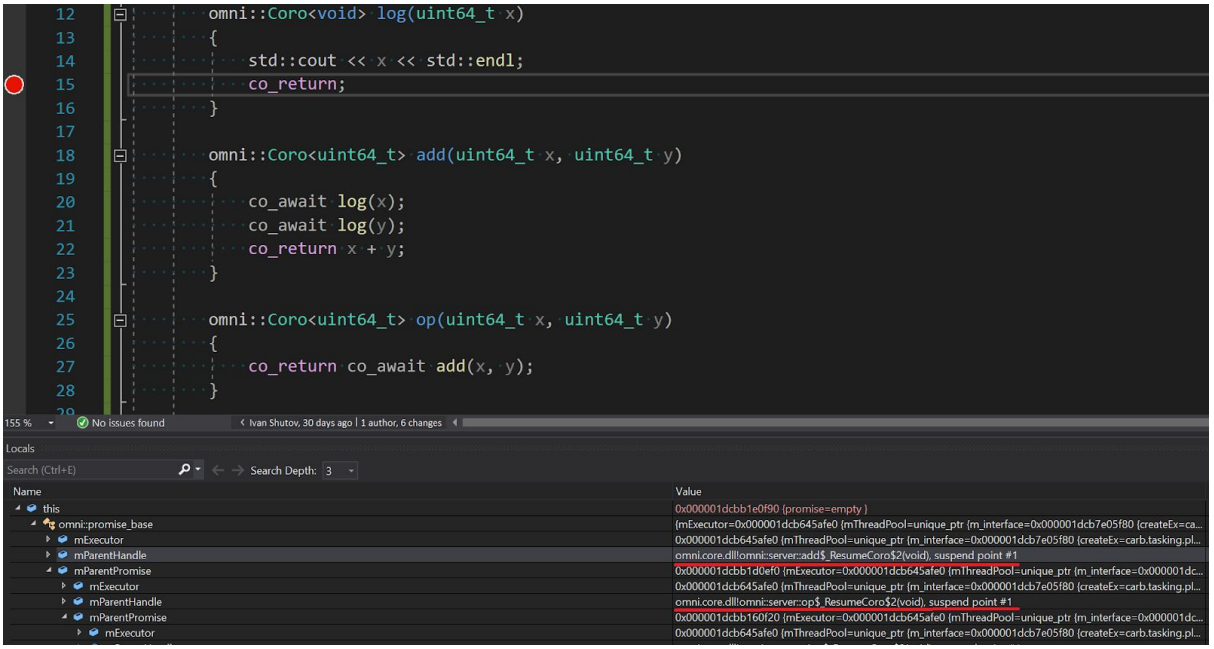
C#

Microsoft did some work in the implementation of C# runtime, such that it does yield asynchronous callstacks. https://github.com/dotnet/corefx/issues/24627

Possible solution

We can store a pointer to the parent coroutine **promise_type** object and this yields the following debugging experience.

Asynchronous callstack:



The image above illustrates that because we added a pointer to the parent coroutine **promise_type** object, we can now unfold the callstack in the form of a linked list chained through **mParentPromise** pointer. In the absence of the possibility to get a **coroutine_handle** from the currently running coroutine, we first need to break on a line which uses new keywords (**co_await**, **co_yield** or **co_return**) and step inside the coroutine support library to get the **promise_type** object related to the current coroutine. This solution presents callstacks as data in the sense that to view such a callstack one would use a watch in the debugger.

Of course it would be nice for the tooling to explicitly support a notion of asynchronous callstacks, but this is a major change and it's not clear how to make this solution general enough and compatible with majority of possible **promise_type** / **Awaiter** interface implementations, where coroutines execution may be lazy or eager or even both.

When and if standard executors will be integrated together with coroutines, this problem will become increasingly noticeable and coroutine support libraries implemented by vendors should probably consider having a pointer to the parent coroutine in a form that is discoverable in the debugger.

As discussed in "Debugging C++ coroutines" paper (p2073r0), if:

- it would be possible to get a **coroutine_handle** object from the currently running coroutine, and
- it would be possible to get **promise_type** object from such handle

then:

- there will be no need to add a separate pointer like **mParentPromise**
- instead it will be possible to unfold the asynchronous callstack through **coroutine_handle** objects. Of course this still depends on a particular implementation of **promise_type** / **Awaiter** interfaces, but in the majority of cases it

is expected that **promise_type** will hold a handle to the parent coroutine serving as continuation.