Coroutines: producer-consumer

Samstag, 21. Oktober 2023 14:31

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

Coroutines provide a more intuitive and structured way of writing asynchronous code by allowing you to write asynchronous operations in a procedural style.
They are a feature introduced in C++20 to simplify asynchronous programming.
Unlike other pre-existing mechanisms like

- std::async tasks
- std::packaged_task
- Events (std::condition_variable & std::mutex)

with ability to synchronize two, or more threads on the result of the task, by establishing the communication channel with two ends:

- std::**promise** that writes into the shared state either result, or the exception, and
- std::**future** (std::shared_future) - a receiving end, that waits on the result of the task (or the exception),

coroutines don't directly involve threads, or any other OS synchronization primitives.
They are rather a pure software abstraction which is based on the coroutine's control object and the **state-machine** logic built around it.

Coroutines are *stackless* - which means that the control object has to be created on the heap.
Coincidentally, it's a library wrapper around the promise_type (*std::coroutine_handle<promise_type>*), but it actually doesn't have anything in common with std::promise.
The **promise_type** is an interface (a *customization point*) that describes the predefined transition states in coroutine's state-machine.
More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/C%2B%2B20/Coroutines.pdf

Coroutines are highly versatile and can be used in various scenarios where you need to manage asynchronous message flow.
One common example is socket-based communication.
Today, I'll try to enlighten coroutines through yet another example: **single producer - single consumer** scenario.

Implementation

First, we need to define **result type** for coroutine

```
class [[nodiscard]] AudioDataResult final
{
        public:
                class promise_type;
                using handle_type = std::coroutine_handle<promise_type>;

                class promise_type
                {
                  ...
                };
};
```

as wrapper around the inner: *promise_type* type.
We decorate the enclosing class with `[[nodiscard]]` attribute, since the result type is control object of the coroutine: that we return to the client code - to manage its suspension/resumption.

```
~AudioDataResult() { if(handle_) { handle_.destroy(); } }
```

The result type is move-only: the copy operations are forbidden - to prevent the control object being multiplicated

```
 // Make the result type move-only, due to exclusive ownership over the handle
AudioDataResult(const AudioDataResult&) = delete;
AudioDataResult& operator= (constAudioDataResult&) = delete;

AudioDataResult(AudioDataResult&& other) noexcept:
handle_(std::exchange(other.handle_, nullptr))
{}

AudioDataResult& operator = (AudioDataResult&& other) noexcept
{
        using namespace std;
        AudioDataResult tmp =std::move(other);
        swap(*this, tmp);
        return *this;
}
```

Let's defined the *promise_type* interface itself

```cpp
// Predefined interface that has to be specify in order to implement
// coroutine's state-machine transitions
class promise_type
{

    public:

        using value_type = std::vector<int>;

        AudioDataResult get_return_object()
        {
            return AudioDataResult{ handle_type::from_promise(*this) };
        }
        std::suspend_never initial_suspend() noexcept { return{}; }
        std::suspend_always final_suspend() noexcept { return{}; }

        void return_void() {}
        void unhandled_exception()
        {
            std::rethrow_exception(std::current_exception());
        }

        // Generates the value and suspend the "producer"
        template <typename Data>
        requires std::convertible_to<std::decay_t<Data>, value_type>
        std::suspend_always yield_value(Data&& value)
        {
            data_ = std::forward<Data>(value);
            data_ready_.store(true, std::memory_order_release);
            return {};
        }

    private:
        value_type data_;
        std::atomic<bool> data_ready_;
};//promise_type interface
```

The *promise_type* defines the necessary infrastructure of coroutine.
Additionally, for any coroutines that want to act as a generator - "***producer***", to yield the values: promise_type has to be enhanced with the yield_value method (co_yield ≡ co_await promise_.yield_value).
Also, to resume the producer, at the point when provided data are consumed - we need to expose appropriate: wrapper method *resume*()

```cpp
 void resume() { if( not handle_.done()) { handle_.resume();} }
```

Now - we need to extend the coroutine to support the ***consumer*** requirements: to be synchronized on the data readiness.
In other words, the consumer will be suspended, until the data are signaled as available by producer.
For that, we need to implement Awaiter interface

```cpp
class promise_type
{
    // Awaiter interface: for consumer waiting on data being ready
    struct AudioDataAwaiter
    {

        explicit AudioDataAwaiter(promise_type& promise) noexcept: promise_(promise) {}

        bool await_ready() const
        {
            return promise_.data_ready_.load(std::memory_order_relaxed);
        }

        void await_suspend(handle_type) const
        {
            static constexpr bool expected = true;
            static constexpr bool replace_with = false;

            utils::exchange(promise_.data_ready_,
                 expected,
                 replace_with,
                 std::memory_order_acquire,
                 std::memory_order_relaxed);
        }

        // Move assignment at client invocation side:
        // const auto data = co_await audioDataResult;
        // This requires that coroutine's result type provides the co_await unary operator

        value_type&& await_resume() const
        {
            return std::move(promise_.data_);
        }
```

```
        private:
            promise_type& promise_;

    };//Awaiter interface

};//promise_type
```

In its state-machine, the *await_ready*() will be first transition state: it will be inspected on data readiness.
If the data are not ready, as next the *await_suspend*() will be called.
Here we actually wait - until the matching flag is being set.
Finally, the *await_resume*() will be called: we "move" the value from the promise_type, by unconditionally cast to the rvalue reference. At the client invocation side, this will cause the move assignment operator on the returned value - data to be called

```
const auto data = co_await audioDataResult;
```

For that to work, result type needs to provide the co_await unary operator, which returns our Awaiter interface

```
class AudioDataResult
{
    auto operator co_await() noexcept
    {
        return promise_type::AudioDataAwaiter{handle_.promise()};
    }
};
```

<Example 1>: https://godbolt.org/z/MzsrdKbKT

The other way around is to use the promise_type::*await_transform*() - to wait on the value stored in the promise_type instance used by the producer

```
class promise_type
{
    auto await_transform(handle_type other)
    {
        // Awaiter interface: remained the same
        struct AudioDataAwaiter
        {
            explicit AudioDataAwaiter(promise_type& promise) noexcept: promise_(promise) {}
            ...
        };

        return AudioDataAwaiter{other.promise()};
    }
};
```

This way, we don't need to specify the co_await unary operator of the result type anymore,
but rather (explicit) conversion operator

```
explicit operator handle_type() const {return handle_;}
```

so that we can pass it at the point when consumer calls co_await, which will internally be translated to await_transform() call

```
const auto data = co_await static_cast<AudioDataResult::handle_type>(audioDataResult);
```

We can illustrate this as: *me.handle_.promise().await_transform(other.handle_)*

<Example 2>: https://godbolt.org/z/57zsK9rEn

Conclusion

In this simple example, producer will be suspended, without any penalty - since after being resumed,
it will provide the very same - upfront known sequence of data.
In real scenario, that is likely not be the case: the producer itself will be probably some kind of mediator - receiver of asynchronously emitted data, that will be retransmitted to the consumer.
For that, some queuing logic needs to be implemented at producer side, to avoid the data loss at the point of being suspended, waiting on consumer to resume it - to compensate the differences in
producer data arrival vs. consumer consumption rate.