Coroutines - part 2

Mittwoch, 8. Januar 2025 14:22

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

Intro

I've already wrote about the coroutines, but this is the topic that I keep revisiting over and over again.
This time, I want to enlighten - demystify this topic through the concreate example.
As you already know - the coroutines can be seen as a state-machines, that on each suspension and resumption go through the predefined states: preserving the coroutine's frame at the point of suspension, so that it can be restored at the point of resumption - this happens by default on the heap, since normal function (routine) stack wouldn't be sufficient here.

It consist of two interfaces:

- The ***Promise*** interface describes - how the coroutines behave in these state transitions, providing the link to the controlling handle: std::coroutine_handle<>
- The ***Awaitable*** interface describes the mechanics to suspend the caller that waits on the certain condition (synchronous, or asynchronous) to be fulfilled before caller will be resumed.

Example

Imagine we have a **state-machine** that manages the lifecycle of our app.
For the ***shut-down sequence***, we usually want to wait on certain tasks to be completed, maybe even in certain order, before the app enters the terminal state.
We can wait on this tasks on the same thread (AOT), or we can assign to each task - its own execution context.

⇒ *How would you accomplish that?*

Well, one way is to use the another state-machine: coroutines.
First, we will implement the generic **join operator**: that will suspend our state-machine, waiting on the arbitrary number of tasks to complete.

There are several things to consider here:

- Synchronization primitives
  We will use ***std::latch*** behind the scene, as a count-down mechanism: to synchronize on the tasks completion.
  Unlike semaphores, the logic with latches is complete opposite.
  As long as the initial counter is non-zero, the latch is blocking the waiting thread (suspended caller).
  That is why, it's initially set to the number of arbitrary tasks to wait for

  ```
  std::latch signal_{sizeof...(Tasks)};
  ```

  The each task decrements it, upon the completion - until all tasks are done, after the caller will be finally resumed

- Arbitrary number of tasks - this can be any invocable: callable object.
  @note Unlike the **when_all_ready()* operator in the corocpp library - where the task (Task<T>) is Awaitable (coroutine) - here the meaning of the task is a simple callable object

  We need to extend the each task with the std::latch - decorator pattern

  ```
  auto signaling_task = [this](auto& task) mutable
  {
      auto job = [this, task]() mutable
      {
          using result_type = std::invoke_result_t<decltype(task)>;
          constexpr bool has_return = not std::is_void_v<result_type>;

          std::conditional_t<has_return, result_type, null_type> result {};

           try
           {
               if constexpr (has_return) result = std::invoke(task);
               else std::invoke(task);
           }
           catch(...)
           {
               const auto exp = std::current_exception();
               if (exception_handler_) std::invoke(exception_handler_, exp);
           }

           signal_.count_down();
           if constexpr (has_return) return result; // return optionally the result of the task
      };
      // Execute task
  ```

```
        if constexpr (std::is_null_pointer_v<Executor>)
            std::invoke(std::move(job));// on the same thread as caller
        else
            std::invoke(executor_, std::move(job));// on a dedicated execution context
    };
```

- We wrap the task invocation around the try/catch block, for **handling the exception**
There are different strategy for that. Either you handle the each exception uniformly - providing the exception handler, or the wrapping the exception along with result into *std::expected*, and return it when caller is resumed. Here, we use the very same exception handler.

- **Executor - or Scheduler** is an <u>execution context</u> that will be attached to each task, if any.
Specifying the *std::nullptr_t* - the each and every task will be executed within the same thread - the one that host the suspended coroutine (caller).
Otherwise, the Executor should be callable - that can be invoked with the given tasks

```
    if constexpr(not std::is_null_pointer_v<Executor>)
        static_assert((std::invocable<Executor, Tasks> && ...), "<Awaitable> Invalid Executor type!");
```

- We will apply the same logic, for all tasks - and wait on their completion being signaled

```
    /*
     * Depends on the executor, the tasks can be spawned in a non-blocking
     * manner (asynchronously), or be executed sequentially (std::nullptr_t) - on
     * the same thread as suspended caller
     */
    auto joiner = [&]<std::size_t...Is>(std::index_sequence<Is...>)
    {
        ( signaling_task(std::get<Is>(tasks_)), ...);
    };

    joiner(std::make_index_sequence<sizeof...(Tasks)>{});

    // Wait on all tasks being completed
    signal_.wait();
```

The entire code for this operator would be

```
template <typename Executor, typename... Tasks>
requires(sizeof...(Tasks) > 0)
auto join_tasks(exception_handler handler, Executor&& executor, Tasks&&...tasks)
{
    // Inner - nested Awaitable interface implementation
    struct Awaitable final :public std::suspend_always
    {
        // C-tor
        explicit Awaitable(exception_handler handler, Executor&& executor, Tasks&&...tasks) noexcept
                : exception_handler_(handler),
                  executor_(std::forward<Executor> executor)),
                  tasks_(std::make_tuple(std::forward<Tasks>(tasks)...))
        {
            if constexpr(not std::is_null_pointer_v<Executor>)
                static_assert((std::invocable<Executor, Tasks> && ...), "<Awaitable> Invalid Executor type!");
        }

        void await_suspend(std::coroutine_handle<> caller)
        {
            /*
             * Equip each task additionally with the std::latch - count-down mechanism (decorator pattern)
             */
             auto signaling_task = [this](auto& task) mutable
             {
                 auto job = [this, task]() mutable
                 {
                     using result_type = std::invoke_result_t<decltype(task)>;
                     constexpr bool has_return = not std::is_void_v<result_type>;

                     std::conditional_t<has_return, result_type, null_type> result {};

                     try
                     {
                         if constexpr (has_return) result = std::invoke(task);
                         else std::invoke(task);
                     }
                     catch(...)
                     {
                         const auto exp = std::current_exception();
                         if(exception_handler_) std::invoke(exception_handler_, exp);
                     }

                     signal_.count_down();
                     if constexpr (has_return) return result;
                };

                // Execute task
                if constexpr (std::is_null_pointer_v<Executor>)
                std::invoke(std::move(job)); // on the same thread as caller
            else
                std::invoke(executor_, std::move(job)); // on a dedicated execution context
        };

        /*
         * Depends on the executor, the tasks can be spawned in a non-blocking
         * manner (asynchronously), or be executed sequentially (std::nullptr_t) - on the same thread as
         * suspended caller
         */
        auto joiner = [&]<std::size_t...Is>(std::index_sequence<Is...>)
        {
            ( signaling_task(std::get<Is>(tasks_)), ...);
        };

        joiner(std::make_index_sequence<sizeof...(Tasks)>{});

        // Wait on all tasks being completed
        signal_.wait();

        // Resume caller
        caller.resume();
    }

    [[maybe_unused]]void await_resume() noexcept {}

private:

    std::latch signal_{sizeof...(Tasks)}; /**Signal the tasks completion*/
    exception_handler exception_handler_; /**Exception handler*/

    using executor_type = std::conditional_t<not std::is_null_pointer_v<Executor>,std::remove_cvref_t<Executor>, std::nullptr_t>;
    executor_type executor_; /** Execution context for the tasks, or std::nullptr_t*/

    std::tuple<std::remove_cvref_t<Tasks>...>tasks_; /**list of the tasks*/
}; // Awaitable

return Awaitable{handler, std::forward<Executor>(executor), std::forward<Tasks>(tasks)...};
}
```

The link to the matching unit-test: https://godbolt.org/z/6qddqobTe

Conclusion

As demonstrated - the coroutines are the new way to write asynchronous, inter-thread message based communication in a procedural way (*structured concurrency*)

```
auto result = co_await Awaitable{};
```

or in our case

```
co_await join_tasks(std::forward<Tasks>(tasks)...);
```

whereby the real synchronization happens behind the customization points exposed by the matching Awaitable interface:
employing the other, already known synchronization mechanisms.
This way, we have a descriptive - functional way of handling the asynchronous tasks in a generic way,
hiding the implementation details behind the library-like utility classes and operators.

Links

*corocpp library
https://github.com/lewissbaker/cppcoro/tree/master?tab=readme-ov-file#when_all_ready