On writing loops in PPL and continuation-passing style, part 1

devblogs.microsoft.com/oldnewthing/20230822-00

August 22, 2023

Raymond Chen

The Parallel Patterns Library (PPL) is based on a continuation-passing style, where you invoke a task, and then attach a callable object that will be given the result. Prior to the introduction of the `await` and `co_await` keywords to C#, JavaScript, and C++, this was your only real choice for asynchronous programming.

Sequential calculations are fairly straightforward in continuation-passing style because you just pass the next step as the continuation.

```
// Synchronous version
auto widget = find_widget(name);
auto success = widget.toggle();
if (!success) report_failure();

// Asynchronous version
find_widget(name).then([=](auto widget) {
    return widget.toggle();
}).then([=](auto success) {
    if (!success) report_failure();
});
```

Iteration is harder to convert to continuation-passing style because you need to restart the task chain, which means you have recursion. A named helper function makes this easier.

```
// Synchronous version
for (int i = 0; i < 3; i++) {
    widgets[i] = create_widget();
}

// Asynchronous version
task<void> create_many_widgets(Widget* widgets, int count)
{
    if (count == 0) return task_from_result();

    return create_widget().then([=](auto widget) {
        widgets[0] = widget;
        return create_many_widgets(widgets + 1, count - 1);
    });
}
```

The asynchronous version first creates one widget and saves it into `widgets[0]`. Then it asks to create `count - 1` more widgets starting at `widgets + 1`. This recursively fills in the remaining widgets. The recursion is broken when the count drops to zero.

My joke is that continuation-passing style forces you to write in Scheme. (Note: Not actually a joke.)

Maybe you can write a helper function to automate looping.

```
template<typename Callable>
task<void> do_while_task(Callable&& callable)
{
    using Decayed = std::decay_t<Callable>;
    struct Repeat {
        Repeat(Callable&& callable) :
            f(std::make_shared<Decayed>(
                std::forward<Callable>(callable))) {}

        std::shared_ptr<Decayed> f;
        task<void> operator()(bool loop) {
            if (loop) {
                return (*f)().then(*this);
            } else {
                return task_from_result();
            }
        }
    };
    Repeat p(std::forward<Callable>(callable));
    return p(true);
}

// Sample usage
do_while_task([i = 0, widgets]() mutable
{
    if (i >= 3) return task_from_result(false);
    return create_widget().then([index = i++, widgets](auto widget)
    {
        widgets[index] = widget;
        return true;
    });
}).then([] {
    printf("Done!\n");
});
```

The idea here is that you pass `do_while_task` a callable object, and the `do_while_task` invokes the callable, expecting it to return a `task<bool>`. If the returned task completes with `true`, then `do_while_task` invokes the callable again, repeating until the callable's returned task finally completes with `false`, at which point we end the task chain by returning a completed task.

The parameter passed to the `task::then()` method is usually a lambda, but it can be any callable. After all, a lambda itself is just a convenient syntax for a callable object. For our usage, we will pass an instance of the `Repeat` class, which is callable thanks to the `operator()` overload.

One of the quirks of the Parallel Patterns Library (PPL) is that the callable passed to `then()` must be copyable.[1] Therefore, we wrap the `callable` inside a `std::shared_ptr` so that the shared pointer can be copied, while still retaining a single copy of the `callable`. This is important because the `callable` passed to `do_while_task` might be stateful, and we need to allow it to retain state across calls.

At each iteration, we check whether the previous iteration said to continue running. If not, then we stop. Otherwise, we invoke the callable and request that we (or at least a copy of ourselves) be called back with the result, thus scheduling the next iteration.

The main function starts the festivities by calling the callable with `true`.

We wrote our `then()` handler to accept a `bool`, which means that it is bypassed when an exception occurs. The exception flows off the end of the task chain and is reported to the caller of `do_while_task`.

Next time, we'll rewrite this in terms of a more traditional recursion.

[1] If PPL support movable callables, then we could have used a `std::unique_ptr<Callable>` and done a `.then(std::move(*this))` to move the unique pointer from one iteration to the next.