Speculation on the design decisions that led to the common ABI for C++ coroutines

devblogs.microsoft.com/oldnewthing/20220103-00

January 3, 2022

Raymond Chen

A little while ago, I discussed the common ABI for C++20 coroutine handles. Recall that the common ABI is

```
struct coroutine_frame_abi
{
    void (*resume)(coroutine_frame_abi*);
    void (*destroy)(coroutine_frame_abi*);
};
```

and that in practice, the implementations set themselves up like this:

```
struct coroutine_frame
{
    void (*resume)(coroutine_frame*);
    void (*destroy)(coroutine_frame*);
    uint16_t index;
    /* other stuff */
};
```

The `index` represents the point inside the coroutine at which execution was suspended. Each time the coroutine suspends, the `index` is updated, and when the coroutine resumes, the `resume` function switches on the `index` to decode where to resume execution.

What other designs could have been used?

One counter-proposal was that instead of updating the index, the code could update the `resume` and `destroy` pointers to point to where to resume next (or what to do if the coroutine is destroyed).

Updating the function pointers would speed up resumption, since it could just jump directly to the resumption point instead of having to execute a `switch` statement.

However, it also comes with a cost.

For one thing, Control Flow Guard is most effective when function pointers are multiples of 16, and the Microsoft compiler will arrange for all function pointer jump targets to be placed on 16-byte boundaries. The clang compiler also supports Control Flow Guard, but I don't know whether it puts jump targets on 16-byte boundaries.

Putting all resumption and destruction points on 16-byte boundaries would on average insert eight bytes for each potential entry point. The destruction entry points can often be quite small, destructing one object and then falling through to another case where the remainder of the objects are destructed.

For example:

```
winrt::IAsyncAction MyCoroutine()
{
    auto p1 = std::make_unique(1);
    co_await Something1();
    {
        auto p2 = std::make_unique(2);
        co_await Something2();
    }
    auto p3 = std::make_unique(3);
    co_return;
}
```

There are four suspension points in this coroutine, once we add the two extra suspension points provided by the promise.

```
winrt::IAsyncAction MyCoroutine()
{
    promise p;
    co_await p.initial_suspend();
    // 1

    auto p1 = std::make_unique(1);
    co_await Something1();
    // 2

    {
        auto p2 = std::make_unique(2);
        co_await Something2();
        // 3
    }

    auto p3 = std::make_unique(3);
    p.return_void();

    co_await p.final_suspend();
    // 4
}
```

Resume	Destroy
goto 1	destruct p
goto 2	destruct p1 and p
goto 3	destruct p2, p1, and p
goto 4	destruct p3, p1, and p

In practice, the four `destroy` functions are going to fall through to each other or at least jump to each other.

```
destroy4:
    lea     rcx, [p3]
    call    std::unique_ptr<int>::~unique_ptr<int>
    jmp     destroy2
destroy3:
    lea     rcx, [p2]
    call    std::unique_ptr<int>::~unique_ptr<int>
destroy2:
    lea     rcx, [p1]
    call    std::unique_ptr<int>::~unique_ptr<int>
destroy1:
    lea     rcx, [p]
    call    promise::~promise
```

If each of these destruction entry points were a runtime function pointer, there would have to be a lot of padding between them to get the start-addresses to align on 16-byte boundaries.

On the other hand, if it's a switch statement or jump table, no such padding is required because the jump targets are kept in the code segment or the read-only data segment, so they are safe from corruption via buffer overflow, use-after-free, or type confusion.

In addition to the instruction padding requirements, replacing the function pointers on suspension is significantly more code:

```
    ; index-based
    mov word ptr [rbx].index, 42 ; update the index

    ; function-pointer-based
    mov rcx, offset resume2
    mov qword ptr [rbx].resume, rcx
    mov rcx, offset destroy2
    mov qword ptr [rbx].destroy, rcx
```

Instead of writing a 16-bit constant, we are writing two 64-bit constants. But the x86-64 cannot write a 64-bit constant directly to memory. You have to pass the value through a register first.

And that constant isn't a constant. It's a relocatable address, which means that you also have to add a relocation record for each of those addresses.[1]

Furthermore, other 64-bit processors cannot load 64-bit immediate constants. If the function isn't too large, you can use instruction-pointer-relative instructions:

```
; index-based
movz    r0, #42
str     r0, [r1, #index]

; function-pointer-based for small functions
adr     x0, resume2
str     x0, [r1, #resume]
adr     x0, destroy2
str     x0, [r1, #destroy]

; function-pointer-based for large functions
adrp    x0, resume2
add     x0, x0, #PageOffset(resume2)
str     x0, [r1, #resume]
adrp    x0, destroy2
add     x0, x0, #PageOffset(destroy2)
str     x0, [r1, #destroy]

; alternate version with constants in memory
; (two pointers = size of four instructions)
ldr     x0, [pc, #...]  ; load constant from memory
str     x0, [r1, #resume]
ldr     x0, [pc, #...]  ; load constant from memory
str     x0, [r1, #destroy]
```

You don't need a switch statement in the `resume` function, but you pay more at each suspension point to set up the function pointers. This is trading a fixed cost for a variable cost. The size of the coroutine switch statement is 34 bytes:

```
0f b7 43 xx             movzx eax, word ptr [rbx+xx]
ff c0                   inc   eax
83 f8 08                cmp   eax, 8
0f 87 xx xx xx xx       ja    fatal_error
48 8d 15 xx xx xx xx    lea   rdx, [__ImageBase]
8b 8c 82 xx xx xx xx    mov   ecx, dword ptr [rdx+rax*4+xxxxxxxx]
48 03 ca                add   rcx, edx
ff e1                   jmp   rcx
```

The index update is six bytes:

```
66 c7 43 xx yy yy       mov  word ptr [rbx+xx], yyyy
```

The double-address update is 27 bytes:

```
48 b9 xx xx xx xx       mov  rcx, xxxxxxxxxxxxxxxx
      xx xx xx xx
48 89 0b                mov  [rbx], rcx
48 b9 xx xx xx xx       mov  rcx, xxxxxxxxxxxxxxxx
      xx xx xx xx
48 89 4b 08             mov  [rbx+8], rcx
```

You save a 34-byte fixed overhead, but each suspension point costs 21 bytes more. This means that once you have a second suspension point, you're at net loss in code size.

A similar calculation plays out for AArch64: The standard resumption dispatcher is eight instructions:

```
        ldrh    x0, [x1, #index]
        add     w0, w0, #1
        cmp     w0, #8
        bhi     fatal_error
        adr     x9, switch_table
        ldrsw   x8, [x9, w0 uxtw #2]
        add     x9, x9, x8, lsl #2
        br      x9
```

Updating the index is two instructions, but updating the two pointers is four instructions for small functions, and six instructions for large functions.[2] Even if you take the smallest extra cost of two instructions, it means that once your coroutine has four suspension points, updating function pointers is going to have a larger net code size.

For all but the simplest coroutines, the index-based version ends up a net win in terms of code size.

[1] I guess you could be sneaky and if you know that only one suspension point precedes the one you are about to reach, you could add the delta to the two addresses. But that works only for straight-line code, and if anything goes slightly wrong, the results can get quite wild.

[2] If you store the addresses as in-memory constants, then it will cost you eight instructions plus two relocations.

Raymond Chen

Follow