Debugging coroutine handles: The Microsoft Visual C++ compiler, clang, and gcc

devblogs.microsoft.com/oldnewthing/20211007-00

October 7, 2021

Raymond Chen

How compilers implement coroutines is an implementation detail which is subject to change at any time. Nevertheless, you may be called upon to debug them, so it's nice to know what you're looking at.

The C++ language requires that any coroutine be resumable from a `coroutine_handle<>`, so there needs to be some vtable-like thing so that calling the `resume()` method on an arbitrary `coroutine_handle<>` resumes the correct coroutine.

Note: The Microsoft Visual C++ compiler coroutine ABI took a breaking change in version 16.8, so I'll cover Microsoft Visual C++ coroutines twice, once in C++17 mode and once again in C++20 mode.

In the Microsoft Visual C++ compiler, the C++17-style coroutine handle is a pointer to a structure we shall call a "frame" for expository purposes.

```
struct coroutine_frame
{
    void (*resume)(coroutine_frame*);
    uint16_t index;
    uint16_t flags;
    promise_type promise;
    parameters...
    locals...
    temporaries...
    other bookkeeping...
};
```

The `index` represents the progress of the coroutine through its function body. The `flags` value is nonzero if the coroutine frame was allocated on the heap.

Constructing a coroutine frame consists of the following steps:

- Allocate memory for the frame, usually from the heap.
- Initialize the `resume` member to point to a custom function specific to the coroutine.

- Initialize the `index` to 2.
- Initialize the `flags` to 1 if the frame was allocated on the heap; otherwise initialize it to zero.

The index is initialized to 2 because the state of a suspended coroutine is always recorded as a nonzero even number.

- Nonzero: I'm guessing that zero is kept as a permanently invalid state to aid in debugging.
- Even: We'll see why later.

When a coroutine suspends, its `index` is updated to remember where the coroutine needs to resume. The coroutine states appear to be numbered in the order in which they appear in the function, so the initial state of the coroutine is 2, the first suspension point is 4, the next one is 6, and so on.[1] Some of the suspension points can get optimized out, say, because the compiler can prove that `await_ready` always returns `true`.

To resume a suspended coroutine, call the `resume` function with a pointer to the coroutine frame. Each coroutine gets a custom `resume` function which uses the index as an index into a jump table to dispatch to the appropriate point in the coroutine where execution should resume.

For 32-bit code, the jump table is an array of addresses to jump to. For 64-bit code, the jump table is an array of relative virtual addresses which need to be added to the module base address to for the code address. Using relative virtual addresses keeps the jump table smaller and also reduces the number of relocations needed.

To destroy a suspended coroutine, set the bottom bit of the index (turning it into an odd number), and then call the `resume` function. The odd entries in the jump table point to cleanup functions which destruct the variables that were live at the point of suspension. And if the `flags` say that the coroutine was allocated on the heap, then it is `delete` from the heap.

The Microsoft Visual C++ compiler uses the naming convention `function$_ResumeCoro$N` for the coroutine `resume` function, for some number *N*. (I haven't yet figured out what the *N* means.) Here's a 64-bit example:

```
function$_ResumeCoro$1:
    mov     [rsp+8], rcx            ; save coroutine frame
    push    rbx
    sub     rsp, 30h                ; build stack frame
    mov     rbx, [rsp+40h]          ; rbx = coroutine frame
    movzx   eax, word ptr [rbx+8]   ; eax = index
    mov     [rsp+20h], ax           ; remember the index
    inc     ax                      ; add 1, just for fun
    cmp     ax, 6
    ja      fatal_error             ; invalid index
    movsx   rax, ax
    lea     rdx, [__ImageBase]      ; get module base address
    mov     ecx, [rdx+rax*4+3158h]  ; get offset from jump table
    add     rcx,rdx                 ; apply offset to base address
    jmp     rcx                     ; jump there
```

Note that the compiler *adds one* to the index before using it to look up the offset in the jump table, so you need to ignore the first entry in the jump table.

The clang compiler uses a slightly different approach:

```
struct coroutine_frame
{
    void (*resume)(coroutine_frame*);
    void (*destroy)(coroutine_frame*);
    uintN_t index;
    /* parameters, local variables, other bookkeeping */
};
```

Instead of encoding the "destroying" state in the bottom bit of the index, clang uses a separate `destroy` function. This means that the indices are small integers, with no special meaning for even/odd values. (Zero is a valid index.) The `resume` and `destroy` functions have separate jump tables, one for resumption and one for destruction, and if the number of states is small, then clang doesn't even bother making a jump table; it just uses a bunch of tests. The size of the variable used to hold the state is chosen to be large enough to hold all of the states. Most reasonable-sized coroutines can get by with an 8-bit index, but the compiler internally supports indices up to 32 bits in size.

The gcc compiler sits somewhere in between the Microsoft and clang compilers.

```
struct coroutine_frame
{
    void (*actor)(coroutine_frame*);
    void (*destroy)(coroutine_frame*);
    uint8_t unused;
    uint8_t flags;
    uint16_t index;
    /* parameters, local variables, other bookkeeping */
};
```

Like the clang compiler, the gcc compiler uses a pair of function pointers, one for resuming the coroutine (which is internally called the `actor`) and one for destroying it. However, the gcc compiler follows the Microsoft C++ convention of using even numbers for suspended states and odd numbers for destroying states. The `destroy` function just sets the bottom bit of the `index` and then jumps to the `actor` function.

Inside the `actor` function, the code checks the bottom bit of the `index` and dispatches from two different jump tables, one for even indices and one for odd indices. Curiously, the table for even indices has `fatal_error` in all the odd slots, and the table for odd indices has `fatal_error` in all the even slots, so really, they could have been combined into a single table. Not sure what what's about.

The `flags` records whether the coroutine function's parameters have been transferred to the frame. This is used when the frame is destroyed to know whether or not there are parameters in the frame which need to be destructed.

Finally, we come to Microsoft Visual C++ coroutines in C++20 mode. As noted in their blog post, the change was made in order to be ABI-compatible with clang and gcc, so that coroutines from all three compilers can interoperate.

```
struct coroutine_frame
{
    void (*resume)(coroutine_frame*);
    void (*destroy)(coroutine_frame*);
    promise_type promise;
    parameters...
    uint16_t index;
    uint16_t flags;
    locals...
    temporaries...
    other bookkeeping...
};
```

The original `resume` function has been split into separate `resume` and `destroy` functions, and the other members of the coroutine frame have been rearranged.

Adding a `destroy` function to the start of the coroutine frame establishes the *de facto* common ABI for coroutine frames:

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

For all four coroutine frame formats, you can figure out what coroutine a coroutine handle corresponds to by dumping the start of the frame and looking at the `resume` pointer. You can also look at the `index` to see where in the coroutine's execution you are, although for Microsoft Visual C++ coroutines in C++20 mode, the index is not at a fixed location, so digging it out will require you to disassemble the `resume` function to see where it reads the index from.

In all cases, you'll have to disassemble the `resume` function to find the jump table (or for clang, the switch statement) but you can then index into that jump table (after adjusting by 1 for the Microsoft C++ compiler) to find the point at which execution is going to resume.

Here's the cookbook in a table:

	Microsoft Visual C++	**clang**	**gcc**
Identify coroutine from handle	Dump first pointer as a function pointer		
Is coroutine destroying?	Index is odd	(no way to tell)	Index is odd
Where will it resume?	Disassemble resumption function add 1 to index look up in jump table	Disassemble resumption function follow switch statement	Disassemble resumption function find the right jump table (even/odd) look up in jump table

[1] I've never created a coroutine with more than 32767 suspension points, nor do I have any interest in trying, so I don't know whether the compiler switches to a 32-bit `index` or whether it just bails out with "Error: Coroutine has too many suspension points."

Raymond Chen

Follow