The AArch64 processor (aka arm64), part 24: Code walkthrough

devblogs.microsoft.com/oldnewthing/20220829-00

August 29, 2022

Raymond Chen

As is traditional, I wrap up the processor overview series with an annotated walkthrough of a simple function. Here's the function again:

```
extern FILE _iob[];

int fclose(FILE *stream)
{
    int result = EOF;

    if (stream->_flag & _IOSTRG) {
        stream->_flag = 0;
    } else {
        int index = stream - _iob;
        _lock_str(index);
        result = _fclose_lk(stream);
        _unlock_str(index);
    }

    return result;
}
```

Here we go.

This function takes a single pointer parameter, which is therefore passed in the *x0* register. No parameters are passed on the stack.

```
; int fclose(FILE *stream)

    stp     x19, x20, [sp,#-0x20]!
    str     x21, [sp,#0x10]
    stp     fp, lr, [sp,#-0x10]!
    mov     fp, sp
```

We start with the function prologue, which creates the stack frame and saves nonvolatile registers that we will be using inside the function.

The first instruction reserves 32 bytes of stack and stores *x19* and *x20* into the first two slots. The pre-increment addressing mode (signaled by the exclamation point) updates the base register *sp* with the effective address, so this both stores the registers to memory as well as moving the stack pointer.

The second instruction stores the *x21* variable into the memory that follows *x20*. The last 8 bytes are not used; they were allocated in order to preserve 16-byte stack pointer alignment.

The third instruction pushes the frame pointer and link register into the stack. Notice that this function adjusted the stack pointer twice. I'm not sure how the compiler decides whether to reserve stack space all at once, or whether to reserve it little by little, like we did here.

After all the registers have been stored, we set *fp* to the current stack pointer, which makes it point to where we stored the previous *fp*, thereby linking a new node onto the chain of stack frames.

Now that the prologue is out of the way, we can start with the function body.

```
    mov     x20, x0             ; x20 = stream
```

The compiler takes the *stream* parameter, which was received in *x0*, and saves it in the nonvolatile register *x20* so it can be preserved across function calls.

```
; int result = EOF;
; if (stream->_flag & _IOSTRG) {

    ldr     w8, [x20, #0xC]     ; w8 = stream->_flag
    mov     w21, #-1            ; w21 = EOF
    tbz     x8, #6, nostring    ; branch if _IOSTRG bit is zero
```

The work for the next two lines of code are interleaved. The compiler appears to have chosen to use *w21* to hold the `result` variable, so it initializes it to `-1`. The disassembler shows it as a `MOV`, but the raw instruction is really a `MOVN w21, #0, LSL #0`.

The initialization of the `result` variable is sandwiched between the test for the `_IOSTRG` bit. We load the value of `_flag` into the *w8* register and test bit 6, which is the bit that corresponds to `_IOSTRG`, branching if the bit is clear (test bit zero).

```
;    stream->_flag = 0;
; } else {

    str     wzr, [x20, #0xC]    ; set _flag to zero
    b       done                ; end of "true" branch
```

If the branch is not taken, we fall through and store a 32-bit zero to `_flag`. That's the end of the "true" branch.

```
;   int index = stream - _iob;

nostring:
    adrp    x8, sample+0x2000   ; load high bits of pointer
    add     x8, x8, #0x0180     ; x8 -> _iob
    sub     x9, x20, x8         ; calculate byte offset
    asr     x19, x9, #4         ; x19 = convert to element offset
```

In the “false” branch, we calculate the stream index. First, we load up the address of the `_iob`. This takes two instructions, the first to load up the page that holds the `_iob` variable, and the second to find the `_iob` within that page.

Subtract the `_iob` from the `stream` to get the byte offset, and convert it to an index by dividing by the size of a single `FILE`, which happens to be 16, so dividing can be done by shifting. The index is kept in *x19*.

```
;   _lock_str(index);

    mov     w0, w19             ; parameter is the index
    bl      _lock_str
```

The index is the sole parameter to `_lock_str`, so we put it into *w0* and call the function.

```
;   result = _fclose_lk(stream);

    mov     x0, x20             ; parameter is the stream
    bl      _fclose_lk          ; call _fclose_lk
    mov     w21, w0             ; save the result
```

Next up is calling `_fclose_lk` with the stream as the parameter. We save the return value into *w21* which represents the `result` variable.

```
;   _unlock_str(index);
; }

    mov     w0, w19             ; parameter is the index
    bl      _unlock_str
```

Unlocking the string is done by index, which is fortunately still sitting around in the *w19* register.

```
; return result;

done:
    mov     w0, w21
```

The function return value goes into *x0*, so we move *w21* (representing `result`) into the lower 32 bits of the *x0* register.

```
; }
    ldp     fp, lr, [sp], #0x10
    ldr     x21, [sp, #0x10]
    ldp     x19, x20, [sp], #0x20
    ret
```

And we're done. Now it's time to clean up. We pop off the previous frame pointer and return address, the restore and pop the other nonvolatile registers we had saved. Finally we perform a `ret` to jump back to the return address in *lr*.

When I do these walkthrough, I look to see if there was anything I could do to tighten up the code. The interesting thing that the compiler failed to recognize is that the lifetimes of `result` and `stream` do not overlap in any meaningful way, so they could share the same register. This reduces the number of registers by one, which saves 16 bytes of stack since we no longer need to save *x21*.

Another trick is to fold the `asr` into the `mov` instruction that sets up the `index` parameter, saving an instruction.

```
; int fclose(FILE *stream)
    stp     x19, x20, [sp,#-0x10]!  ; NEW! Need only 0x10 bytes
                                ; NEW! Don't need to save x21
    stp     fp, lr, [sp,#-0x10]!
    mov     fp, sp

    mov     x20, x0             ; x20 = stream

; int result = EOF;
; if (stream->_flag & _IOSTRG) {

    ldr     w8, [x20, #0xC]     ; w8 = stream->_flag
    tbz     x8, #6, nostring    ; branch if _IOSTRG bit is zero

;    stream->_flag = 0;
; } else {

    str     wzr, [x20, #0xC]    ; set _flag to zero
                                ; NEW! "stream" is dead, so
                                ;      w20 now represents "result"
    mov     w20, #-1            ; result = EOF
    b       done                ; end of "true" branch

;   int index = stream - _iob;

nostring:
    adrp    x8, sample+0x2000   ; load high bits of pointer
    add     x8, x8, #0x0180     ; x8 -> _iob
    sub     x19, x20, x8        ; calculate byte offset (x19)

;   _lock_str(index);

                                ; NEW! Convert byte offset to index
                                ;      on the fly
    asr     w0, w19, #4         ; parameter is the index
    bl      _lock_str

;   result = _fclose_lk(stream);
;   _unlock_str(index);
; }

    mov     x0, x20             ; parameter is the stream
    bl      _fclose_lk          ; call _fclose_lk

                                ; NEW! "stream" is dead, so
                                ;      w20 now represents "result"
    mov     w20, w0             ; save the result

                                ; NEW! Convert byte offset to index
                                ;      on the fly
    asr     w0, w19, #4         ; parameter is the index
    bl      _unlock_str
```

```
; return result;

done:
    mov     w0, w20

; }
    ldp     fp, lr, [sp], #0x10
                                ; NEW! Don't need to restore x21
    ldp     x19, x20, [sp], #0x10
    ret
```

This is really just recreational optimization at this point. The extra few instructions in the compiler-generated code is not going to be noticeable here, seeing as the `fclose` function is probably going to do things like close file handles, which are far more expensive than just a few instructions.

This concludes our quick overview of the ARM processor in 64-bit mode. Now when you have to look at a crash dump on an ARM64 system, you might have a clue about what you're looking at.

Raymond Chen

Follow