The ARM processor (Thumb-2), part 18: Other kinds of prologues and epilogues

devblogs.microsoft.com/oldnewthing/20210623-00

June 23, 2021

Raymond Chen

Last time, we looked at the standard function prologue and epilogue. There are some variations to the standard that you may encounter from time to time.

Lightweight leaf functions are functions which meet all of the following criteria:

- Modify only the non-preserved registers: *r0* through *r3* and *r12*, and *d0* through *d7* and *d16* through *r31*, and flags.
- Do not use any stack aside from inbound parameter space.

Lightweight leaf functions do not create a stack frame. They must keep the return address in the *lr* register for the entire lifetime so that the kernel can unwind the function to its caller. The requirement that it use only non-preserved registers allows the kernel to unwind without using any unwind codes, since there are no registers that need to be restored during unwinding.

Conversely, any function that lacks unwind codes is assumed to be a lightweight leaf function.

Another variation is the shrink-wrapped function. This is a function that starts out with a small stack frame (or no stack frame at all, pretending to be a lightweight leaf function), in the hope that it can early-out. If not, then it expands to a full stack frame.

If a function uses 16 or fewer bytes of local variables and outbound parameters, it can include up to four dummy registers to the initial `push`:

```
push    {r0-r7,r11,lr}
```

The part you recognize is the saving of registers *r4* through *r7*, plus the frame pointer and return address. The sneaky part is that it also saves registers *r0* through *r3*. These extra registers are pushed, not so much because the function wants to save them, but because pushing four additional registers implicitly subtracts 4 ×4 = 16 bytes from the *sp* register, allocating the local variables and outbound parameters as part of the initial `push`.

In the epilogue, you can use the reverse trick to clean up those extra 16 bytes as part of the final pop:

```
pop     {r0-r7,r11,pc}
```

However, if your function needs to return a value in *r0* (and possibly *r1*), you can't pop them in your optimized epilogue, because that would clobber your return value. You'll have to use an old-fashioned `add sp, sp, #n` to discard those bytes from the stack.

If the function is variadic, it will probably start with a

```
push    {r0-r3}
```

This pushes the first 16 bytes of parameters onto the stack, so that they line up exactly adjacent to the stack-based parameters. That way, the code that walks the parameter list can just walk through memory uniformly.

This extra push instruction in the prologue requires a change to the epilogue, because our usual trick of popping the return address into *pc* isn't going to work.

```
add     sp, sp, #0x20       ; free locals and outbound stack parameters
pop     {r4-r7,r11}         ; restore registers but leave return address
ldr     pc, [sp], #0x14     ; return and clean extra stack space
```

Things start out innocently enough, but this time, the `pop` instruction leaves the return address on the stack, and the *r0* through *r3* registers are still on the stack, too. At this point, we have this diagram:

return address	
previous *r11*	← *r11* (frame chain)
⋮	
stack param	
saved *r3*	
saved *r2*	
saved *r1*	
saved *r0*	
return address	← *sp*

The magic instruction that finishes the function is

```
ldr     pc, [sp], #0x14     ; return and clean extra stack space
```

Let's take this instruction apart.

First, it loads *pc* from the stack pointer. Loading a value into *pc* acts like a jump instruction, so the next instruction to execute when this one is complete will be the instruction at the return address.

The `, #0x14` suffix means that this is using the post-increment addressing mode. After the register is loaded from memory, the base register (*sp*) is incremented by `0x14`. This moves the stack pointer past the saved return address as well as the 16 bytes occupied by the registers *r0* through *r3* we had pushed at function entry.

The last trick I'll talk about is tail call optimization. The epilogue for this function goes like this:

```
add     sp, sp, #0x20       ; free locals and outbound stack parameters
pop     {r4-r7,r11,lr}      ; restore registers and set lr to return address
b       next_function
```

After cleaning up the local variables and outbound stack parameters, we pop off everything that we saved, but instead of putting the return address into *pc* like we usually do, we pop it back into *lr*. This preserves the requirement that on entry to a function, the *lr* register holds the return address. We can now jump directly to the entry point of the tail call target.

Well, that was an exciting tour of function prologues and epilogues. Next time, we'll look at common code sequences you should learn to recognize.

Raymond Chen

Follow