EL0<->EL1 switching

Hello There!

Let's continue on our journey of understanding a mammoth that is the ARM64 architecuture.

In the Previous artcile, we went throught different exception levels. Would encourage perusing through it once in case you'd want to jog your memories. In this one, we would together want to attempt to deep dive into switching between two of the exception levels - `EL0` and `EL1`.

We'll walk through the process of a system call from EL0 to EL1 and the return journey, focusing on context switching, the vector table, synchronous exception handling, and key registers like ESR_EL1 and ELR_EL1. We will see from the lens of code snippets to see how implementation might look in action. Finally, we will piece together the snippets to arrive at full switching flow.

Quick Recap

ARMv8-A organizes privilege levels into Exception Levels (EL0 to EL3). * EL0 is where user applications run—unprivileged, restricted access. * EL1 is typically the OS kernel, with more privileges like accessing system registers. * Switching between these levels happens during exceptions, like system calls, interrupts, or faults. * A system call (via the `svc` instruction) is a synchronous exception that moves execution from EL0 to EL1. The return to EL0 is handled by the `eret` instruction.

So that's it. Thanks for reading! Not quite though, right? Our interest is going in the details and that's what we are going to do.

Triggering the System Call from EL0

Let's start with a basic example and say I'm writing a user program in EL0, and I want to make a system call to, say, write to a file (like Linux's `write` syscall). The `svc` instruction is my gateway. It generates a synchronous exception, causing the processor to switch to EL1 and jump to a predefined handler in the vector table.

Here's a simple EL0 code snippet to trigger a system call:

```
// user program in EL0 making a system call (e.g., write syscall)
mov x8, #64           // syscall number for write (Linux ARM64)
mov x0, #1            // file descriptor (stdout = 1)
adr x1, print_msg     // pointer to message
mov x2, #12           // length of message
```

```
svc #0                  // trigger system call to EL1


print_msg:
    .asciz "Hey from EL0, this message is for EL1\n"
```

What's happening here? I load the system call number (64 for `write` in Linux ARM64) into `x8`, set up arguments in `x0-x2` (per the Linux ABI), and issue `svc #0`. The immediate value (`#0`) is mostly ignored in practice but can be used by the kernel to differentiate types of supervisor calls. When `svc` executes, the processor (automatically happens in h/w, no s/w intervention is required):

- Switches to EL1.
- Saves the program counter (PC) to ELR_EL1 (Exception Link Register for EL1).
- Saves the status register (PSTATE) to SPSR_EL1 (Saved Program Status Register for EL1).
- Jumps to the vector table entry for synchronous exceptions.

Note: The `svc` instruction doesn't save general-purpose registers (`x0-x30`). That's on the kernel to handle, or you risk clobbering user state.

Homework: Imagine any malicious user passes garbage or some unsavory data in registers (e.g., an invalid pointer in `x1`), the kernel needs robust validation to avoid crashes. Can you come up with some modifcations in our snippet to achieve that?

The Vector Table and Synchronous Exception Handling

When the `svc` instruction fires, the processor looks at the Vector Base Address Register (VBAR_EL1 - it is programmed during the boot phase, we will talk about this in future artciles when we talk about the boot flow) to find the vector table—a table of exception handlers in EL1. The table has entries for different exception types, including synchronous exceptions like system calls.

For a 64-bit EL0 app, the processor jumps to the "Synchronous, Lower EL, AArch64" entry.

Code snippet of what a vector table would look like:

```
// vector table in EL1
.align 11       // align to 2KB boundary (ARMv8a architecture requirement)


vector_table:
    // synchronous exception from current EL with SP0
    b sync_handler
    .align 7    // each entry is 128 bytes (ARMv8a architecture requirement)
    // other entries (interrupts, FIQ, etc.)
```

```
    b .
    .align 7
    b .
    .align 7
    b .
    .align 7
    // synchronous exception from lower EL (EL0, AArch64)
    b sync_lower_el

sync_handler:
    // handle synchronous exceptions (e.g., svc from EL1)
    b .

sync_lower_el:
    // save context, parse the ESR, handle syscall, restore the context, return to EL0
    bl save_context             // save registers (Explained in detail in the next section)
    bl parse_and_handle_syscall // call syscall handler (Explained in detail in the next section)
    bl restore_context          // restore the registers (Explained in detail in the next section)
    eret                        // return to EL0
```

The vector table must be aligned to a 2KB boundary (hence `.align 11`). Each entry is 128 bytes, allowing for small handlers or branches to larger ones. For our system call, the processor jumps to `sync_lower_el`. The handler saves critical registers and calls a syscall handler. The `eret` instruction restores execution to EL0.

Context Switching

In EL1, the kernel saves the EL0 context (general-purpose registers, etc.) to ensure the user program can resume. On return, it restores the context before `eret`.

Context save/restore sub-routines:

```
// context save in EL1
save_context:
    sub sp, sp, #256           // allocate space for context
    stp x0, x1, [sp, #16 * 0]  // save x0, x1
    stp x2, x3, [sp, #16 * 1]  // save x2, x3
    stp x4, x5, [sp, #16 * 2]
    stp x6, x7, [sp, #16 * 3]
    stp x8, x9, [sp, #16 * 4]
    stp x10, x11, [sp, #16 * 5]
    stp x12, x13, [sp, #16 * 6]
```

```
    stp x14, x15, [sp, #16 * 7]
    stp x16, x17, [sp, #16 * 8]
    stp x18, x19, [sp, #16 * 9]
    stp x20, x21, [sp, #16 * 10]
    stp x22, x23, [sp, #16 * 11]
    stp x24, x25, [sp, #16 * 12]
    stp x26, x27, [sp, #16 * 13]
    stp x28, x29, [sp, #16 * 14]
    stp x30, xzr, [sp, #16 * 15]  // save x30, pad with zero
    ret

// context restore in EL1
restore_context:
    ldp x0, x1, [sp, #16 * 0]     // restore x0, x1
    ldp x2, x3, [sp, #16 * 1]     // restore x2, x3
    ldp x4, x5, [sp, #16 * 2]
    ldp x6, x7, [sp, #16 * 3]
    ldp x8, x9, [sp, #16 * 4]
    ldp x10, x11, [sp, #16 * 5]
    ldp x12, x13, [sp, #16 * 6]
    ldp x14, x15, [sp, #16 * 7]
    ldp x16, x17, [sp, #16 * 8]
    ldp x18, x19, [sp, #16 * 9]
    ldp x20, x21, [sp, #16 * 10]
    ldp x22, x23, [sp, #16 * 11]
    ldp x24, x25, [sp, #16 * 12]
    ldp x26, x27, [sp, #16 * 13]
    ldp x28, x29, [sp, #16 * 14]
    ldp x30, xzr, [sp, #16 * 15]
    add sp, sp, #256            // free up the space (remember we allocated this in save_context)
    ret
```

Here is the key is store all the registers (`x0-x30`) to a stack frame (256 bytes). Why all registers? The kernel might clobber them (we don't want to put any restrictions on the kernel, do we now?), and the user expects preservation of its registers (except `x0` for return values). The restore function pops registers back. You'd see a very similar looking code in any major kernel source out there.

Homework: You'd see a very similar looking code in any major kernel source out there. There might be some extra registers (e.g Floating point registers, SVE/NEON registers) which might also be saved/restored. Analyzed the Linux source and jot-down those differences.

Parsing ESR_EL1

The Exception Syndrome Register (ESR_EL1) identifies the exception cause. Remember, h/w will only be able to identify that a synchronous exception from lower EL and the control would be transferred to the appropriate entry in vector table which is `sync_lower_el`. We will have to identify if it is indeed a `svc` call or not, and for that, we will have to decode the exception syndrome register.

For a system call, it confirms an `svc` instruction and provides details.

```
// Parse annd handle syscalls
parse_and_handle_syscall:
   // parse ESR_EL1 to identify exception type and handle them accordingly
    mrs x0, esr_el1        // read ESR_EL1
    ubfx x1, x0, #26, #6   // extract Exception Class bits - BITs 31:26
    cmp x1, #0x15          // EC for SVC from AArch64 = 0x15
    b.eq handle_svc        // branch to SVC handler (at this point we have with certainity figured it
    // handle other exceptions
    b other_exception


handle_svc:
    ubfx x2, x0, #0, #16   // extract ISS (Instruction Specific Syndrome)
    // x2 contains SVC immediate
    ret


other_exception:
    // handle other cases
    b .
```

The Exception Class (EC, bits 31:26) is `0x15` for `svc` from AArch64. The ISS (bits 15:0) holds the `svc` immediate.

ELR_EL1 and Returning to EL0

ELR_EL1 holds the return address (usually the instruction after `svc`). The kernel can modify it for special cases.

```
// prepare to return to EL0
return_to_el0:
    // ELR_EL1 holds return address
    // SPSR_EL1 holds saved PSTATE
    bl restore_context
    eret                   // return to EL0
```

The `eret` instruction restores PC from ELR_EL1 and state from SPSR_EL1.

Complete Example

Let’s piece everything together:

```
// EL1 syscall handler
.align 11
vector_table:
    b .
    .align 7
    b .
    .align 7
    b .
    .align 7
    b .
    .align 7
    b sync_lower_el

sync_lower_el:
    // save context
    sub sp, sp, #256
    stp x0, x1, [sp, #16 * 0]
    stp x2, x3, [sp, #16 * 1]
    stp x4, x5, [sp, #16 * 2]
    stp x6, x7, [sp, #16 * 3]
    stp x8, x9, [sp, #16 * 4]
    stp x10, x11, [sp, #16 * 5]
    stp x12, x13, [sp, #16 * 6]
    stp x14, x15, [sp, #16 * 7]
    stp x16, x17, [sp, #16 * 8]
    stp x18, x19, [sp, #16 * 9]
    stp x20, x21, [sp, #16 * 10]
    stp x22, x23, [sp, #16 * 11]
    stp x24, x25, [sp, #16 * 12]
    stp x26, x27, [sp, #16 * 13]
    stp x28, x29, [sp, #16 * 14]
    stp x30, xzr, [sp, #16 * 15]

    // parse ESR_EL1
    mrs x0, esr_el1
    ubfx x1, x0, #26, #6
    cmp x1, #0x15
```

```
    b.ne other_exception

    // handle syscall (e.g., write)
    cmp x8, #64
    b.eq handle_write
    b other_syscall

handle_write:
    // simplified write syscall
    // In proper design, this would be a correct place to jump to C realm to handle write()
    bl kernels_write_implementation
    mov x0, #42        // return value
    b return_to_el0

other_syscall:
    // handle other syscalls
    mov x0, #-1        // error return
    b return_to_el0

other_exception:
    // handle other exceptions
    mov x0, #-1
    b return_to_el0

return_to_el0:
    // restore context
    ldp x0, x1, [sp, #16 * 0]
    ldp x2, x3, [sp, #16 * 1]
    ldp x4, x5, [sp, #16 * 2]
    ldp x6, x7, [sp, #16 * 3]
    ldp x8, x9, [sp, #16 * 4]
    ldp x10, x11, [sp, #16 * 5]
    ldp x12, x13, [sp, #16 * 6]
    ldp x14, x15, [sp, #16 * 7]
    ldp x16, x17, [sp, #16 * 8]
    ldp x18, x19, [sp, #16 * 9]
    ldp x20, x21, [sp, #16 * 10]
    ldp x22, x23, [sp, #16 * 11]
    ldp x24, x25, [sp, #16 * 12]
    ldp x26, x27, [sp, #16 * 13]
    ldp x28, x29, [sp, #16 * 14]
```

```
    ldp x30, xzr, [sp, #16 * 15]
    add sp, sp, #256
    eret
```

Homework: I've deliberately kept a subtle bug in the above snippet. Can you catch it? Hint - it is related to return values being correctly communicated back.

In the next one, we will talk about the switching of other layers and even secure and non-secure worlds. We will take a simple use-case and trace its flow. See you in the next one!