Lab 3: Interrupts and Exceptions

习题解答

1. 中断流程

> 请简要描述一下在你实现的操作系统中，中断时 CPU 进行了哪些操作。

1. 首先关闭所有中断，不再接受来自外部的中断请求，所有的中断只会在用户态发生，即只有从 EL0 到 EL1 的中断
2. 根据当前中断类型，跳转到中断向量表 `vectors` 的对应位置
3. 保存上下文，将所有通用寄存器和 3 个特殊寄存器 ELR_EL1（中断前 PC）、SPSR_EL1（中断前 PSTATE）、SP_EL0（用户态栈指针 SP）压入堆栈（建立 Trap frame）
4. 跳转到中断函数 `trap` 入口，根据中断路由选择当前的中断服务（局部时钟 timer 中断、全局时钟 clock 中断、UART 输入中断等）
5. 中断处理程序完成后，恢复上下文，将所有通用寄存器和 3 个特殊寄存器 ELR_EL1、SPSR_EL1、SP_EL0 弹出堆栈还原；中断返回，主程序继续执行

2. Trap frame 设计

> 请在 `inc/trap.h` 中设计你自己的 trap frame，并简要说明为什么这么设计。

本操作系统中，Trap frame 包含 31 个通用寄存器和 3 个特殊寄存器 SP_EL0, SPSR_EL1, ELR_EL1。

为什么要保存所有通用寄存器，而不是仅 callee-saved 或 caller-saved 寄存器？因为这是操作系统内核中断，而不是普通用户态的函数调用，CPU 需要充分保障程序数据安全，callee-saved 和 caller-saved 只是君子协定。

为什么要保存这 3 个特殊寄存器？因为中断返回（`eret`）时需要还原中断前 PSTATE（保存在 SPSR_EL1）、中断前 PC（保存在 ELR_EL1），以及还原用户态栈指针 SP（保存在 SP_EL0）。[1]

```
struct trapframe {
    // General-Purpose Registers
    uint64_t x0;
    uint64_t x1;
    uint64_t x2;
    uint64_t x3;
    uint64_t x4;
    uint64_t x5;
    uint64_t x6;
    uint64_t x7;
    uint64_t x8;
    uint64_t x9;
    uint64_t x10;
    uint64_t x11;
    uint64_t x12;
    uint64_t x13;
    uint64_t x14;
    uint64_t x15;
    uint64_t x16;
    uint64_t x17;
    uint64_t x18;
    uint64_t x19;
    uint64_t x20;
    uint64_t x21;
    uint64_t x22;
    uint64_t x23;
    uint64_t x24;
    uint64_t x25;
    uint64_t x26;
    uint64_t x27;
    uint64_t x28;
    uint64_t x29;  // Frame Pointer
    uint64_t x30;  // Procedure Link Register

    // Special Registers
    uint64_t sp_el0;    // Stack Pointer
    uint64_t spsr_el1;  // Program Status Register
    uint64_t elr_el1;   // Exception Link Register
};
```

3. Trap frame 构建与恢复

> 请补全 `kern/trapasm.S` 中的代码，完成 trap frame 的构建、恢复。

3.1 Trap frame 构建

将所有通用寄存器和 3 个特殊寄存器 ELR_EL1、SPSR_EL1、SP_EL0 压入堆栈，然后跳转到中断函数 `trap` 入口。

```
# kern/trapasm.S

/* vectors.S send all traps here. */
.global alltraps
alltraps:
    /* Build your trap frame. */

    stp x29, x30, [sp, #-16]!
    stp x27, x28, [sp, #-16]!
    stp x25, x26, [sp, #-16]!
    stp x23, x24, [sp, #-16]!
    stp x21, x22, [sp, #-16]!
    stp x19, x20, [sp, #-16]!
    stp x17, x18, [sp, #-16]!
    stp x15, x16, [sp, #-16]!
    stp x13, x14, [sp, #-16]!
    stp x11, x12, [sp, #-16]!
    stp x9, x10, [sp, #-16]!
    stp x7, x8, [sp, #-16]!
    stp x5, x6, [sp, #-16]!
    stp x3, x4, [sp, #-16]!
    stp x1, x2, [sp, #-16]!

    mrs x3, elr_el1
    mrs x2, spsr_el1
    mrs x1, sp_el0
    stp x3, x0, [sp, #-16]!
    stp x1, x2, [sp, #-16]!

    /* Call trap(struct *trapframe). */

    add x0, sp, #0
    bl  trap
```

3.2 Trap frame 恢复

将所有通用寄存器和 3 个特殊寄存器 ELR_EL1、SPSR_EL1、SP_EL0 弹出堆栈还原，然后中断返回（`eret`）。

```
# kern/trapasm.S

/* Return falls through to trapret. */
.global trapret
trapret:
    /* Restore registers. */

    ldp x1, x2, [sp], #16
    ldp x3, x0, [sp], #16
    msr sp_el0, x1
    msr spsr_el1, x2
    msr elr_el1, x3

    ldp x1, x2, [sp], #16
    ldp x3, x4, [sp], #16
    ldp x5, x6, [sp], #16
    ldp x7, x8, [sp], #16
    ldp x9, x10, [sp], #16
    ldp x11, x12, [sp], #16
    ldp x13, x14, [sp], #16
    ldp x15, x16, [sp], #16
    ldp x17, x18, [sp], #16
    ldp x19, x20, [sp], #16
    ldp x21, x22, [sp], #16
    ldp x23, x24, [sp], #16
    ldp x25, x26, [sp], #16
    ldp x27, x28, [sp], #16
    ldp x29, x30, [sp], #16

    eret
```

测试环境

- OS: Ubuntu 18.04.5 LTS (WSL2 4.4.0-19041-Microsoft)
- Compiler: gcc version 8.4.0 (Ubuntu/Linaro 8.4.0-1ubuntu1~18.04)
  - Target: aarch64-linux-gnu
- Debugger: GNU gdb 8.2 (Ubuntu 8.2-0ubuntu1~18.04)
  - Target: aarch64-linux-gnu
- Emulator: QEMU emulator version 5.0.50
- Using GNU Make 4.1

---

1. ARM Cortex - A Series Programmer's Guide for ARMv8-A ↩