《操作系统》

实验四报告

目录

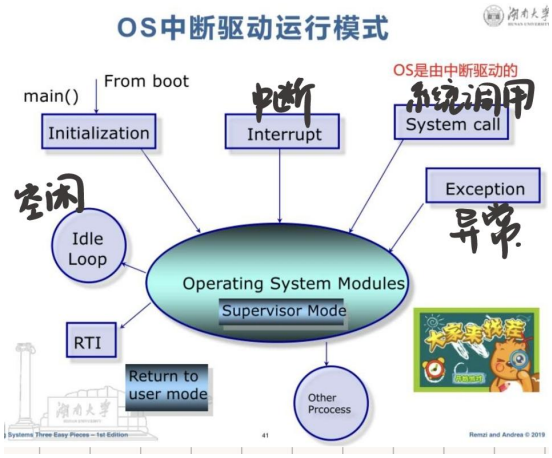
1 实验代码分析

1.1 实验目的

本实验和实验五的目的在于深刻理解中断的原理和机制，掌握 CPU 访问中断控制器的方法，掌握 Arm 体系结构的中断机制和规范，实现时钟中断服务和部分异常处理等。

1.2 ARMv8 的中断与异常处理

操作系统是一个多入口的程序，执行陷阱（Trap）指令，出现异常、发生中断时都会陷入到操作系统。



AArch64 中共有 4 个异常级别，分别为 EL0，EL1，EL2 和 EL3。在 AArch64 中，Interrupt 是 Exception 的子类型，称为异常。AArch64 中有四种类型的异常：

Sync（Synchronous exceptions，同步异常），在执行时触发的异常，例如在尝试访问不存在的内存地址时。

IRQ （Interrupt requests，中断请求），由外部设备产生的中断。

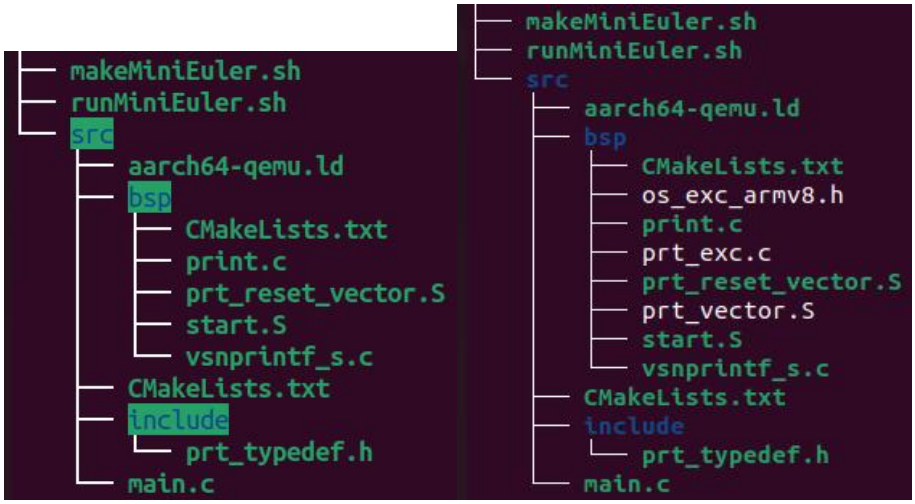
FIQ （Fast Interrupt Requests，快速中断请求），类似于 IRQ，但具有更高的优先级，因此 FIQ 中断服务程序不能被其他 IRQ 或 FIQ 中断。

SError （System Error，系统错误），用于外部数据中止的异步中断。

当异常发生时，处理器将执行与该异常对应的异常处理代码。在 ARM 架构中，这些异常处理代码将会被保存在内存的异常向量表中，每一个异常级别（EL0，EL1，EL2 和 EL3）都有其对应的异常向量表。需要注意的是，与 x86 等架构不同，该表包含的是要执行的指令，

而不是函数地址。

1.3 代码结构



可以对应 lab2 的结构来看，make 和 run 两个脚本文件用于自动化编译和执行；aarch64-qemu.ld 是创建的自定义链接器脚本，通常包含对内存布局的描述、符号解析的规则、代码和数据段的布局以及如何将这些段映射到输出文件中的特定位置；**bsp 目录存放与硬件紧密相关的代码**，include 目录中存放项目的大部分头文件；prt_typedef.h 头文件：它是 UniProton 所使用的基本数据类型和结构的定义，如 U8、U16、U32、U64 等；main.c 是我们的主函数，Start 和 prt_reset_vector.S 也是执行的一部分。print.c 和 vsnprintf_s.c 用于实验二的串口输出。

本实验在 bsp 文件夹中新增了 prt_vector.S 、prt_exc.c 和 os_exc_armv8.h ，同时在 src/bsp/CMakeLists.txt 中增加了链接方式，并修改了 main.c 文件和 start.S 文件。

1.4 异常向量表

1.4.1 新建 src/bsp/prt_vector.S 文件

定义异常向量表，针对 4 组，每组 4 类异常共 16 类异常均定义有其对应的入口，且其入口均定义为 EXC_HANDLE vecId handler 的形式。

在 prt_reset_vector.S 中的 OsEnterMain: 标号后加入代码，**设置 EL1 级别的异常向量表。**获取异常向量表的地址，并将其加载到寄存器 x0 中，并将 x0 寄存器中的值写入到 VBAR_EL1 寄存器中。VBAR_EL1 用于存储 EL1（异常级别 1）异常向量表的基地址。

```
DAIF_MASK = 0x1C0       // disable SError Abort, IRQ, FIQ

    .global  OsVectorTable
    .global  OsEnterMain

    .section .text.startup, "ax"
OsEnterMain:
    BL      main

    MOV     x2, DAIF_MASK // bits [9:6] disable SError Abort, IRQ, FIQ
    MSR     DAIF, x2 // 把通用寄存器 x2 的值写入系统寄存器 DAIF 中

EXITLOOP:
    B EXITLOOP
```

在 prt_reset_vector.S 中的 OsEnterMain: 标号后加入代码

```
OsVectTblInit: // 设置 EL1 级别的异常向量表
    LDR x0, =OsVectorTable
    MSR VBAR_EL1, X0
```

定义宏 EXC_HANDLE，把这部分代码放到 src/bsp/prt_vector.S 文件的开头，它的主要作用是一发生异常就立即保存 CPU 寄存器的值，然后跳转到异常处理函数进行异常处理。

实现异常处理函数，包括 OsExcDispatch 和 OsExcDispatchFromLowEl。

OsExcDispatch 首先读取 4 个异常处理寄存器，然后将其保存到栈中，调用实际的异常处理 OsExcHandleEntry 函数。执行完 OsExcHandleEntry 函数后，依序恢复寄存器的值。这就是操作系统课程中重点讲述的上下文的保存和恢复过程。

OsExcDispatchFromLowEl 与 OsExcDispatch 的操作除调用的实际异常处理函数不同外其它完全一致。

1.4.2 新建 src/bsp/prt_exc.c 文件

实现实际的 OsExcHandleEntry 和 OsExcHandleFromLowElEntry 异常处理函数。

在这里的异常处理仅输出了一行字符串。

1.4.3 新建 src/bsp/os_exc_armv8.h 文件

上面两个异常处理函数的第 2 个参数是 struct ExcRegInfo* 类型，而在 src/bsp/prt_vector.S 中我们为该参数传递的是栈指针 sp（如下图）。所以该结构需与异常处理寄存器保存的顺序保持一致。在文件中，定义了 ExcRegInfo 结构。

```
13      mov     x0, x1  // x0： 异常类型
12      mov     x1, sp  // x1: 栈指针
11      bl      OsExcHandleEntry  // 跳转到实际的 C 处理函数， x0, x1分别为该函数的第1，2个参数。
```

1.5 触发异常与系统调用

注释掉 FPU 启用代码，构建系统并执行发现没有任何信息输出。

CPU 启动后进入的是 EL1 或以上级别，修改 main 函数，修改后代码的主要目的是从 EL1 切换到 EL0 级别，并通过 SVC 执行一个系统调用。

SPSR_EL1 寄存器中设置目标异常级别，ELR_EL1 用于保存异常返回时将要执行的指

令的地址。执行异常返回，返回到之前由 ELR_EL1 寄存器指定的地址，并且按照 SPSR_EL1 寄存器中的设置切换到目标异常级别。执行 SVC（Supervisor Call）异常，异常号是 0。

```
// 回到异常 EL 0级别，模拟系统调用，查看异常的处理，了解系统调用实现机制。
// 《Bare-metal Boot Code for ARMv8-A Processors》
OS_EMBED_ASM(
    "MOV    X1, #0b00000\n" // Determine the EL0 Execution state.
    "MSR    SPSR_EL1, X1\n"
    "ADR    x1, EL0Entry\n" // Points to the first instruction of EL0 code
    " MSR    ELR_EL1, X1\n"
    "eret\n"  // 返回到 EL 0 级别
    "EL0Entry: \n"
    "MOV x0, %0 \n" //参数1
    "MOV x8, #1\n" //在linux中,用x8传递 syscall number，保持一致。
    "SVC 0\n"    // 系统调用
    "B .\n" // 死循环，以上代码只用于演示，EL0级别的栈未正确设置
    ::"r"(&Test_SVC_str[0])
);
```

在 src/bsp/prt_exc.c 修改 OsExcHandleFromLowElEntry 函数实现系统调用，做了几个字符串输出。

1.6 构建项目与执行结果

本实验逻辑为：

prt_vector.S 文件定义了异常向量表和其所用宏，实现异常处理函数 OsExcDispatch 和 OsExcDispatchFromLowEl（函数外层，进行保存上下文，需调用具体的处理函数）。

prt_reset_vector.S 中的 OsEnterMain:标号后加入代码，设置 EL1 级别的异常向量表。

prt_exc.c 实现了具体的异常处理函数：输出字符串提示调用成功。系统调用实现时修改了 OsExcHandleFromLowElEntry 函数。

os_exc_armv8.h 定义异常处理函数调用中所需的 ExcRegInfo 结构。

执行结果：

```
guoruiling@guoruiling-virtual-machine:~/lab4$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab4/build
Scanning dependencies of target bsp
[ 12%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 25%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 37%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 50%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 62%] Building C object bsp/CMakeFiles/bsp.dir/prt_exc.c.obj
[ 75%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_vector.S.obj
[ 75%] Built target bsp
[ 87%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
```

```
guoruiling@guoruiling-virtual-machine:~/lab4$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
----------- ------------    ------    -----------
 ************ ************   ********   ************
----         ------------  ----------  ----    ---
************     ****     ****    **** *********
 ------------     ----     ------------ ---------
       *****     ****     ************ ****  ****
 ------------     ----     ----    ---- ----   ----
************     ****     ****    **** ****    ****
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

Catch a SVC call.
syscall number: 1, param 0: 0x40024cf0
Hello, my first system call!
```

2 实验任务

2.1 作业 1

题目：查找启用 FPU 前异常出现的位置和原因。禁用 FPU 后 PRT_Printf 工作不正常，需通过调试跟踪查看异常发生的位置和原因 ，查看 elr_el1 esr_el1 寄存器。

首先查询这两个寄存器的资料。

ELR_EL1（异常链接寄存器）：

该寄存器主要用于在发生系统调用、异常或中断时，存储当前程序的程序计数器（PC）值。这允许在系统恢复时能够返回到正确的执行位置。

ESR_EL1（异常状态寄存器）：

该寄存器主要用于在操作系统内核中处理异常情况。它主要在权限级别 EL1 下被访问，以查明具体触发异常的原因。

ESR_EL1 寄存器主要包含以下部分：

EC（异常分类）：描述异常触发的原因，例如 EC=0b010101 时，表示从 AArch64 状态的用户程序执行 SVC 指令而触发的异常，即系统调用。

IL（同步异常的指令长度）：表示异常指令的长度，通常用于判断是 32 位还是 64 位指令。ISS（具体指令特征）：提供关于触发异常的指令的更多详细信息。

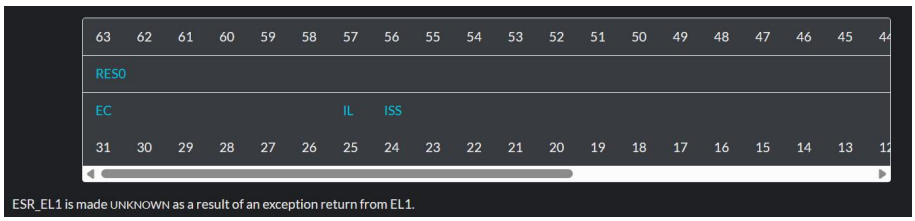
```
(gdb) si
22          BIC    sp, x1, #0xf
(gdb) si
Start () at /home/guoruiling/lab4/src/bsp/start.S:31
31          B      OsEnterMain
(gdb) si
OsEnterMain () at /home/guoruiling/lab4/src/bsp/prt_reset_vector.S:9
9           LDR x0, =OsVectorTable
(gdb) si
10          MSR VBAR_EL1, X0
(gdb) si
11          BL      main
(gdb) si
main () at /home/guoruiling/lab4/src/main.c:8
8           const char Test_SVC_str[] = "Hello, my first system call!";
(gdb) si
0x00000000400021e0      8           const char Test_SVC_str[] = "Hello, my first system call!";
(gdb) si
0x00000000400021e4      7       {
(gdb) si
0x00000000400021e8      8           const char Test_SVC_str[] = "Hello, my first system call!";
(gdb) si
OsVectorTable () at /home/guoruiling/lab4/src/bsp/prt_vector.S:136
136         EXC_HANDLE  4 OsExcDispatch
(gdb) si
0x0000000040004204 in OsVectorTable () at /home/guoruiling/lab4/src/bsp/prt_vector.S:136
136         EXC_HANDLE  4 OsExcDispatch
(gdb) i r ESR_EL1
ESR_EL1        0x1fe00000          534773760
(gdb)  i r ELR_EL1
ELR_EL1        0x400021e8          1073750504
(gdb) x/i 0x400023d4
   0x400023d4 <TryPrintf+80>:   add     x6, sp, #0x40
```

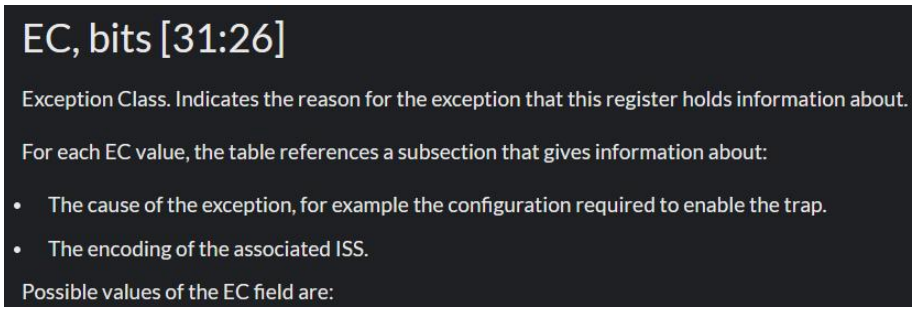
根据 GDB 调试，找到出错的具体位置，并查看两个寄存器的值，分别为

ESR_EL1　　0x1fe00000　　534773760

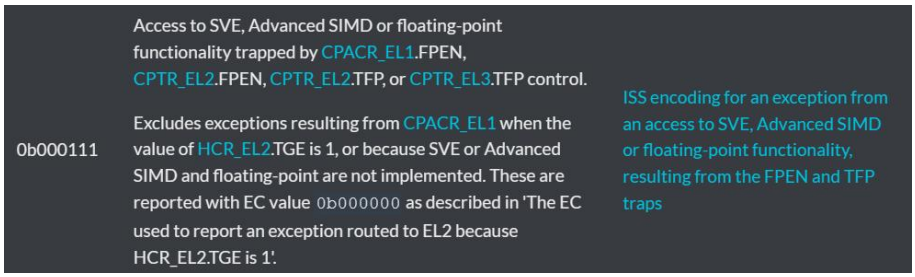
ELR_EL1　　0x400021e8　　1073750504

根据技术手册，我们可以知道：





ESR_EL1 中的异常触发原因（异常分类）为 31-26 位，对于 0x1fe00000，这个值为 000111，找到技术手册中对应的错误：



我们可以知道，是由于无法访问浮点功能出错。

ELR_EL1 中存储的是异常返回时的返回地址，根据这个地址，我们可以知道出错的位置，查看该地址的值，可以发现是 TryPrintf 函数中 add x6,sp,#0x40 这步出错。