# 一、实验名称

异常处理

# 二、实验目的

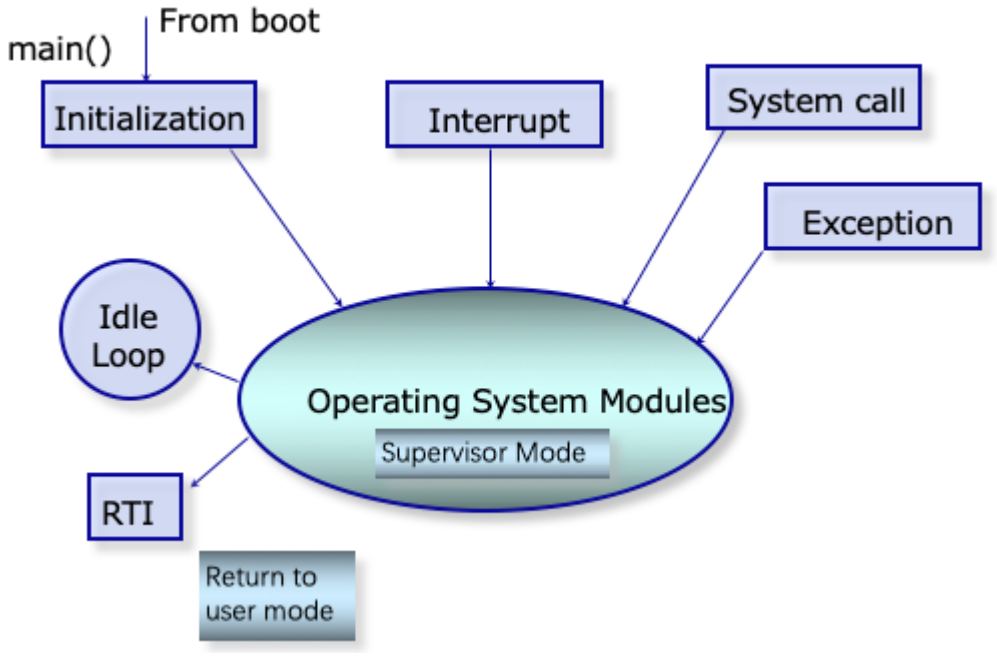
（1）深刻理解中断的原理和机制；
（2）掌握CPU访问中断控制器的方法；
（3）掌握Arm体系结构的中断机制和规范；
（4）实现时钟中断服务和部分异常处理等。

# 三、实验任务

## 1.异常向量表的建立

### （1）陷入操作系统

如下图所示，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。

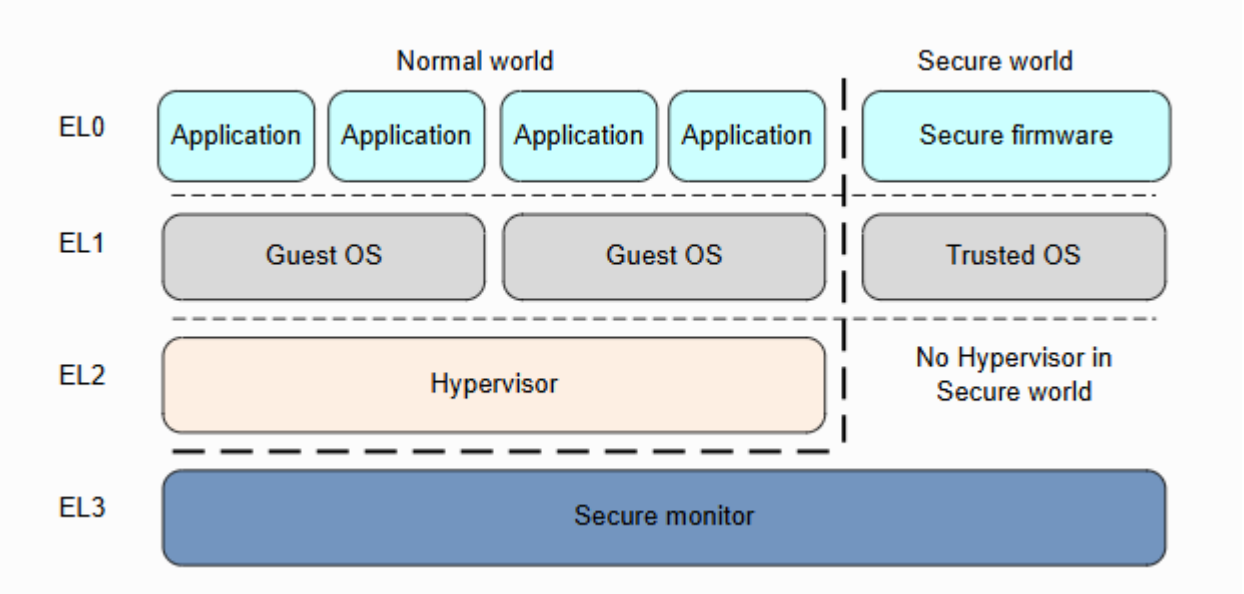



### （2）ARMv8的中断与异常处理

注意

访问Arm官网下载并阅读 ARM Cortex-A Series Programmer's Guide for ARMv8-A 和 AArch64 Exception and Interrupt Handling 等技术参考手册。

ARMv8 架构定义了两种执行状态(Execution States)，AArch64 和 AArch32。分别对应使用64位宽通用寄存器或32位宽通用寄存器的执行 1 。




上图所示为AArch64中的异常级别(Exception levels)的组织。可见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 2 ：

- 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。

> [异常级别说明]
>
> - **EL0**: 用户模式（User Mode），应用程序运行在此级别。
> - **EL1**: 内核模式（Kernel Mode），操作系统内核运行在此级别。
> - **EL2**: Hypervisor模式（Hypervisor Mode），虚拟化扩展时使用。
> - **EL3**: 安全监控模式（Secure Monitor Mode），处理安全相关任务。

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* 3 。

异常向量表的基地址由VBAR_ELn给出，然后每个表项都有一个从该基地址定义的偏移量。每个表有16个表项，每个表项的大小为128（0x80）字节（32 条指令）。 该表实际上由4组，每组4个表项组成。 分别是：

- 发生于当前异常级别的异常且SPSel寄存器选择SP0 4 ， Sync、IRQ、FIQ、SError对应的4个异常处理。
- 发生于当前异常级别的异常且SPSel寄存器选择SPx 4 ， Sync、IRQ、FIQ、SError对应的4个异常处理。
- 发生于较低异常级别的异常且执行状态为AArch64， Sync、IRQ、FIQ、SError对应的4个异常处理。
- 发生于较低异常级别的异常且执行状态为AArch32， Sync、IRQ、FIQ、SError对应的4个异常处理。

> [SPSel寄存器]
>
> - 控制栈指针的选择：

```
- **SP0**：用户栈指针（SP_EL0），用于EL0（用户态）。
- **SPx**：内核栈指针（SP_ELx，如SP_EL1），用于EL1及以上（内核态）。
```

## （3）新建异常向量表

新建 src/bsp/prt_vector.S 文件，参照这里 3 定义异常向量表如下：

```
.section .os.vector.text, "ax"      # 定义一个名为 ".os.vector.text" 的内存段
                                    # 属性 "ax":
                                    #   "a" = 可分配（Allocatable）
                                    #   "x" = 可执行（Executable）
.global  OsVectorTable              # 声明 OsVectorTable 为全局符号（可供链接器使
用）
.type  OsVectorTable,function       # 定义 OsVectorTable 的类型为函数（符合ARM架
构要求）
.align 13                           # 对齐到 2^13 = 8192 字节边界
                                    # ARMv8 要求异常向量表必须对齐到 8KB 边界
OsVectorTable:                      # 异常向量表的起始地址（标签）
.set    VBAR, OsVectorTable         # 定义宏 VBAR，值为 OsVectorTable 的地址
                                    # VBAR 是 ARM 的 Vector Base Address
Register
                                    # CPU 会从 VBAR 加载异常向量表的基地址
.org VBAR                           # 将后续代码的起始地址对齐到 VBAR 的地址
                                    # 即从 OsVectorTable 的起始位置开始填充
#异常向量表条目
#当前EL（Exception Level）使用SP_EL0（用户模式）
.org (VBAR + 0x00)                  # 偏移 0x0000：同步异常（Synchronous）
                                    # 当前 EL（如 EL1）使用 SP_EL0（用户栈）
                                    # 例如：EL1 下的非法指令、数据中止等
    EXC_HANDLE 0 OsExcDispatch      # 异常编号 0，跳转到 OsExcDispatch 处理
```

```
.org (VBAR + 0x80)                 # 偏移 0x080: IRQ/vIRQ（中断请求）
                                   # 当前 EL 使用 SP_EL0
    EXC_HANDLE 1 OsExcDispatch     # 异常编号 1，跳转到 OsExcDispatch

.org (VBAR + 0x100)                # 偏移 0x100: FIQ/vFIQ（快速中断）
    EXC_HANDLE 2 OsExcDispatch     # 异常编号 2

.org (VBAR + 0x180)                # 偏移 0x180: SERROR（系统错误）
                                   # 例如：异步外部abort（如内存总线错误）
    EXC_HANDLE 3 OsExcDispatch     # 异常编号 3
#当前EL使用 SP_ELx（内核模式，如 EL1/EL2）
.org (VBAR + 0x200)                # 偏移 0x200: 同步异常（SP_ELx）
                                   # 当前 EL 使用 SP_ELx（内核栈）
    EXC_HANDLE 4 OsExcDispatchFromLowEl  # 异常编号 4
                                   # 用于从低 EL（如 EL0）切换到高 EL 时的异常处理

.org (VBAR + 0x280)                # 偏移 0x280: IRQ/vIRQ（SP_ELx）
    EXC_HANDLE 5 OsExcDispatch     # 异常编号 5

.org (VBAR + 0x300)                # 偏移 0x300: FIQ/vFIQ（SP_ELx）
    EXC_HANDLE 6 OsExcDispatch     # 异常编号 6

.org (VBAR + 0x380)                # 偏移 0x380: SERROR（SP_ELx）
    EXC_HANDLE 7 OsExcDispatch     # 异常编号 7

#EL切换（目标EL使用AArch64）
.org (VBAR + 0x400)                # 偏移 0x400: 同步异常（目标 EL 为 AArch64）
                                   # 例如：从 EL0 触发异常后切换到 EL1
    EXC_HANDLE 8 OsExcDispatchFromLowEl  # 异常编号 8

.org (VBAR + 0x480)                # 偏移 0x480: IRQ/vIRQ（目标 EL 为
AArch64）
    EXC_HANDLE 9 OsExcDispatch     # 异常编号 9

.org (VBAR + 0x500)                # 偏移 0x500: FIQ/vFIQ（目标 EL 为
AArch64）
    EXC_HANDLE 10 OsExcDispatch    # 异常编号 10

.org (VBAR + 0x580)                # 偏移 0x580: SERROR（目标 EL 为 AArch64）
    EXC_HANDLE 11 OsExcDispatch    # 异常编号 11

#EL切换（目标EL使用AArch32）
```

```
.org (VBAR + 0x600)                 # 偏移 0x600: 同步异常（目标 EL 为 AArch32）
                                   # 例如：从 EL1 触发异常后切换到 EL0（32 位模
式）
    EXC_HANDLE 12 OsExcDispatch    # 异常编号 12

.org (VBAR + 0x680)                 # 偏移 0x680: IRQ/vIRQ（目标 EL 为
AArch32）
    EXC_HANDLE 13 OsExcDispatch    # 异常编号 13

.org (VBAR + 0x700)                 # 偏移 0x700: FIQ/vFIQ（目标 EL 为
AArch32）
    EXC_HANDLE 14 OsExcDispatch    # 异常编号 14

.org (VBAR + 0x780)                 # 偏移 0x780: SERROR（目标 EL 为 AArch32）
    EXC_HANDLE 15 OsExcDispatch    # 异常编号 15
```

可以看到：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。

## 补充说明：

1. **EXC_HANDLE 宏**
   - 宏展开后通常包含跳转指令（如 `B` 或 `LDR`），将控制权转移到具体的异常处理函数（如 `OsExcDispatch`）。
   - 参数 `0-15` 是异常编号，用于区分不同异常类型。
2. **SP_EL0 vs SP_ELx**
   - **SP_EL0**：用户模式栈指针（EL0 用户态使用）。
   - **SP_ELx**：高异常级别栈指针（如 EL1/EL2 内核态使用）。
3. **AArch64 vs AArch32**
   - **AArch64**：ARMv8 的 64 位执行状态。
   - **AArch32**：ARMv7 兼容的 32 位执行状态。

提示:CPSR 寄存器中有当前栈的选择 bits[0] 0:SP_EL0,1:SP_ELX

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

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

# 2.上下文保存与恢复

## （1）EXC_HANDLE宏

EXC_HANDLE 实际上是一个宏，其定义如下。

```
.global OsExcHandleEntry
.type   OsExcHandleEntry, function

.macro SAVE_EXC_REGS  // 保存通用寄存器的值到栈中
    stp    x1, x0, [sp,#-16]!
    stp    x3, x2, [sp,#-16]!
    stp    x5, x4, [sp,#-16]!
    stp    x7, x6, [sp,#-16]!
    stp    x9, x8, [sp,#-16]!
    stp    x11, x10, [sp,#-16]!
    stp    x13, x12, [sp,#-16]!
    stp    x15, x14, [sp,#-16]!
    stp    x17, x16, [sp,#-16]!
    stp    x19, x18, [sp,#-16]!
    stp    x21, x20, [sp,#-16]!
    stp    x23, x22, [sp,#-16]!
    stp    x25, x24, [sp,#-16]!
    stp    x27, x26, [sp,#-16]!
    stp    x29, x28, [sp,#-16]!
    stp    xzr, x30, [sp,#-16]!
.endm

.macro RESTORE_EXC_REGS  // 从栈中恢复通用寄存器的值
    ldp    xzr, x30, [sp],#16
    ldp    x29, x28, [sp],#16
    ldp    x27, x26, [sp],#16
    ldp    x25, x24, [sp],#16
    ldp    x23, x22, [sp],#16
    ldp    x21, x20, [sp],#16
    ldp    x19, x18, [sp],#16
    ldp    x17, x16, [sp],#16
    ldp    x15, x14, [sp],#16
    ldp    x13, x12, [sp],#16
    ldp    x11, x10, [sp],#16
    ldp    x9, x8, [sp],#16
    ldp    x7, x6, [sp],#16
    ldp    x5, x4, [sp],#16
    ldp    x3, x2, [sp],#16
    ldp    x1, x0, [sp],#16
.endm
```

```
.macro EXC_HANDLE vecId handler
    SAVE_EXC_REGS // 保存寄存器宏

    mov x1, #\vecId // x1 记录异常类型
    b   \handler // 跳转到异常处理
.endm
```

提示:注意把这部分代码放到 src/bsp/prt_vector.S 文件的开头
EXC_HANDLE 宏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。

## （2）在 src/bsp/prt_vector.S 文件中实现异常处理函数

随后，我们继续在 src/bsp/prt_vector.S 文件中实现异常处理函数，包括 OsExcDispatch 和 OsExcDispatchFromLowEl。

```
    .global OsExcHandleEntry
     .type   OsExcHandleEntry, function

     .global OsExcHandleEntryFromLowEl
     .type   OsExcHandleEntryFromLowEl, function



     .section .os.init.text, "ax"
     .globl OsExcDispatch
     .type OsExcDispatch, @function
     .align 4
OsExcDispatch:
     mrs    x5, esr_el1
     mrs    x4, far_el1
     mrs    x3, spsr_el1
     mrs    x2, elr_el1
     stp    x4, x5, [sp,#-16]!
     stp    x2, x3, [sp,#-16]!

     mov    x0, x1  // x0:  异常类型
     mov    x1, sp  // x1: 栈指针
     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。

     ldp    x2, x3, [sp],#16
     add    sp, sp, #16        // 跳过far, esr, HCR_EL2.TRVM==1的时候，EL1不能
```

```
写far, esr
    msr    spsr_el1, x3
    msr    elr_el1, x2
    dsb    sy
    isb

    RESTORE_EXC_REGS // 恢复上下文

    eret //从异常返回


    .globl OsExcDispatchFromLowEl
    .type OsExcDispatchFromLowEl, @function
    .align 4
OsExcDispatchFromLowEl:
    mrs    x5, esr_el1
    mrs    x4, far_el1
    mrs    x3, spsr_el1
    mrs    x2, elr_el1
    stp    x4, x5, [sp,#-16]!
    stp    x2, x3, [sp,#-16]!

    mov    x0, x1
    mov    x1, sp
    bl     OsExcHandleFromLowElEntry

    ldp    x2, x3, [sp],#16
    add    sp, sp, #16        // 跳过far, esr, HCR_EL2.TRVM==1的时候，EL1不能
写far, esr
    msr    spsr_el1, x3
    msr    elr_el1, x2
    dsb    sy
    isb

    RESTORE_EXC_REGS // 恢复上下文

    eret //从异常返回
```

[补充说明]

> ESR_EL1：记录异常类型（如中断、系统错误）和错误信息。
> FAR_EL1：记录触发异常的内存地址（如数据访问中止）。
> SPSR_EL1：保存异常发生时的CPU状态（如条件标志、中断屏蔽位）。

> ELR_EL1：保存异常返回地址（下一条指令地址）。
> stp：将寄存器对保存到栈中，!表示栈指针自动调整。

**OsExcDispatch:首先保存了4个系统寄存器到栈中，然后调用实际的异常处理 OsExcHandleEntry 函数。当执行完 OsExcHandleEntry 函数后，我们需要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。

## （3）异常处理函数

新建 src/bsp/prt_exc.c 文件，实现实际的 OsExcHandleEntry 和 OsExcHandleFromLowElEntry 异常处理函数。

```
#include "prt_typedef.h"

#include "os_exc_armv8.h"

extern U32 PRT_Printf(const char *format, ...);

// ExcRegInfo 格式与 OsExcDispatch 中寄存器存储顺序对应

void OsExcHandleEntry(U32 excType, struct ExcRegInfo *excRegs)

{

    PRT_Printf("Catch a exception.\n");

}

// ExcRegInfo 格式与 OsExcDispatchFromLowEl 中寄存器存储顺序对应

void OsExcHandleFromLowElEntry(U32 excType, struct ExcRegInfo *excRegs)

{

    PRT_Printf("Catch a exception from low exception level.\n");

}
```

注意到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是栈指针 sp。所以该结构需与异常处理寄存器保存的顺序保持一致。

新建 src/bsp/os_exc_armv8.h 文件，定义 ExcRegInfo 结构。

```
// 防止头文件重复包含（标准头文件保护宏）
#ifndef ARMV8_EXC_H
#define ARMV8_EXC_H

// 包含自定义类型定义（假设定义了 uintptr_t 等类型）
#include "prt_typedef.h"

// 定义 ARMv8 通用寄存器数量（X0 ~ X30 共 31 个）
#define XREGS_NUM       31

// 异常上下文寄存器信息结构体
// 注：内存布局必须与 TskContext 结构体完全一致（用于上下文切换）
struct ExcRegInfo {
    uintptr_t elr;        // 异常链接寄存器（Exception Link Register）
    // - 存储异常返回地址（PC 值）
    // - 对应硬件寄存器 ELR_EL1/ELR_EL3

    uintptr_t spsr;       // 保存的处理器状态寄存器（Saved Program Status
Register）
    // - 保存异常发生前的 CPSR 状态（条件标志、异常屏蔽位等）
    // - 对应硬件寄存器 SPSR_EL1/SPSR_EL3

    uintptr_t far;        // 故障地址寄存器（Fault Address Register）
    // - 记录导致数据异常的虚拟地址（如数据中止异常）
    // - 对应硬件寄存器 FAR_EL1/FAR_EL3

    uintptr_t esr;        // 异常综合寄存器（Exception Syndrome Register）
    // - 编码异常类型（EC 字段）和具体原因（ISS 字段）
    // - 例如：区分同步异常、IRQ、系统错误等

    uintptr_t xzr;        // 零寄存器（Zero Register，X30 的别名）
    // - 硬件强制实现为零的寄存器（读操作返回 0，写操作无效）
    // - 在上下文中可能用于临时存储或对齐

    uintptr_t xregs[XREGS_NUM]; // 通用寄存器数组（X0 ~ X29）
    // - 索引与寄存器编号对应关系：xregs[0] = X30, xregs[1] = X29, ...,
xregs[30] = X0
    // - 注意：存储顺序与硬件压栈顺序相反（需匹配 TskContext 布局）
};

#endif /* ARMV8_EXC_H */
```

提示：注意把上面的新增文件加入构建系统。

```
set(SRCS start.S prt_reset_vector.S print.c prt_vector.S
fscanf_s.c   memset_s.c     secureinput_a.c   sprintf_s.c
strtok_s.c    vsnprintf_s.c  wcscat_s.c    wmemmove_s.c
fwscanf_s.c  output.inl     secureinput_w.c   sscanf_s.c
swprintf_s.c  vsprintf_s.c   wcscpy_s.c    wscanf_s.c
gets_s.c     scanf_s.c      secureprintoutput.h    strcat_s.c
swscanf_s.c   vsscanf_s.c    wcsncat_s.c       input.inl
secinput.h     secureprintoutput_a.c  strcpy_s.c   vfscanf_s.c
vswprintf_s.c  wcsncpy_s.c   memcpy_s.c   securecutil.c
secureprintoutput_w.c  strncat_s.c  vfwscanf_s.c  vswscanf_s.c
wcstok_s.c    memmove_s.c  securecutil.h  snprintf_s.c
strncpy_s.c  vscanf_s.c    vwscanf_s.c    wmemcpy_s.c os_exc_armv8.h
prt_exc.c )

add_library(bsp OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成
库
```

### （4）触发异常

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，通过调试将会观察到异常。

```
(base) xiaoye@长乐:~/OSlab/lab4$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
```

## 3.系统调用

### （1）在 main 函数中调用一条系统调用

提示：下面请启用 FPU。
系统调用是通用操作系统为应用程序提供服务的方式，理解系统调用对理解通用操作系统的实现非常重要。下面我们来实现1条简单的系统调用。
EL 0 是用户程序所在的级别，而在lab1中我们已经知道CPU启动后进入的是EL1或以上级别。
在 main 函数中我们首先返回到 EL0 级别，然后通过 SVC 调用一条系统调用。

```
// 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- Processors》

OS_EMBED_ASM(
    "MOV   X1, #0b00000\n" // 设置SPSR_EL1寄存器为EL0执行状态（AArch64模式）
    // SPSR_EL1用于保存异常返回时的处理器状态（参考[6](@ref)）
    // 0b00000表示EL0（用户模式）+ 无异常掩码 + 无调试异常（参考[4](@ref)）
```

```
    "MSR   SPSR_EL1, X1\n" // 将X1的值写入SPSR_EL1寄存器
    // 通过MSR指令设置异常返回状态寄存器，准备返回EL0（参考[6](@ref)）

    "ADR   x1, EL0Entry\n" // 将EL0Entry标签地址加载到x1寄存器
    // ADR指令计算相对地址，用于定位EL0代码入口（参考[10](@ref)）

    "MSR   ELR_EL1, X1\n"  // 设置ELR_EL1寄存器为EL0Entry地址
    // ELR_EL1保存异常返回时的程序计数器值（参考[6](@ref)）

    "eret\n"               // 异常返回指令
    // 从ELR_EL1恢复PC，从SPSR_EL1恢复状态，跳转到EL0执行（参考[6](@ref)）

    "EL0Entry: \n"         // EL0代码入口标签

    "MOV x0, %0 \n"        // 将参数字符串地址存入x0寄存器（第一个参数）
    // Linux系统调用约定：x0-x5传递参数（参考[3](@ref)）

    "MOV x8, #1\n"         // 将系统调用号1（sys_write）存入x8寄存器
    // ARM64系统调用号通过x8传递（参考[3](@ref)）

    "SVC 0\n"              // 触发软中断（Supervisor Call）
    // 通过SVC指令进入EL1异常处理流程（参考[4](@ref)）

    "B .\n"                // 死循环（防止异常处理后继续执行）
    // 实际应用中需配置EL0栈指针（参考[1](@ref)的注释）

    ::"r"(&Test_SVC_str[0]) // 内联汇编参数传递
);

// 在 EL1 级别上模拟系统调用
// OS_EMBED_ASM("SVC 0"); // 直接触发SVC指令（未演示）
return 0;
```

备注

OS_EMBED_ASM 在 prt_typedef.h 中定义为 **asm volatile**，用于 C 与 ASM 混合编程。

SVC 是 arm 中的系统调用指令，相当于 x86 中的 int 指令。

ARM64系统调用号通过x8传递。

备注

汇编语法可以参考 GNU ARM Assembler Quick Reference 5 和 Arm Architecture Reference Manual Armv8 (Chapter C3 A64 Instruction Set Overview) 6

**内联汇编中Clobbers的用途到底是什么？** 7

1.Clobbers 是一个以逗号分隔的寄存器列表（该列表中还可以存放一些特殊值，用于表示一些特殊用途）。
2. 它的目的是为了告知编译器，Clobbers 列表中的寄存器会被该asm语句中的汇编代码隐性修改。
3. 由于 Clobbers 里的寄存器会被asm语句中的汇编代码隐性修改，编译器在为 input operands 和 output operands 挑选寄存器时，就不会使用 Clobbers 里指定的寄存器，这样就避免了发生数据覆盖等逻辑错误。
4. 通俗来讲，Clobbers 的用途就是为了告诉编译器，我这里指定的这些寄存器在该asm语句的汇编代码中用了，你在编译这条asm语句时，如果需要用到寄存器，别用我这里指定的这些，否则就都乱了。
5. Clobbers 里的特殊值可以为 cc，用于表示该平台的 flags 寄存器会被隐性修改（比如 x86 平台的 eflags 寄存器）。
6. Clobbers 里的特殊值也可以为 memory，用于表示某些内存数据会被隐性使用或隐性修改，所以在执行这条asm语句之前，编译器会保证所有相关的、涉及到内存的寄存器里的内容会被刷到内存中，然后再执行这条asm语句。在执行完这条asm语句之后，这些寄存器的值会再被重新load回来，然后再执行这条asm语句后面的逻辑。这样就保证了所有操作用到的数据都是最新的，是正确的。

## （2）系统调用实现

在 src/bsp/prt_exc.c 修改 OsExcHandleFromLowElEntry 函数实现 1 条系统调用。

```
#include "prt_typedef.h"

#include "os_exc_armv8.h"

extern void TryPutc(unsigned char ch);

void MyFirstSyscall(char *str)

{

    while (*str != '\0') {

        TryPutc(*str);

        str++;

    }

}
```

```
extern U32 PRT_Printf(const char *format, ...);

// ExcRegInfo 格式与 OsExcDispatch 中寄存器存储顺序对应

void OsExcHandleEntry(U32 excType, struct ExcRegInfo *excRegs)

{

    PRT_Printf("Catch a exception.\n");

}

// ExcRegInfo 格式与 OsExcDispatchFromLowEl 中寄存器存储顺序对应

void OsExcHandleFromLowElEntry(U32 excType, struct ExcRegInfo *excRegs)

{

    int ExcClass = (excRegs->esr&0xfc000000)>>26;

    if (ExcClass == 0x15){ //SVC instruction execution in AArch64 state.

        PRT_Printf("Catch a SVC call.\n");

        // syscall number存在x8寄存器中, x0为参数1

        int syscall_num = excRegs->xregs[(XREGS_NUM - 1)- 8]; //uniproton存储的顺序x0在高，x30在低

        uintptr_t param0 = excRegs->xregs[(XREGS_NUM - 1)- 0];

        PRT_Printf("syscall number: %d, param 0: 0x%x\n", syscall_num, param0);



        switch(syscall_num){

            case 1:

                MyFirstSyscall((void *)param0);

                break;
```

```
            default:

                PRT_Printf("Unimplemented syscall.\n");

        }

    }else{

        PRT_Printf("Catch a exception.\n");
        }

}
```

```
(base) xiaoye@长乐:~/OSlab/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: 0x4004b0c0
Hello, my first system call!
```

# 四、lab4 作业

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 elr_el1 esr_el1 寄存器。

禁用FPU后无输出；启用之后输出正常。

## （1）调试

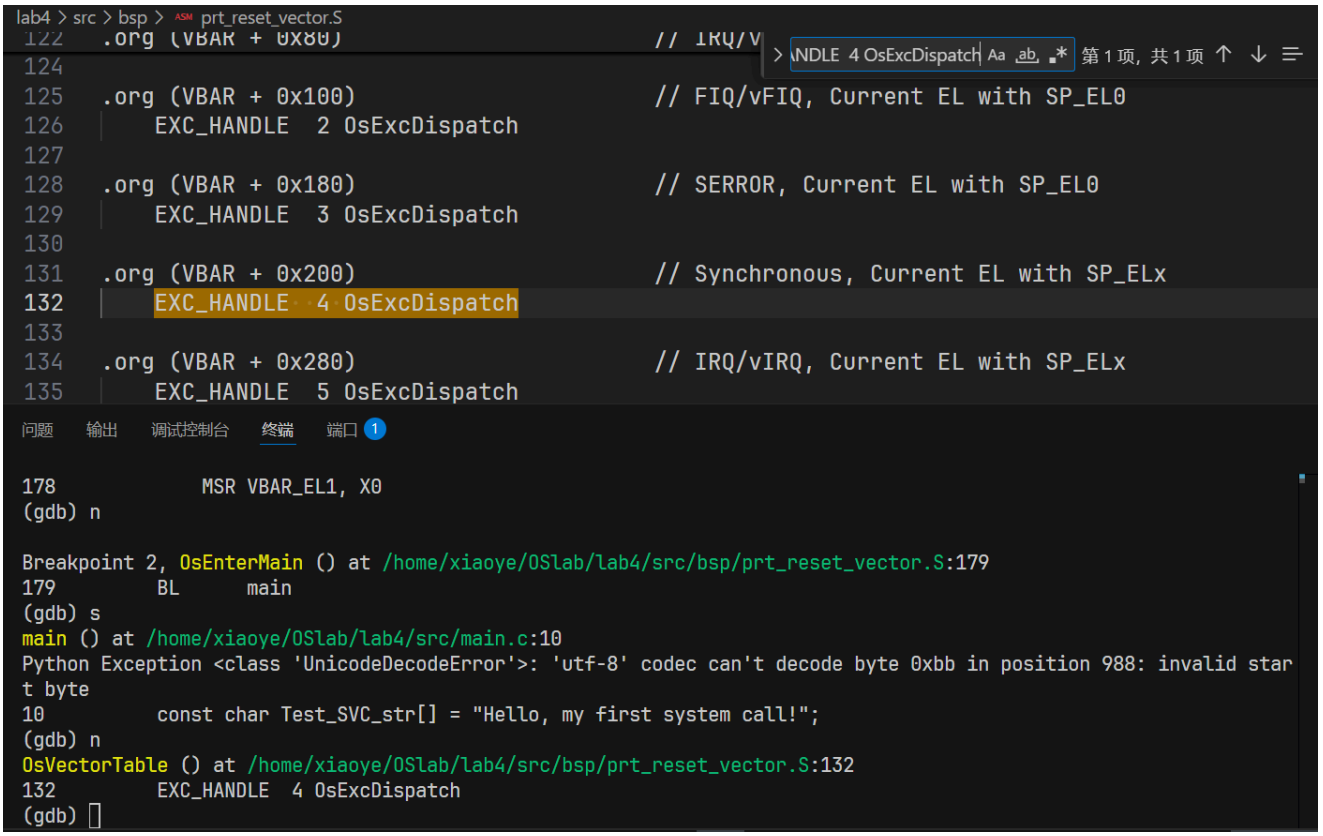
现在禁用FPU，开始调试。

```
__text_start () at /home/xiaoye/OSlab/lab4/src/bsp/start.S:11
11          MRS     x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
(gdb) break OsEnterMain
Breakpoint 1 at 0x40000030: file /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S, line 177.
(gdb) n
12          MOV     x2, #0x4 // CurrentEL EL1:  bits [3:2] = 0b01
(gdb) n
13          CMP     w6, w2
(gdb) n
15          BEQ Start // 若 CurrentEl 为 EL1 级别，跳转到 Start 处执行，否则死循环。
(gdb) n
21          LDR     x1, =__os_sys_sp_end // ld文件中定义，堆栈设置
(gdb) n
22          BIC     sp, x1, #0xf
(gdb) n
```

```
Breakpoint 1, OsEnterMain () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:177
177             LDR x0, =OsVectorTable
(gdb) disassemble
Dump of assembler code for function OsEnterMain:
=> 0x0000000040000030 <+0>:     ldr     x0, 0x40000048 <EXITLOOP+4>
   0x0000000040000034 <+4>:     msr     vbar_el1, x0
   0x0000000040000038 <+8>:     bl      0x40002bd4 <main>
   0x000000004000003c <+12>:    mov     x2, #0x1c0                      // #448
   0x0000000040000040 <+16>:    msr     daif, x2
End of assembler dump.
(gdb) break *0x0000000040000038
Breakpoint 2 at 0x40000038: file /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S, line 179.
(gdb) n
178             MSR VBAR_EL1, X0
(gdb) n
```

```
Breakpoint 2, OsEnterMain () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:179
179         BL      main
(gdb) s
main () at /home/xiaoye/OSlab/lab4/src/main.c:10
Python Exception <class 'UnicodeDecodeError'>: 'utf-8' codec can't decode byte 0xbb in position 988: invalid star
t byte
10          const char Test_SVC_str[] = "Hello, my first system call!";
(gdb) n
OsVectorTable () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:132
132         EXC_HANDLE  4 OsExcDispatch
```

查找对应的代码




我们发现它属于同步异常，当前异常级别与 SP_ELx。

异常编号4，跳转到OsExcDispatch处理

```
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:61
61          mrs     x5, esr_el1
(gdb) break OsExcHandleEntry
Breakpoint 3 at 0x4000d1e0: file /home/xiaoye/OSlab/lab4/src/bsp/prt_exc.c, line 16.
(gdb) n
62          mrs     x4, far_el1
(gdb) n
63          mrs     x3, spsr_el1
(gdb) n
64          mrs     x2, elr_el1
(gdb) n
65          stp     x4, x5, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:66
66          stp     x2, x3, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:68
68          mov     x0, x1  // x0:  异常类型
(gdb) n
69          mov     x1, sp  // x1: 栈指针
(gdb) n
70          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。
(gdb) s
```

跳转到实际的 C 处理函数

```
Breakpoint 3, OsExcHandleEntry (excType=4, excRegs=0x4004af90) at /home/xiaoye/OSlab/lab4/src/bsp/prt_exc.c:16
Python Exception <class 'UnicodeDecodeError'>: 'utf-8' codec can't decode byte 0xb8 in position 259: invalid star
t byte
16          PRT_Printf("Catch a exception.\n");
(gdb)
```

**查看elr_el1 esr_el1 寄存器**

```
(gdb) i r ELR_EL1
ELR_EL1        0x40002be0          1073753056
(gdb) i r ESR_EL1
ESR_EL1        0x1fe00000          534773760
```

**紧接着查看相应的地址内容，并通过反汇编查看对应main函数的指令**

```
(gdb) x/a 0x40002be0
0x40002be0 <main+12>:   0x910043f33dc00000
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000040002bd4 <+0>:     adrp    x0, 0x40020000 <OsVectorTable>
   0x0000000040002bd8 <+4>:     add     x0, x0, #0xa4b
   0x0000000040002bdc <+8>:     stp     x19, x30, [sp, #-48]!
   0x0000000040002be0 <+12>:    ldr     q0, [x0]
```

**综上所述，异常发生的位置是0x40002be0**

## （2）寄存器分析

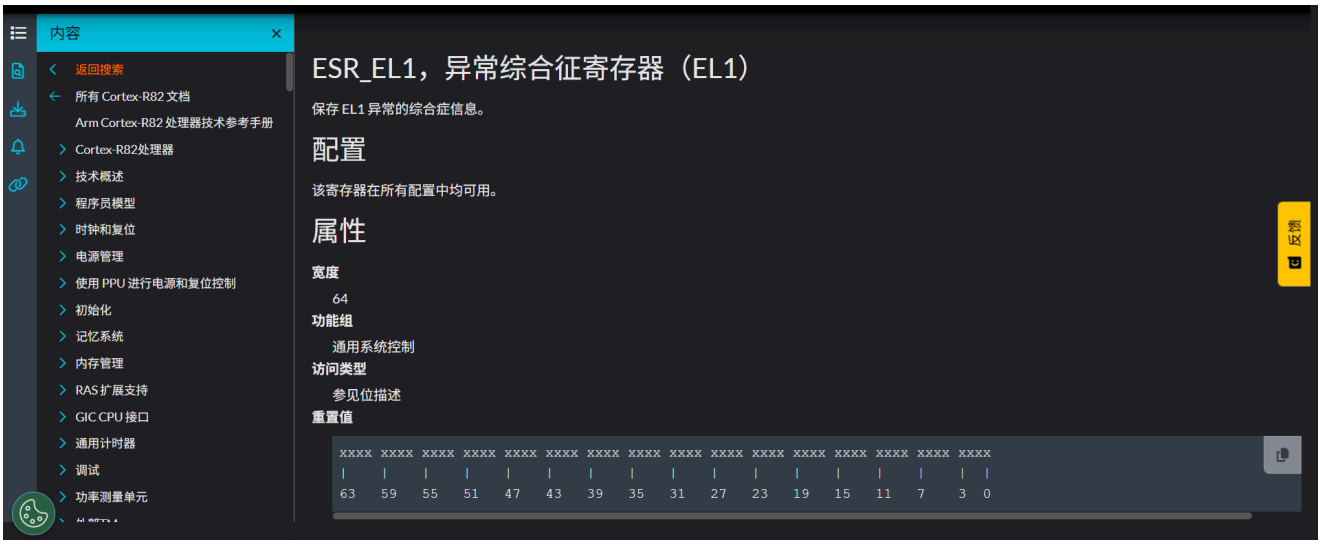
## ELR_EL1寄存器

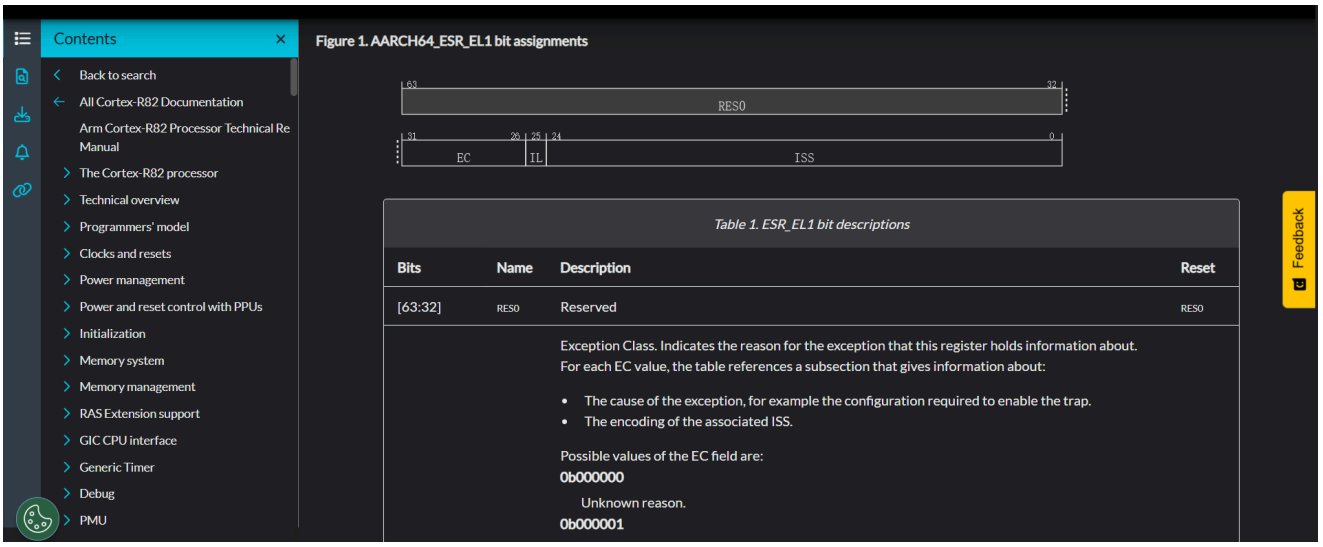

结合调试结果，分析可知，异常发生时正在执行的指令是ldr q0, [x0]

### 代码分析

ldr:LoadRegister的缩写，表示加载数据到寄存器。
q0:目标寄存器，这里是SIMD寄存器中的第一个寄存器，用于存储双精度浮点数。
[x]:内存地址，存储数据的位置。x0是一个通用寄存器，存储了要访问的内存地址。
该指令的含义是从内存地址x0中加载一个双精度浮点数（128位）到q0寄存器中。

## ESR_EL1寄存器



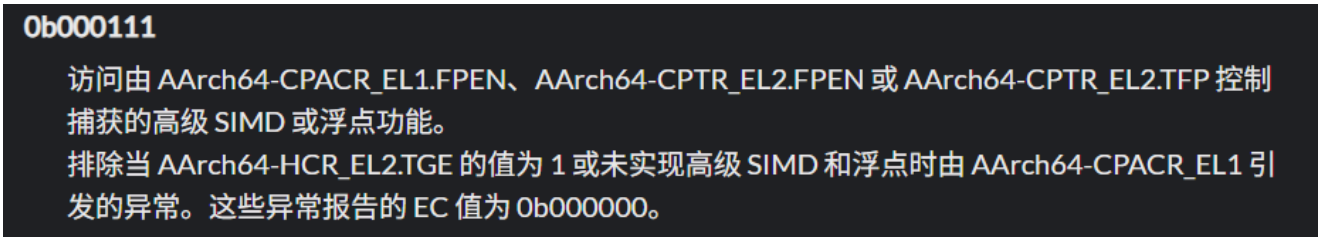



异常综合寄存器（ESR_ELn）用于向异常处理程序提供有关异常原因的信息。其结构解释如下：

- **位 [31:26]**：这些位表示异常类别，允许处理程序区分各种可能的异常原因。这包括未分配的指令、特权操作引发的异常、浮点操作异常、监管模式调用（SVC）、虚拟化调用（HVC）、安全监控器调用（SMC）、数据异常以及对齐异常。
- **位 [25]**：此位表示捕获指令的长度。对于16位指令，该位设置为0；对于32位指令，该位设置为1。此位还针对某些异常类别进行设置。
- **位 [24:0]**：该字段形成指令特定综合（ISS）字段，包含特定于该异常类型的信息。例如，当执行系统调用指令（SVC、HVC或SMC）时，该字段包含与操作码相关联的即时值。例如，如果操作码是SVC 0×123456，则ISS字段可能包含值0x123456。

ESR_ELn 寄存器仅在同步异常和SError时更新。对于IRQ或FIQ，该寄存器不会更新，因为这些中断处理程序通常从通用中断控制器（GIC）的寄存器获取状态信息。

结合调试结果，分析可知，EC（异常类别）= 000111。查看文档，发现它是因访问受控的FPU或高级SIMD功能触发的异常。




**综上所述，异常原因是main函数尝试对一个双精度浮点数（128位）进行加载的操作，但是由于FPU被禁用，因此产生了异常。该异常是同步异常。**

## （3）为什么没有输出

```
(gdb) n
PRT_Printf (format=0x400211dd "Catch a exception.\n") at /home/xiaoye/OSlab/lab4/src/bsp/print.c:127
127         va_start(vaList, format);
(gdb) n
128         count = TryPrintf(format, vaList);
(gdb) n
OsVectorTable () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:132
132         EXC_HANDLE  4 OsExcDispatch
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:61
61          mrs     x5, esr_el1
(gdb) n
62          mrs     x4, far_el1
(gdb) n
63          mrs     x3, spsr_el1
(gdb) n
64          mrs     x2, elr_el1
(gdb) n
65          stp     x4, x5, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:66
66          stp     x2, x3, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:68
68          mov     x0, x1  // x0:  异常类型
(gdb) n
69          mov     x1, sp  // x1: 栈指针
(gdb) n
70          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。
```

```
Breakpoint 3, OsExcHandleEntry (excType=4, excRegs=0x4004ad60) at /home/xiaoye/OSlab/lab4/src/bsp/prt_exc.c:16
16          PRT_Printf("Catch a exception.\n");
(gdb) n
PRT_Printf (format=0x400211dd "Catch a exception.\n") at /home/xiaoye/OSlab/lab4/src/bsp/print.c:127
127         va_start(vaList, format);
(gdb) n
128         count = TryPrintf(format, vaList);
(gdb) n
OsVectorTable () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:132
132         EXC_HANDLE  4 OsExcDispatch
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:61
61          mrs     x5, esr_el1
(gdb) n
62          mrs     x4, far_el1
(gdb) n
63          mrs     x3, spsr_el1
(gdb) n
64          mrs     x2, elr_el1
(gdb) n
65          stp     x4, x5, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:66
66          stp     x2, x3, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:68
68          mov     x0, x1  // x0:  异常类型
(gdb) n
69          mov     x1, sp  // x1: 栈指针
```

```
70          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。
(gdb) n

Breakpoint 3, OsExcHandleEntry (excType=4, excRegs=0x4004ab30) at /home/xiaoye/OSlab/lab4/src/bsp/prt_exc.c:16
16          PRT_Printf("Catch a exception.\n");
(gdb) n
PRT_Printf (format=0x400211dd "Catch a exception.\n") at /home/xiaoye/OSlab/lab4/src/bsp/print.c:127
127         va_start(vaList, format);
(gdb) n
128         count = TryPrintf(format, vaList);
(gdb) n
OsVectorTable () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:132
132         EXC_HANDLE  4 OsExcDispatch
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:61
61          mrs    x5, esr_el1
(gdb) n
62          mrs    x4, far_el1
(gdb) n
63          mrs    x3, spsr_el1
(gdb) n
64          mrs    x2, elr_el1
(gdb) n
65          stp    x4, x5, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:66
66          stp    x2, x3, [sp,#-16]!
(gdb) n
OsExcDispatch () at /home/xiaoye/OSlab/lab4/src/bsp/prt_reset_vector.S:68
```

继续调试，发现在print.c中执行完第128行之后，就会产生异常。os调用异常向量表，异常处理函数，进入print.c后执行到第128行后，再次产生异常。如此循环往复，永远不会有输出。注意到，该异常和上面分析的同步异常是同一类型。查看ELR_EL1。

```
(gdb) i r ELR_EL1
ELR_EL1        0x40002df8          1073753592
(gdb) x/a 0x40002df8
0x40002df8 <PRT_Printf+44>:     0x3d801be13d8017e0
(gdb) disassemble PRT_Printf
Dump of assembler code for function PRT_Printf:
   0x0000000040002dcc <+0>:     sub     sp, sp, #0x110
   0x0000000040002dd0 <+4>:     stp     x1, x2, [sp, #216]
   0x0000000040002dd4 <+8>:     add     x1, sp, #0x110
   0x0000000040002dd8 <+12>:    stp     x1, x1, [sp, #48]
   0x0000000040002ddc <+16>:    add     x1, sp, #0xd0
   0x0000000040002de0 <+20>:    str     x1, [sp, #64]
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000040002de4 <+24>:    mov     w1, #0xffffffc8                 // #-56
   0x0000000040002de8 <+28>:    str     w1, [sp, #72]
   0x0000000040002dec <+32>:    mov     w1, #0xffffff80                 // #-128
   0x0000000040002df0 <+36>:    str     w1, [sp, #76]
   0x0000000040002df4 <+40>:    add     x1, sp, #0x10
   0x0000000040002df8 <+44>:    str     q0, [sp, #80]
   0x0000000040002dfc <+48>:    str     q1, [sp, #96]
--Type <RET> for more, q to quit, c to continue without paging--
```

引起异常的指令是str q0, [sp, #80]。

> [] 代码分析
> str:Store Register的缩写，表示将寄存器的值存储到内存中。
> q0:目标寄存器，这里是SIMD寄存器中的第一个寄存器，用于存储双精度浮点数。
> sp:基址寄存器。
> #80:表示立即数偏移，即基址加上 80 字节后的内存地址为目标地址。

原因就是它使用了q0这个和浮点数有关的寄存器，而我们禁用了FPU。先是main函数ldr指令引起异常，后是每次调用PRT_Printf时候，str指令都会引起异常。一直循环，没有输出。

好啦。到这里，我们就分析结束啦。
总的来说，还是对os内部异常的处理有了更加系统全面真实的感受和理解。