0. 前言

实验概述

运行实验代码，了解OSTD的异常分发机制，上手编写简单的系统调用

实验内容

1. 拉取实验所需环境
2. 学习异常和中断概念
3. 查看简单的异常处理函数
4. 学习系统调用概念
5. 查看自定义的系统调用处理函数
6. 了解OSTD异常分发机制的实现
7. 上手实践，添加新的系统调用类型。

1. 实验环境及代码

拉取实验所需环境以及代码：

```
podman pull glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0
mkdir os-lab4
podman run -it -v ./os-lab4:/root/os-lab glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0

git clone -b lab4 https://github.com/sdww0/sustech-os-lab.git
cd sustech-os-lab
```

运行实验代码:

```
riscv64-unknown-linux-gnu-gcc -static -nostdlib ./user/hello.S -o user_prog
cargo osdk run --scheme riscv --target-arch=riscv64
```

Code Reference: https://asterinas.github.io/book/ostd/a-100-line-kernel.html

2. Exceptions and Interrupts

不同架构上对于异常 (Exception) 和中断 (Interrupt) 的定义可能不一致，来自RISCV手册对于两种概念的解释：

1. We use the term **exception** to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.
2. We use the term **interrupt** to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control.

本节课主要集中在OSTD对于异常分发机制的使用与讲解。

> **OSTD中的中断分发**

> 在OSTD中，中断会通过单独的中断分发链，与异常走的不是同一个分发流程。有关OSTD的外部中断分发链，感兴趣的同学可以查看`ostd/src/trap/handler.rs`文件，在这里不会进行展开。

3. OSTD的异常分发机制

OSTD封装了异常，并采用分发机制供上层使用。在`crate_user_task`中可以看到有以下代码：

```
loop {
    // The execute method returns when system
    // calls or CPU exceptions occur or some
    // events specified by the kernel occur.
    let return_reason = user_mode.execute(|| false);

    // The CPU registers of the user space
    // can be accessed and manipulated via
    // the `UserContext` abstraction.
    let user_context = user_mode.context_mut();
    match return_reason {
        ReturnReason::UserSyscall => {
            handle_syscall(user_context, current.user_space().unwrap())
        }
        ReturnReason::UserException => {
            handle_exception(user_context, current.user_space().unwrap())
        }
        ReturnReason::KernelEvent => {}
    }
}
```

在这里可以看到`return_reason`中包含三种不同的情况，分别为`UserException`，`UserSyscall`与`KernelEvent`。解释分别如下：

1. `UserException`代表除系统调用之外的异常，如加载地址错误。需要注意的是，在RISCV中系统调用也归属于异常的一部分，但因为它是操作系统的重要一环，因此OSTD在这里进行了区分。
2. `UserSyscall`代表系统调用。
3. `KernelEvent`是自定义的一种返回原因，在这里不展开分析。

3.1 UserException

实验代码下的`user/load_page_fault.S`汇编程序会通过加载地址`0xdeadbeef`上存放的内容，展示CPU异常中的加载地址错误。该地址是一个我们随便编写的非法地址，CPU在加载时无法找到该地址所存放的内容，随后触发异常，进入到`handle_exception`。我们可以通过以下步骤运行代码并观察结果：

1. 编译汇编程序：`riscv64-unknown-linux-gnu-gcc -static -nostdlib ./user/load_page_fault.S -o user_prog`
2. 运行内核：`cargo osdk run --scheme riscv --target-arch=riscv64`

> **交叉编译**：在当前编译平台下（如x86），编译出的程序能运行在体系架构不同的另一个平台上（如RISCV或ARM），但编译平台却不能运行该程序。

> 在编译汇编程序时，我们没有使用`gcc`来进行编译，而是用了RISCV提供的交叉编译工具`riscv64-unknown-linux-gnu-gcc`来进行编译，这个原因是我们需要编译出不同平台的汇编程序，目前主流的pc机仍是以x86架构为主的CPU。该交叉编译工具的下载和编译需要耗费较多时间，因此我们直接在镜像中准备好了该工具。

3.2 UserSyscall（系统调用）

系统调用是操作系统为用户态运行的进程提供的一系列服务接口，这些接口可以允许用户态程序与硬件设备进行交互。该交互形式与正常的函数调用类似，只不过会新增一个用户态与内核态切换的开销。在代码中可以看到有`handle_syscall`函数，里面定义了三种系统调用，其中`SYS_WRITE`与`SYS_EXIT`属于POSIX标准，`SYS_DUMMY_CALL`则是自定义的系统调用，其拥有以下功能：

1. 打印传入的第一个系统调用参数
2. 返回第二个参数值给用户态

```rust
fn handle_syscall(user_context: &mut UserContext, user_space: &UserSpace) {
    const SYS_DUMMY_CALL: usize = 1;
    const SYS_WRITE: usize = 64;
    const SYS_EXIT: usize = 93;

    match user_context.a7() {
        SYS_WRITE => {
            ....
        }
        SYS_DUMMY_CALL => {
            let value = user_context.a0();
            println!("Value from userland program: 0x{:x}", value);
            user_context.set_a0(user_context.a1());
        }
        SYS_EXIT => exit_qemu(QemuExitCode::Success),
        _ => unimplemented!(),
    }
}
```

不同的架构中对于系统调用的寄存器用途会有不同的规范，在RISC-V中，我们一般使用a0-a5作为系统调用的传递参数，a7作为系统调用号信息，a0作为系统调用返回值。让我们来分析一下系统调用拥有的功能，第一个是获取需要的系统调用寄存器数据，即打印出a0的内容，第二个为返回第二个参数给用户态，这一步则需要将a1寄存器的内容放入a0中。

运行本节代码：

1. 编译汇编程序：`riscv64-unknown-linux-gnu-gcc -static -nostdlib ./user/dummy.S -o user_prog`
2. 运行内核：`cargo osdk run --scheme riscv --target-arch=riscv64`

> **POSIX标准**
>
> POSIX：可移植操作系统接口（Portable Operating System Interface of UNIX），提供了一套供操作系统遵循的准则来简化跨平台软件开发的任务。类Unix系统如Linux会采用该标准以复用庞大的用户态应用程序生态。
>
> POSIX准则会定义一系列的系统调用接口，比如在该项目中的SYS_WRITE与SYS_EXIT便是该接口规范中的对应系统调用号。

3.3 了解分发机制的实现

在之前的内容中，我们从一个使用者的角度简单学习了如何处理异常。这种简单来源于OSTD内部对于中断的封装使得开发者无需关注汇编和硬件的配置。我们仍要了解操作系统从陷入内核态的时刻到我们的处理函数具体做了什么。

在之前的CPU异常和系统调用处理中，一个关键函数是`user_mode.execute()`，该函数可以总结成两个功能：1. 以指定的CPU寄存器信息进入到用户态；2. 当用户态触发中断或异常时，更新用户态CPU寄存器信息，返回原因。

3.3.1 进入用户态

首先来看该函数是如何进入到用户态，通过追溯`execute`函数调用链，我们可以找到以下进入调用链：

1. `UserMode::execute`
2. `UserContext::execute`
3. `self.user_context.run()`
4. `run_user` in assembly

在最底层的`run_user`汇编函数中，我们可以找到一段由`STORE_SP`，以及`LOAD_SP`组合成的指令，这些指令实际上是OSTD定义的宏，里面会保存或加载指定的CPU寄存器状态，从函数整体来看，`run_user`做了以下事情：1. 保存内核态CPU寄存器；2. 加载用户态CPU寄存器；3. 使用`sret`跳转到用户态。

3.3.2 用户态返回

用户态返回的流程依赖于架构中的特殊寄存器信息，这个/这些寄存器信息决定了用户态触发中断或异常后，CPU该跳转到哪条指令，或者切换到哪个栈。在RISCV中，会使用`sscratch`以及`stvec`来该操作，除此之外，RISCV还定义了一系列寄存器，帮助操作系统处理中断或异常如`scause`，`sepc`与`stval`。

1. `sscratch`：用户态和内核态切换上下文的一个中间寄存器，在一般的程序中我们需要用一个`temp`的变量来交换两个变量的值，`sscratch`便是起到了这个作用
2. `stvec`：存放了用户态陷入到内核态时需要跳转的地址，在OSTD中这个地址会设置为`trap_entry`。
3. `scause`：提供当前中断或异常的类型信息
4. `sepc`：提供中断或异常发生时的指令地址
5. `stval`：提供中断或异常发生的附加信息

在了解上述知识后，我们来看OSTD在异常发生时的处理流程：

1. 在`trap_entry`中，会根据异常发生之前的特权级，将用户态和内核态的处理流程分离开，我们主要关注用户态的处理函数。
2. 用户态处理函数会做三件事情：(1) 保存用户态寄存器信息到指定位置；(2) 恢复进入用户态时保存的内核态CPU寄存器；(3) 使用`ret`返回到`run_user`。
3. 从`run_user`返回到`UserContext::execute`，在这里会根据`scause`判断异常类型，进入不同的流程。

上述讲解只是给一个异常的大致分发流程，里面还会使用`sscratch`来辅助分发流程，有关实现细节可以参考三个文件：(1) riscv/trap.S; (2) riscv/cpu/mod.rs; (3) user.rs;

4. 上手练习

结合系统调用，将作业一中的`ReadOnlyFile`放到内核中，体验往操作系统中添加一个小功能的过程，注意需要进行以下改动：

1. `std::cell::RefCell` -> `ostd::sync::SpinLock`
2. 使用到`RefCell::borrow_mut`的地方，替换为`SpinLock::lock`方法
3. `HashSet` -> `BTreeSet`
4. 删除`latest_read_time`与涉及到的方法
5. 根据报错信息，导入需要的包

并实现以下功能：

1. 使用`spin::Once`新建一个静态文件：`static FILE: Once<ReadOnlyFile> = Once::new();`
2. 添加新的系统调用，名为SYS_CREATE，用用户态传入的字符串初始化该文件
3. 添加新的系统调用，名为SYS_PRINT，打印该文件存放的字符串，如果没初始化不打印
4. ... more?

> 一个操作系统需要非常多的系统调用与功能模块协调运行，以及需要应对各种极端参数并需要进行错误处理，大家可以尝试完善该功能，在面对极端参数时仍能正常处理