# 教程

## 程序运行状态

use riscv::register::{sstatus::Sstatus, scause::Scause};

为什么要这么定义?不能定义在结构体中吗

## 状态的保存与恢复

这里涉及到汇编指令

**教程中的解释：**

为了状态的保存与恢复，我们可以先用栈上的一小段空间来把需要保存的全部通用寄存器和 CSR 寄存器保存在栈上，保存完之后在跳转到 Rust 编写的中断处理函数；而对于恢复，则直接把备份在栈上的内容写回寄存器。

首先分析教程中的汇编代码

# 宏：将寄存器存到栈上

.macro SAVE reg, offset

sd \reg, \offset\*8(sp) #定义SAVE宏

.endm

# 宏：将寄存器从栈中取出

.macro LOAD reg, offset

ld \reg, \offset\*8(sp) #定义LOAD宏

.endm

.section .text

#.section指示把代码划分成若干个段（Section），程序被操作系统加载执行时，每个段被加载到不同的地址 .text段保存代码，是只读和可执行的

.globl \_\_interrupt #声明\_\_interrupt是全局可见的

# 进入中断

# 保存 Context 并且进入 Rust 中的中断处理函数#interrupt::handler::handle\_interrupt()

\_\_interrupt:

# 在栈上开辟 Context 所需的空间

addi sp, sp, -34\*8 #在栈上分配8\*34个空间 适配寄存器

# 保存通用寄存器，除了 x0（固定为 0）

SAVE x1, 1

# 将原来的 sp（sp 又名 x2）写入 2 位置

addi x1, sp, 34\*8 #x1指向栈顶 也就是sp

SAVE x1, 2 #将sp存到2

# 其他通用寄存器

SAVE x3, 3

SAVE x4, 4

SAVE x5, 5

SAVE x6, 6

SAVE x7, 7

SAVE x8, 8

SAVE x9, 9

SAVE x10, 10

SAVE x11, 11

SAVE x12, 12

SAVE x13, 13

SAVE x14, 14

SAVE x15, 15

SAVE x16, 16

SAVE x17, 17

SAVE x18, 18

SAVE x19, 19

SAVE x20, 20

SAVE x21, 21

SAVE x22, 22

SAVE x23, 23

SAVE x24, 24

SAVE x25, 25

SAVE x26, 26

SAVE x27, 27

SAVE x28, 28

SAVE x29, 29

SAVE x30, 30

SAVE x31, 31

# 取出 CSR 并保存

csrr s1, sstatus #将sstatus取出 赋给s1

csrr s2, sepc

SAVE s1, 32 #将sstatus存在32

SAVE s2, 33 #为什么只存到了33?

# 调用 handle\_interrupt，传入参数

# context: &mut Context

mv a0, sp

# scause: Scause

csrr a1, scause

# stval: usize

csrr a2, stval

jal handle\_interrupt #跳转到函数开始的位置

.globl \_\_restore

# 离开中断

# 从 Context 中恢复所有寄存器，并跳转至 Context 中 sepc 的位置

\_\_restore:

# 恢复 CSR

LOAD s1, 32

LOAD s2, 33

csrw sstatus, s1

csrw sepc, s2

# 恢复通用寄存器

LOAD x1, 1

LOAD x3, 3

LOAD x4, 4

LOAD x5, 5

LOAD x6, 6

LOAD x7, 7

LOAD x8, 8

LOAD x9, 9

LOAD x10, 10

LOAD x11, 11

LOAD x12, 12

LOAD x13, 13

LOAD x14, 14

LOAD x15, 15

LOAD x16, 16

LOAD x17, 17

LOAD x18, 18

LOAD x19, 19

LOAD x20, 20

LOAD x21, 21

LOAD x22, 22

LOAD x23, 23

LOAD x24, 24

LOAD x25, 25

LOAD x26, 26

LOAD x27, 27

LOAD x28, 28

LOAD x29, 29

LOAD x30, 30

LOAD x31, 31

# 恢复 sp（又名 x2）这里最后恢复是为了上面可以正常使用 LOAD 宏

LOAD x2, 2

sret #从内核态跳到用户态

issue上提出的方法

# Essential for substitution %i

.altmacro #启用备用宏模式

# length of general purpose registers in bytes

.set reg\_size, 8

# No. of registers inside a context frame

.set context\_size, 34

# Load register relative to sp

.macro load reg, offset #定义load宏

ld \reg, \offset \* reg\_size(sp)

.endm

.macro load\_gp n

load x\n, \n #定义load\_gp宏

.endm

...

.global \_\_restore #声明\_\_restore全局可见

\_\_restore:

# Restore csr registers

#这里没有定义初始空间

load t0, 32

load t1, 33

csrw sstatus, t0

csrw sepc, t1

.set i, 3

.rept 29

load\_gp %i

.set i, i + 1

.endr

# Restore ra and sp last

load x1, 1

load x2, 2

# Return to the address stored in sepc

sret

#少了些东西 没有进入函数 应该只是个实例 但是可以用于原来汇编的修改

在上述代码中

mv a0, sp #sp 复制到a0中

csrr a1, scause

csrr a2, stval #这两个csr的值写入a1 a2

首先 a0是x10 a1是x11 a2是x12 对应ecall里面的三个参数

他们两个不用保存起来吗?

看了issue后说 scause stval没有被修改 之后再细看

## 中断处理流程

#[no\_mangle]

pub fn handle\_interrupt(context: &mut Context, scause: Scause, stval: usize) {

panic!("Interrupted: {:?}", scause.cause());

}

汇编中的jal handle\_interrupt 当中断触发时跳转

stvec::write(\_\_interrupt as usize, stvec::TrapMode::Direct)

A direct mode mtvec means that all traps will go to the exact same function, 'm going to use the direct mode. Then, we can parse out the cause using Rust's match statement.

简化分析,用rust的match分析

## 时钟中断

panic!(

"Unresolved interrupt: {:?}\n{:x?}\nstval: {:x}",

scause.cause(),

context,

stval

);

这里有一个错误 context 没有用Debug 最新版本已经实现了

在main.rs中 我们需要增加一个loop函数 然后时钟中断可以打断它

直接去掉返回值会报错

# 实验题

1. **简述：在 rust\_main 函数中，执行 ebreak 命令后至函数结束前，sp 寄存器的值是怎样变化的？**

首先我们看汇编的文件

\_\_interrupt:

# 在栈上开辟 Context 所需的空间

addi sp, sp, -34\*8 #在栈上分配8\*34个空间 适配寄存器

# 保存通用寄存器，除了 x0（固定为 0）

SAVE x1, 1

# 将原来的 sp（sp 又名 x2）写入 2 位置

addi x1, sp, 34\*8 #x1指向栈顶 也就是sp

SAVE x1, 2 #将sp存到2

# 其他通用寄存器

SAVE x3, 3

…

SAVE x31, 31

# 取出 CSR 并保存

csrr s1, sstatus #将sstatus取出 赋给s1

csrr s2, sepc

SAVE s1, 32 #将sstatus存在32

SAVE s2, 33 #为什么只存到了33?

mv a0, sp

csrr a1, scause

csrr a2, stval

jal handle\_interrupt #跳转到函数开始的位置

将这个寄存器组保存在栈上，然后执行handle\_interrupt函数 接着跳转到 breakpoint函数 在最后\_\_restore sp会变为原来的值。

1. **如果去掉 rust\_main 后的 panic 会发生什么，为什么**

会发生报错，因为返回值类型是!

对比文档，发现理解出偏差了—— ——

1. **实验**

Trap::Exception(Exception::LoadFault)=>panic!("Exception::LoadFault")

添加这一句即可

if let 0=stval{

println!("Success");

}else{

panic!("Exception::LoadFault");

}

地址出错时的判断

第三题的解法是按照给的标准解法来解的，自己想的解法是错误的—— ——