Lab 1: RV64 内核引导与时钟中断处理

实验目的

- 学习 RISC-V 汇编， 编写 head.S 实现跳转到内核运行的第一个 C 函数。
- 学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出。
- 学习 Makefile 相关知识， 补充项目中的 Makefile 文件， 来完成对整个工程的管理。
- 学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 处理的初始化。
- 理解 CPU 上下文切换机制，并正确实现上下文切换功能。
- 编写 trap 处理函数，完成对特定 trap 的处理。
- 调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。

实验环境

- Environment in Lab0

实验基础知识介绍

RV64 内核引导

前置知识

为了顺利完成 OS 实验，我们需要一些前置知识和较多调试技巧。在 OS 实验中我们需要 **RISC-V汇编** 的前置知识，课堂上不会讲授，请同学们通过阅读以下四份文档自学：

- RISC-V Assembly Programmer's Manual
- RISC-V Unprivileged Spec
- RISC-V Privileged Spec
- RISC-V 手册（中文）

> 注：RISC-V 手册（中文）中有一些 Typo，请谨慎参考。

RISC-V 的三种特权模式

RISC-V 有三个特权模式：U (user) 模式、S (supervisor) 模式和 M (machine) 模式。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

其中：

- M 模式是对硬件操作的抽象，有**最高**级别的权限

- S 模式介于 M 模式和 U 模式之间，在操作系统中对应于内核态 (Kernel)。当用户需要内核资源时，向内核申请，并切换到内核态进行处理
- U 模式用于执行用户程序，在操作系统中对应于用户态，有**最低**级别的权限

3.1.3 从计算机上电到 OS 运行

我们以最基础的嵌入式系统为例，计算机上电后，首先硬件进行一些基础的初始化后，将 CPU 的 Program Counter 移动到内存中 Bootloader 的起始地址。
Bootloader 是操作系统内核运行之前，用于初始化硬件，加载操作系统内核。
在 RISC-V 架构里，Bootloader 运行在 M 模式下。Bootloader 运行完毕后就会把当前模式切换到 S 模式下，机器随后开始运行 Kernel。

这个过程简单而言就是这样：

```
   Hardware              RISC-V M Mode            RISC-V S Mode
+------------+         +--------------+         +----------+
|  Power On  |  ---->  |  Bootloader  |  ---->  |  Kernel  |
+------------+         +--------------+         +----------+
```

SBI 与 OpenSBI

SBI (Supervisor Binary Interface) 是 S-mode 的 Kernel 和 M-mode 执行环境之间的接口规范，而 OpenSBI 是一个 RISC-V SBI 规范的开源实现。RISC-V 平台和 SoC 供应商可以自主扩展 OpenSBI 实现，以适应特定的硬件配置。

简单的说，为了使操作系统内核适配不同硬件，OpenSBI 提出了一系列规范对 M-mode 下的硬件进行了统一定义，运行在 S-mode 下的内核可以按照这些规范对不同硬件进行操作。

为降低实验难度，我们选择 OpenSBI 作为 Bootloader 来完成机器启动时 M-mode 下的硬件初始化与寄存器设置，并使用 OpenSBI 所提供的接口完成诸如字符打印的操作。

在实验中，QEMU 已经内置了 OpenSBI 作为 Bootloader，我们可以使用 `-bios default` 启用。如果启用，QEMU 会将 OpenSBI 代码加载到 0x80000000 起始处。OpenSBI 初始化完成后，会跳转到 0x80200000 处（也就是 Kernel 的起始地址）。因此，我们所编译的代码需要放到 0x80200000 处。

如果你对 RISC-V 架构的 Boot 流程有更多的好奇，可以参考这份 bootflow。

Makefile

Makefile 可以简单的认为是一个工程文件的编译规则，描述了整个工程的编译和链接流程。在 Lab0 中我们已经使用了 make 工具利用 Makefile 文件来管理整个工程。在阅读了 Makefile介绍 这一章节后，同学们可以根据工程文件夹里 Makefile 的代码来掌握一些基本的使用技巧。

内联汇编

内联汇编（通常由 asm 或者 __asm__ 关键字引入）提供了将汇编语言源代码嵌入 C 程序的能力。
内联汇编的详细介绍请参考 Assembler Instructions with C Expression Operands 。
下面简要介绍一下这次实验会用到的一些内联汇编知识：

内联汇编基本格式为：

```
    __asm__ volatile (
        "instruction1\n"
        "instruction2\n"
        ......
        ......
        "instruction3\n"
        : [out1] "=r" (v1),[out2] "=r" (v2)
        : [in1] "r" (v1), [in2] "r" (v2)
        : "memory"
    );
```

其中，三个 `:` 将汇编部分分成了四部分：

- 第一部分是汇编指令，指令末尾需要添加 '\n'。
- 第二部分是输出操作数部分。
- 第三部分是输入操作数部分。
- 第四部分是可能影响的寄存器或存储器，用于告知编译器当前内联汇编语句可能会对某些寄存器或内存进行修改，使得编译器在优化时将其因素考虑进去。

这四部分中后三部分不是必须的。

示例一

```
unsigned long long s_example(unsigned long long type,unsigned long long arg0) {
    unsigned long long ret_val;
    __asm__ volatile (
        "mv x10, %[type]\n"
        "mv x11, %[arg0]\n"
        "mv %[ret_val], x12"
        : [ret_val] "=r" (ret_val)
        : [type] "r" (type), [arg0] "r" (arg0)
        : "memory"
    );
    return ret_val;
}
```

示例一中指令部分，`%[type]`、`%[arg0]` 以及 `%[ret_val]` 代表着特定的寄存器或是内存。

输入输出部分中，`[type] "r" (type)` 代表着将 `()` 中的变量 `type` 放入寄存器中（`"r"` 指放入寄存器，如果是 `"m"` 则为放入内存），并且绑定到 `[]` 中命名的符号中去。`[ret_val] "=r" (ret_val)` 代表着将汇编指令中 `%[ret_val]` 的值更新到变量 `ret_val` 中。

示例二

```
#define write_csr(reg, val) ({
    __asm__ volatile ("csrw " #reg ", %0" :: "r"(val)); })
```

示例二定义了一个宏，其中 `%0` 代表着输出输入部分的第一个符号，即 `val`。

`#reg` 是c语言的一个特殊宏定义语法，相当于将reg进行宏替换并用双引号包裹起来。

例如 `write_csr(sstatus,val)` 经宏展开会得到：

```
({
    __asm__ volatile ("csrw " "sstatus" ", %0" :: "r"(val)); })
```

编译相关知识介绍

vmlinux.lds

GNU ld 即链接器，用于将 `*.o` 文件（和库文件）链接成可执行文件。在操作系统开发中，为了指定程序的内存布局，ld 使用链接脚本（Linker Script）来控制，在 Linux Kernel 中链接脚本被命名为 vmlinux.lds。更多关于 ld 的介绍可以使用 `man ld` 命令。

下面给出一个 vmlinux.lds 的例子：

```
/* 目标架构 */
OUTPUT_ARCH( "riscv" )

/* 程序入口 */
ENTRY( _start )

/* kernel代码起始位置 */
BASE_ADDR = 0x80200000;

SECTIONS
{
    /* . 代表当前地址 */
    . = BASE_ADDR;

    /* 记录kernel代码的起始地址 */
    _skernel = .;

    /* ALIGN(0x1000) 表示4KB对齐 */
    /* _stext, _etext 分别记录了text段的起始与结束地址 */
    .text : ALIGN(0x1000){
        _stext = .;

        *(.text.entry)
        *(.text .text.*)

        _etext = .;
    }

    .rodata : ALIGN(0x1000){
        _srodata = .;

        *(.rodata .rodata.*)

        _erodata = .;
    }

    .data : ALIGN(0x1000){
        _sdata = .;

        *(.data .data.*)

        _edata = .;
```

```
    }

    .bss : ALIGN(0x1000){
        _sbss = .;

        *(.bss.stack)
        sbss = .;
        *(.bss .bss.*)

        _ebss = .;
    }

    /* 记录kernel代码的结束地址 */
    _ekernel = .;
}
```

首先我们使用 OUTPUT_ARCH 指定了架构为 RISC-V ，之后使用 ENTRY 指定程序入口点为 `_start` 函数，程序入口点即程序启动时运行的函数，经过这样的指定后在head.S中需要编写 `_start` 函数，程序才能正常运行。

链接脚本中有 `.` `*` 两个重要的符号。单独的 `.` 在链接脚本代表当前地址，它有赋值、被赋值、自增等操作。而 `*` 有两种用法，其一是 `*()` 在大括号中表示将所有文件中符合括号内要求的段放置在当前位置，其二是作为通配符。

链接脚本的主体是SECTIONS部分，在这里链接脚本的工作是将程序的各个段按顺序放在各个地址上，在例子中就是从0x80200000地址开始放置了 `.text` ， `.rodata` ， `.data` 和 `.bss` 段。各个段的作用可以简要概括成：

段名	主要作用
.text	通常存放程序执行代码
.rodata	通常存放常量等只读数据
.data	通常存放已初始化的全局变量、静态变量
.bss	通常存放未初始化的全局变量、静态变量

在链接脚本中可以自定义符号，例如以上所有 `_s` 与 `_e` 开头的符号都是我们自己定义的。

更多有关链接脚本语法可以参考这里。

vmlinux

vmlinux 通常指 Linux Kernel 编译出的可执行文件 (Executable and Linkable Format / ELF)，特点是未压缩的，带调试信息和符号表的。在整套 OS 实验中，vmlinux 通常指将你的代码进行编译，链接后生成的可供 QEMU 运行的 RV64 架构程序。如果对 vmlinux 使用 `file` 命令，你将看到如下信息：

```
$ file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically
linked, not stripped
```

System.map

System.map 是内核符号表（Kernel Symbol Table）文件，是存储了所有内核符号及其地址的一个列表。“符号”通常指的是函数名，全局变量名等等。使用 `nm vmlinux` 命令即可打印 vmlinux 的符号表，符号表的样例如下：

```
0000000000000800 A __vdso_rt_sigreturn
ffffffe000000000 T __init_begin
ffffffe000000000 T _sinittext
ffffffe000000000 T _start
ffffffe000000040 T _start_kernel
ffffffe000000076 t clear_bss
ffffffe000000080 t clear_bss_done
ffffffe0000000c0 t relocate
ffffffe00000017c t set_reset_devices
ffffffe000000190 t debug_kernel
```

使用 System.map 可以方便地读出函数或变量的地址，为 Debug 提供了方便。

RV64 时钟中断处理

如果完成了 **3.1** 中的 **RV64 内核引导**，我们能成功地将一个最简单的 OS 启动起来， 但还没有办法与之交互。我们在课程中讲过操作系统启动之后由**事件**（ **event** ）驱动，在本次实验的后半部分中，我们将引入一种重要的事件 **trap**，trap 给了 OS 与硬件、软件交互的能力。在 **3.1** 中我们介绍了在 RISC-V 中有三种特权级 (M 态、 S 态、 U 态)， 在 Boot 阶段， OpenSBI 已经帮我们将 M 态的 trap 处理进行了初始化，这一部分不需要我们再去实现，因此后续我们重点关注 S 态的 trap 处理。

RISC-V 中的 Interrupt 和 Exception

什么是 Interrupt 和 Exception

> 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. 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. We use the term **trap** to refer to **the transfer of control to a trap handler** caused by either an exception or an interrupt.

上述是 RISC-V Unprivileged Spec 1.6 节中对于 `Trap`、 `Interrupt` 与 `Exception` 的描述。总结起来 `Interrupt` 与 `Exception` 的主要区别如下表：

Interrupt	Exception
Hardware generate	Software generate
These are **asynchronous external requests** for service (like keyboard or printer needs service).	These are **synchronous internal requests** for service based upon abnormal events (think of illegal instructions, illegal address, overflow etc).
These are **normal events** and shouldn’t interfere with the normal running of a computer.	These are **abnormal events** and often result in the termination of a program

上文中的 `Trap` 描述的是一种控制转移的过程, 这个过程是由 `Interrupt` 或者 `Exception` 引起的。这里为了方便起见，我们在这里约定 `Trap` 为 `Interrput` 与 `Exception` 的总称。

相关寄存器

除了32个通用寄存器之外，RISC-V 架构还有大量的 **控制状态寄存器** `Control and Status Registers(CSRs)`，下面将介绍几个和 trap 机制相关的重要寄存器。

Supervisor Mode 下 trap 相关寄寄存器:

- `sstatus` (Supervisor Status Register)中存在一个 SIE (Supervisor Interrupt Enable) 比特位，当该比特位设置为 1 时，会**响应**所有的 S 态 trap， 否则将会禁用所有 S 态 trap。
- `sie` (Supervisor Interrupt Eable Register)。在 RISC-V 中，`Interrupt` 被划分为三类 `Software Interrupt`， `Timer Interrupt`， `External Interrupt`。在开启了 `sstatus[SIE]` 之后，系统会根据 `sie` 中的相关比特位来决定是否对该 `Interrupt` 进行**处理**。
- `stvec` (Supervisor Trap Vector Base Address Register) 即所谓的“中断向量表基址”。 `stvec` 有两种模式：`Direct 模式`，适用于系统中只有一个中断处理程序, 其指向中断处理入口函数 （ 本次实验中我们所用的模式 ） 。`Vectored 模式`，指向中断向量表， 适用于系统中有多个中断处理程序 （ 该模式可以参考 RISC-V 内核源码 ） 。
- `scause` (Supervisor Cause Register), 会记录 trap 发生的原因，还会记录该 trap 是 `Interrupt` 还是 `Exception`。
- `sepc` (Supervisor Exception Program Counter), 会记录触发 exception 的那条指令的地址。

Machine Mode 异常相关寄寄存器:

- 类似于 Supervisor Mode， Machine Mode 也有相对应的寄存器，但由于本实验同学不需要操作这些寄存器，故不在此作介绍。

以上寄存器的详细介绍请同学们参考 RISC-V Privileged Spec

相关特权指令

- `ecall` (Environment Call)，当我们在 S 态执行这条指令时，会触发一个 `ecall-from-s-mode-exception`，从而进入 M Mode 下的处理流程(如设置定时器等)；当我们在 U 态执行这条指令时，会触发一个 `ecall-from-u-mode-exception`，从而进入 S Mode 下的处理流程 (常用来进行系统调用)。
- `sret` 用于 S 态 trap 返回, 通过 `sepc` 来设置 `pc` 的值， 返回到之前程序继续运行。

以上指令的详细介绍请同学们参考 RISC-V Privileged Spec

上下文处理

由于在处理 trap 时，有可能会改变系统的状态。所以在真正处理 trap 之前，我们有必要对系统的当前状态进行保存，在处理完成之后，我们再将系统恢复至原先的状态，就可以确保之前的程序继续正常运行。
这里的系统状态通常是指寄存器，这些寄存器也叫做CPU的上下文 (`Context`).

trap 处理程序

trap 处理程序根据 `scause` 的值， 进入不同的处理逻辑，在本次试验中我们需要关心的只有 `Superviosr Timer Interrupt` 。

时钟中断

时钟中断需要 CPU 硬件的支持。CPU 以“时钟周期”为工作的基本时间单位，对逻辑门的时序电路进行同步。而时钟中断实际上就是“每隔若干个时钟周期执行一次的程序”。下面介绍与时钟中断相关的寄存器以及如何产生时钟中断。

- `mtime` 与 `mtimecmp` (Machine Timer Register)。 `mtime` 是一个实时计时器， 由硬件以恒定的频率自增。 `mtimecmp` 中保存着下一次时钟中断发生的时间点，当 `mtime` 的值大于或等于 `mtimecmp` 的值，系统就会触发一次时钟中断。因此我们只需要更新 `mtimecmp` 中的值，就可以设置下一次时钟中断的触发点。 `OpenSBI` 已经为我们提供了更新 `mtimecmp` 的接口 `sbi_set_timer` (见 `lab1` 4.4节)。
- `mcounteren` (Counter-Enable Registers)。由于 `mtime` 是属于 M 态的寄存器，我们在 S 态无法直接对其读写， 幸运的是 OpenSBI 在 M 态已经通过设置 `mcounteren` 寄存器的 `TM` 比特位， 让我们可以在 S 态中可以通过 `time` 这个**只读**寄存器读取到 `mtime` 的当前值，相关汇编指令是 `rdtime` 。

以上寄存器的详细介绍请同学们参考 RISC-V Privileged Spec

实验步骤

准备工程

从 repo 同步实验代码框架。为了减少大家的工作量，在这里我们提供了简化版的 `printk` 来输出格式化字符串。

```
├── arch
│   └── riscv
│       ├── include
│       │   ├── defs.h
│       │   └── sbi.h
│       ├── kernel
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sbi.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── printk.h
│   ├── stddef.h
│   └── types.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── printk.c
└── Makefile
```

完成 **RV64 内核引导**，需要完善以下文件：

- arch/riscv/kernel/head.S
- lib/Makefile

- arch/riscv/kernel/sbi.c
- arch/riscv/include/defs.h

完成 **RV64 时钟中断处理**，需要完善 / 添加以下文件：

- arch/riscv/kernel/head.S
- arch/riscv/kernel/entry.S
- arch/riscv/kernel/trap.c
- arch/riscv/kernel/clock.c

RV64 内核引导

完善 Makefile 脚本

```
C_SRC       = $(sort $(wildcard *.c))
OBJ         = $(patsubst %.c,%.o,$(C_SRC))
CROSS_=riscv64-linux-gnu-
GCC=${CROSS_}gcc

all: $(OBJ)

%.o: %.c
    ${GCC}  ${CFLAG} -c $<
x
clean:
    $(shell rm *.o 2>/dev/null)
```

补充 `sbi.c`

OpenSBI 在 M 态，为 S 态提供了多种接口，比如字符串输入输出。因此我们需要实现调用 OpenSBI 接口的功能。给出函数定义如下：

```
struct sbiret {
    long error;
    long value;
};

struct sbiret sbi_ecall(int ext, int fid,
                    uint64 arg0, uint64 arg1, uint64 arg2,
                    uint64 arg3, uint64 arg4, uint64 arg5);
```

sbi_ecall 函数中，需要完成以下内容：

1. 将 ext (Extension ID) 放入寄存器 a7 中，fid (Function ID) 放入寄存器 a6 中，将 arg0 ~ arg5 放入寄存器 a0 ~ a5 中。
2. 使用 `ecall` 指令。`ecall` 之后系统会进入 M 模式，之后 OpenSBI 会完成相关操作。
3. OpenSBI 的返回结果会存放在寄存器 a0 ， a1 中，其中 a0 为 error code， a1 为返回值， 我们用 sbiret 来接受这两个返回值。

同学们可以参照内联汇编的示例一完成该函数的编写。
编写成功后，调用 `sbi_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0)` 将会输出字符'0'。其中 `0x1` 代表 `sbi_console_putchar` 的 ExtensionID， `0x0` 代表FunctionID, 0x30代表'0'的ascii值，其余参数填0。

请在 `arch/riscv/kernel/sbi.c` 中补充 `sbi_ecall()`。

sbi_ecall() 代码如下：

```
#include "types.h"
#include "sbi.h"

struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
                        uint64 arg1, uint64 arg2,
                        uint64 arg3, uint64 arg4,
                        uint64 arg5)
{
    struct sbiret ret;
    //OpenSBI 的返回结果会存放在寄存器 a0，a1 中,其中a0为error code，a1为返回值
    //我们用 sbiret 来接受这两个返回值
    __asm__ volatile (
            "mv a7, %[ext]\n"
            "mv a6, %[fid]\n"
            "mv a5, %[arg5]\n"
            "mv a4, %[arg4]\n"
            "mv a3, %[arg3]\n"
            "mv a2, %[arg2]\n"
            "mv a1, %[arg1]\n"
            "mv a0, %[arg0]\n"
            //将 ext (Extension ID) 放入寄存器 a7 中，fid (Function ID) 放入寄存器 a6
中
            //将 arg0 ~ arg5 放入寄存器 a0 ~ a5 中
            "ecall\n"   //使用ecall指令 ecall之后系统会进入M模式
            "mv %[error], a0\n"
            "mv %[value], a1\n"
            : [error] "=r" (ret.error), [value] "=r" (ret.value)
    : [ext] "r" (ext), [fid] "r" (fid), [arg0] "r" (arg0),
    [arg1] "r" (arg1), [arg2] "r" (arg2), [arg3] "r" (arg3),
    [arg4] "r" (arg4), [arg5] "r" (arg5)
    : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
    );
    return ret;
}
```

修改 defs

```
#ifndef _DEFS_H
#define _DEFS_H

#include "types.h"

#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    asm volatile ("csrr %[v], " #csr    \
    : [v] "=r" (__v)::);     \
    __v;                                    \
```

```
})

#define csr_write(csr, val)                         \
({                                                  \
    uint64 __v = (uint64)(val);                     \
    asm volatile ("csrw " #csr ", %0"               \
                  : : "r" (__v)                     \
                  : "memory");                      \
})

#endif
```

RV64 时钟中断处理

`vmlinux.lds`

```
/* 目标架构 */
OUTPUT_ARCH( "riscv" )

/* 程序入口 */
ENTRY( _start )

/* kernel代码起始位置 */
BASE_ADDR = 0x80200000;

SECTIONS
{
    /* . 代表当前地址 */
    . = BASE_ADDR;

    /* 记录kernel代码的起始地址 */
    _skernel = .;

    /* ALIGN(0x1000) 表示4KB对齐 */
    /* _stext, _etext 分别记录了text段的起始与结束地址 */
    .text : ALIGN(0x1000){
        _stext = .;

        *(.text.init)
        *(.text.entry)
        *(.text .text.*)

        _etext = .;
    }

    .rodata : ALIGN(0x1000){
        _srodata = .;

        *(.rodata .rodata.*)

        _erodata = .;
    }
```

```
    .data : ALIGN(0x1000){
        _sdata = .;

        *(.data .data.*)

        _edata = .;
    }

    .bss : ALIGN(0x1000){
        _sbss = .;

        *(.bss.stack)
        *(.sbss .sbss.*)
        *(.bss .bss.*)

        _ebss = .;
    }

    /* 记录kernel代码的结束地址 */
    _ekernel = .;
}
```

开启 trap 处理

在运行 `start_kernel` 之前，我们要对上面提到的 CSR 进行初始化，初始化包括以下几个步骤：

1. 设置 `stvec`， 将 `_traps` (`_trap` 在 4.3 中实现) 所表示的地址写入 `stvec`，这里我们采用 `Direct 模式`，而 `_traps` 则是 trap 处理入口函数的基地址。
2. 开启时钟中断，将 `sie[STIE]` 置 1。
3. 设置第一次时钟中断，参考 `clock_set_next_event()` (`clock_set_next_event()` 在 4.3.4 中介绍) 中的逻辑用汇编实现。
4. 开启 S 态下的中断响应， 将 `sstatus[SIE]` 置 1。

```
.extern start_kernel

    .section .text.init
    .globl _start
_start:
    la sp, boot_stack_top # set the stack pointer

    # set stvec
    la t0, _traps
    csrw stvec, t0

    # set on STIE bit of sie
    csrr t0, sie
    ori t0, t0, 0x20
    csrw sie, t0

    # set on SIE bit of sstatus
    csrr t0, sstatus
    ori t0, t0, 0x2
```

```
    csrw sstatus, t0

    # set first time interrupt
    jal ra, clock_set_next_event

    jal x0, start_kernel  # jump to start_kernel

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 1028 * 4 # stack size 4KB

    .globl boot_stack_top
boot_stack_top:
```

实现上下文切换

我们要使用汇编实现上下文切换机制， 包含以下几个步骤：

1. 在 `arch/riscv/kernel/` 目录下添加 `entry.S` 文件。
2. 保存 CPU 的寄存器（上下文）到内存中（栈上）。
3. 将 `scause` 和 `sepc` 中的值传入 trap 处理函数 `trap_handler` (`trap_handler` 在 4.4 中介绍) ，我们将会在 `trap_handler` 中实现对 trap 的处理。
4. 在完成对 trap 的处理之后， 我们从内存中（栈上）恢复CPU的寄存器（上下文）。
5. 从 trap 中返回。

```
    .section .text.entry
    .align 2
    .globl _traps
_traps:
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -256
    sd x1, 0(sp)
    sd x2, 8(sp)
    sd x3, 16(sp)
    sd x4, 24(sp)
    sd x5, 32(sp)
    sd x6, 40(sp)
    sd x7, 48(sp)
    sd x8, 56(sp)
    sd x9, 64(sp)
    sd x10, 72(sp)
    sd x11, 80(sp)
    sd x12, 88(sp)
    sd x13, 96(sp)
    sd x14, 104(sp)
    sd x15, 112(sp)
    sd x16, 120(sp)
    sd x17, 128(sp)
    sd x18, 136(sp)
    sd x19, 144(sp)
    sd x20, 152(sp)
    sd x21, 160(sp)
```

```
    sd x22, 168(sp)
    sd x23, 176(sp)
    sd x24, 184(sp)
    sd x25, 192(sp)
    sd x26, 200(sp)
    sd x27, 208(sp)
    sd x28, 216(sp)
    sd x29, 224(sp)
    sd x30, 232(sp)
    sd x31, 240(sp)
    csrr t0, sepc
    sd t0, 248(sp)

    # 2. call trap_handler
    csrr a0, scause # check which are the argument registers
    csrr a1, sepc
    jal x1, trap_handler

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
    ld t0, 248(sp)
    csrw sepc, t0
    ld x1, 0(sp)
    ld x3, 16(sp)
    ld x4, 24(sp)
    ld x5, 32(sp)
    ld x6, 40(sp)
    ld x7, 48(sp)
    ld x8, 56(sp)
    ld x9, 64(sp)
    ld x10, 72(sp)
    ld x11, 80(sp)
    ld x12, 88(sp)
    ld x13, 96(sp)
    ld x14, 104(sp)
    ld x15, 112(sp)
    ld x16, 120(sp)
    ld x17, 128(sp)
    ld x18, 136(sp)
    ld x19, 144(sp)
    ld x20, 152(sp)
    ld x21, 160(sp)
    ld x22, 168(sp)
    ld x23, 176(sp)
    ld x24, 184(sp)
    ld x25, 192(sp)
    ld x26, 200(sp)
    ld x27, 208(sp)
    ld x28, 216(sp)
    ld x29, 224(sp)
    ld x30, 232(sp)
    ld x31, 240(sp)
    ld x2, 8(sp)
    addi sp, sp, 256

    # 4. return from trap
    sret
```

实现 trap 处理函数

1. 在 `arch/riscv/kernel/` 目录下添加 `trap.c` 文件。
2. 在 `trap.c` 中实现 trap 处理函数 `trap_handler()`, 其接收的两个参数分别是 `scause` 和 `sepc` 两个寄存器中的值。

```
#include "clock.h"
#include "printk.h"

void trap_handler(unsigned long scause, unsigned long sepc) {
    unsigned long interrupt_sig = 0x8000000000000000;
    if (scause & interrupt_sig){ // it's interrupt
        scause = scause - interrupt_sig;
        unsigned long timer_int = 0x5;
        if (!(scause ^ timer_int)){ // it's Supervisor timer interrupt
            printk("kernel is running!\n[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
        }
    }
}
```

实现时钟中断相关函数

1. 在 `arch/riscv/kernel/` 目录下添加 `clock.c` 文件。
2. 在 `clock.c` 中实现 get_cycles ( ) : 使用 `rdtime` 汇编指令获得当前 `time` 寄存器中的值。
3. 在 `clock.c` 中实现 clock_set_next_event ( ) : 调用 `sbi_ecall`，设置下一个时钟中断事件。

```
// clock.c

// QEMU中时钟的频率是10MHz, 也就是1秒钟相当于10000000个时钟周期。
unsigned long TIMECLOCK = 10000000;

unsigned long get_cycles() {
    // 编写内联汇编，使用 rdtime 获取 time 寄存器中 (也就是mtime 寄存器 )的值并返回
    // YOUR CODE HERE

}

void clock_set_next_event() {
    // 下一次 时钟中断 的时间点
    unsigned long next = get_cycles() + TIMECLOCK;

    // 使用 sbi_ecall 来完成对下一次时钟中断的设置
    // YOUR CODE HERE
}
```

编译及测试

```
Launch the qemu ......

OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2023 Hello RISC-V
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
```

思考题

1. **请总结一下 RISC-V 的 calling convention，并解释 Caller / Callee Saved Register 有什么区别?**

   RISC-V的调用约定（calling convention）是一组规则，用于定义如何在RISC-V体系结构中进行函数调用和参数传递。RISC-V的调用约定包括如何管理寄存器、栈以及参数传递。Caller / Callee Saved Register是调用约定的一部分，它们有助于管理寄存器的状态。

   RISC-V的调用约定主要包括以下内容：

   1. **参数传递：** 函数参数通常由一些固定的寄存器传递，如`a0`、`a1`、`a2`等，或者通过栈传递。
   2. **返回值：** 函数的返回值通常存储在`a0`寄存器中。
   3. **寄存器使用：** 调用者（Caller）和被调用者（Callee）各自有一组寄存器，用于保存临时数据。Caller Saved寄存器用于保存调用前的状态，而Callee Saved寄存器用于在函数内保存寄存器的状态，以确保调用函数不会破坏这些寄存器的内容。

4. **栈帧：** 被调用的函数通常会创建一个栈帧，用于保存局部变量、寄存器保存区域以及其他信息。Caller和Callee之间协调栈指针以保证栈的一致性。

Caller Saved Register（Caller保存寄存器）： 由调用者（Caller）负责保存和恢复，它们用于保存调用函数之前的寄存器状态；在函数调用之前，Caller Saved寄存器中的内容会被保存，以便在函数返回后能够还原。这是因为被调用函数可能会修改这些寄存器的内容。

Callee Saved Register（Callee保存寄存器）： 由被调用者（Callee）负责保存和恢复，它们用于保存被调用函数内的寄存器状态；被调用函数在函数入口保存Callee Saved寄存器，以确保不破坏Caller的寄存器内容，并在函数退出时还原。

区别：

Caller Saved寄存器由调用者负责保存，通常用于保存调用前的寄存器状态，以确保函数调用后能够恢复这些寄存器的内容。被调用函数不需要关心这些寄存器。

Callee Saved寄存器由被调用者负责保存，用于保存函数内的寄存器状态，以确保被调用函数执行期间不会破坏这些寄存器的内容，同时需要在函数返回时还原这些寄存器的状态。Caller不需要了解和操作这些寄存器。

2. **编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值（截图）。**

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/os23fall-stu/src/lab1$ cat System.map
0000000080200000 t $x
0000000080200034 t $x
0000000080200154 t $x
000000008020017c t $x
00000000802001e0 t $x
00000000802002c4 t $x
0000000080200340 t $x
0000000080200384 t $x
0000000080200394 t $x
00000000802003e4 t $x
00000000802008c0 t $x
0000000080200000 A BASE_ADDR
0000000080202000 D TIMECLOCK
0000000080202008 d _GLOBAL_OFFSET_TABLE_
0000000080204010 B _ebss
0000000080202008 D _edata
0000000080204010 B _ekernel
00000000802010a8 R _erodata
0000000080200940 T _etext
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080200034 T _traps
0000000080203000 B boot_stack
0000000080204010 B boot_stack_top
000000008020017c T clock_set_next_event
0000000080200154 T get_cycles
00000000802008c0 T printk
0000000080200394 T putc
00000000802001e0 T sbi_ecall
0000000080200340 T start_kernel
0000000080200384 T test
00000000802002c4 T trap_handler
00000000802003e4 t vprintfmt
```

3. **用 `csr_read` 宏读取 `sstatus` 寄存器的值，对照 RISC-V 手册解释其含义（截图）。**

补全宏定义

```
#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    asm volatile ("csrr %[v], " #csr        \
    : [v] "=r" (__v)::);                    \
    __v;                                    \
})
```

读取寄存器的值并输出

```
    uint64 sstatus;
    sstatus = csr_read(sstatus);
    printk("sstatus = %d\n", sstatus);
```

输出截图

```
Boot HART MEDELEG         : 0x000000000000b109
2023 Hello RISC-V
sstatus = 24578
```

sstatus=24578=OX0000 0000 0000 0000 0110 0000 0000 0010

31	30 20	19	18	17	16 15	14 13	12 9	8	7 6	5	4	3 2	1	0
SD	WPRI	MXR	SUM	WPRI	XS[1:0]	FS[1:0]	WPRI	SPP	WPRI	SPIE	UPIE	WPRI	SIE	UIE
1	11	1	1	1	2	2	4	1	2	1	1	2	1	1

对照手册，这个值的含义为 FS[1:0] = 11, SIE = 1

SIE = 1 表示允许Supervisor特权级别中的中断处理，即处理器可以响应中断请求

FS = 11 表示浮点运算被启用，但是处于 dirty 状态

4. **用 `csr_write` 宏向 `sscratch` 寄存器写入数据，并验证是否写入成功（截图）。**

```
    uint64 sscratch;
    sscratch = csr_read(sscratch);
    printk("sscratch_0 = %d\n",sscratch);

    uint64 sscratch_0 = 0x12345678;
    csr_write(sscratch, sscratch_0);
    sscratch = csr_read(sscratch);
    printk("sscratch_1 = %d\n",sscratch);
```

```
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2023 Hello RISC-V
sstatus = 24578
sscratch_0 = 0
sscratch_1 = 305419896
```

OX12345678 = 305419896,故写入成功。

5. **Detail your steps about how to get `arch/arm64/kernel/sys.i`**

安装aarch64后编译

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i
```

在 arch/arm64/kernel 目录下，得到 sys.i

```
# 0 "arch/arm64/kernel/sys.c"
# 1 "/mnt/c/users/21006/desktop/os/linux-6.6-rc4//"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "./../include/linux/compiler-version.h" 1
# 0 "<command-line>" 2
# 1 "./../include/linux/kconfig.h" 1




# 1 "./include/generated/autoconf.h" 1
# 6 "./../include/linux/kconfig.h" 2
# 0 "<command-line>" 2
# 1 "./../include/linux/compiler_types.h" 1
# 80 "./../include/linux/compiler_types.h"
# 1 "./include/linux/compiler_attributes.h" 1
# 81 "./../include/linux/compiler_types.h" 2
# 153 "./../include/linux/compiler_types.h"
# 1 "./include/linux/compiler-gcc.h" 1
# 154 "./../include/linux/compiler_types.h" 2
# 167 "./../include/linux/compiler_types.h"
# 1 "./arch/arm64/include/asm/compiler.h" 1
# 168 "./../include/linux/compiler_types.h" 2


struct ftrace_branch_data {
 const char *func;
 const char *file;
 unsigned line;
 union {
  struct {
   unsigned long correct;
   unsigned long incorrect;
  };
  struct {
   unsigned long miss;
   unsigned long hit;
  };
  unsigned long miss_hit[2];
 };
};
```

6. **Find system call table of Linux v6.0 for `ARM32`, `RISC-V(32 bit)`, `RISC-V(64 bit)`, `x86(32 bit)`, `x86_64`**
   List source code file, the whole system call table with macro expanded, screenshot every step.

   ARM32

```
......
syscall 0, sys_restart_syscall
syscall 1, sys_exit
syscall 2, sys_fork
syscall 3, sys_read
syscall 4, sys_write
syscall 5, sys_open
syscall 6, sys_close
syscall 8, sys_creat
syscall 9, sys_link
RISC-V(64 and 32 bit)
make(for 64 bit)
make(for 32 bit)
syscall 10, sys_unlink
syscall 11, sys_execve
syscall 12, sys_chdir
syscall 14, sys_mknod
```

```
syscall 15, sys_chmod
syscall 16, sys_lchown16
syscall 19, sys_lseek
syscall 20, sys_getpid
syscall 21, sys_mount
syscall 23, sys_setuid16
syscall 24, sys_getuid16
syscall 26, sys_ptrace
syscall 29, sys_pause
syscall 33, sys_access
syscall 34, sys_nice
syscall 36, sys_sync
syscall 37, sys_kill
syscall 38, sys_rename
syscall 39, sys_mkdir
syscall 40, sys_rmdir
syscall 41, sys_dup
syscall 42, sys_pipe
syscall 43, sys_times
syscall 45, sys_brk
syscall 46, sys_setgid16
syscall 47, sys_getgid16
syscall 49, sys_geteuid16
syscall 50, sys_getegid16
```

RISC-V(32 bit) / RISC-V(64 bit)

```
[241] = __riscv_sys_perf_event_open,

[242] = __riscv_sys_accept4,



[243] = __riscv_sys_recvmmsg,
# 632 "./include/uapi/asm-generic/unistd.h"
[260] = __riscv_sys_wait4,



[261] = __riscv_sys_prlimit64,

[262] = __riscv_sys_fanotify_init,

[263] = __riscv_sys_fanotify_mark,

[264] = __riscv_sys_name_to_handle_at,

[265] = __riscv_sys_open_by_handle_at,



[266] = __riscv_sys_clock_adjtime,



[267] = __riscv_sys_syncfs,

[268] = __riscv_sys_setns,

[269] = __riscv_sys_sendmmsg,

[270] = __riscv_sys_process_vm_readv,

[271] = __riscv_sys_process_vm_writev,

[272] = __riscv_sys_kcmp,

[273] = __riscv_sys_finit_module,
```

x86

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ cat arch/x86/entry/syscalls/syscall_64.tbl
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
11      common  munmap                  sys_munmap
12      common  brk                     sys_brk
13      64      rt_sigaction            sys_rt_sigaction
14      common  rt_sigprocmask          sys_rt_sigprocmask
15      64      rt_sigreturn            sys_rt_sigreturn
16      64      ioctl                   sys_ioctl
17      common  pread64                 sys_pread64
18      common  pwrite64                sys_pwrite64
19      64      readv                   sys_readv
20      64      writev                  sys_writev
21      common  access                  sys_access
22      common  pipe                    sys_pipe
```

x86(32 bit)

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ cat arch/x86/entry/syscalls/syscall_32.tbl
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
# sys_*() system calls and compat_sys_*() compat system calls if
# IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
# parameter.
#
# The abi is always "i386" for this file.
#
0       i386    restart_syscall         sys_restart_syscall
1       i386    exit                    sys_exit
2       i386    fork                    sys_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open                        compat_sys_open
6       i386    close                   sys_close
7       i386    waitpid                 sys_waitpid
8       i386    creat                   sys_creat
9       i386    link                    sys_link
10      i386    unlink                  sys_unlink
11      i386    execve                  sys_execve                      compat_sys_execve
12      i386    chdir                   sys_chdir
13      i386    time                    sys_time32
14      i386    mknod                   sys_mknod
15      i386    chmod                   sys_chmod
16      i386    lchown                  sys_lchown16
17      i386    break
18      i386    oldstat                 sys_stat
19      i386    lseek                   sys_lseek                       compat_sys_lseek
```

7. **Explain what is ELF file? Try readelf and objdump command on an ELF file, give screenshot of the output.**
   Run an ELF file and cat `/proc/PID/maps` to give its memory layout.

   readelf操作:

   readelf -h xxx

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13976 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

readelf -s xxx

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ readelf -s test

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _[...]@GLIBC_2.34 (2)
    23: 0000000000001168     0 FUNC    GLOBAL HIDDEN    17 _fini
    24: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    25: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    26: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
    27: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
    28: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   26 _end
    29: 0000000000001060    38 FUNC    GLOBAL DEFAULT   16 _start
    30: 0000000000004010     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    31: 0000000000001149    30 FUNC    GLOBAL DEFAULT   16 main
    32: 0000000000004010     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    33: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    34: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@G[...]
    35: 0000000000001000     0 FUNC    GLOBAL HIDDEN    12 _init
```

readelf -S xxx

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ readelf -S test
There are 31 section headers, starting at offset 0x3698:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       00000000000000a8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000480  00000480
       000000000000008d  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           000000000000050e  0000050e
       000000000000000e  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000520  00000520
       0000000000000030  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000550  00000550
       00000000000000c0  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000000610  00000610
       0000000000000018  0000000000000018  AI       6    24     8
  [12] .init             PROGBITS         0000000000001000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000001020  00001020
       0000000000000020  0000000000000010  AX       0     0     16
  [14] .plt.got          PROGBITS         0000000000001040  00001040
       0000000000000010  0000000000000010  AX       0     0     16
  [15] .plt.sec          PROGBITS         0000000000001050  00001050
       0000000000000010  0000000000000010  AX       0     0     16
```

objdump操作：

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ objdump -d test

test:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 9a 2f 00 00       push   0x2f9a(%rip)        # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       f2 ff 25 9b 2f 00 00    bnd jmp *0x2f9b(%rip)        # 3fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102d:       0f 1f 00                nopl   (%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       f2 e9 e1 ff ff ff       bnd jmp 1020 <_init+0x20>
    103f:       90                      nop

Disassembly of section .plt.got:

0000000000001040 <__cxa_finalize@plt>:
    1040:       f3 0f 1e fa             endbr64
    1044:       f2 ff 25 ad 2f 00 00    bnd jmp *0x2fad(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
    104b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .plt.sec:

0000000000001050 <puts@plt>:
```

运行ELF文件并获取 /proc/PID/maps

```
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ ps aux | grep test
mahong     818 10.8  0.0   2768   980 pts/2    S+   17:53   0:03 ./test
mahong     828  0.0  0.0   4020  2056 pts/1    S+   17:54   0:00 grep --color=auto test
mahong@DESKTOP-7H0E88I:/mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5$ sudo cat /proc/818/maps
56533fd42000-56533fd43000 r--p 00000000 00:2f 1801439850951985          /mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5/test
56533fd43000-56533fd44000 r-xp 00001000 00:2f 1801439850951985          /mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5/test
56533fd44000-56533fd45000 r--p 00002000 00:2f 1801439850951985          /mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5/test
56533fd45000-56533fd46000 r--p 00002000 00:2f 1801439850951985          /mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5/test
56533fd46000-56533fd47000 rw-p 00003000 00:2f 1801439850951985          /mnt/c/Users/MaHong/Desktop/JuniorFirst/OS/OS_EX/linux-6.5.5/test
565340354000-565340375000 rw-p 00000000 00:00 0                         [heap]
7f451a244000-7f451a247000 rw-p 00000000 00:00 0
7f451a247000-7f451a26f000 r--p 00000000 08:10 31896                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f451a26f000-7f451a404000 r-xp 00028000 08:10 31896                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f451a404000-7f451a45c000 r--p 001bd000 08:10 31896                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f451a45c000-7f451a460000 r--p 00214000 08:10 31896                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f451a460000-7f451a462000 rw-p 00218000 08:10 31896                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f451a462000-7f451a46f000 rw-p 00000000 00:00 0
7f451a477000-7f451a479000 rw-p 00000000 00:00 0
7f451a479000-7f451a47b000 r--p 00000000 08:10 31893                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f451a47b000-7f451a4a5000 r-xp 00002000 08:10 31893                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f451a4a5000-7f451a4b0000 r--p 0002c000 08:10 31893                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f451a4b1000-7f451a4b3000 r--p 00037000 08:10 31893                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f451a4b3000-7f451a4b5000 rw-p 00039000 08:10 31893                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffc0a317000-7ffc0a338000 rw-p 00000000 00:00 0                         [stack]
7ffc0a3c5000-7ffc0a3c9000 r--p 00000000 00:00 0                         [vvar]
7ffc0a3c9000-7ffc0a3ca000 r-xp 00000000 00:00 0                         [vdso]
```

8. **在我们使用make run时， OpenSBI 会产生如下输出:**

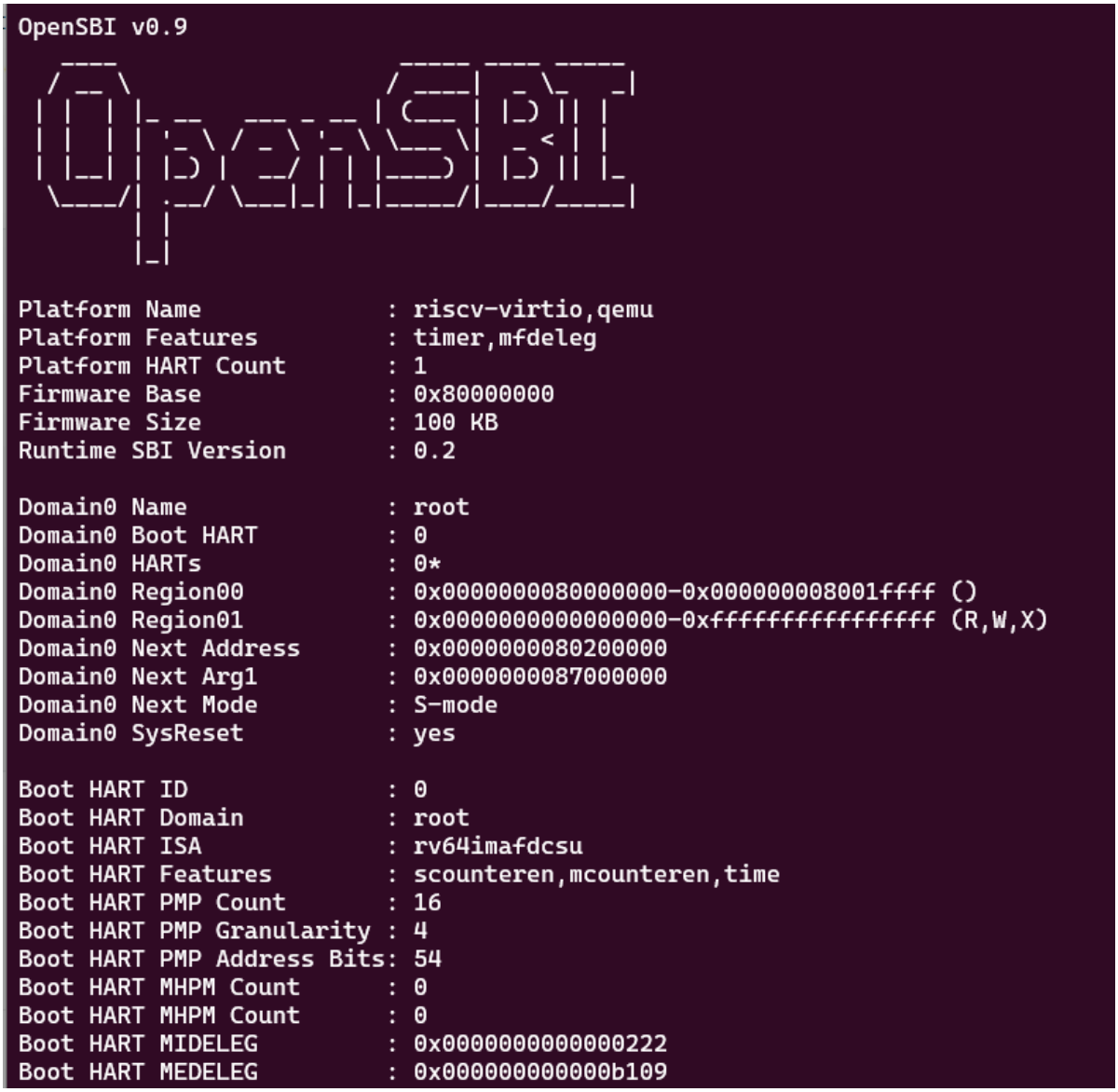
```
OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|


......
```

```
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

......
```

通过查看 `RISC-V Privileged Spec` 中的 `medeleg` 和 `mideleg`，解释上面 `MIDELEG` 值的含义。



`medeleg`和`mideleg`都属于 Machine Trap Delegation Registers，其中`medeleg`指 Machine exception delegation register，而`mideleg`指 Machine interrupt delegation register。`MIDELEG`值为 0x0000000000000222 代表第 2、6、10 位置为了 1，说明实现了虚拟机管理程序扩展（hypervisor extension）。