lab 0.5

练习要求

- 使用gdb调试QEMU模拟的RISC-V计算机加电开始运行到执行应用程序的第一条指令（即跳转到0x80200000）
- 这个阶段的执行过程，说明RISC-V硬件加电后的几条指令的位置,完成的功能

练习1: 使用GDB验证启动流程

文件组成

- 内核启动
  - `kern/init/entry.S`: OpenSBI启动之后将要跳转到的一段汇编代码。在这里进行内核栈的分配，然后转入C语言编写的内核初始化函数。
  - `kern/init/init.c`： C语言编写的内核入口点。主要包含`kern_init()`函数，从`kern/entry.S`跳转过来完成其他初始化工作
- 设备驱动
  - `kern/driver/console.c(h)`: 在QEMU上模拟式，唯一的"设备"是虚拟的控制台，通过OpenSBI接口使用。简单封装了OpenSBI的字符串读写接口，向上提供输入输出库
- 库文件(封装各类函数)
  - `libs/riscv.h`: 以宏的方式，定义了riscv指令集的寄存器和指令。如果在C语言里使用riscv指令，需要通过内联汇编和寄存器的编号。这个头文件把寄存器编号和内联汇编都封装成宏，使得我们可以用类似函数的方式在C语言里执行一句riscv指令。
  - `libs/sbi.c(h)`: 封装OpenSBI接口为函数。如果想在C语言里使用OpenSBI提供的接口，需要使用内联汇编。这个头文件把OpenSBI的内联汇编调用封装为函数。
  - `libs/defs.h`: 定义了一些常用的类型和宏。 例如bool 类型
  - `libs/string.c(h)`: 一些对字符数组进行操作的函数
  - `kern/libs/stdio.c`, `libs/readline.c`, `libs/printfmt.c`: 实现了一套标准输入输出，功能类似于C语言的`printf()` 和`getchar()`。需要内核为输入输出函数提供两个桩函数（stub): 输出一个字符的函数，输入一个字符的函数。在这里，是`cons_getc()`和`cons_putc()`。
- 编译、连接脚本
  - `tools/kernel.ld`: ucore的链接脚本(link script), 告诉链接器如何将目标文件的section组合为可执行文件。
  - `tools/function.mk`: 定义Makefile中使用的一些函数
  - `Makefile`: GNU make编译脚本

启动流程

最小的可执行内核的执行流是：

加电——OpenSBI启动——跳转至0x80200000 (`kern/init/entry.S`) ——进入 `kern_init()`函数（`kern/init/init.c`)，从`kern/entry.S`跳转过来完成其他初始化工作——调用`cprintf()`输出一行信息——结束

`cprintf()`函数的执行流为:

接受若干变量作为参数——解析结构化的字符串，把待输出的各种变量转化为一串字符—— 调用`console.c`提供的字符输出接口依次输出所有字符（实际上`console.c`又封装了`sbi.c`向上提供的OpenSBI接口)

实验模拟

1. 加电、OpenSBI启动

最小可执行内核里, 我们主要完成两件事:

```
1. 内核的内存布局和入口点设置
1. 通过sbi封装好输入输出函数
```

QEMU会帮我们模拟一块riscv64的CPU，一块物理内存，还会借助电脑显示屏和键盘模拟命令行的输入和输出。但缺少硬盘的模拟

在操作系统执行之前，必然有一个其他程序执行，bootloader（负责开机boot，和load(加载OS到内存里)），完成"把操作系统加载到内存"这个工作，完成后把CPU的控制权交给操作系统。

在QEMU模拟的riscv计算机里，我们使用QEMU自带的bootloader—— OpenSBI固件。

即Qemu 开始执行任何指令之前，首先两个文件将被加载到 Qemu 的物理内存中：

1. **作为 bootloader 的 OpenSBI.bin 被加载到物理内存以物理地址 0x80000000 开头的区域上**
2. **内核镜像 os.bin 被加载到以物理地址 0x80200000 开头的区域上**

此时过程为:

操作系统的二进制可执行文件被OpenSBI加载到内存中，然后OpenSBI会把CPU的"当前指令指针"(pc, program counter)跳转到内存里的一个位置，开始执行内存中那个位置的指令。

之后OpenSBI还要把CPU的program counter跳转到一个位置,开始操作系统的执行

实际上，有两种不同的可执行文件格式：`elf`(e是executable的意思， l是linkable的意思，f是format的意思)和`bin`(binary)，为了正确地和上一阶段的 OpenSBI 对接，我们需要保证内核的第一条指令位于物理地址 0x80200000 处，因为这里的代码是地址相关的，这个地址是由处理器，即Qemu指定的。为此，需要将内核镜像预先加载到 Qemu 物理内存以地址 0x80200000 开头的区域上。一旦 CPU 开始执行内核的第一条指令，证明计算机的控制权已经被移交给我们的内核

`elf`文件：包含一个文件头(ELF header), 包含冗余的调试信息，指定程序每个section的内存布局，需要解析program header才能知道各段(section)的信息。如果我们已经有一个完整的操作系统来解析elf文件，那么elf文件可以直接执行。但是对于OpenSBI来说，elf格式还是太复杂了。

`bin`文件：简单地在文件头之后解释自己应该被加载到什么起始位置。

elf文件和bin文件的区别：可以认为bin文件会把elf文件指定的每段的内存布局都映射到一块线性的数据里，这块线性的数据（或者说程序）加载到内存里就符合elf文件之前指定的布局

因此我们任务明确为：

1. **得到内存布局合适的elf文件，然后把它转化成bin文件（这一步通过objcopy实现）**
2. **然后加载到QEMU里运行（QEMU自带的OpenSBI会干这个活）**

2.kern_init()

在`kern/init/init.c`编写函数`kern_init`,作为"真正的"内核入口点。为了让我们能看到一些效果，我们希望它能在命令行进行格式化输出。

如果我们在linux下运行一个C程序，需要格式化输出，该`#include<stdio.h>`，但是实际上依赖于`glibc`提供的运行时环境，也就是一定程度上依赖于操作系统提供的支持。可是我们并没有把`glibc`移植到ucore里。此时，我们只能在ucore里重新实现，QEMU里的OpenSBI固件提供了输入一个字符和输出一个字符的接口，会把这个接口一层层封装起来，提供`stdio.h`里的格式化输出函数`cprintf()`来使用。

下一步中，将从OpenSBI的接口一层层封装到格式化输入输出函数。

3.OpenSBI接口封装与cprintf()执行

OpenSBI作为运行在M态的软件（或者说固件）,提供了一些接口供我们编写内核的时候使用。我们可以通过`ecall`指令(environment call)调用OpenSBI。通过寄存器传递给OpenSBI一个"调用编号"，如果编号在`0-8`之间，则由OpenSBI进行处理，否则交由我们自己的中断处理程序处理（暂未实现）。

C语言并不能直接调用`ecall`,需要通过内联汇编来实现。

这样我们就可以通过`sbi_console_putchar()`来输出一个字符。接下来我们要做的事情就像月饼包装，把它封了一层又一层。`console.c`只是简单地封装一下

```
// kern/driver/console.c
#include <sbi.h>
#include <console.h>

void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }
```

`stdio.c`里面实现了一些函数，注意我们已经实现了ucore版本的puts函数:`cputs()`

我们还在`libs/printfmt.c`实现了一些复杂的格式化输入输出函数。最后得到的`cprintf()`函数仍在`kern/libs/stdio.c`定义，功能和C标准库的`printf()`基本相同

`printfmt.c`还依赖一个头文件`riscv.h`,这个头文件主要定义了若干和riscv架构相关的宏，尤其是将一些内联汇编的代码封装成宏，使得我们更方便地使用内联汇编来读写寄存器。

```
// libs/riscv.h
...
#define read_csr(reg) ({ unsigned long __tmp; \
  asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
  __tmp; })
//通过内联汇编包装了 csrr 指令为 read_csr() 宏
#define write_csr(reg, val) ({ \
  if (__builtin_constant_p(val) && (unsigned long)(val) < 32) \
    asm volatile ("csrw " #reg ", %0" :: "i"(val)); \
  else \
    asm volatile ("csrw " #reg ", %0" :: "r"(val)); })
...
```

4.具体实现

目标工作：

1. 编译所有的源代码
2. 把目标文件链接起来
3. 生成elf文件
4. 生成bin硬盘镜像
5. 用qemu跑起来

为了实现这些功能，我们直接调用Makefile。

make 和 Makefile

GNU make是一种代码维护工具，在大中型项目中，它将根据程序各个模块的更新情况，自动的维护和生成目标代码。

make命令执行时，需要一个 makefile （或Makefile）文件，以告诉make命令需要怎么样的去编译和链接程序

我们的`Makefile`还依赖`tools/function.mk`。make命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

makefile的基本规则简洁

```
target ... : prerequisites ...
    command
    ...
    ...
```

- target也就是一个目标文件，可以是object file，也可以是执行文件。还可以是一个标label
- prerequisites就是，要生成那个target所需要的文件或是目标
- command也就是make需要执行的命令（任意的shell命令）

这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在 command中。

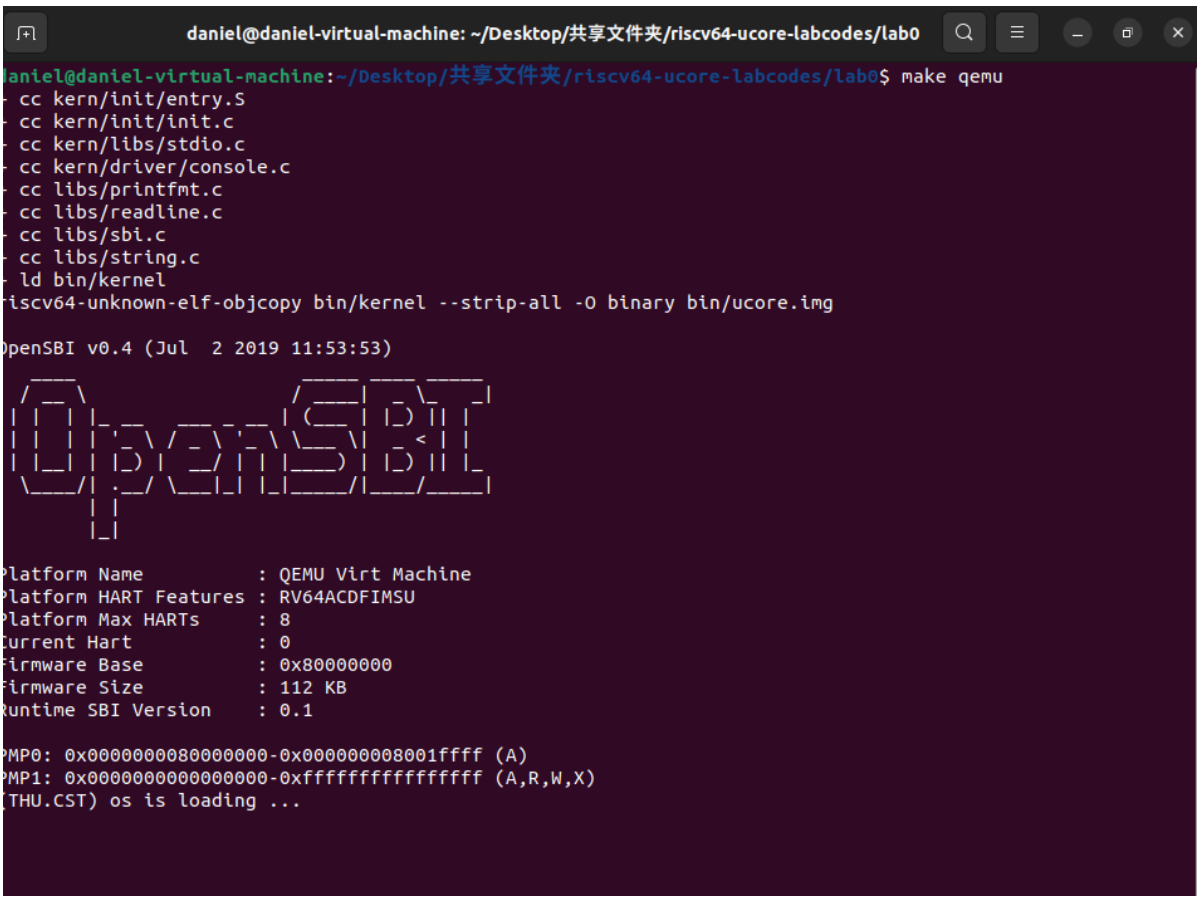
如果prerequisites中有一个以上的文件比target文件要新，那么command所定义的命令就会被执行。这就是makefile的规则。也就是makefile中最核心的内容

Runing ucore

在源代码的根目录下`make qemu`, 在makefile中运行的代码为

```
.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#    $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
    $(V)$(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000
```

这段代码就是我们启动qemu的命令，这段代码首先通过宏定义$(UCOREIMG) $(SWAPIMG) $(SFSIMG)的函数进行目标文件的构建，然后使用qemu语句进行qemu启动加载内核。



它输出一行`(THU.CST) os is loading`，然后进入死循环。

补充练习：知识点补充

> 对于有**复杂的读写时序要求**的设备，比如硬盘，CPU是无法直接读取的，需要借助驱动程序告知CPU如何与设备通讯，读取数据。这些驱动程序存储在一个不需要驱动程序，CPU就可以直接读取的地方，如早期的ROM芯片，以及后期相对普及的NOR Flash芯片，在CPU启动时，会首先运行这些代码，用这些代码实现对硬盘、内存和其他复杂设备的读取

> **固件(firmware)**是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件,固件可为设备更复杂的软件，提供标准化的操作环境。在基于 riscv 的计算机系统中，OpenSBI 是固件。OpenSBI运行在**M态（M-mode）**，因为固件需要直接访问硬件。（U-mode是用户程序、应用程序的特权级，S-mode是操作系统内核的特权级，M-mode是固件的特权级。）

地址选择的细节：

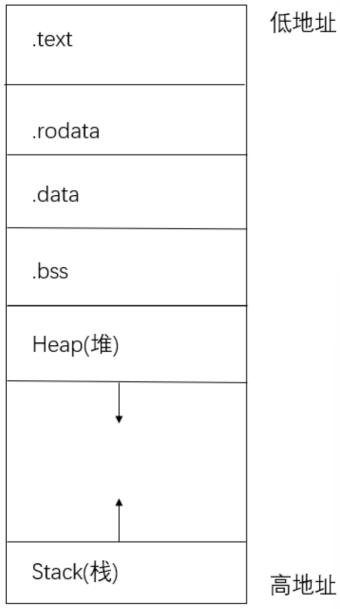
> QEMU模拟的这款riscv处理器的**复位地址**是0x1000，而不是0x80000000
>
> 所谓**复位地址**，指的是CPU在上电的时候，或者按下复位键的时候，PC被赋的初始值
>
> RISCV的设计标准相对灵活，它允许芯片的实现者自主选择复位地址，因此不同的芯片会有一些差异。QEMU-4.1.1是选择了一种实现方式进行模拟
>
> 在QEMU模拟的这款riscv处理器中，将复位向量地址初始化为0x1000，再将PC初始化为该复位地址,复位代码主要是将计算机系统的各个组件（包括处理器、内存、设备等）置于初始状态，并且会启动Bootloader(此处的位置为0x80000000)，Bootloader将加载操作系统内核并启动操作系统的执行。

代码地址的相关性：

> 为什么内核一定要被加载到0x80200000的位置开始呢?
>
> 以一行C语言代码为例，`int sum = 0;`，在编译和链接的时候就会分配空间的具体地址，假设这个地址是`pa_sum`。`sum +=5;`这样的语句时，相应的机器指令是，将`pa_sum`位置的值加载入寄存器，完成加法计算，再将这个值写回内存中对应的`pa_sum`处
>
> 而此时生成的指令，会将`pa_sum`的值（假设为0x55aa55aa），直接写入到指令的编码中，如`load r1 0x55aa55aa`
>
> 如此一来，则这一段代码就成了**地址相关代码**，即指令中的访存信息在编译完成后即已成为绝对地址，那么在运行之前，自然需要将所需要的代码加载到指定的位置

一般来说，程序会分为如下这些段（section）：

> .text 段，即代码段，存放汇编代码
>
> .rodata 段，只读数据，通常是程序中的常量
>
> .data 段，被初始化的可读写数据，通常保存全局变量
>
> .bss 段，存放被初始化为 00 的可读写数据，与 .data 段的不同之处在于初始化为 00
>
> stack ，即栈，运行过程中的局部变量，以及负责函数调用时的各种机制。它从高地址向低地址增长
>
> heap ，即堆，用来支持程序运行过程中内存的动态分配



gnu工具链中，包含一个链接器`ld`主要作用是：把输入文件(往往是 .o文件)链接成输出文件(往往是elf文件)。

函数调用与calling convention:

> 编译器将高级语言源代码翻译成汇编代码。对于汇编语言而言，在最简单的编程模型中，所能够利用的只有指令集中提供的指令、各通用寄存器、 CPU 的状态、内存资源。因此在高级语言中，进行一次函数调用，需要考虑，参数传递，返回值传递，函数开始地址传递，函数返回后能从期望位置继续执行。
>
> 通常编译器按照某种规范去翻译所有的函数调用，这种规范被称为 calling convention。此外，为了实现函数调用，我们需要预先分配一块内存作为 **调用栈**

---

lab1：断,都可以断

实验目的

在最小可执行内核的基础上，支持中断机制，并且用时钟中断来检验我们的中断处理系统。

实验要求

- 理解内核启动中的程序入口操作
  - 阅读 kern/init/entry.S内容代码，结合操作系统内核启动流程，说明指令la sp, bootstacktop 完成的操作，以及操作的目的。
  - tail kern_init完成的操作，操作的目的。
- 完善中断处理
  - 完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写kern/trap/trap.c函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”100 ticks”，在打印完10行后调用sbi.h中shut_down()函数关机。
- 扩展练习1：描述与理解中断流程
  - 描述ucore中处理中断异常的流程（从异常的产生开始），其中mov a0，sp的目的。
  - SAVE_ALL中寄寄存器保存在栈中的位置是什么确定的。
  - 对于任何中断，__alltraps 中都需要保存所有寄存器吗?
- 扩展练习2：理解上下文切换机制
  - 在trapentry.S中汇编代码 csrw sscratch, sp。
  - csrrw s0, sscratch, x0实现了什么操作，其目的是?
  - save all里面保存了stval scause这些csr，而在restore all里面却不还原它们，那这样store的意义何在?

练习1：内核启动中的程序入口操作

项目组成：

- 硬件驱动层：`kern/driver/clock.c(h)`：通过`OpenSBI`的接口, 可以读取当前时间(rdtime), 设置时钟事件(sbi_set_timer)，是时钟中断必需的硬件支持`kern/driver/intr.c(h)`：中断也需要CPU的硬件支持，这里提供了设置中断使能位的接口其实只封装了一句riscv指令）。
- 初始化：`kern/init/init.c`: 需要调用中断机制的初始化函数。
- 中断处理：`kern/trap/trapentry.S`: 我们把中断入口点设置为这段汇编代码。这段汇编代码把寄存器的数据挪来挪去，进行上下文切换。

  `kern/trap/trap.c(h)`: 分发不同类型的中断给不同的handler, 完成上下文切换之后对中断的具体处理，例如外设中断要处理外设发来的信息，时钟中断要触发特定的事件。中断处理初始化的函数也在这里，主要是把中断向量表(stvec)设置成所有中断都要跳到`trapentry.S`进行处理。

内核启动流程:

内核初始化函数`kern_init()`的执行流:

——从`kern/init/entry.S`进入

——输出一些信息说明正在初始化

——设置中断向量表(stvec）跳转到的地方为`kern/trap/trapentry.S`里的一个标记

——在`kern/driver/clock.c`设置第一个时钟事件，使能时钟中断

——设置全局的S mode中断使能位

—— 现在开始不断地触发时钟中断

产生一次时钟中断的执行流:

——set_sbi_timer()通过OpenSBI的时钟事件触发一个中断，跳转到`kern/trap/trapentry.S`的`__alltraps`标记

——保存当前执行流的上下文，并通过函数调用，切换为`kern/trap/trap.c`的中断处理函数`trap()`的上下文，进入`trap()`的执行流

——切换前的上下文作为一个结构体，传递给`trap()`作为函数参数

—— `kern/trap/trap.c`按照中断类型进行分发(`trap_dispatch(), interrupt_handler()`)

——执行时钟中断对应的处理语句，累加计数器，设置下一次时钟中断

——完成处理，返回到`kern/trap/trapentry.S`

——恢复原先的上下文，中断处理结束

练习2：完善中断处理

riscv64 中断介绍

中断概念

中断（interrupt）机制，就是不管CPU现在手里在干啥活，收到“中断”的时候，都先放下来去处理其他事情，处理完其他事情可能再回来干手头的活。中断机制需要软件硬件一起来支持。硬件进行中断和异常的发现，然后交给软件来进行处理。

中断分类

异常(Exception)：在执行一条指令的过程中发生了错误，此时我们通过中断来处理错误

陷入(Trap)：主动通过一条指令停下来，并跳转到处理函数。

外部中断(Interrupt)： CPU 的执行过程被外设发来的信号打断，此时我们必须先停下来对该外设进行处理。外部中断是异步(asynchronous)的，CPU 并不知道外部中断将何时发生。

由于中断处理需要进行较高权限的操作，中断处理程序一般处于**内核态**

riscv64 权限模式

M-mode是 RISC-V 中 hart可以执行的最高权限模式。默认情况下,发生所有异常(不论在什么权限模式下)的时候,控制权都会被移交到 M 模式的异常处理程序。

S-mode是支持现代类 Unix 操作系统的权限模式，支持基于页面的虚拟内存机制是其核心。Unix 系统中的大多数exception都应该进行 S 模式下的系统调用。

寄存器

`sstatus`寄存器二进制位`SIE`数值为0的时候，如果当程序在S态运行，将禁用全部中断。

`sstatus`寄存器二进制位`UIE`可以在置零的时候禁止用户态程序产生中断。

`stvec`即所谓的"中断向量表基址"，作用就是把不同种类的中断映射到对应的中断处理程序。与此同时，`stvec`会把最低位的两个二进制位用来编码一个"模式"。如果是"00"就说明更高的SXLEN-2个二进制位存储的是唯一的中断处理程序的地址，如果是"01"说明更高的SXLEN-2个二进制位存储的是中断向量表基址。

当我们触发中断进入 S 态进行处理时，以下寄存器会被硬件自动设置，将一些信息提供给中断处理程序：

- **sepc**：它会记录触发中断的那条指令的地址
- **scause**：它会记录中断发生的原因，还会记录该中断是不是一个外部中断
- **stval**：它会把发生问题的目标地址或者出错的指令记录下来，这样我们在中断处理程序中就知道处理目标了

特权指令

RISCV支持以下和中断相关的特权指令：

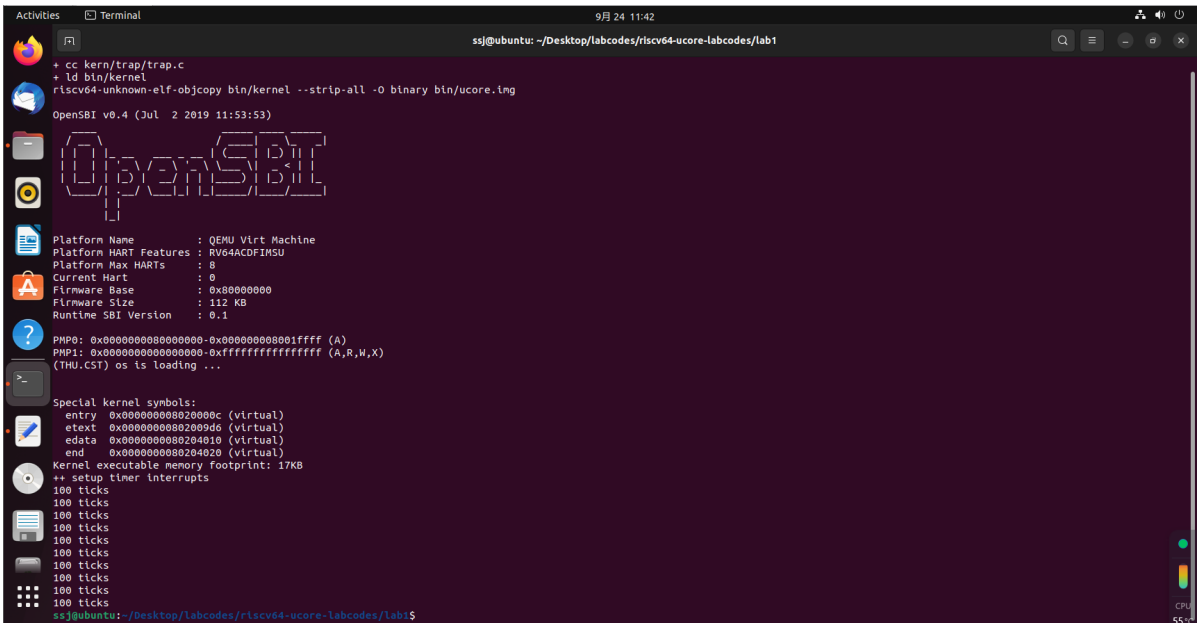
- **ecall**：当我们在 S 态执行这条指令时，会触发一个 ecall-from-s-mode-exception，从而进入 M 模式中的中断处理流程。当我们在 U 态执行这条指令时，会触发一个 ecall-from-u-mode-exception，从而进入 S 模式中的中断处理流程
- **sret**：用于 S 态中断返回到 U 态，实际作用为pc←sepc，即返回到通过中断进入 S 态之前的地址。
- **ebreak**：触发一个断点中断从而进入中断处理流程。
- **mret**：用于 M 态中断返回到 S 态或 U 态，实际作用为pc←mepc，返回到通过中断进入 M 态之前的地址。

具体实现代码

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1; //抹掉scause最高位代表"这是中断不是异常"的1
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
            .....
         case IRQ_S_TIMER:
            //时钟中断
            /* LAB1 EXERCISE2   YOUR CODE : 学号*/
           ticks++;
```

```
            if(ticks%TICK_NUM ==0)
            {print_ticks();}
            if(ticks%TICK_NUM ==0)
            sbi_shutdown();
             /*你的代码*/

             break;
             .....
           default:
             print_trapframe(tf);
             break;
    }
}
```



扩展练习 Challenge1：描述与理解中断流程

内容介绍

Trap中断处理程序

中断处理需要初始化，所以我们在`init.c`里调用一些初始化的函数

```
#include <trap.h>
int kern_init(void) {
...
    //trap.h的函数，初始化中断
    idt_init();  // init interrupt descriptor table

    //clock.h的函数，初始化时钟中断
    clock_init();
    //intr.h的函数，使能中断
    intr_enable();
    while (1)
        ;
}
```

```
// kern/trap/trap.c
void idt_init(void) {
    extern void __alltraps(void);
    //约定：若中断前处于S态，sscratch为0
    //若中断前处于U态，sscratch存储内核栈地址
    //那么之后就可以通过sscratch的数值判断是内核态产生的中断还是用户态产生的中断
    //我们现在是内核态所以给sscratch置零
    write_csr(sscratch, 0);
    //我们保证__alltraps的地址是四字节对齐的，将__alltraps这个符号的地址直接写到stvec寄存器
    write_csr(stvec, &__alltraps);
}
```

trap.c的中断处理函数trap, 实际上把中断处理,异常处理的工作分发给interrupt_handler(), exception_handler(),在其中我们实现了简单的时钟中断功能。即每次触发时钟中断的时候，我们会给一个计数器加一，并且设定好下一次时钟中断。当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断。

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1; //抹掉scause最高位代表“这是中断不是异常”的
1
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
            .....
         case IRQ_S_TIMER:
            //时钟中断
            /* LAB1 EXERCISE2   YOUR CODE : 学号*/

            /*你的代码*/

            break;
            .....
          default:
            print_trapframe(tf);
            break;
    }
}
```

Q1：描述ucore中处理中断异常的流程（从异常的产生开始），其中mov a0，sp的目的是什么?

具体流程在上述内容中，mov a0，sp的目的是：

```
move  a0, sp #传递参数。
    #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
    #trap是trap.c里面的一个C语言函数，也就是我们的中断处理程序
    jal trap
```

Q2：SAVE_ALL中寄寄存器保存在栈中的位置是什么确定的?

其保存的位置是由栈顶指针sp决定的，剩余寄存器的位置在sp之下线性排列

```
addi sp, sp, -36 * REGBYTES #REGBYTES是riscv.h定义的常量，表示一个寄存器占据几个字节
  #让栈顶指针向低地址空间延伸 36个寄存器的空间，可以放下一个trapFrame结构体。
  #除了32个通用寄存器，我们还要保存4个和中断有关的CSR
```

Q3：对于任何中断，__alltraps 中都需要保存所有寄存器吗?

对照表格，结构体中pushregs应该是能保存所有寄存器的，但不是任何中断都需要保留所有寄存器。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

```
struct pushregs {
    uintptr_t zero;  // Hard-wired zero
    uintptr_t ra;    // Return address
    uintptr_t sp;    // Stack pointer
    uintptr_t gp;    // Global pointer
    uintptr_t tp;    // Thread pointer
    uintptr_t t0;    // Temporary
    uintptr_t t1;    // Temporary
    uintptr_t t2;    // Temporary
    uintptr_t s0;    // Saved register/frame pointer
    uintptr_t s1;    // Saved register
    uintptr_t a0;    // Function argument/return value
    uintptr_t a1;    // Function argument/return value
    uintptr_t a2;    // Function argument
    uintptr_t a3;    // Function argument
    uintptr_t a4;    // Function argument
    uintptr_t a5;    // Function argument
    uintptr_t a6;    // Function argument
    uintptr_t a7;    // Function argument
    uintptr_t s2;    // Saved register
    uintptr_t s3;    // Saved register
    uintptr_t s4;    // Saved register
    uintptr_t s5;    // Saved register
    uintptr_t s6;    // Saved register
    uintptr_t s7;    // Saved register
    uintptr_t s8;    // Saved register
    uintptr_t s9;    // Saved register
    uintptr_t s10;   // Saved register
    uintptr_t s11;   // Saved register
    uintptr_t t3;    // Temporary
    uintptr_t t4;    // Temporary
    uintptr_t t5;    // Temporary
    uintptr_t t6;    // Temporary
};
```

扩增练习 Challenge2：理解上下文切换机制

内容介绍：

下文切换(context switch)机制:

- 保存CPU的寄存器（上下文）到内存中（栈上）
- 从内存中（栈上）恢复CPU的寄存器

为了方便我们组织上下文的数据,我们定义一个结构体.

```
struct trapframe {
    struct pushregs gpr;
    uintptr_t status; //RISCV寄存器状态
    uintptr_t epc; //记录触发中断的那条指令的地址
    uintptr_t badvaddr; //读取这个寄存器来获取引发异常的虚拟地址
    uintptr_t cause; //记录中断发生的原因
};
```

首先我们定义一个汇编宏 `SAVE_ALL`，用来保存所有寄存器到栈顶

```
.macro SAVE_ALL #定义汇编宏

    csrw sscratch, sp #保存原先的栈顶指针到sscratch
 #依次保存32个通用寄存器。但栈顶指针需要特殊处理。
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    STORE x3, 3*REGBYTES(sp)
    ...
    STORE x31, 31*REGBYTES(sp)
 # RISCV不能直接从CSR写到内存，需要csrr把CSR读取到通用寄存器，再从通用寄存器STORE到内存
    csrrw s0, sscratch, x0
    csrr s1, sstatus
    csrr s2, sepc
    csrr s3, sbadaddr
    csrr s4, scause

    STORE s0, 2*REGBYTES(sp)
    ...
    STORE s4, 35*REGBYTES(sp)
    .endm #汇编宏定义结束
```

然后是恢复上下文的汇编宏，恢复的顺序和当时保存的顺序反过来，先加载两个CSR（控制和状态寄存器），再加载通用寄存器

```
.macro RESTORE_ALL

LOAD s1, 32*REGBYTES(sp)
LOAD s2, 33*REGBYTES(sp)

# 注意之前保存的几个CSR并不都需要恢复
csrw sstatus, s1
csrw sepc, s2

# 恢复sp之外的通用寄存器
LOAD x1, 1*REGBYTES(sp)
LOAD x3, 3*REGBYTES(sp)
...
LOAD x31, 31*REGBYTES(sp)
# 最后恢复sp
LOAD x2, 2*REGBYTES(sp)
.endm
```

此时编写真正的中断入口点 `trapentry.S`

```
.globl __alltraps

.align(2) #中断入口点 __alltraps必须四字节对齐
__alltraps:
    SAVE_ALL #保存上下文
    move  a0, sp #传递参数。
    #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
     #trap是trap.c里面的一个C语言函数，也就是我们的中断处理程序
    jal trap
    #trap函数指向完之后，会回到这里向下继续执行__trapret
   .globl __trapret
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

问题回答：

Q1：在trapentry.S中汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么?

csrw sscratch, sp ： 其作用为保存原先的栈顶指针到sscratch

csrrw s0, sscratch, x0 ： RISCV不能直接从CSR写到内存, 需要csrr把CSR读取到通用寄存器，再从通用寄存器STORE到内存，此为保存当前状态时的操作。

Q2：save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢?

首先需要明确这些scr的作用，其中：

```
struct trapframe {
    struct pushregs gpr;
    uintptr_t status; //RISCV寄存器状态
    uintptr_t epc; //记录触发中断的那条指令的地址
    uintptr_t badvaddr; //读取这个寄存器来获取引发异常的虚拟地址
    uintptr_t cause; //记录中断发生的原因
};
```

因此，为了恢复中断前状态，即寄存器状态，和pc寄存器的值，只需要保存stval scause这些csr，而与异常具体信息有关的scr则不用。

储存这些信息的意义：该结构体保存了异常发生的具体信息，在调用trap函数时，需要用到

badvaddr，cause来确定具体的异常信息，执行相应功能。

扩展练习Challenge3：完善异常中断

代码编写

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            break;
            .....
        case CAUSE_ILLEGAL_INSTRUCTION:
            //非法指令异常处理
            /* LAB1 CHALLENGE3   YOUR CODE : 2113157/
            /*(1)输出指令异常类型（ Illegal instruction）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
             */
            cprintf("Illegal instruction\n");
            cprintf("Illegal instruction address: %p\n", tf->epc);
            tf->epc += 4;
            break;
        case CAUSE_BREAKPOINT:
            //非法指令异常处理
            /* LAB1 CHALLLENGE3   YOUR CODE :  2113157/
            /*(1)输出指令异常类型（ breakpoint）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
             */
            cprintf("Breakpoint\n");
            cprintf("Breakpoint address: %p\n", tf->epc);
            tf->epc += 4;
            break;
            .....
            default:
            print_trapframe(tf);
            break;
    }
}
```