实验分工：
由于lab0.5和lab1比较基础，涉及配置环境、对RISCV和OpenSBI的基础理解，因此小组三人都对所有练习进行了实现，然后一起讨论遇到的问题以及总结知识点。

Lab 0.5

实验目的

- 熟悉使用qemu和gdb进行调试工作,使用gdb调试并验证QEMU模拟的RISC-V计算机从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。

实验过程

RISC—V加电后的启动流程主要分为三个阶段。 第一阶段，将必要的文件载入到 Qemu 物理内存之后，Qemu CPU 的程序计数器（PC）会被初始化为 0x1000 ， 因此 Qemu 实际执行的第一条指令位于物理地址 0x1000 ，处理器将从此处开始执行复位代码，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，在这里QEMU的复位代码指定加载Bootloader的位置为0x80000000，即执行完复位代码后，跳转到物理地址 0x80000000 对应的指令处。

第二阶段，由于 Qemu 的第一阶段固定跳转到 0x80000000 ，所以需要将作为 bootloader 的 OpenSBI.bin 被加载到以物理地址 0x80000000 开头的区域上，这样就能保证 0x80000000 处正好保存 bootloader 的第一条指令。在 OpenSBI 的初始化工作完成之后，它会跳转到0x80200000地址处，并将计算机控制权移交给下一阶段的软件——内核镜像。

第三阶段，为了正确地和上一阶段的 OpenSBI 对接，需要保证内核的第一条指令位于物理地址 0x80200000 处。为此，我们需要将内核镜像os.bin预先加载到 Qemu 以物理地址 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，也就达到了本节的目标。

知识点

- 操作系统作为一个程序，必须加载到内存里才能执行，并不是计算机加电后直接运行。所以需要bootloader将操作系统加载入内存。
- 程序的分段：一般来说，一个程序按照功能不同会分为.text 段，即代码段，存放汇编代码； .rodata 段，即只读数据段，存放只读数据，通常是程序中的常量； .data 段，存放被初始化的可读写数据，通常保存程序中的全局变量； .bss 段，存放被初始化为 00 的可读写数据 stack ，即栈，用来存储程序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。
- 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，将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。

Lab 1

实验目的

- 学习riscv 的中断相关知识

- 学习中断前后如何实现上下文环境的保存与恢复
- 处理最简单的断点中断和时钟中断

练习1

entry.S是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。

la sp, bootstacktop

la即load address，表示将bootstacktop的地址加载到sp寄存器，即栈寄存器中，将栈顶的值赋为kern_init的入口地址。

tail kern_init

调用kern_init函数，在一个函数的最后调用另一个函数，称为尾调用，指令为tail call，可简写为tail。

练习2

时钟中断类型的处理过程如下：

set_sbi_timer()通过OpenSBI的时钟事件触发一个中断，跳转到trapentry.S的__alltraps标记，保存当前执行流的上下文，以待后续的恢复。
通过函数调用，切换为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，里面包括32个通用寄存器及4个控制状态寄存器的值。
trap.c按照中断类型进行分发(trap_dispatch() 或 interrupt_handler())。执行时钟中断对应的处理语句，累加计数器，同时设置下一次时钟中断。
完成处理，返回到trapentry.S。
恢复原先的上下文，中断处理结束。

实现代码如下：

```
case IRQ_S_TIMER:
            // (1)设置下次时钟中断- clock_set_next_event()
            clock_set_next_event();
            // (2)计数器（ticks）加一
            ticks++;
            // (3)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，同时打印次数（num）加一
            if(ticks % TICK_NUM == 0){
                cprintf("100 ticks\n");
                num++;
            }
            // (4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
            if(num == 10){
                sbi_shutdown();
            }
            break;
```

Challenge 1

- 中断处理流程

  当 CPU 执行指令时发生异常、陷入或者外部中断之后，CPU 会自动执行一系列操作，保存当前执行流的上下文，即 CPU 寄存器的值。

  之后转移控制到异常处理程序，异常处理程序的入口地址由中断向量表 (stvec) 提供，中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序，那么可以让stvec直接指向那个中断处理程序的地址。CPU 将控制转移到异常处理程序的入口处，开始执行异常处理代码。

  异常处理程序根据异常类型和原因来执行相应的处理代码。

  当异常处理程序完成任务后，需要恢复被保存的寄存器的值。

  最后，使用异常返回指令将控制权交还给原来被中断的指令，从而恢复正常的程序执行流程。

- mov a0，sp的目的是什么?
  mov a0，sp指令将栈顶指针的值存入a0寄存器中，a0寄存器传递参数给接下来调用的函数trap。

- SAVE_ALL中寄存器保存在栈中的位置是什么确定的?
  由栈顶指针和预先规定好的存储顺序决定的。

- 对于任何中断，__alltraps 中都需要保存所有寄存器吗？ 需要。因为判断哪个寄存器中的值是需要保存的效率太低，复杂度高，直接保存所有寄存器的值开销并不会过大，更方便。

Challenge 2

- csrw sscratch, sp 保存原先的栈顶指针到sscratch。
  由于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，第一步读取到通用寄存器使用指令 csrrw s0, sscratch, x0 。将sscratch先存到s0寄存器中，再将x0存入sscratch中，即把sscratch赋值为0，目的是在发生递归异常时，异常向量可以知道该异常来自内核。
- 控制状态寄存器中存储的是有关中断处理的信息，例如中断类型，中断发生的原因以及一些 辅助信息，仅供中断处理使用，与程序的正常执行没有关系。所以只需恢复通用寄存器就足 够用于返回原本执行的程序。

Challenge 3

- 非法指令异常处理

```
case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
             /* LAB1 CHALLENGE3   YOUR CODE :  */
            /*(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("Exception type:Illegal instruction");
            cprintf("Illegal instruction caught at:0x%016llx\n",tf->epc);
            tf->epc += 2;
            break;
```

- 断点异常处理

```
case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB1 CHALLLENGE3   YOUR CODE :  */
            /*(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("Exception type: breakpoint");
            cprintf("breakpoint caught at 0x%016llx\n",tf->epc);
            tf->epc += 2;
            break;
```

知识点

- S模式支持内存分页机制，通过查看entry.s文件以及向上探究，发现操作系统内核占两页，而每页内存大小为4096字节，即4MB，因此内核大小为8MB;
- RISCV一般来说有三种模式：M模式、S模式、U模式，权限从高到低。在M模式下可以实现和固件的交互、操作系统内核态可以处于S模式。
- 操作系统中断处理方法总结如下：
  发生异常、陷入或外部中断时，CPU保存执行流上下文->转移到异常处理程序入口处->执行相应处理代码->恢复原中断处上下文->返回原来中断指令处，恢复正常的程序执行流程。
- 需要保存的特殊寄存器（CSRs）有：
  sstatus：有一个二进制位SIE和UIE，以SIE为例，如果该位为0，则操作系统在S态时将禁用全部中断。（相当于中断使能位）
  sepc：保存中断指令地址。
  scause：保存中断原因和是否是外部中断。
  stval：保存中断辅助信息，便于中断处理程序的执行。
  badvaddr：当访问无效或超内存地址时，badvaddr会保存该条指令的地址。
- RISCV不能直接实现CSR和内存的交互，因此需要通用寄存器作为桥梁。
- 区分不同的中断类型：
  外部中断（External Interrupt）：由外部设备触发的中断，例如键盘输入、鼠标点击、网络数据到达等。

  内部中断（Internal Interrupt）：由处理器内部发生的事件触发的中断，例如除零错误、越界访问等。

  异常中断（Exception Interrupt）：由程序执行的异常情况触发的中断，例如非法指令、非法地址等。

  陷阱中断（Trap Interrupt）：由软件故意产生的中断，通常用于系统调用和调试的目的。
- 想要实现自己封装函数比如cprintf或者获取当前时间等，可以通过调用OpenSBI内部程序实现。