OS Lab0.5&Lab1

操作系统课程实验报告

实验名称：Lab0.5 最小可执行内核&Lab1 中断处理机制

小组成员：2113999 陈翊炀 2112849唐静蕾 2113388 高涵

Ⅰ.Lab0.5 最小可执行内核

一、实验目的

实验0.5主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识。

二、实验内容

练习1:使用GDB验证启动流程

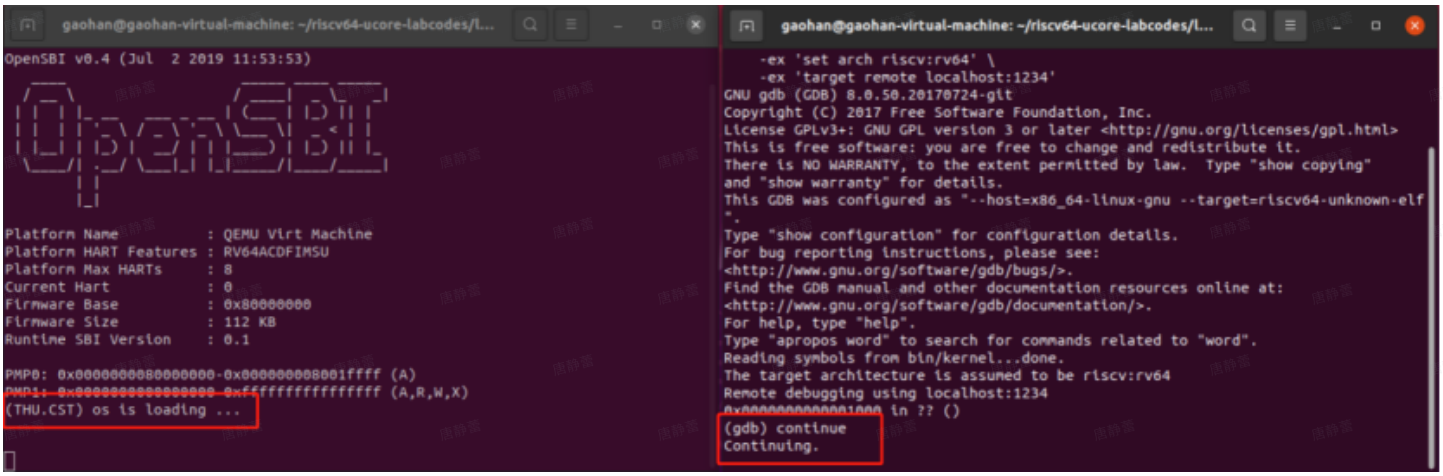
> 为了熟悉使用qemu和gdb进行调试工作,使用gdb调试QEMU模拟的RISC-V计算机加电开始运行到执行应用程序的第一条指令（即跳转到0x80200000）这个阶段的执行过程，说明RISC-V硬件加电后的几条指令在哪里？完成了哪些功能？要求在报告中简要写出练习过程和回答。

Pt1：说明RISC-V硬件加电后的几条指令在哪里?完成了哪些功能?

首先，我们可以观察得到RISCV硬件加电后系统执行了什么：

加电 -> OpenSBI启动 -> 跳转到 0x80200000 (kern/init/entry.S）->进入kern_init()函数（kern/init/init.c) ->调用cprintf()输出一行信息->结束

具体执行代码及截图如图：



分析之后发现加电后首先启动openSBI，查看执行结果，确实与代码所示一致，即：在进入kern init()函数之后输入continue执行指令会实现对代码的执行，分析代码发现，其主要功能是会在命令行输出`(THU.CST) os is loading ...`，之后的代码是死循环while(1)。

根据代码显示确实如此，代码在持续运行（continuing）没有结束，直到手动关闭命令行。

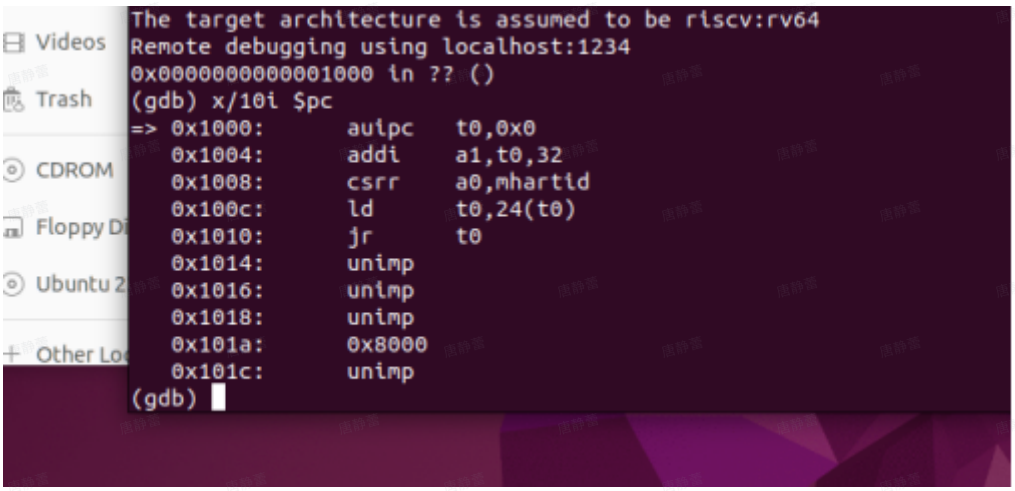
Pt2:对指令具体调试分析

我们使用远程调试手段，输入

```
x/10i $pc
```

终端显示即将执行的10条汇编指令：

```
0x1000:    auipc    t0,0x0
0x1004:    addi     a1,t0,32
0x1008:    csrr     a0,mhartid
0x100c:    ld       t0,24(t0)
0x1010:    jr       t0
0x1014:    unimp
0x1016:    unimp
0x1018:    unimp
0x101a:    0x8000
0x101c:    unimp
```



通过上面的汇编代码我们可以得知，首先执行的是地址0x1000的一段代码，然后才跳转到t0地址，我们可以得出起始地址为 `0x1000` 的初步结论。

查阅资料学习后，可知这段代码实际上是qemu自带的启动代码，它执行完0x1010这条代码后就跳转至t0地址，t0地址就是 `0x80000000` ，即我们所需要加载的作为 bootloader 的 OpenSBI.bin 的地址。

对这十条汇编代码进行分析，首先，实验手册中显示：

> “在QEMU模拟的riscv计算机里，我们使用QEMU自带的bootloader: OpenSBI固件，那么在 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：即作为 bootloader 的 OpenSBI.bin 被加载到物理内存以物理地址 0x80000000 开头的区域上，同时内核镜像 os.bin 被加载到以物理地址 0x80200000 开头的区域上。”但是具体执行时第一条即将执行的指令地址是0x1000，不是开头的0x80000000物理地址，对此，手册中写到：“QEMU模拟的这款riscv处理器的

> 复位地址是0x1000，而不是0x80000000（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）”

因此在执行时首条指令地址为0x1000.现在解决了PC初始值的问题，分析一下每条指令对应的实现功能：

|***auipc t0, 0x0***|*：这是一个 “Add Upper Immediate to PC” 指令，将立即数 0 加到 PC 的上半部分，然后将结果存储在寄存器 t0 中。

|***addi a1, t0, 32：***|*这是一个 “Add Immediate” 指令，将寄存器 t0 的值与立即数 32 相加，然后将结果存储在寄存器 a1 中。

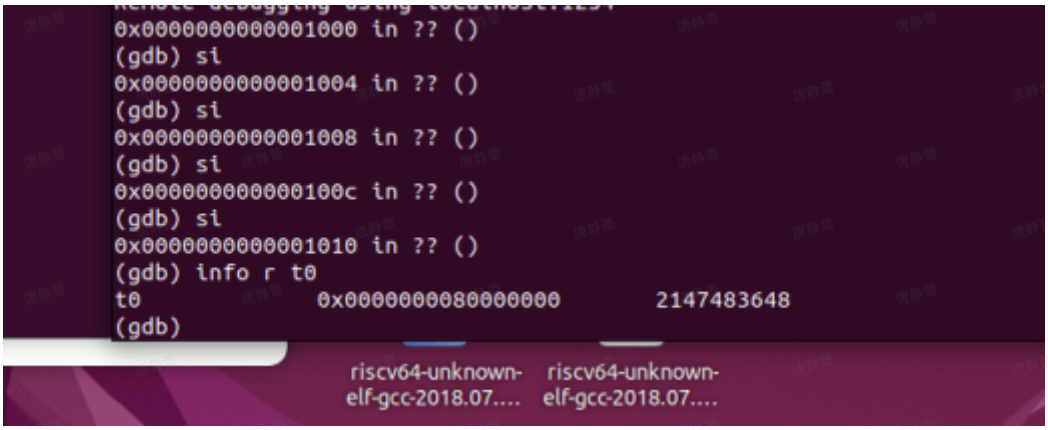
|***csrr a0, mhartid：***|*这是一个 “Control and Status Register Read” 指令，用于从 mhartid 寄存器中读取当前硬件线程 ID，并将结果存储在寄存器 a0 中。

|***ld t0, 24(t0)：***|*这是一个 “Load” 指令，用于从 t0 寄存器的偏移地址为 24 的内存位置中加载数据，并将结果存储在寄存器 t0 中。

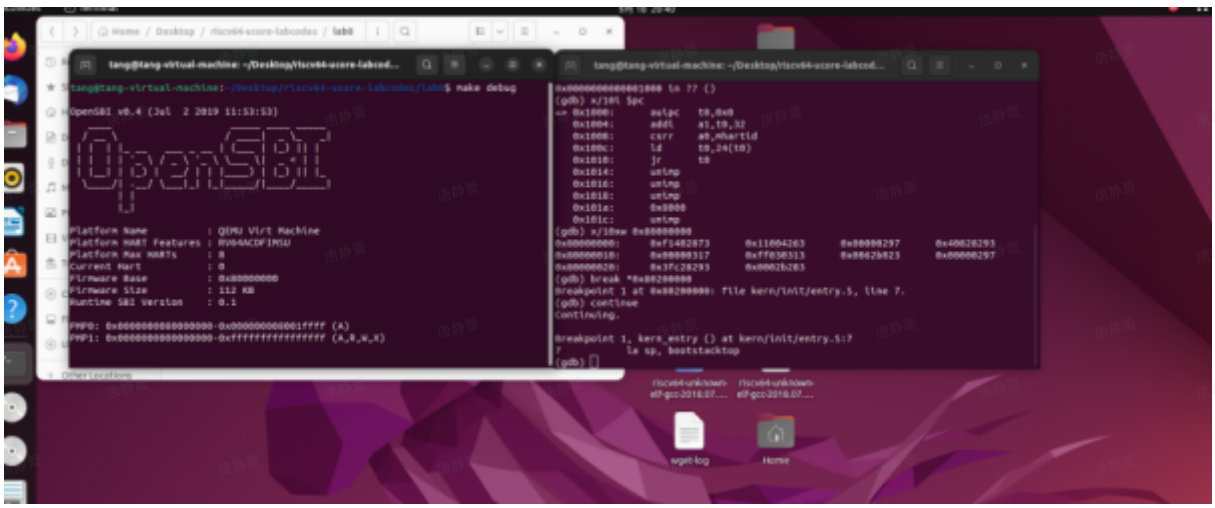
|***jr t0：***|*这是一个 “Jump Register” 指令，用于将程序控制转移到寄存器 t0 中存储的地址。

总体来说，这些指令的功能是将立即数 0 加到 PC 上半部分，然后将结果存储在 `t0` 中，接着将 t0 的值与立即数32 相加，将结果存储在 `a1` 中，读取当前硬件线程 ID 并将结果存储在 a0 中，从 `t0` 的偏移地址为 24 的内存位置加载数据并存储在 `t0` 中，最后将程序控制转移到 `t0` 中存储的地址。

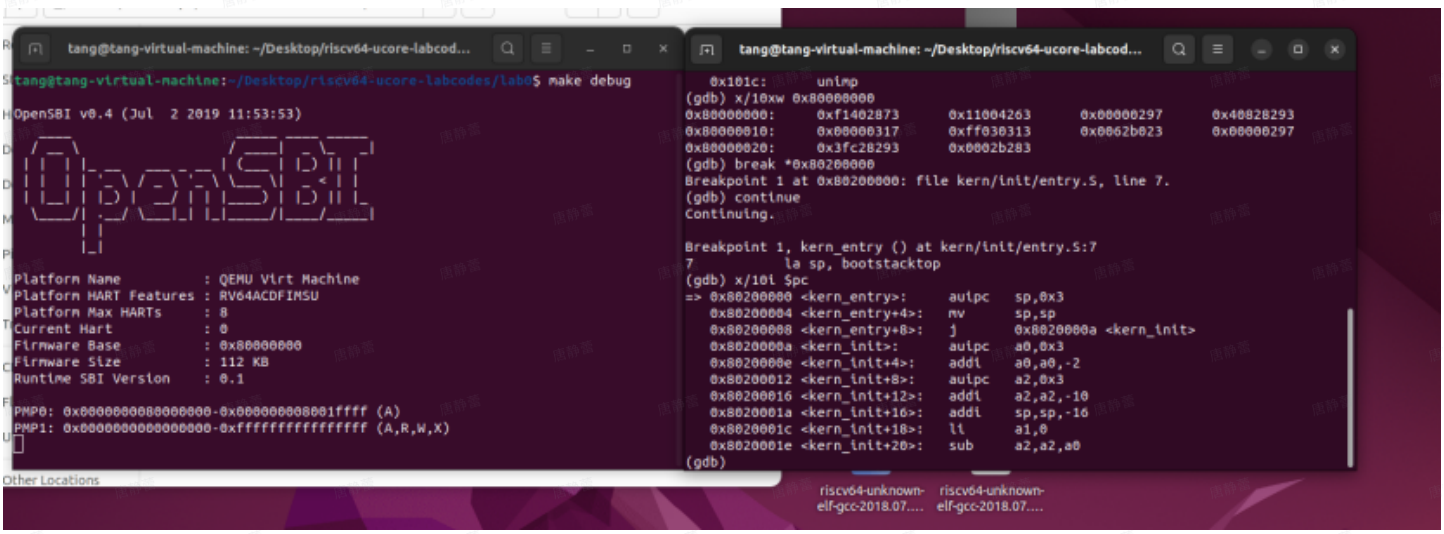
单步执行汇编代码，执行到地址 **`0x1010`** 处后查看t0寄存器中的值，为 **`0x80000000`** ，即我们需要跳转到的地址



在 `0x80200000` 处设置断点后用 `continue` 指令执行，发现作为 `bootloader` 的OpenSBI加载完毕，准备开始执行我们的程序



此时接着查看之后的10条汇编指令，发现已经是我们的程序的汇编代码，至此RISC-V计算机加电开始运行到执行应用程序的步骤都已经结束。



II .Lab1 中断处理机制

一、实验目的

实验1主要讲解的是中断处理机制。操作系统是计算机系统的监管者，必须能对计算机系统状态的突发变化做出反应，这些系统状态可能是程序执行出现异常，或者是突发的外设请求。当计算机系统遇到突发情况时，不得不停止当前的正常工作，应急响应一下，这是需要操作系统来接管，并跳转到对应处理函数进行处理，处理结束后再回到原来的地方继续执行指令。这个过程就是中断处理过程。

二、实验内容

练习1：理解内核启动中的程序入口操作

```
1  la sp, bootstacktop
```

其中指令 `la` 是将某个标签的地址加载到寄存器中；sp是栈指针寄存器，指向栈顶；

这条指令完成了将 `bootstacktop` 返回地址保存在 `sp` 寄存器中的操作。其目的是为了在函数退出时能够正确返回到调用函数的地址。

```
1 tail kern_init
```

此条指令完成了函数调用的操作。它相当于一个尾调用，将控制权直接转移到kern_init函数，并不会返回到调用它的函数。其目的是为了在内核启动时将控制权传递给kern_init函数，以便执行操作系统的初始化。

练习2：完善中断处理 （记得提交源代码）

> 编程完善 `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中的 `shut_down` 函数关机

完善后的 `trap.c` 代码：

```
1 void trap(struct trapframe *tf) {
```

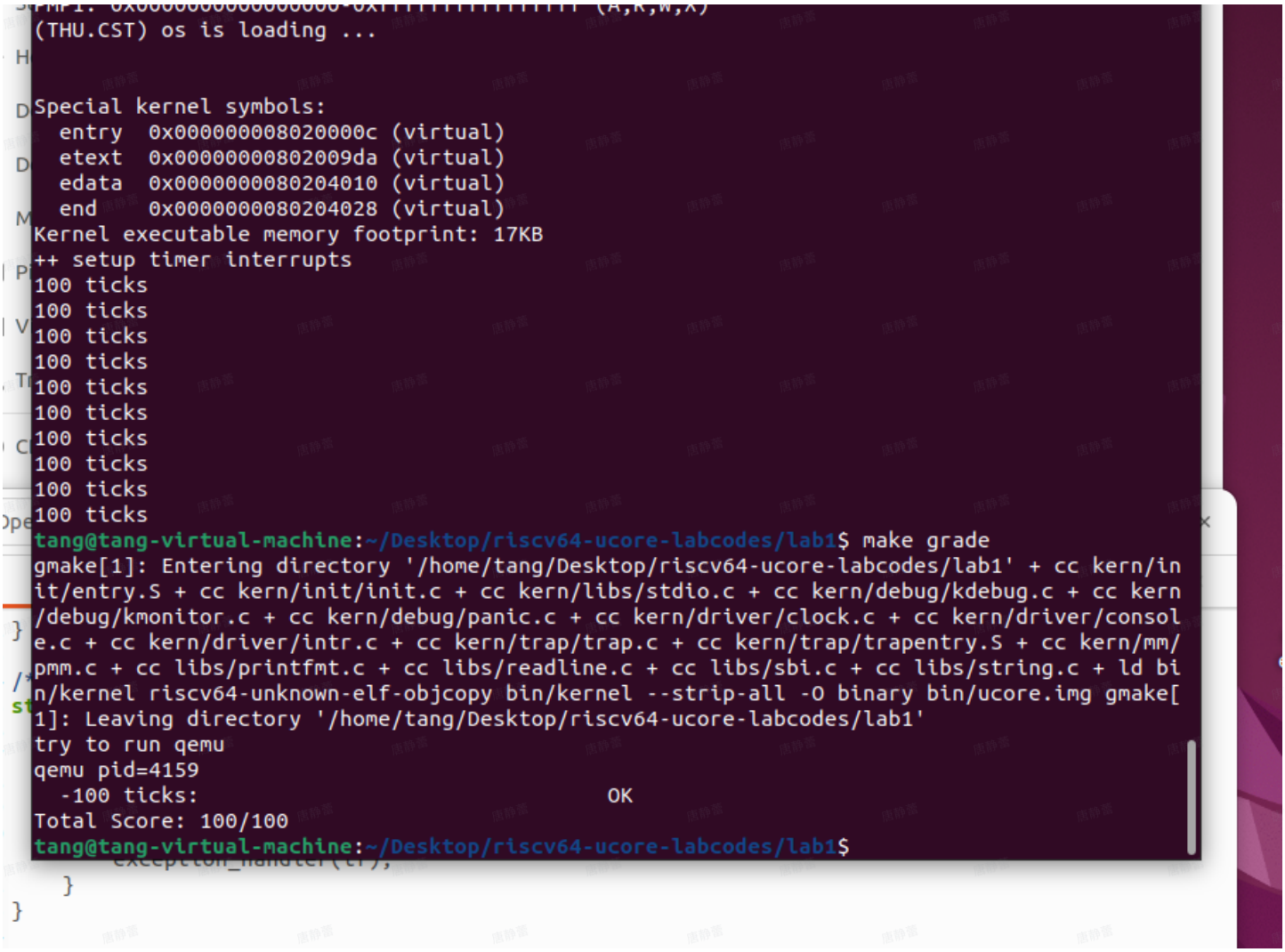
```
trap_dispatch(tf);
ticks++;//时钟终端计次
if(TICK_NUM==ticks){//当计次达到100次时，将计次清零并在屏幕上打印一行“100 ticks”
ticks=0;
print_ticks();
num++;//num是计打印文字次数的变量
}
if(num==10){//当打印了10行后调用sbi.h中的shut_down()函数关机
sbi_shutdown();
}
 }
```

本题练习代码实现看加粗部分：

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_SOFT:
            cprintf("Supervisor software interrupt\n");
            break;
        case IRQ_H_SOFT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_SOFT:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_TIMER:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_TIMER:
            // "All bits besides SSIP and USIP in the sip register are
            // read-only." -- privileged spec1.9.1, 4.1.4, p59
            // In fact, Call sbi_set_timer will clear STIP, or you can clear it
            // directly.
            // cprintf("Supervisor timer interrupt\n");
             /* LAB1 EXERCISE2   YOUR CODE : 2112849  */
            clock_set_next_event();//(1)设置下次时钟中断- clock_set_next_event()
            ticks++; //(2)计数器（ticks）加一
            if(TICK_NUM==ticks){ //(3)当计数器加到100的时候，我们会输出一个`100ticks
            ticks=0;
            print_ticks();
            num++;
```

```
32                 }
33             if(num==10){//(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
34 
35             sbi_shutdown();
36         }
37             break;
38         case IRQ_H_TIMER:
39             cprintf("Hypervisor software interrupt\n");
40             break;
41         case IRQ_M_TIMER:
42             cprintf("Machine software interrupt\n");
43             break;
44         case IRQ_U_EXT:
45             cprintf("User software interrupt\n");
46             break;
47         case IRQ_S_EXT:
48             cprintf("Supervisor external interrupt\n");
49             break;
50         case IRQ_H_EXT:
51             cprintf("Hypervisor software interrupt\n");
52             break;
53         case IRQ_M_EXT:
54             cprintf("Machine software interrupt\n");
55             break;
56         default:
57             print_trapframe(tf);
58             break;
59     }
60 }
61 
62 
```

通过 `make grade` 的测试：



challenge1描述与理解中断流程

> 回答：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？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？请说明理由。

在ucore中，处理中断异常的流程如下几步：

异常的产生：当发生中断或异常时，处理器会中断当前正在执行的指令，保存当前的上下文，并开始执行相应的中断处理程序。

中断向量表：ucore使用中断向量表来确定中断或异常的处理程序入口地址。中断向量表是一个数组，每个元素对应一个中断或异常，并包含相应的处理程序入口地址。

中断处理程序的入口地址：通过中断向量表，根据中断号或异常类型找到相应的处理程序入口地址。

保存寄存器：处理程序开始执行时，会保存当前的上下文，即将寄存器的值保存到栈中。这样做的目的是为了保护当前任务的上下文，以便在处理完中断或异常后能够恢复到之前的状态。

mov a0, sp的目的：这条指令是将栈指针的值保存到a0寄存器中。栈指针指向当前栈帧的栈顶，保存它的值是为了在处理完中断或异常后能够恢复到正确的栈帧。

保存所有寄存器的位置：在ucore中，寄存器的保存位置是由SAVE_ALL宏决定的。SAVE_ALL宏会将寄存器的值保存到栈中，具体的保存位置是根据栈指针的值来确定的。在某些情况下，可能需要在栈中

保存一些临时变量，所以寄存器的保存位置不是固定的。

__alltraps中不一定需要保存所有寄存器：这是因为有些中断并不涉及某些特定寄存器的操作和使用，例如不需要中断处理程序时，或是一些临时寄存器没有必要保存，我们仅需保存必要的某些寄存器就可以完成中断处理操作和恢复。但是值得注意的是，在课堂上也提到了对于操作系统管理者来说，很难确定程序使用者具体对哪个寄存器进行了调用和保存等改变，因此采用保存所有寄存器实现中断异常恢复是最保险的方法，虽然可能浪费资源。

challenge 2 理解上下文切换机制

> 回答：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？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？

A_1

对这两条汇编代码而言：

1. `csrw sscratch, sp`：

   这条指令将当前栈指针 `sp` 的值写入到 `sscratch` 控制状态寄存器（CSR）中。

   目的是将当前的栈指针 `sp` 存储到 `sscratch` 寄存器中，以便在后续的异常处理中保存和恢复栈指针。

2. `csrrw s0, sscratch, x0`：

   这条指令执行了两个操作：首先，它从 `sscratch` CSR 中读取值，并将其存储在寄存器 `s0` 中；然后，它将 `x0` 寄存器的值（通常为零）写入 `sscratch` CSR。

   目的是将 `sscratch` 寄存器的值复制到 `s0` 寄存器中，并将 `sscratch` 寄存器中的值设置为 0。

从源代码的注释中可以得知，将 `sscratch` 寄存器设置为 0，是以便在发生递归异常时，异常向量知道它是来自内核的，从而在对应的部分进行处理。

A_2

从源代码中可以得知，在还原时，`s0`，`s3`，`s4`（对应 `sscratch`，`sbadaddr`，`scause` 三个CSR）没有被还原。

- 上一问的回答中已经提到，`sscratch` 被有意设置成0用以辨识，因此直接储存了它的值而没有恢复。
- `sbadaddr` 寄存器存储了最后一个异常的虚拟地址，即导致异常或故障的虚拟地址。在处理异常后，不需要恢复 `sbadaddr` 寄存器的值，可能因为它通常是只读的，并且异常已经处理完毕。

- 查阅RISC-V手册可以推测，`scause` 寄存器存储了导致异常或中断的原因和类型，因此它仅仅被用于判断，不需要进行还原。

将这三个寄存器的值进行保存，首先仍然有助于异常处理程序获取关于异常的有用信息，以此进行适当的处理和记录。

其次，可能是由于非法指令可以加在任意位置，比如在通过内联汇编加入，也可以直接修改汇编，将这三个值进行另外的保存可能是为了防止恶意代码检测字符串而对它们进行修改或在其他情况下被意外修改，从而影响异常处理程序的跳转与处理。

challenge 3 完善异常中断

> 编程完善在触发一条非法指令异常 mret和，在 kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“Illegal instruction caught at 0x(地址)”，“ebreak caught at 0x（地址）”与“Exception type:Illegal instruction"，“Exception type: breakpoint”。

代码实现：

`trap.c` 部分：

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            break;
        case CAUSE_FAULT_FETCH:
            break;
        case CAUSE_ILLEGAL_INSTRUCTION:
           // 非法指令异常处理
            cprintf("Exception type: Illegal instruction\n");
             cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
            tf->epc += 4;
            /* LAB1 CHALLENGE3   YOUR CODE 2112849 :  */
            /*(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
            */
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
            cprintf("Exception type: breakpoint\n");
            cprintf("ebreak caught at 0x%08x\n", tf->epc);
            /* LAB1 CHALLLENGE3   YOUR CODE  2112849 :  */
            /*(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
            */
            break;
        case CAUSE_MISALIGNED_LOAD:
            break;
        case CAUSE_FAULT_LOAD:
            break;
        case CAUSE_MISALIGNED_STORE:
            break;
        case CAUSE_FAULT_STORE:
            break;
        case CAUSE_USER_ECALL:
            break;
        case CAUSE_SUPERVISOR_ECALL:
            break;
        case CAUSE_HYPERVISOR_ECALL:
            break;
        case CAUSE_MACHINE_ECALL:
            break;
        default:
            print_trapframe(tf);
            break;
    }
}
```

`int.c` 部分：

```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    cons_init();  // init the console

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);

    print_kerninfo();

    // grade_backtrace();

    idt_init();  // init interrupt descriptor table

    // rdtime in mbare mode crashes
    clock_init();  // init clock interrupt
```

```
18
19      intr_enable();  // enable irq interrupt
20
21
22  __asm__ __volatile__("ebreak");//__asm__ __volatile__("ebreak") 是一条内联汇编指令
23  //ebreak 指令是一个特权指令，用于在程序中插入一个断点，以便在调试或测试过程中停止程序执行
24      while (1)
25          ;
26  }
```

观察 `trap.c` 部分的代码可以发现在 `case_ILLEGAL_INSTRUCTION` 部分我们需要将tf->epc寄存器向前移动4个字节以跳过触发异常的指令，继续执行接下来的指令，这样做是为了确保异常处理程序不会再次进入同一个异常点，从而导致无限循环。而在 `case CAUSE_BREAKPOINT` 中，在处理断点异常时不需要将 `tf->epc` 寄存器向前移动，这是由于断点异常是由 `ebreak` 指令触发的，在执行 `ebreak` 指令后，CPU会将下一条指令的地址存储在 `tf->epc` 寄存器中，作为断点异常的返回地址，所以不需要将 `tf->epc` 寄存器向前移动。

测试结果：