Lab 1 中断处理机制

小组成员：李帅东、楚乾靖、于成俊

实验目的

学习并掌握以下几个方面：

- 有关riscv中断的知识
- 中断前后上下文的保存与恢复、
- 处理断点中断和时钟中断

练习

练习1：理解内核启动中的程序入口操作

要求：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` 完成了什么操作，目的是什么？ `tail kern_init` 完成了什么操作，目的是什么？

```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

- `La sp,bootstacktop` 是一条地址加载指令，用于将 `bootstacktop` 符号的地址加载到栈顶指针 `sp` 当中。目的是将 `bootstacktop` 所表示的内存地址当作内核堆栈的顶部，以便在内核执行时正确使用栈内存。
- `tail kern_init` 是一个尾调用（tail call）指令，用于跳转到 `kern_init` 函数的代码块，以完成内核初始化。尾调用可以使得跳转更加高效，因为它避免了额外的函数调用/返回开销。

练习2：完善中断处理 （需要编程）

要求：请编程完善 `trap.c` 中的中断处理函数trap，在 `kern/trap/trap.c` 中处理时钟中断的部分编写代码，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。运行整个系统，大约每1秒会输出一次”100 ticks”，输出10行。

-
  - 我们需要“每隔若干时间就发生一次时钟中断”，但是`OpenSBI`提供的接口一次只能设置一个时钟中断事件。因此需要在每次中断后重新设置下一次的时钟中断，即调用`clock_set_next_event()`函数。否则会一次性输出10行“100 ticks”。
  - 我们需要每遇到100次时钟中断后，调用print_ticks子程序，所以当计数器ticks等于100时，要归零，重新计数。

代码如下：

```
        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 :  */
            /*(1)设置下次时钟中断- clock_set_next_event()
             *(2)计数器（ticks）加一
             *(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
            * (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
            */
            clock_set_next_event();
            ticks++;
            if(ticks==100){
               print_ticks();
               num++;
               ticks=0;
            }
            if(num==10){
               sbi_shutdown();
            }
            break;
```

- 为了验证上述代码的功能，在命令行输入`“make qemu”`，显示结果如下，

```
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
```

扩展练习 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。

- 处理中断流程：
  - 在中断产生后，CPU跳转至中断向量表基址（stvec）中找到相应的中断处理程序，在中断处理之前，先保存上下文（context），即保存所有寄存器到栈顶（实际上把一个trapFrame结构体放到了栈顶），然后处理中断，处理完成后，恢复context。
  - 以时钟中断为例，调用`clock_init()`进行初始化。在该函数中，调用`set_csr()`函数，将`sie`中的时钟中断使能信号打开。之后调用`sbi_set_timer()`函数。在时间到达`timebase`时发生中断。进入终端入口后，先保存现场没在通过`trap.c`执行`trap_dispatch()`函数，之后恢复现场。

```
#define IRQ_S_TIMER 5
/*kern/driver/clock.c*/
#define MIP_STIP (1<<IRQ_S_TIMER)
void clock_init(void) {
    set_csr(sie, MIP_STIP);//将sie置0
    clock_set_next_event();//设定下一个时钟中断
    // initialize time counter 'ticks' to zero
    ticks = 0;
    cprintf("++ setup timer interrupts\n");
}
void clock_set_next_event(void) { sbi_set_timer(get_cycles() + timebase); }
/* kern/trap/trap.c */
//判断是外部中断还是内部中断
static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}
```

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
              ...
        case IRQ_S_TIMER:
            cprintf("时钟中断的处理");
              break;
             ……
     }
}
```

- `mov a0，sp` 的目的：这条指令是函数调用过程中的传参指令，参数为 `trapFrame`，将其入栈，为调用trap（）函数做准备。
- ```
  .macro SAVE_ALL

  csrw sscratch, sp

  addi sp, sp, -36 * REGBYTES
  ```

  从这段代码可以看出，它首先保存原先的栈顶指针到 `sscratch`，然后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。所以位置由 `sp` 和偏移量确定。
- 是否需要保存所有寄存器：需要。
  - 原因一：哪个寄存器是否需要保存不好判断，代码实现难，容易出错。
  - 原因二：寄存器数量不是很多，保存对内存空间要求小，为了安全，全部保存。

扩展练习 Challenge2：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？

- 实现的操作和目的：
  - `csrw sscratch, sp`：保存原先的栈顶指针到 `sscratch`。 `csrw` 是 RISC-V 指令中的一种特权指令，用于在当前特权级别下的导致异常的指令之后设置一个特权寄存器的值。 `sscratch` 是一个特权寄存器，用于保存异常处理期间临时存储的数据。在异常处理过程中，为了保存当前上下文和现场状态，可以将栈指针的值存储到 `sscratch` 寄存器中，以备将来恢复上下文时使用。
  - `csrrw s0, sscratch, x0`：
  - 将 `sscratch` 的值读入 `s0`。因为 `RISCV` 不能直接从 `CSR` 写到内存，所以需要用 `csrrw` 把 `CSR` 读取到通用寄存器，再从通用寄存器 `STORE` 到内存。然后将 `sscratch` 置为0，清空先前存储的数据，确保在异常处理或其他特权级操作中不会使用旧的、无效或不一致的数据。
- 意义：
  - `scause` 用于记录中断发生原因、记录是不是外部中断。
  - `stval` 用于记录处理中断的辅助信息，指令获取(instruction fetch)、访存、缺页异常等，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。
  - 恢复上下文时，不用 `scauce` 和 `stval` 来记录中断的相关信息了，即它们不是上下文，因此不用恢复这些 `csr` 了。
  ```

扩展练习Challenge3：完善异常中断

题目：编程完善在触发一条非法指令异常（如`mret`），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`”。

- 根据题意，编写相应的代码：

```
        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("Illegal instruction caught at 0x%016llx\n", tf->epc);
            tf->epc+=4;
            break;
        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("ebreak caught at 0x%016llx\n", tf->epc);
            tf->epc+=4;
            break;
```

- 为了触发非法指令异常，选择在时钟中断处理部分，通过内联汇编加入非法指令：

```
case IRQ_S_TIMER:
            clock_set_next_event();
            ticks++;
            if(ticks==100){
             print_ticks();
             // 非法指令
             asm volatile (
               "mret"
             );
             num++;
             ticks=0;
            }
            if(num==10){
            sbi_shutdown();
            }
            break;
```

- 验证：

```
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0x0000000080200478
100 ticks
```

查验成绩：

```
ycj@ubuntu:~/riscv64-ucore-labcodes/lab1$ make grade
make[1]: Entering directory '/home/ycj/riscv64-ucore-labcodes/lab1' + cc kern/in
it/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c
+ cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/driver/clock.c + c
c kern/driver/console.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc kern/
trap/trapentry.S + cc kern/mm/pmm.c + cc libs/string.c + cc libs/printfmt.c + cc
 libs/readline.c + cc libs/sbi.c + ld bin/kernel riscv64-unknown-elf-objcopy bin
/kernel --strip-all -O binary bin/ucore.img make[1]: Leaving directory '/home/yc
j/riscv64-ucore-labcodes/lab1'
try to run qemu
qemu pid=3170
  -100 ticks:                                   OK
Total Score: 100/100
```

实验总结

项目构成：

硬件驱动：

- `kern/driver/clock.c(h)`：时钟中断需要必要的硬件支持，该文件调用了`OpenSBI`的`API`接口以读取当
  前时间、设置时钟事件
- `kern/driver/intr.c(h)`：终端需要CPU硬件支持，此处提供了设置中断使能位的接口

初始化：

- `kern/init/init.c`：调用了中断机制的初始化函数

中断处理：

- `kern/trap/trapentry.S`：中断入口点，操作寄存器数据，实现上下文切换
- `kern/trap/trap.c(h)`：上下文切换后的中断处理具体操作函数，判断中断类型、执行对应的`handler`。中断初始化函数也在此处。

执行流：

- 内核初始化函数`kern_init()`执行流：从`kern/init/entry.S`进入->输出初始化提示信息->设置中断向量表（`stvec`）跳转位置为`kern/trap/trapentry.S`中的一个标记->在`kern/driver/clock.c`设置第一个时钟事件、使能时钟中断->设置全局S mode中断使能位->不断触发时钟中断。
- 产生一次始终中断执行流（上述执行流的最后一步）：`set_sbi_timer()`触发中断，跳转到`kern/trap/trapentry.S`中的`__alltraps`标记->保存当前执行流上下文，通过函数调用切换为`kern/trap/trap.c`的中断处理函数`trap()`的上下文并执行之；切换前的上下文作为一个结构体传递给`trap()`作为参数->`kern/trap/trap.c`按照中断类型进行分发（`trap_dispatch()，interrupt_handler()`）->执行时钟中断对应处理语句，累加计数器设置下一次时钟中断->完成处理返回`kern/trap/trapentry.S`->恢复原先的上下文，中断处理结束。

实验收获：

了解了中断分类：

- **异常exception**：正在执行的指令发生错误。如"访问无效地址"，"0作除数"，"缺页"。其中，"缺页"可以恢复，"0作除数"不能恢复。
- **陷入Trap**：主动通过一条指令停下来，跳转到处理函数。如"通过`ecall`的`syscall`"，"通过`ebreak`的`breakpoint`"。
- **外部中断Interrupt**：`CPU`执行过程被外设信号打断，我们必须停下来先对外设进行处理。如：定时器倒计时结束，串口收到数据。这是异步的，CPU并不知道外部中断何时发生，在正常执行代码时，有了中断才去处理。中断处理需要较高权限，中断处理程序在内核态。

`riscv64`的权限模式：

- **Machine mode**:`Risc-v`硬件线程最高权限的模式。**一旦发生异常，就会被移交到M模式处理程序**，所有`Riscv`处理器都必须实现的唯一权限模式
- **S mode**:`Unix`系统中的大多数exception都应该进行 S 模式下的系统调用。M 模式的异常处理程序可以将异
  常重新导向 S 模式，也支持通过**异常委托机制**（Machine Interrupt Delegation,机器中断委托）选择性地将中断和同步异常直接交给 S 模式处理,而完全绕过 M 模式。

寄存器：`riscv`有大量的 控制状态寄存器`CSRs`

- **sstatus** (Supervisor Status Register): 寄存器中的二进制位
  - **SIE**（Supervisor interrupt enable) : 在`RISCV`标准里是2^1 对应的二进制位。数值为0的时候，如果当程序在S态运行，将禁用全部中断。（对于在U态运行的程序，`SIE`这个二进制位的数值没有任何意义）。
  - **UIE**（user interrupt enable)：可以在置零的时候禁止用户态程序产生中断。
- **stvec**（Supervisor Trap Vector Base Address Register）中断向量表基址
  - 作用：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，直接让 **stvec** 直接指向中断处理程序的地址。
  - 最低2位的二进制位
    - 00： 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。
    - 01：更高的`SXLEN-2`个二进制位存储的是中断向量表基址，通过不同的异常原因来索引中断向量表。
    - 用62位表示64位的地址，由于`RISCV`架构要求这个地址是四字节对齐的，所以总是在较高的62位后补两个0。

特权指令

- `ecall`：S态执行时，触发 `ecall-from-s-mode-exception`，进入M模式的中断处理流程。U态时，触发`ecall-from-u-mode-exception`，进入S态处理。
- `sret`：S态中断返回U态，即`pc<-sepc`
- `ebreak`:触发断点进入中断。
- `mret`：用于 M 态中断返回到 S 态或 U 态，实际作用为`pc<-mepc`