Lab3

何潇 王豪杰 杨竣羽

练习一：完善中断处理（需要编程）

由于在打印10行结果后需要调用`sbi.h`中的`shut_down()`关机，所以先引用`sbi.h`头文件

```
#include <sbi.h>
```

接着定义变量print_num来记录一共打印了几次“100 ticks”

```
static int print_num = 0;
```

下面是对时钟中断处理部分的补充代码

```
clock_set_next_event();
ticks++;
if (ticks % TICK_NUM == 0) {
    print_ticks();
    print_num++;     //记录当前打印次数
    if (print_num == 10) {
        sbi_shutdown();  //打印十次后关机
    }
}
```

扩展练习一：描述与理解中断流程

要求：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。

- ucore 中处理中断异常的流程大致分为三个阶段：异常的产生、中断处理程序的执行和恢复现场。(考虑从 U 模式陷入 S 模式。不考虑 M 模式)

1. 异常的产生

- 首先，当异常发生时，CPU会自动将当前的程序计数器（PC）保存到 sepc 寄存器中，将异常原因保存到 scause 寄存器中。对于某些特定异常（如页故障），还会将出错的虚拟地址存入 stval 寄存器。保存到 sepc 寄存器中的值对于异常来说，通常是触发异常的指令地址，而对于中断来说，则是被打断的指令地址。
- 接下来，保存相关的辅助信息。保存并修改中断使能状态。将当前的中断使能状态 sstatus.SIE 保存到 sstatus.SPIE(记录进入中断前的中断使能状态) 中，并且会将 sstatus.SIE 清零，从而禁用 S 模式下的中断。这是为了保证在处理中断时不会被其他中断打断。

- 然后，保存当前的特权级信息。将当前特权级（即 U 模式，值为 0）保存到sstatus.SPP中，并将当前特权级切换到 S 模式。此时，系统已经进入 S 模式，准备跳转到中断处理程序。将pc设置为stvec寄存器中的值，并跳转到中断处理程序的入口__alltraps。

2. 上下文保存及中断处理程序的执行

- 首先我们需要初始化stvec寄存器，也就是stvec直接指向唯一的中断处理程序入口点，所有类型的中断和异常都会跳转到这里。中断处理需要中断当前事务去处理异常，之后再继续执行原来的事务。所以我们需要利用上下文切换机制，保存当前上下文，然后切换到内核态去处理中断。在RISC-V架构中，我们使用 __alltraps 作为中断和异常的入口点，它将保存所有通用寄存器（除了 sp ）和一些特权模式相关的控制、状态寄存器到栈上，通过宏 SAVE_ALL 来实现。然后调用中断处理函数，在中断处理函数中根据异常类型进行相应的处理。最后通过 RESTORE_ALL 恢复寄存器状态，恢复到被打断的地方继续执行。
- 中断处理 trap 函数中定义了两个简单的函数 interrupt_handler() 和 exception_handler()，分别处理中断和异常。在中断发生时，会调用 interrupt_handler() 函数来处理；而在异常发生时，则调用 exception_handler() 函数进行处理。这两个函数会根据不同的异常类型进行相应的处理操作。

3. 恢复现场

- 处理完中断后，需要将之前保存的寄存器恢复到原来的状态，以便程序能够继续执行被打断的地方。这通常是通过从栈中恢复寄存器的值来实现的。程序从 trap() 函数返回后，执行流会进入 __trapret，执行宏 RESTORE_ALL 来恢复寄存器状态。最后通过 sret 指令返回到用户态，继续执行被打断的程序。

关于 mov a0, sp 的目的：

- 在 SAVE_ALL 宏之后，栈指针 sp 指向了内核栈上刚刚构建完成的 trapframe 结构体的起始地址。这个结构体包含了中断发生时 CPU 的全部状态。为什么一定要传给 a0 呢？因为根据 RISC-V 调用的约定，函数的第一个参数需要通过 a0 寄存器传递。通过 mov a0, sp 的操作，将栈顶指针的值（即 trapframe 结构体的地址）传递给 a0 寄存器，后续 trap(tf) 函数就可以调用了。这样做的目的是为了在中断处理函数中能够通过 tf 指针来访问和修改所有被保存的寄存器值。

SAVE_ALL 中寄存器保存在栈中的位置是如何确定的?

- 在 RISC-V 中断处理框架中， SAVE_ALL 宏的本质是按照 struct trapframe 的成员布局，依次将寄存器的值压入栈中。由于C语言里面的结构体，是若干个变量在内存里直线排列。也就是说，一个trapFrame结构体占据36个uintptr_t的空间，里面依次排列通用寄存器x0到x31,然后依次排列4个和中断相关的CSR，SAVE_ALL 宏的汇编代码在执行时，就是按照这个预定义的偏移量，将 sp 加上相应的偏移后，再把寄存器的值存储到那个内存地址。

对于任何中断，__alltraps 中都需要保存所有寄存器吗?

- 是的，对于任何中断或异常情况，__alltraps 中都需要保存所有寄存器。这是因为中断和异常处理程序需要能够恢复到被打断的上下文状态，操作系统无法预先知道程序正在使用哪些寄存器，必须保存所以的寄存器，防止在恢复时出错，虽然少保存一些寄存器能带来一些资源和性能上的优化，但是这会增加中断处理程序的复杂度，并且在某些情况下可能会导致无法正确恢复上下文。因此，为了保证系统的稳定性和可靠性，通常的做法是保存所有寄存器。

扩展练习二：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?

- csr 是控制状态寄存器的缩写，在 RISC-V 架构中，csr 指令用于读写 CSR（Control and Status Register） 寄存器。csrwrw 指令是CSR读写指令，它将源寄存器的内容写入到目标 CSR 中，并返回该CSR的原始值。csrw 则是 crsrw 的简化形式，只写入 CSR 但不返回原始值。
- 先讲 csrrw s0, sscratch, x0; 这条指令将 sscratch 的值先读取到寄存器 s0 中，然后将 x0（即零，x0 是一个硬连线为 0 的寄存器）的值写回到 sscratch 中。由于 x0 的值总是 0，所以这条指令最后会将 sscratch 清零，标记现在处于特权态。目的是为了判断这次陷入是从用户态还是特权态发生的，同时在特权级切换时，保存和恢复 sscratch 的值。
- csrw sscratch, sp; 实现了将 sp 寄存器的值写入 sscratch CSR。sscratch 是 Scratch 寄存器，用于在特权级切换时保存临时数据。在这里，它将当前栈指针（sp）的值保存在 sscratch 中，以便在从 S 模式返回到 U 模式时能够恢复。
- 在 save all 中保存了 stval, scause 这些 csr，而在 restore all 里面却不还原它们的原因是：这些寄存器包含了异常发生时的附加信息（例如触发中断\异常的原因和值），而在处理完中断后通常不需要这些附加信息(可以理解为冗余的信息)，我们只需要恢复中断发生前寄存器的值即可回到原来运行状态，每次中断发生后，我们又会重新设置这些寄存器的值，也就不需要在处理完成后还原它们。

扩展练习Challenge3：完善异常中断

该部分主要可以分为2大部分：

- 完善异常中断处理
- 捕获并处理非法指令（Illegal instruction）与断点（ebreak）异常

可以通过下面几个方面完成：

1、实验目的

- 修改“kern/trap/trap.c”代码中异常处理逻辑，使系统能正确识别两类异常：

异常类型	含义	对应 `scause` 值 (Machine 模式说明)
Illegal Instruction	CPU 执行了无效/未实现的指令	2
Breakpoint	执行了 `ebreak` 指令	3

当捕获到这两种异常时，系统应打印类似如下内容：

```
Illegal instruction caught at 0x80200140
Exception type: Illegal instruction
```

或

```
ebreak caught at 0x80200148
Exception type: breakpoint
```

2、异常（trap）机制回顾

主要内容可以看“实验答辩.md”文档。

3、实验操作

根据实验要求，要在异常处理部分补充上“Illegal instruction”和“breakpoint”两种异常处理类型，并且输出异常指令地址，更新tf->epc寄存器，保存异常发生时的指令地址，若不修改该寄存器，那么返回时会回到原地址，出现死循环。

具体修改如下列所示：

```
case CAUSE_ILLEGAL_INSTRUCTION:
    cprintf("Exception type: Illegal instruction\n");
    cprintf("Illegal instruction at 0x%lx\n", tf->epc); //"0x%lx"表示的是打印一个64位无符号长整数（即地址值），以十六进制格式输出。后面的“tf->epc”表示的是异常发生时，CPU自动保存的寄存器sepc的值。
    tf->epc += 4;  // 跳过这条非法指令，因为在RISC-V里指令宽度为4，所以此处通过+4跳过指令。
    break;

case CAUSE_BREAKPOINT:
    cprintf("Exception type: breakpoint\n");
    cprintf("Breakpoint at 0x%lx\n", tf->epc);
    tf->epc += 4;  // 同样跳过断点指令
    break;
```

因为此时还不能使用标准C库的printf，所以使用内核态的函数cprintf打印类型、地址。