

# 南开大学操作系统实验报告

学号：2110598 姓名：许宸

学号：2113384 姓名：刘新宇

学号：2112487 姓名：刘轩宇

## 实验题目 — Lab 1 中断机制

### 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。

本章将学到：

- riscv 的中断相关知识
- 中断前后如何进行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

### 环境

#### 软件环境

ubuntu22.04, QEMU-4.1.1

### 实验步骤与内容

#### （一）理解内核启动的程序入口操作

```
la sp, bootstacktop
```

该段汇编代码使用了'la'指令，将栈指针sp设置为bootstacktop的地址，初始化操作系统内核的堆栈，内核将使用bootstacktop处的内存位置作为其堆栈空间。

```
tail kern_init
```

tail指令通常用于实现子程序的调用和返回，它的作用是将当前程序的控制流转移到一个子程序。因此该段汇编代码表示了将目前程序的控制流转移到kern\_init子程序中，即初始化操作系统内核中的各种资源和状态的程序。tail(尾调用)是一种特殊的函数调用，在调用函数后不会返回到调用者，而是直接跳转到新函数，而且不会保留调用者的栈帧，故在内核初始化后没有不必要的栈帧被保留下来，节省了空间。

## (二) 完善中断处理

请编程完善 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“100ticks”，在打印完 10 行后调用 sbi.h 中的 shut\_down() 函数关机。

实现代码如下

```
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();// 设置下一次的时钟中断
    ticks++;
    if(ticks % TICK_NUM == 0){
        print_ticks();//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
        num1++;//打印次数+1
        if(num1==10){
            sbi_shutdown();
        }//当打印次数为10时，调用<sbi.h>中的关机函数关机
    }
    break;
```

结果如下:

```

Special kernel symbols:
  entry 0x000000008020000c (virtual)
  etext 0x0000000080200a6a (virtual)
  edata 0x0000000080204010 (virtual)
  end   0x0000000080204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
app@app-virtual-machine:~/riscv64-ucore-labcodes/lab1$ S

```

实验结果符合预期，进行下一步实验。

### (三) 扩展练习 Challenge1: 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。

1. 异常产生：通常操作系统的中断由异常(Exception)或外部中断(Interrupt)产生，他们分别是因为执行一条指令的过程中发生了错误或是CPU的执行过程被外设发来的信号打断。当异常发生时，处理器会根据异常类型跳转到相应的异常处理例程。
2. 异常处理程序跳转：产生中断后，CPU会跳到 `stvec`，在 `trap.c` 文件的初始化函数 `idt_init` 中，我们将 `stvec` 的值设置为 `_alltraps` 的地址，当发生中断或异常时，CPU将跳转到该地址开始执行异常处理流程。
3. `_alltraps` 的具体处理如下：

```

_alltraps:
    SAVE_ALL

    move a0, sp
    jal trap
    # sp should be the same as before "jal trap"

    .globl __trapret

```

首先执行宏 `SAVE_ALL` 进行寄存器状态保存，然后将 `sp` 栈指针寄存器的状态保存到 `a0` 中，再进行寄存器状态的保存后，跳转到异常处理程序的程序 `trap` 中。

4. 在 `trap.c` 中，有 `exception_handler()` 和 `interrupt_handler()` 两个函数分别对异常和中断进行处理。
5. 在 `trap.c` 处理完异常或中断之后，执行 `__trapret`，用于从异常处理函数返回到被中断的代码处。  
`RESTORE_ALL` 是一个宏，用于从栈中恢复寄存器的状态。然后执行 `sret` 恢复到用户态。

以上中断异常的大概流程描述完成。`move a0, sp` 的目的是将当前栈指针（`SP`）保存在 `a0` 寄存器中，这样异常处理函数可以知道进入异常时的栈指针的值。

由`addi sp, sp, -36 * REGBYTES`可以得知，首先要将`sp`的值移动 $-36 * \text{REGBYTES}$  (36个寄存器大小，分别保存除`x2`外的`x0-x31`和`s0-s4`)，即应该由我们所需要保存状态的寄存器的数量和所占用的总空间决定。

不是对于任何中断或异常都需要保存所有寄存器。具体需要保存哪些寄存器取决于异常的性质和异常处理函数的需求。有些异常处理函数可能只需要保存和恢复一部分寄存器，因为它们并不修改或使用所有寄存器。这样可以提高异常处理的效率，因为不必保存和恢复不必要的寄存器。但在某些情况下，特别是在进行上下文切换时，需要保存所有寄存器以确保上下文的完全还原。因此，是否需要保存所有寄存器取决于具体的实现和需求。

#### (四) 扩增练习 Challenge2：理解上下文切换机制

在`trapentry.S` 中汇编代码`csw sscratch, sp; csw s0, sscratch, x0` 实现了什么操作，目的是什么？`saveall`里面保存了`stval` `cause` 这些`csr`，而在`restoreall` 里面却不还原它们？那这样`store`的意义何在呢？

答： `csw sscratch, sp`：这条指令将栈指针 `sp` 的当前值写入到 `sscratch` 寄存器中。`sscratch` 寄存器通常用于保存一个临时的、可用于异常处理的值。在这里，将 `sp` 写入 `sscratch` 的目的是保存当前的栈指针，以便在异常处理过程中恢复。`csw s0, sscratch, x0`：这条指令将 `x0` 寄存器的值写入 `sscratch` 寄存器，并将 `sscratch` 寄存器的旧值读取到 `s0` 寄存器中。这样做的目的是把`sscratch` 寄存器设置为零，并将旧的 `sscratch` 寄存器的值保存在 `s0` 中。通常，在异常处理开始时，会将 `sscratch` 寄存器设置为零，以指示异常处理程序正在内核态执行，然后在处理完成后，可以恢复 `sscratch` 寄存器的旧值，以确保异常处理程序能够正常返回。

至于为什么 `saveall` 保存了一些 CSR（控制和状态寄存器）而 `restoreall` 没有还原它们，这是因为 CSR 的保存和恢复通常取决于异常处理的具体需求。在异常处理的过程中，某些 CSR 可能需要在异常处理过程中保持不变，而其他 CSR 可能需要保存和还原。

#### (五) 扩展练习 Challenge3：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`”。

在`init.c`程序中，在`kern_init()`函数中，进行内核初始化，再进行`idit_init()`即异常表加载后我们添加内置汇编程序分别来触发异常和中断如下：

```
__asm__ volatile("ebreak"); // breakpoint
__asm__ volatile(".word 0x00000000"); // Illegal instruction
```

两段程序用处如下：

```
__asm__ volatile("ebreak");
```

这是一个用于触发断点异常的指令。在RISC-V架构中，`ebreak` 指令用于引发一个断点异常，通常用于调试和异常处理。当处理器执行到这个指令时，它会跳转到相应的异常处理程序，以便进行调试或其他必要的操作。

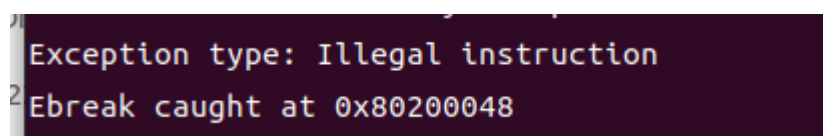
```
__asm__ volatile(".word 0x00000000");
```

这是一个用于生成特定的机器码指令的内联汇编语句。在这种情况下，它生成的是一个机器码值为0的指令，它实际上是一个非法的指令。当处理器执行到这个非法指令时，会触发非法指令异常（Illegal Instruction Exception）。

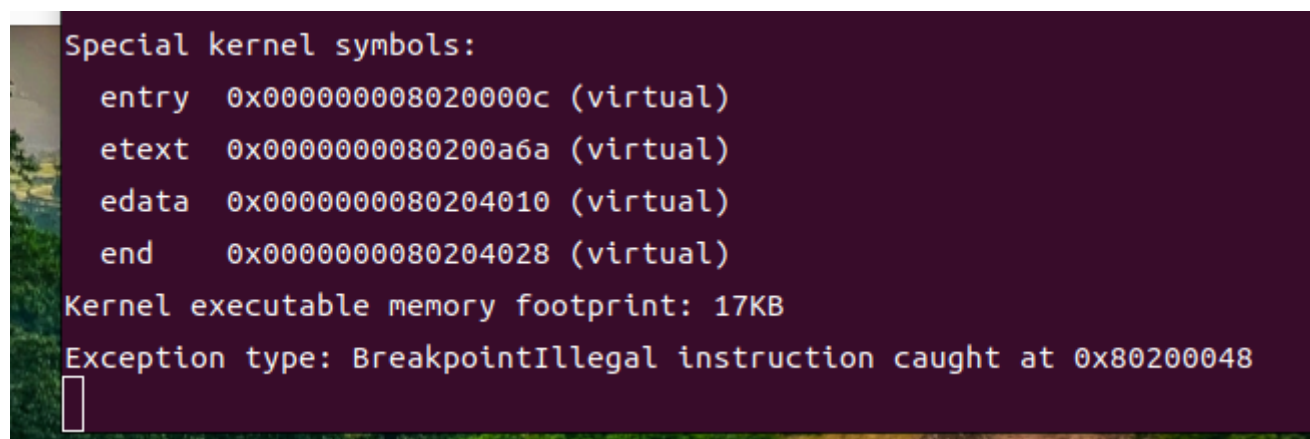
在\kern\trap.c文件修改exception\_handler处理异常函数，代码如下：

```
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\n");
    cprintf("Ebreak caught at 0x%08x\n", tf->epc);
    tf->epc += 4; // 更新 tf->epc寄存器
    break;
case CAUSE_BREAKPOINT:
    //断点异常处理
    /* LAB1 CHALLENGE3 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("Illegal instruction caught at 0x%08x\n", tf->epc);
    tf->epc += 4; // 更新 tf->epc寄存器
    break;
```

结果截图如下：



```
Exception type: Illegal instruction
Ebreak caught at 0x80200048
```



```
Special kernel symbols:
  entry  0x000000008020000c (virtual)
  etext  0x0000000080200a6a (virtual)
  edata  0x0000000080204010 (virtual)
  end    0x0000000080204028 (virtual)
Kernel executable memory footprint: 17KB
Exception type: BreakpointIllegal instruction caught at 0x80200048
```