

操作系统Lab1实验报告

学号：2210620 姓名：何畅

学号：2212117 姓名：胡雨欣

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。

本章你将学到：

- 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

实验内容

1.理解内核启动中的程序入口操作

```
kern_entry:
    la sp, bootstacktop

    tail kern_init
```

`la` 是 "load address" 指令，它将 `bootstacktop` 的地址加载到栈指针寄存器（`sp`）中。

`bootstacktop` 是内核栈的栈顶位置。

`tail` 是汇编中的一个优化指令，它相当于一个无条件跳转，并且在跳转时不保留当前函数的调用栈。此指令会直接跳转到 `kern_init` 函数，并将控制权交给它。

2.完善中断处理

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。

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%TICK_NUM==0){
        print_ticks();
        num++;
    }
    if(num==10){
        sbi_shutdown();
    }
    break;

```

结果如下：

```

Special kernel symbols:
  entry 0x000000008020000a (virtual)
  etext 0x0000000080200a1a (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
hechang@hechang-virtual-machine:/mnt/hgfs/Ubuntu/riscv64-ucore-labcodes/lab1$

```

3.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？

中断异常的流程：

1. 异常产生：异常，指在执行一条指令的过程中发生了错误；外部中断，简称中断，指的是 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. 当异常或中断发生时，CPU会跳到 `stvec`（中断向量表基址），保存CPU的寄存器（上下文）到内存中（栈上）。在 ucore 的 `trap.c` 中，`stvec` 在 `idt_init()` 函数中被设置为 `__alltraps` 的地址，`__alltraps` 会在接下来几问介绍。

3. 之后程序执行 `trap.c`，`trap.c` 的中断处理函数 `trap`，把中断处理、异常处理的工作分发给了 `interrupt_handler()`、`exception_handler()`，这些函数再根据中断或异常的不同类型来处理。
4. 在处理完异常或中断后，会回到 `__alltraps` 向下继续执行 `__trapret`，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。

`__alltraps` 的代码如下：

```
__alltraps:
    SAVE_ALL

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

    .globl __trapret
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

`mov a0, sp` 将当前栈指针（SP）保存在 `a0` 寄存器中，这样 `trap` 函数可以知道进入异常时的栈指针的值，进而访问保存的上下文信息。

`SAVE_ALL` 中，运行了 `addi sp, sp, -36 * REGBYTES`，即寄存器保存在栈中的位置由当前栈指针的位置和寄存器数量、单个寄存器大小共同决定。

对于某些中断，是可以选择不保存所有寄存器的，对于某些简单且无需上下文切换的中断，可以选择不保存所有寄存器。这种优化的依据是中断处理过程不会修改太多寄存器，且处理过程简短，因此可以只保存临时寄存器或部分调用者寄存器。如果中断处理较为复杂，或者涉及嵌套中断和任务切换，则必须保存所有寄存器以确保安全性和一致性。

4.理解上下文切换机制

- 在 `trapentry.S` 中汇编代码 `csw 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？

`csw sscratch, sp`：这条指令的含义是将当前的栈指针 `sp` 写入到 `sscratch` 寄存器中。`sscratch` 是一个在 RISC-V 特权架构中定义的 CSR（控制状态寄存器），主要用于陷入异常或中断时保存任意数据。在这个上下文中，`sscratch` 寄存器暂时用来保存栈指针的值，以便在异常处理过程中恢复。

`csrrw s0, sscratch, x0`：`csrrw` 指令是将 `sscratch` 中的值读到寄存器 `s0` 中，并同时 `x0`（寄存器 `x0` 总是等于 0）写入 `sscratch`。这样做的主要目的是：将 `sscratch` 的内容（即中断或异常发生时的栈指针）读取到 `s0`，保存这个值，以便稍后恢复。并且将 `sscratch` 清零。这样，如果在处理中再次发生异常，处理器就能判断这次陷入是来自内核（因为 `sscratch` 为 0），而不是用户态，从而区分异常的来源。

对于为什么不还原 `csr`，保存这些寄存器是为了让异常处理函数可以访问异常的详细信息（如异常原因和地址），而不还原是因为 `stval` 和 `scause` 是与中断或异常相关的信息，并且在处理中并不需要被恢复。它们在处理完异常后就不再有用，并且这些寄存器的值会在下一次中断或异常发生时被新的值覆盖。因此，保存它们只是为了在处理过程中使用，而不是为了在异常处理结束后恢复。

5.完善异常中断

编程完善在触发一条非法指令异常 `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”。

```
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("Illegal instruction caught at 0x%08x\n",tf->epc);
    tf->epc+=2;//恢复处理，跳过触发异常的指令
    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("Ebreak caught at 0x%08x\n", tf->epc);
    tf->epc+=2;
    break;
```

在进行 `idt_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：

```
// 插入非法指令
__asm__ volatile("unimp"); // 触发非法指令异常

// 插入断点指令
__asm__ volatile("ebreak"); // 触发断点异常
```

`volatile` 是一个关键字，用来告知编译器某个变量的值可能会随时发生变化，并且不能依赖编译器的优化。

`unimp` 指令是一个未定义的指令，表示这是一个非法或未实现的指令，并且这个指令是**2字节**指令，所以 `tf->epc+=2`。在 RISC-V 中，它属于一种保留指令，用于测试或触发异常处理。

`ebreak` 指令用于引发一个断点异常，通常用于调试和异常处理，并且这个指令是**2字节**指令，所以 `tf->epc+=2`。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。

运行结果如图：

```
Special kernel symbols:
  entry  0x0000000008020000a (virtual)
  etext  0x00000000080200a1e (virtual)
  edata  0x00000000080204010 (virtual)
  end    0x00000000080204028 (virtual)
Kernel executable memory footprint: 17KB
Exception type: Illegal instruction
Illegal instruction caught at 0x8020018c
Exception type: Breakpoint
Ebreak caught at 0x8020018e
++ 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
hechang@hechang-virtual-machine:~/桌面/lab1$
```

重要的知识点

- 1.异常是什么，外部中断是什么。
- 2.异常和中断怎么处理，对于可恢复的异常和中断怎么处理。
- 3.对应OS原理的知识点：保存上下文和恢复上下文很像switch_to的实现，即进程切换。
- 4.OS原理中很重要，但在实验中没有对应上的知识点：进程调度。