操作课程实验报告

Lab1：断,都可以断

网络空间安全学院 信息安全-法学双学位班

小组成员：苏浩天 苟辉铭 王崇宇

练习一

- Q：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？
- 1、la sp,bootstacktop：

  功能：将 bootstacktop 对应的地址赋值给 sp 寄存器，目的是初始化栈，为栈分配内存空间。这样做的目的是确保函数的参数变量以及返回地址等相关信息可以正确地使用和恢复。
- 2、tail kern_init：

  功能：尾调用，将执行转到 kern*init 函数，将控制权传递给kern*init 函数，但是由于是尾调用，它不保存当前函数的返回地址，而是结束时跳转到被调用函数起始地址，这样可以在一些情况下避免堆栈的冗余。目的是进入操作系统的入口，跳转到初始化函数，同时避免此次函数调用对 sp 的影响。

练习二

- Q：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”100 ticks”，在打印完10行后调用sbi.h中的shut*down()函数关机。
- 要求完成问题1提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程和定时器中断中断处理的流程。实现要求的部分代码后，运行整个系统，大约每1秒会输出一次”100 ticks”，输出10行。

```c
clock_set_next_event(); // 设置下次时钟中断
ticks++; // 计数器加一
if (ticks % 100 == 0) { // 每100次时钟中断
    print_ticks(); // 调用print_ticks函数
    print_count++; // 打印次数加一
    if (print_count >= 10) { // 打印10次后关机
        sbi_shutdown(); // 调用关机函数
    }
}
```

```

## 实现过程

- 这段代码是一个简单的时钟中断处理程序的片段，主要用来实现每100次时钟中断打印一次信息，并在打印10次后关闭系统。

- `clock_set_next_event();`：
  - 该函数调用会设置下一个时钟中断事件。它通常是在操作系统或嵌入式系统中用于配置定时器，以便在指定的时间间隔后触发中断。
- `ticks++;`：
  - `ticks`是一个计数器变量，用于记录自系统启动以来发生的时钟中断次数。每当中断发生时，该计数器就会增加1。
- `if (ticks % 100 == 0) {`：
  - 这个条件检测`ticks`是否是100的倍数。如果是，意味着已经发生了100次时钟中断。在这种情况下，程序会执行以下的代码块。
- `print_ticks();`：
  - 这个函数调用用于打印当前的时钟计数或其他相关信息。具体内容取决于`print_ticks()`函数的实现。
- `print_count++;`：
  - `print_count`是一个记录打印次数的变量，每当打印一次信息时，这个计数器就会加1。
- `if (print_count >= 10) {`：
  - 这个条件检查`print_count`是否达到了10。如果已打印10次，则进入该条件块。
- `sbi_shutdown();`：
  - 如果满足条件，将调用`sbi_shutdown()`函数来关闭系统。这可能是一个安全的关机程序，确保在关闭系统前执行必要的清理操作。

## 整体流程

- 当系统每次时钟中断发生时，会调用`clock_set_next_event()`来准备下一个中断事件。
- 同时，`ticks`计数器会增加。
- 每当`ticks`成为100的倍数时，会调用`print_ticks()`函数打印信息，并将`print_count`加1。
- 一旦打印次数达到了10次，系统将调用`sbi_shutdown()`函数关闭。
- 定时器中断处理流程：
  - 当发生时钟中断时，会跳转到寄存器`stvec`保存的地址执行指令，即`__alltraps`的位置继续执行。
  - 接着保存所有的寄存器，然后执行`mov a0,sp`将`sp`保存到`a0`中，之后跳转到`trap`函数继续执行。
  - 调用`trap_dispatch`函数，判断异常是中断，跳转到处理函数`interrupt_handler`处继续执行。
  - 根据`cause`的值，跳转到`IRQ_S_TIMER`处继续执行。
- 得到结果如下所示：
```

```
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
sht@sht-virtual-machine:~/Desktop/riscv64-ucore-labcodes/lab1$
```

扩展练习一

- Q：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。
- 1、异常处理步骤：
  - 异常产生后，跳转到寄存器 stvec 保存的地址执行指令。内核初始化时将该寄存器设置为 __alltraps，因此会跳转到 trapentry.S 中的 __alltraps 标签执行，这正是异常处理程序的入口点。
  - 保存所有寄存器，然后执行 mov a0,sp 将 sp 保存到 a0 中，之后跳转到 trap 函数继续执行。
  - 调用 trap_dispatch 函数，判断异常类型，分别跳转到中断或异常处理函数，根据 cause 的值执行相应处理程序。
- 处理结束后，通过RESTORE_ALL恢复全部寄存器的状态，随即通过sret返回用户模式。
- 2、执行 mov a0,sp 的原因：
  - mov a0,sp的意思是将当前的堆栈指针的值赋值给a0寄存器。根据 RISC-V 的函数调用规范，a0~a7 寄存器用于存储函数参数。由于 trap 函数只有一个参数，即指向结构体的指针，因此需要将该结构体的首地址保存在寄存器 a0 中。
- 3、寄存器保存位置的决定：
  - 寄存器保存的位置由结构体 trapframe 和 pushregs 的定义顺序决定，因为这些寄存器后续会作为 trap 函数参数的一部分，他们的顺序决定了偏移量的大小。具体为相对于栈指针，通过计算偏移量乘字节数确定具体位置。
- 4、保存所有寄存器的必要性：
  - 保存所有寄存器是必要的，因为这些寄存器都将用于函数 trap 参数的一部分。如果不保存，函数参数不完整。若修改 trap 参数结构体的定义，部分寄存器（如0寄存器）可以不保存，因为其值永远为 0。不过在一些特殊的异常情况下，部分寄存器的值可以不保存，这样可以提升效率，由于我们的操作系统较为简单，保存所有的寄存器不会产生特别多的开销，所以采取的保存全部寄存器的决策。

扩展练习二

- Q：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？

- 1、目的是使用 s0 表示函数调用前的栈顶位置。将 sscratch 置为0，以便在发生递归异常时，通过检查sscratch的值，异常向量便能得知其来自于内核，有助于在确保异常处理的正确执行，不会丢失关键信息。

```assembly
csrw sscratch, sp      // 将 sp 的值赋值给 sscratch
csrrw s0, sscratch, x0 // 将 sscratch 的值赋给 s0，并将 sscratch 置为0 (x0寄存器的值恒为0)
```

- 2、不还原 CSR 的原因：因为异常已经由 trap 处理过，没有必要再去还原。这些 CSR 包含导致异常或中断的信息，在处理异常或中断时仍可能需要读取，以确定异常原因。将这些状态寄存器作为参数传递给 trap 函数，确保在处理异常时能够保留关键的执行上下文。处理操作结束后，异常已经消失，此时再存储相关信息没有意义。

扩展练习三

- Q: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”。
- 编程实现如下所示：

```c
case CAUSE_ILLEGAL_INSTRUCTION:
    // 处理非法指令异常
    /* LAB1 CHALLENGE3 YOUR CODE: */
    /* (1) 打印异常类型为非法指令 */
    /* (2) 打印触发非法指令的地址 */
    /* (3) 更新异常程序计数器（epc），指向下一条指令 */
    cprintf("Exception type: Illegal instruction\n"); // 输出异常类型
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc); // 输出异常地址
    tf->epc += 4; // 将 epc 更新到下一条指令地址，假设指令长度为4字节
    break;

case CAUSE_BREAKPOINT:
    // 处理断点异常
    /* LAB1 CHALLENGE3 YOUR CODE: */
    /* (1) 打印异常类型为断点 */
    /* (2) 打印触发断点的地址 */
    /* (3) 更新异常程序计数器（epc），指向下一条指令 */
    cprintf("Exception type: breakpoint\n"); // 输出异常类型
    cprintf("ebreak caught at 0x%08x\n", tf->epc); // 输出异常地址
    tf->epc += 2; // 将 epc 更新到下一条指令地址，ebreak 指令占用2字节
    break;
```

- 在`kern_init`函数中，`intr_enable();`之后写入两行

```c
asm("mret");
asm("ebreak");
```

- 得到结果如下所示：

```
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Exception type:Illegal instruction
Illegal instruction caught at 0x8020004e
Exception type: breakpoint
ebreak caught at 0x80200052
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
sht@sht-virtual-machine:~/Desktop/riscv64-ucore-labcodes/lab1$
```

重要的知识点

相关联的：

中断是指在处理器正常执行指令的过程中，硬件或软件向处理器发出的信号，迫使处理器暂停当前执行的指令序列，转而执行相应的中断处理程序。

分类：

- **硬件中断**：由硬件设备发出的中断信号，如键盘、鼠标、硬盘、网络接口等发出的中断。常见的硬件中断包括时钟中断、I/O设备中断等。
- **软件中断**：由程序发出的中断信号，通常通过执行专门的中断指令（如 x86 的 `INT` 指令）来触发。
- **异常**：特殊类型的中断，通常由执行指令过程中出现的错误（如除零错误、非法指令、缺页异常等）引发。

不同类型的中断在操作系统中的作用、触发机制和处理方式都有所不同。

- **触发方式**：硬件中断由外设引发，软件中断由程序主动请求，异常由 CPU 在执行过程中检测到错误而触发。
- **优先级**：异常和不可屏蔽中断的优先级通常最高，硬件中断优先级取决于中断控制器的设定，软件中断的优先级一般较低，但可根据需求调整。

- **处理方式**：硬件中断和软件中断通常需要保存当前程序状态，执行中断处理程序；异常处理更复杂，可能需要错误恢复、资源释放等。

未涉及的：

- **高级调度策略**：课堂介绍了各种调度策略，如短进程优先、最高响应比优先、多级反馈队列等。这些调度算法的实现和对比，尤其是针对不同类型任务（I/O密集型、CPU密集型）进行调度策略优化。异常处理中可以通过不同的调度方式进行优化。

- **内存分配算法与内存碎片整理**：课堂介绍了基本的内存分配方法，如伙伴系统、最佳匹配、最先匹配、最差匹配。这些内存分配策略如何在操作系统中具体实现和优化本实验中没有涉及。同样如何在内存管理中处理碎片、实现碎片整理也是优化的一种方式。

- **分页机制中的细节**：课堂上讲解了分页管理、页表机制、缺页异常的处理流程、快表的使用等。而实验中未涉及这些机制的具体实现。
- **进程间通信机制**：课堂讲解了进程间的调度与管理，实验中未涉及通信问题。