lab0.5和lab1实验报告

小组成员：王昱 王梓丞 孟启轩

使用GDB验证启动流程

1.**系统初始化**

`make gdb`启动。`0x1000`是该系统的复位地址，每次系统都会从复位地址开始运行，以执行相关的初始化和准备工作。

```
0x0000000000001000 in ?? ()
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      0x8000
   0x101c:      unimp
```

输入指令`l`，得到指令执行的源码

```
(gdb) l
1       #include <mmu.h>
2       #include <memlayout.h>
3
4           .section .text,"ax",%progbits
5           .globl kern_entry
6       kern_entry:
7           la sp, bootstacktop     //将地址 bootstacktop 加载到栈指针寄存器 sp
8
9           tail kern_init
10
```

可以看到，进入`entry.s`，OpenSBI启动后进行内核栈的分配,再调用`kern_init` 进行内核初始化。

2.执行到0x1010后跳转到0x80000000，开始执行bootloader

```
(gdb) x/10i $pc
=> 0x80000000:  csrr    a6,mhartid
   0x80000004:  bgtz    a6,0x80000108
   0x80000008:  auipc   t0,0x0
   0x8000000c:  addi    t0,t0,1032
   0x80000010:  auipc   t1,0x0
   0x80000014:  addi    t1,t1,-16
   0x80000018:  sd      t1,0(t0)
   0x8000001c:  auipc   t0,0x0
   0x80000020:  addi    t0,t0,1020
   0x80000024:  ld      t0,0(t0)
(gdb)
```

```
(gdb) si
0x0000000080000004 in ?? ()
(gdb) l
11      .section .data
12          # .align 2^12
13          .align PGSHIFT
14          .global bootstack
15      bootstack:
16          .space KSTACKSIZE
17          .global bootstacktop
18      bootstacktop:
(gdb)
```

3.**真正的入口**kern_init break *0x80200000 在内核初始化函数处打断点。发现将要执行 la sp,bootstacktop 。但此处仍然不是真正的入口点。

```
(gdb) break *0x80200000
Breakpoint 2 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) continue
Continuing.

Breakpoint 2, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop     //将地址 bootstacktop 加载到栈指针寄存器 sp
(gdb)
```

继续单步执行，直到进入init.c中的kern_init函数，才真正进入程序内核。

```
(gdb) si
9           tail kern_init
(gdb) si
kern_init () at kern/init/init.c:8
8           memset(edata, 0, end - edata);
(gdb) l
3       #include <sbi.h>
4       int kern_init(void) __attribute__((noreturn));
5
6       int kern_init(void) {
7           extern char edata[], end[];
8           memset(edata, 0, end - edata);
9
10          const char *message = "(THU.CST) os is loading ...\n";
11          cprintf("%s\n\n", message);
12          while (1)
(gdb)
```

理解内核启动中的程序入口操作

问题 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？ **回答** 在操作系统的启动过程中，尤其是在切换到内核模式之前，通常需要设置一个干净的栈空间以供内核使用。la sp, bootstacktop：la 指令是 Load Address（加载地址）的简写，这里用于将地址 bootstacktop 加载到栈指针寄存器 sp（Stack Pointer）。 这行代码的目的是初始化栈指针 sp 为 bootstacktop。 bootstacktop 是内核启动时为栈（stack）预留的顶端地址，为接下来的内核代码提供一个栈空间，以保证内核代码执行时有足够的空间用于函数调用、参数传递、临时数据存储等。正确初始化栈指针是启动内核的重要步骤。

操作系统在完成基本的硬件设置和栈指针初始化后，会调用内核的初始化函数，完成更复杂的设置，包括内存管理、设备驱动初始化、文件系统挂载等操作。tail kern_init：使用 RISC-V 的 tail 指令，它相当于一个无返回的跳转，直接跳转到 kern_init 函数。与普通的 jal 指令不同，tail 不会保存返回地址（不会将返回地址保存在寄存器中），因此不会增加函数调用栈的深度。 目的是跳转到 kern_init 是为了开始执行内核的初始化代码，kern_init 通常负责内核的进一步初始化，例如设置页表、内存管理器、硬件中断等。这种跳转方式在操作系统启动流程中很常见，因为从引导程序到内核初始化不需要再返回到前面的代码,可以节省栈空间并简化控制流。

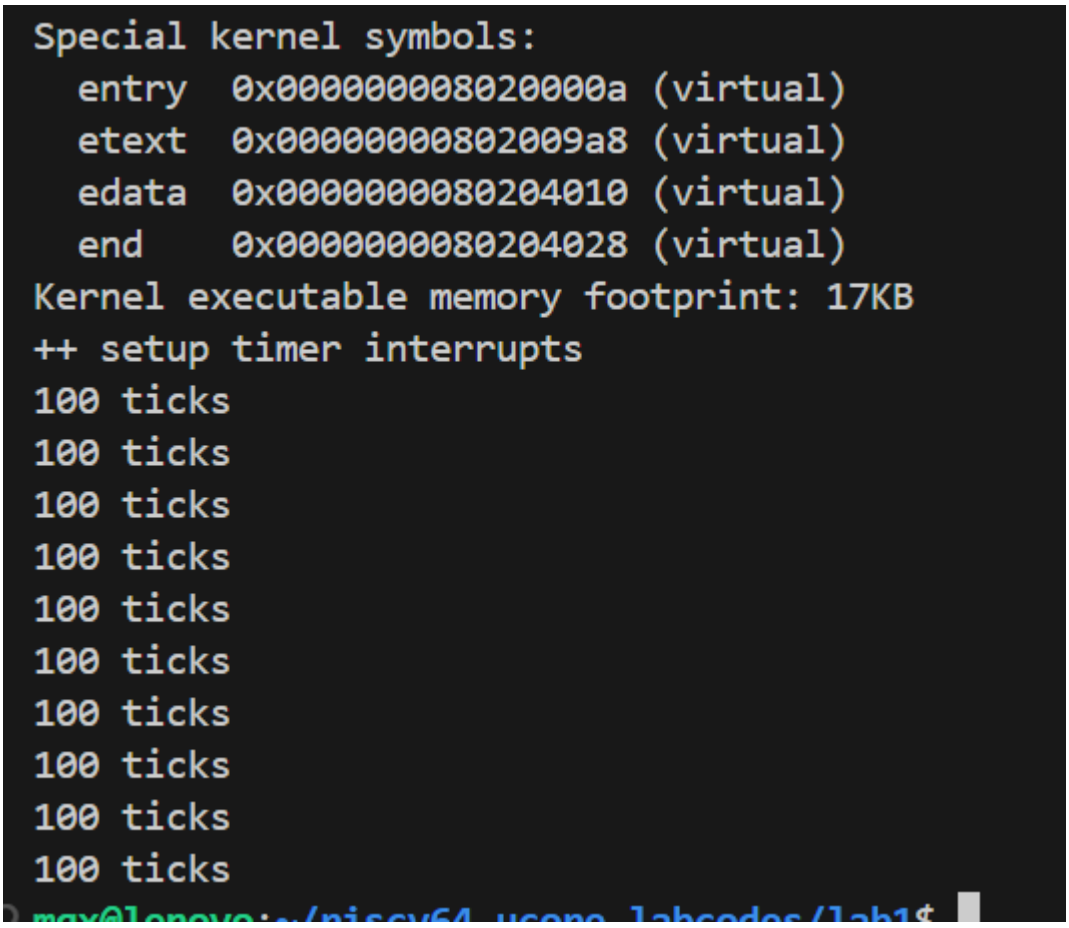
完善中断处理

根据提示信息补全trap()，每100次时钟中断打印一次100 ticks，每次中断后重新设置下一次的时钟中断，即调用clock_set_next_event() 。否则会一次性输出10行 100 ticks 。等ticks到100后，打印次数加1，ticks归零。

```
case IRQ_S_TIMER:
     /* LAB1 EXERCISE2   our 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;
```

可以看到每隔一秒钟打印一次`100 ticks`，打印10次结束。



Challenge 1：描述与理解中断流程

问题

1. 描述 uCore 中处理中断异常的流程（从异常的产生开始）。
2. `mov a0, sp` 的目的是什么?
3. `SAVE_ALL` 中寄存器保存在栈中的位置是什么确定的?
4. 对于任何中断，`__alltraps` 中都需要保存所有寄存器吗？请说明理由。

回答

1. **uCore 中处理中断异常的流程**
   当中断或异常发生时，处理器会将当前的 CPU 状态（包括通用寄存器和部分 CSR 寄存器）保存在栈中，并转移控制权至内核的中断或异常处理程序。对于 RISC-V 架构，处理器首先跳转到 `__alltraps` 汇编代码入口，进入后，先通过 `SAVE_ALL` 宏保存当前的上下文。然后，程序会进入 `trap()` 函数，检查 `scause` 寄存器的值，判断中断或异常类型，调用相应的处理函数。处理完成后，控制流回到 `__alltraps`，通过 `RESTORE_ALL` 宏恢复保存的上下文，最后执行 `sret` 返回用户态继续执行。

2. **`mov a0, sp` 的目的**
   `mov a0, sp` 是将当前栈指针寄存器的值存储到 `a0` 寄存器中。`a0` 是第一个函数参数寄存器，在进入 C 语言的中断处理函数时，内核会将 `trapframe` 传递给处理函数，因此需要将栈指针传递过去。`trapframe` 结构体位于栈上，通过 `sp` 可以访问它。

3. **`SAVE_ALL` 中寄存器保存在栈中的位置**
   每个寄存器保存的位置是根据栈指针 `sp` 和固定的偏移量计算得出。例如，`STORE x1, 1*REGBYTES(sp)` 将寄存器 `x1` 保存到当前栈指针偏移 `1*REGBYTES` 处。每个寄存器的保存位置在栈中的偏移量根据寄存器编号固定，以便处理完中断后能够按照对应的顺序恢复。

4. **`__alltraps` 中是否需要保存所有寄存器?**
   是的，通常情况下所有寄存器都需要保存。中断或异常处理可能需要较长时间处理其他任务，因此为了保证中断返回后程序能够继续正确执行，必须保存所有的寄存器内容，以便在处理完成后恢复现场。

Challenge 2：理解上下文切换机制

问题

1. `csrw sscratch, sp; csrrw s0, sscratch, x0` 实现了什么操作？目的是什么？
2. `SAVE_ALL` 里面保存了 `stval` 和 `scause` 这些 CSR，而在 `RESTORE_ALL` 里面却不还原它们？这样 `STORE` 的意义何在呢？

回答

1. **`csrw sscratch, sp; csrrw s0, sscratch, x0` 实现的操作**

   - `csrw sscratch, sp`: 将当前的栈指针 `sp` 保存到 `sscratch` 寄存器中。`sscratch` 是一个临时寄存器，用于保存 CPU 状态（栈指针）的某些信息。
   - `csrrw s0, sscratch, x0`: 将 `sscratch` 的值写入 `s0` 寄存器（也就是保存的栈指针），并将 `x0` 写入 `sscratch`，清空 `sscratch`。

   这样做的目的是为了安全地保存栈指针，以便在中断处理期间不丢失原始的栈指针。

2. **`stval` 和 `scause` 的 `STORE` 意义**
   保存 `stval` 和 `scause` 是为了记录当前中断或异常的原因信息。这些 CSR 在处理中断或异常时是有用的，特别是在调试和诊断问题时。不过，`stval` 和 `scause` 不需要恢复，因为处理完后它们的值已经无关紧要。保存它们仅是为了在中断处理函数中使用，而不是为了恢复原有值。

Challenge 3：完善异常中断处理

问题 编写代码捕获并处理非法指令和断点异常，输出以下信息：

- "Illegal instruction caught at 0x(地址)"
- "ebreak caught at 0x（地址）"
- "Exception type: Illegal instruction"
- "Exception type: Breakpoint"

代码

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_ILLEGAL_INSTRUCTION:
            // 非法指令异常处理
            /* LAB1 CHALLENGE3 */
            /*(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%016llx\n", tf->epc);
            // %016llx 格式化为16位的十六进制数字
            tf->epc += 4;  // 更新epc寄存器，跳过当前非法指令
            break;

        case CAUSE_BREAKPOINT:
            // 断点异常处理
            /* LAB1 CHALLENGE3 */
            /*(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 \n");
            cprintf("ebreak caught at 0x%016llx\n", tf->epc);
            // ebreak 指令长度为2字节
            tf->epc += 2;  // 更新epc，跳过ebreak指令
            break;

        default:
            print_trapframe(tf);
            break;
    }
}
```

知识点总结

面向Ucore的操作系统上电初步理解

在QEMU中模拟RISC-V计算机硬件时，系统复位向量地址为0x1000，程序计数器（PC）初始化为该地址。OpenSBI（引导加载程序）将操作系统加载到内存中，首先将OpenSBI.bin加载到0x80000000，然后将内核镜像os.bin加载到0x80200000（该地址在kern/init/entry.S中指定），随后操作系统开始执行。

问题：0x80000000地址的来源是什么?

回答：根据汇编代码，指令`jr t0`会跳转到0x80000000，此时t0的值为0x1018，表明复位时的程序决定了这个地址。0x80200000的地址由QEMU的CPU决定。

- **地址无关代码**：可以在主存的任何位置运行，不受特定地址限制的机器代码。
- **地址相关代码**：需要在特定地址运行，不能在其他位置正确执行。

Ucore文件结构

(1) kern

- `init/entry.S`：OpenSBI启动后跳转的汇编代码，作为操作系统的初始代码。
- `init/init.c`：操作系统初始化代码。
- `driver/console.c(h)`：控制台输入输出驱动，依赖sbi.c(h)。
- `libs/stdio.c` & `readline.c` & `printfmt.c`：实现标准输入输出功能。
- `errors.h`：定义内核错误类型的宏。

(2) libs

- `riscv.h`：定义RISC-V指令集的寄存器和指令，支持在C程序中使用内联汇编。
- `sbi.c(h)`：将OpenSBI接口封装为C函数，便于使用。
- `defs.h`：定义常用类型和宏。
- `string.c(h)`：提供字符数组操作函数，类似C中的string.h。

(3) tools

- `kernel.ld`：链接脚本。
- `function.mk`：定义makefile中使用的函数。

(4) makefile

- GNU make编译脚本。

内存布局

ELF文件和bin文件是两种不同的可执行文件格式。bin文件简单地指明加载起始位置，而ELF文件则包含冗余调试信息，指定每个段的内存布局。将合适的ELF文件转换为bin文件后可以加载到QEMU中，从而节省内存。

程序通常包含多个段：

- `.text`：代码段，存放汇编代码。
- `.rodata`：只读数据段。
- `.data`：存放已初始化的可读写数据，通常是全局变量。
- `.bss`：存放初始化为0的可读写数据。

栈（stack）负责函数调用和局部变量，堆（heap）用于动态内存分配。链接器将输入文件的各个段链接在一起，链接脚本描述如何映射输入文件的节到输出文件的节，并规定内存布局。入口点在链接脚本中定义为`kern_entry`，而在`kernel/init/entry.S`中调用`kern_init`作为真正的入口点，该函数初始化内存并输出操作系统正在加载。

权限模式与特权指令

(1) 三种状态/环境

- **User模式**：权限最低，类似于Linux中的用户态。
- **Supervisor模式**：Linux操作系统运行的特权级别，高于User模式。
- **Machine模式**：CPU上电后运行的最高特权级别。

从U到S再到M，权限逐步提升，可以使用更多特权指令和寄存器。

(2) 状态转换指令

使用`ecall`指令进行环境调用，C语言中可通过内联汇编调用。

makefile初步

- **target**：目标文件（对象文件或可执行文件）。
- **prerequisites**：生成目标所需的文件。
- **command**：make需要执行的命令。

Lab 1 知识点总结

硬件驱动

- `kern/driver/clock.c(h)`：实现时钟中断的硬件支持，通过OpenSBI API读取当前时间并设置时钟事件。
- `kern/driver/intr.c(h)`：提供CPU硬件支持的接口，用于设置中断使能位。

初始化

- `kern/init/init.c`：调用中断机制的初始化函数。

中断处理

- `kern/trap/trapentry.S`：中断的入口点，负责操作寄存器以实现上下文切换。
- `kern/trap/trap.c(h)`：处理上下文切换后的具体操作，包括判断中断类型和执行相应的处理程序，同时进行中断初始化。

执行流

内核初始化函数 `kern_init()` 的执行流程如下：

1. 从 `kern/init/entry.S` 进入，输出初始化信息。
2. 设置中断向量表（`stvec`），跳转至 `kern/trap/trapentry.S`。
3. 在 `kern/driver/clock.c` 中设置第一个时钟事件并使能时钟中断。
4. 设置全局S模式的中断使能位，进入持续的时钟中断触发状态。

产生时钟中断的执行流程：

- 调用 `set_sbi_timer()` 触发中断，跳转至 `kern/trap/trapentry.S` 的 `__alltraps` 标记。
- 保存当前上下文，通过函数调用切换到 `kern/trap/trap.c` 的中断处理函数 `trap()`，并将上下文作为参数传递。

- 在 `kern/trap/trap.c` 中，根据中断类型进行分发处理（`trap_dispatch()` 和 `interrupt_handler()`），执行时钟中断对应的处理逻辑，更新计数器并设置下一次时钟中断。
- 处理完成后返回 `kern/trap/trapentry.S`，恢复原上下文，中断处理结束。

中断

- **异常 (Exception)**：执行中的指令发生错误（如无效地址访问、除以零、缺页等）。某些异常（如缺页）可以恢复，其他（如除以零）则无法恢复。
- **陷入 (Trap)**：通过特定指令主动停下来，跳转至处理函数（如 `ecall` 和 `ebreak`）。
- **外部中断 (Interrupt)**：外设信号打断CPU执行，需优先处理（如定时器到期、串口接收数据）。外部中断是异步的，CPU需在正常执行时响应中断。中断处理需要高权限，处理程序在内核态执行。

寄存器

- **sstatus (Supervisor Status Register)**：包含状态信息。
- **SIE (Supervisor Interrupt Enable)**：在S态运行时，数值为0时禁用全部中断。
- **UIE (User Interrupt Enable)**：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，不同种类的中断映射至对应的处理程序。

中断向量表

若有多个中断处理程序，`stvec`指向基址，依赖异常类型索引不同处理程序。地址需四字节对齐，使用62位表示64位地址，低两位总是补零。