

## 练习1

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？

## 内核启动流程

进入lab1文件夹下，执行

```
make debug
```

然后

```
make gdb
```

可以看到如下界面：

```
haipenglai@haipenglai-virtual-machine:~/Desktop/code/OperatingSystem/riscv/riscv64-ucore-labcodes/lab0$ make gdb
riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'

GNU gdb (GDB) 8.0.50.20170724-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...done.
The target architecture is assumed to be riscv:rv64
Remote debugging using localhost:1234
0x000000000001000 in ?? ()
(gdb)
(gdb) x/10i 0x00001000
=> 0x1000:      auipc   t0,0x0
0x1004:      addi    a2,t0,40
0x1008:      csrr    a0,mhartid
0x100c:      ld      a1,32(t0)
0x1010:      ld      t0,24(t0)
0x1014:      jr      t0
0x1018:      unimp
0x101a:      0x8000
```

可以见到，当前pc寄存器指向的是0x1000。因为，QEMU 模拟的 riscv 处理器的复位地址是 0x1000。

然后执行

```
x/10i $pc
```

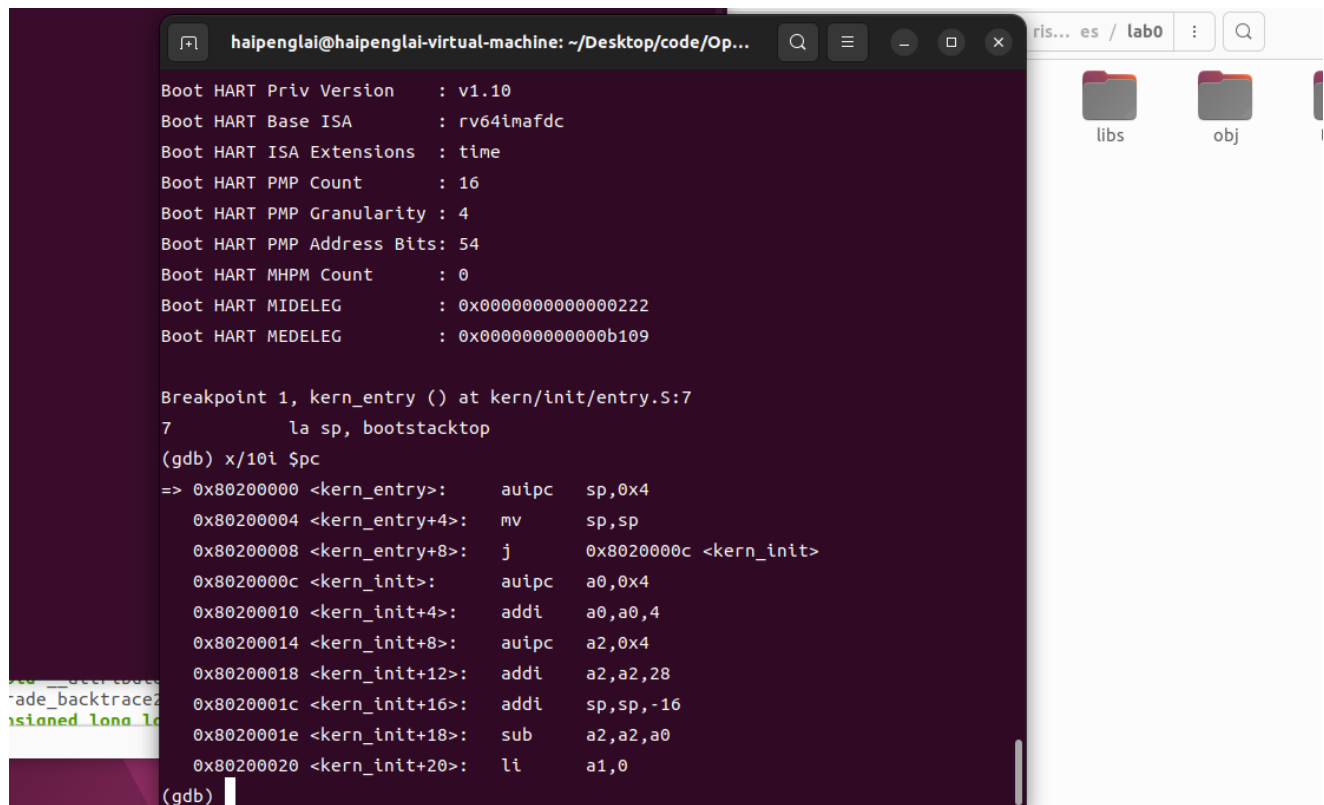
可以看到当前pc所在位置0x00001000所对应的10条指令

它们依次是：



在0x80200000打上断点，可以看到执行到0x80200000的第一条指令(内核的第一条指令)：

la sp,bootstacktop

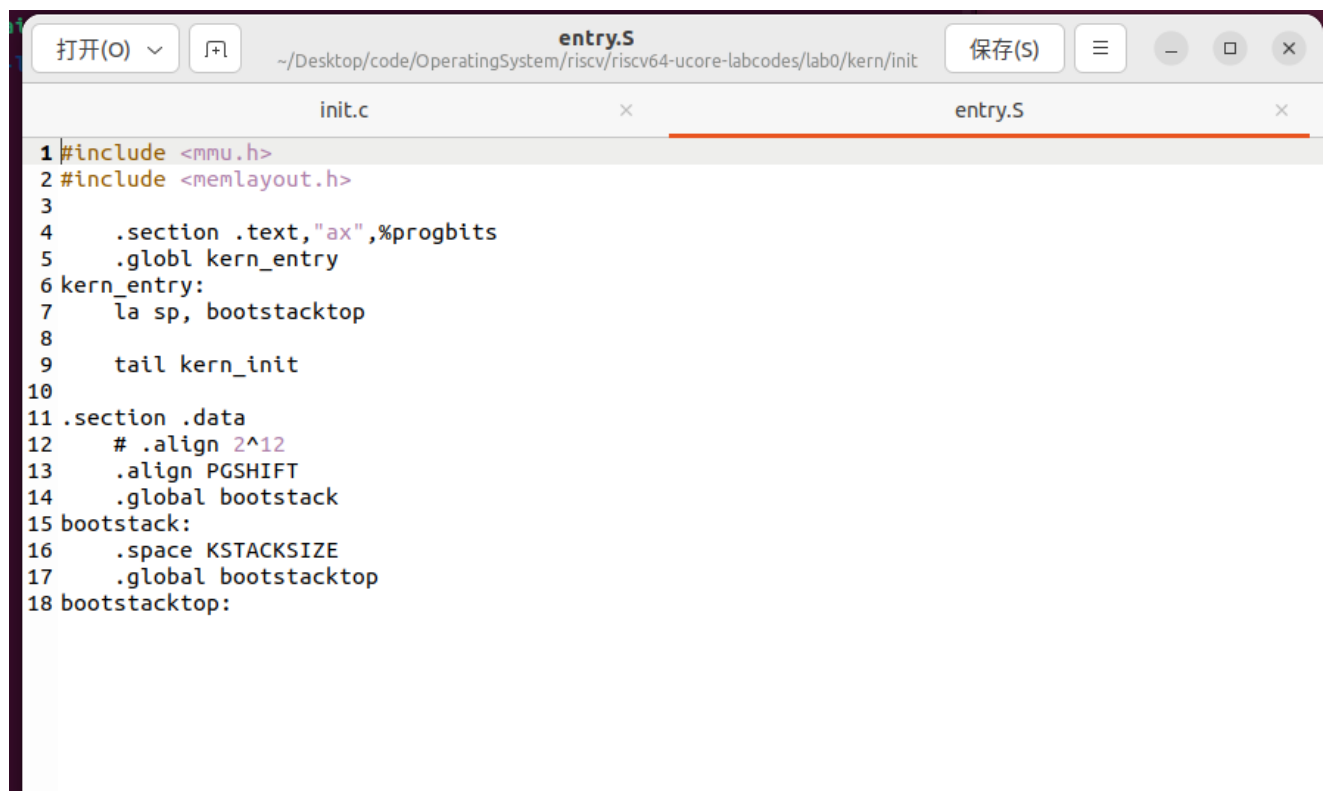


```
haipenglai@haipenglai-virtual-machine: ~/Desktop/code/Op...
Boot HART Priv Version : v1.10
Boot HART Base ISA : rv64imafdc
Boot HART ISA Extensions : time
Boot HART PMP Count : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 0
Boot HART MIDELEG : 0x0000000000000222
Boot HART MEDELEG : 0x000000000000b109

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7      la sp, bootstacktop
(gdb) x/10i $pc
=> 0x80200000 <kern_entry>: auipc sp,0x4
0x80200004 <kern_entry+4>: mv sp,sp
0x80200008 <kern_entry+8>: j 0x8020000c <kern_init>
0x8020000c <kern_init>: auipc a0,0x4
0x80200010 <kern_init+4>: addi a0,a0,4
0x80200014 <kern_init+8>: auipc a2,0x4
0x80200018 <kern_init+12>: addi a2,a2,28
0x8020001c <kern_init+16>: addi sp,sp,-16
0x8020001e <kern_init+18>: sub a2,a2,a0
0x80200020 <kern_init+20>: li a1,0
(gdb)
```

(lab0和lab1这些代码都是一样的)

而这一条指令恰好对应着kern/init/entry.s(内核/初始化/内核入口)当中kern\_entry的指令



```
entry.S
~/Desktop/code/OperatingSystem/riscv/riscv64-ucore-labcodes/lab0/kern/init
init.c x entry.S
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
8
9     tail kern_init
10
11 .section .data
12     # .align 2^12
13     .align PGSIZE
14     .global bootstack
15 bootstack:
16     .space KSTACKSIZE
17     .global bootstacktop
18 bootstacktop:
```

对于代码的解析：

1. kern\_entry: 内核入口点。
2. la sp, bootstacktop: 使用 la (load address) 指令，将 sp (栈指针寄存器) 设置为 bootstacktop 的地址，将栈指针设置为栈顶部。这一步操作初始化了内核的栈，以准备处理函数调用和中断。

3. tail kern\_init: 使用 tail 指令, **跳转到 kern\_init 标签所指位置**。这是内核初始化的入口, 一旦内核的栈准备好, 控制权就传递给了 kern\_init, **开始执行内核的初始化过程**。

而这里的la sp, bootstacktop对应的就是这两句代码:

```
0x80200000 <kern_entry>:    auipc    sp,0x4
0x80200004 <kern_entry+4>:    mv      sp,sp
```

其中第一句话的效果是:  $sp = pc + 4 \ll 12$ , 也就是  $sp = 0x80200000 + 0x4000 = 0x80204000$ , 执行后可以被验证:

```
7          la sp, bootstacktop
(gdb) info register
ra          0x000000008000aa80      2147527296
sp          0x0000000080204000      2149597184
gp          0x0000000000000000      0
```

之后的mv sp,sp没有作用。接下来下一句执行的就是tail kern\_init

```
(gdb) si
9          tail kern_init
(gdb) info register
ra          0x000000008000aa80      2147527296
sp          0x0000000080204000      2149597184
gp          0x0000000000000000      0
```

它对应的代码是:

```
=> 0x80200000 <kern_entry>:    auipc    sp,0x4
    0x80200004 <kern_entry+4>:    mv      sp,sp
    0x80200008 <kern_entry+8>:    j      0x8020000c <kern_init>
    0x8020000c <kern_init>:      auipc    a0,0x4
    0x80200010 <kern_init+4>:    addi     a0,a0,4
    0x80200014 <kern_init+8>:    auipc    a2,0x4
    0x80200018 <kern_init+12>:   addi     a2,a2,28
    0x8020001c <kern_init+16>:   addi     sp,sp,-16
    0x8020001e <kern_init+18>:   sub      a2,a2,a0
    0x80200020 <kern_init+20>:   li       a1,0
```

也就是跳转到0x8020000c,去执行kernal\_init里面的代码, 也就是下面的代码

```
kern_init () at kern/init/init.c:18
18          memset(edata, 0, end - edata);
(gdb) |
```

它恰好对应kern/init/init.c里面的代码

```
5 #include <kdebug.h>
6 #include <kmonitor.h>
7 #include <pmm.h>
8 #include <riscv.h>
9 #include <stdio.h>
10 #include <string.h>
11 #include <trap.h>
12
13 int kern_init(void) __attribute__((noreturn));
14 void grade_backtrace(void);
15
16 int kern_init(void) {
17     extern char edata[], end[];
18     memset(edata, 0, end - edata);
19
20     cons_init(); // init the console
21
22     const char *message = "(THU.CST) os is loading ...\n";
23     cprintf("%s\n\n", message);
24
25     print_kerninfo();
26
27     // grade_backtrace();
28
29     idt_init(); // init interrupt descriptor table
30
31     // rdtm in mbare mode crashes
32     clock_init(); // init clock interrupt
33
34     intr_enable(); // enable irq interrupt
35
36     while (1)
```

## 练习1的答案

因此，我们得出结论,在kern/init/entry.s当中：

(1) la sp, bootstacktop 完成的操作是：

让sp=0x80204000

目的是：

将栈指针设置为栈顶部，初始化了内核的栈，以准备处理函数调用和中断。

(2) tail kern\_init 完成的操作是

让pc跳转到0x8020000c,去执行init.c里面的代码memset(edata, 0, end - edata);

目的是：

控制权就传递给kern\_init，开始执行内核的初始化过程

## 练习2

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。

首先需要定义一个

```
volatile size_t print_time=0;
```

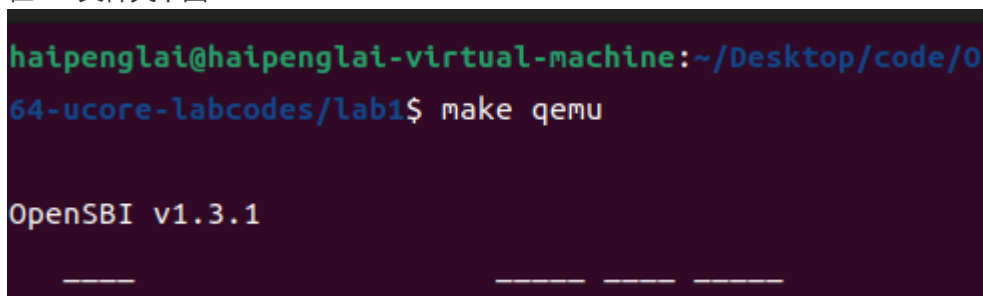
记录打印的次数

然后根据注释写出代码即可：

```
/*(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
*/
//
//下面是我写的代码，根据要求进行了翻译
clock_set_next_event();
num++;
if (num == TICK_NUM) {
    print_ticks();
    num = 0; // 重置计数器
    print_time++; //这里注意是先++后判断关机
    if (print_time == 10) {
        sbi_shutdown(); // 关机
    }
}
break;
```

效果如下

在lab1文件夹下面：



```
haipenglai@haipenglai-virtual-machine:~/Desktop/code/04-ucore-labcodes/lab1$ make qemu

OpenSBI v1.3.1

-----
```

大约每隔一秒输出一行100ticks，总共输出十次

```

    edata 0x0000000080204010 (virtual)
    end   0x0000000080204030 (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

```

### Challenge1: 描述与理解中断流程

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。

而在 `trap.c` 里面，有如下代码：

将 `stvec` 寄存器设为 `__alltraps`

```

30 void idt_init(void) {
31     extern void __alltraps(void);
32     /* Set sscratch register to 0, indicating to exception
       vector that we are
33     * presently executing in the kernel */
34     write_csr(sscratch, 0);
35     /* Set the exception vector address */
36     write_csr(stvec, &__alltraps);
37 }

```

- 出现中断（异常）时，CPU会跳到 `stvec` 寄存器，而 `stvec` 寄存器指向 `alltraps`，`alltraps` 在 `trapentry.S` 内被定义

```

__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

```

效果是保存所有寄存器，跳转到trap，恢复所有寄存器，退出中断处理程序。

而 trap 在 trap.c 内 被 定 义 :

```
void trap(struct trapframe *tf) { trap_dispatch(tf); }
```

又调用了trap\_dispatch

```

/* trap_dispatch - dispatch based on what type of trap occurred */
static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}

```

trap\_dispatch根据类型（中断interruption，异常exception），调用不同的处理函数，对于中断，调用的是interrupt\_handler,里面又根据异常类型分类处理

```

void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_SOFT:
            cprintf("Supervisor software interrupt\n");
            break;
        case IRQ_H_SOFT:
            cprintf("Hypervisor software interrupt\n");
            break;
    }
}

```

对于异常，调用exception\_handler根据异常类型分类处理



```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            break;
        case CAUSE_FAULT_FETCH:
            break;
        case CAUSE_ILLEGAL_INSTRUCTION:

```

## 异常的产生

开机后，会依次执行init/enrty.s,init/init.c

而在init.c里面，会执行clock\_init(),它位于kern/driver/clock.c下面

```
Breakpoint 1, idt_init () at kern/trap/trap.c:34
34      write_csr(sscratch, 0);
(gdb) finish
Run till exit from #0  idt_init () at kern/trap/trap.c:34
kern_init () at kern/init/init.c:32
32      clock_init(); // init clock interrupt
(gdb)
```

```
/* *
 * clock_init - initialize 8253 clock to interrupt 100 times per second,
 * and then enable IRQ_TIMER.
 */
void clock_init(void) {
    // enable timer interrupt in sie
    set_csr(sie, MIP_STIP);
    // divided by 500 when using Spike(2MHz)
    // divided by 100 when using QEMU(10MHz)
    // timebase = sbi_timebase() / 500;
    clock_set_next_event();

    // initialize time counter 'ticks' to zero
    ticks = 0;

    cprintf("++ setup timer interrupts\n");
}
```

首先执行第一句话：set\_csr(sie, MIP\_STIP);其中csr是control status register（状态控制寄存器），STIP - Supervisor Timer Interrupt Pending是计时中断标志位，这句话设置了状态寄存器的计时中断标志位为1，意味着**当计时器产生中断时，内核将能够捕获到这个中断**。

然后执行了clock\_set\_next\_event()

这个函数的定义如下：

```
void clock_set_next_event(void) {
    sbi_set_timer(get_cycles() + timebase);
}
```

调用了kern/libs/sbi.c下面的函数sbi\_set\_timer

```
void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}
```

它的效果是调用sbi\_call

```
//sbi_call函数是我们关注的核心
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n" //mv操作把参数的数值放到寄存器里
        "ecall\n" //参数放好之后，通过ecall，交给OpenSBI来执行
        "mv %[ret_val], x10"
        //OpenSBI按照riscv的calling convention,把返回值放到x10寄存器里
        //我们还需要自己通过内联汇编把返回值拿到我们的变量里
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r"
        ↪ (arg2)
        : "memory"
    );
    return ret_val;
}
```

而这里的ecall(environment)会触发中断。并且触发中断的时间是由get\_cycles() + timebase传入的，相当于timebase这么多个周期后就会触发中断，如果触发中断的时候再次设置clock\_set\_next\_event()，那么每隔timebase这么多个周期后就会反复触发中断。

而timebase=100000=10<sup>5</sup>，而QEMU的时钟周期是10MHz也就是10<sup>7</sup>，也就是0.01s之后就会触发中断，每秒触发100次中断。这一点可以被实验验证，如果把timebase改为10<sup>4</sup>，触发tick的速度将会变成原来的10倍。

总而言之，clock\_init()的效果是：在0.01s之后触发一次中断，后面中断会不断迭代下去。

**问题：**mov a0, sp 的目的是什么？

**为了传递 sp 的值给 trap 函数作为参数。**

sp恰好指向trapframe的起始地址，sp+0对应着zero寄存器，sp+1\*registerSize对应着ra寄存器。

由于void trap(struct trapframe \*tf)需要用到trapframe 的指针，恰好就是sp，所以需要传递 sp 的值给 trap 函数作为参数，而参数传递通过的是a0寄存器，因此需要把sp放到a0里面。

SAVE\_ALL中寄存器保存在栈中的位置是什么确定的？

**是在trapentry.s的 SAVE\_ALL 宏结合trap.h的trapframe确定的，在trapentry.s从低地址到高地址存入了如下寄存器**

```

1 #include <riscv.h>
2
3 .macro SAVE_ALL
4
5     csrw sscratch, sp
6
7     addi sp, sp, -36 * REGBYTES
8     # save x registers
9     STORE x0, 0*REGBYTES(sp)
10    STORE x1, 1*REGBYTES(sp)
11    STORE x3, 3*REGBYTES(sp)
12    STORE x4, 4*REGBYTES(sp)
13    STORE x5, 5*REGBYTES(sp)
14    STORE x6, 6*REGBYTES(sp)
15    STORE x7, 7*REGBYTES(sp)
16    STORE x8, 8*REGBYTES(sp)
17    STORE x9, 9*REGBYTES(sp)
18    STORE x10, 10*REGBYTES(sp)
19    STORE x11, 11*REGBYTES(sp)
20    STORE x12, 12*REGBYTES(sp)
21    STORE x13, 13*REGBYTES(sp)
22    STORE x14, 14*REGBYTES(sp)
23    STORE x15, 15*REGBYTES(sp)
24    STORE x16, 16*REGBYTES(sp)
25    STORE x17, 17*REGBYTES(sp)
26    STORE x18, 18*REGBYTES(sp)
27    STORE x19, 19*REGBYTES(sp)
28    STORE x20, 20*REGBYTES(sp)
29    STORE x21, 21*REGBYTES(sp)
30    STORE x22, 22*REGBYTES(sp)
31    STORE x23, 23*REGBYTES(sp)
32    STORE x24, 24*REGBYTES(sp)
33    STORE x25, 25*REGBYTES(sp)

```

而这些寄存器逐一对应着trapfram的寄存器，例如x0对应着zero寄存器

```

6 struct pushregs {
7     uintptr_t zero; // Hard-wired zero
8     uintptr_t ra; // Return address
9     uintptr_t sp; // Stack pointer
10    uintptr_t gp; // Global pointer
11    uintptr_t tp; // Thread pointer
12    uintptr_t t0; // Temporary
13    uintptr_t t1; // Temporary
14    uintptr_t t2; // Temporary
15    uintptr_t s0; // Saved register/frame pointer
16    uintptr_t s1; // Saved register
17    uintptr_t a0; // Function argument/return value
18    uintptr_t a1; // Function argument/return value
19    uintptr_t a2; // Function argument
20    uintptr_t a3; // Function argument
21    uintptr_t a4; // Function argument
22    uintptr_t a5; // Function argument
23    uintptr_t a6; // Function argument
24    uintptr_t a7; // Function argument
25    uintptr_t s2; // Saved register
26    uintptr_t s3; // Saved register
27    uintptr_t s4; // Saved register
28    uintptr_t s5; // Saved register
29    uintptr_t s6; // Saved register
30    uintptr_t s7; // Saved register
31    uintptr_t s8; // Saved register
32    uintptr_t s9; // Saved register
33    uintptr_t s10; // Saved register
34    uintptr_t s11; // Saved register
35    uintptr_t t3; // Temporary
36    uintptr_t t4; // Temporary
37    uintptr_t t5; // Temporary
38    uintptr_t t6; // Temporary
39 };
40
41 struct trapframe {
42     struct pushregs gpr;
43     uintptr_t status;
44     uintptr_t epc;
45     uintptr_t badvaddr;
46     uintptr_t cause;
47 };
48

```

对于任何中断，\_\_alltraps 中都需要保存所有寄存器吗？

不需要。1. 有时候可以只把需要用到的寄存器存入，2. zero 寄存器是只读的，不需要保存。

## Challenge2

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？

`csrw sscratch, sp` 的操作是将 **sp 寄存器的值写入 sscratch 寄存器**。

`csrrw s0, sscratch, x0` 则是将 `s0` 寄存器设置为 `sscratch` 寄存器的值，**并且让 `sscratch=0`，表示进入到了内核态**。

而 `sscratch` 的值就是 `sp` 的值，这个操作将 **sp 寄存器的值保存在栈上**。【这里很巧妙，`STORE s0, 2*REGBYTES(sp)` 说明把 `sp` 寄存器保存到了栈上对应 `sp` 的值】

trap.c	×	trap.h
5		
6	<b>struct</b>	
7	<code>uintptr_t zero;</code>	<code>// Hard-wired zero</code>
8	<code>uintptr_t ra;</code>	<code>// Return address</code>
9	<code>uintptr_t sp;</code>	<code>// Stack pointer</code>
10	<code>uintptr_t gp;</code>	<code>// Global pointer</code>
11	<code>uintptr_t tp;</code>	<code>// Thread pointer</code>
12	<code>uintptr_t t0;</code>	<code>// Temporary</code>
13	<code>uintptr_t t1;</code>	<code>// Temporary</code>
14	<code>uintptr_t t2;</code>	<code>// Temporary</code>
15	<code>uintptr_t s0;</code>	<code>// Saved register/frame pointer</code>
16	<code>uintptr_t s1;</code>	<code>// Saved register</code>
17	<code>uintptr_t a0;</code>	<code>// Function argument/return value</code>
18	<code>uintptr_t a1;</code>	<code>// Function argument/return value</code>
19	<code>uintptr_t a2;</code>	<code>// Function argument</code>

`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？

`stval scauseglai` 只需要使用一次，用来判断中断和异常类型，后面即使修改了也不需要还原。`store` 的意义就是为了能够使用这一次。

## Challenge 3

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 trap.c 内的 exceptionHandler，写上中断处理程序：**

输出一句话 `Illegal instruction exception written by dyx`

输出异常指令地址

跳过异常指令

```
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
    */
    //下面是我写的代码，根据要求进行了翻译
```

```

    cprintf("Illegal instruction exception written by dyx\n");
    cprintf("Exception program counter (epc): %p\n", tf->epc);
    tf->epc += 4; // 更新 epc 寄存器, 跳过异常指令
    //=====
    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
    */
    //下面是我写的代码, 根据要求进行了翻译
    cprintf("Breakpoint exception\n");
    cprintf("Exception program counter (epc): %p\n", tf->epc);
    tf->epc += 4; // 更新 epc 寄存器, 跳过异常指令
    //=====

```

然后在init.c内触发异常:

trap.c	×	trap.h	×	trapentry.S
<pre> 10 #include &lt;string.h&gt; 11 #include &lt;trap.h&gt; 12 13 int kern_init(void) __attribute__((noreturn)); 14 void grade_backtrace(void); 15 16 int kern_init(void) { 17     extern char edata[], end[]; 18     memset(edata, 0, end - edata); 19 20     cons_init(); // init the console 21 22     const char *message = "(THU.CST) os is loading ...\n"; 23     cprintf("%s\n", message); 24 25     print_kerninfo(); 26 27     // grade_backtrace(); 28 29     idt_init(); // init interrupt descriptor table 30 31     // rdttime in mbare mode crashes 32     clock_init(); // init clock interrupt 33 34     intr_enable(); // enable irq interrupt 35 36     __asm__ __volatile__( 37         "mret"); 38 39     while (1) 40         ; 41 } 42 43 44 void __attribute__((noinline)) 45 grade_backtrace2(unsigned long long ara0, unsigned long long ara1, unsigned long long ara2, unsigned long long ara3) { </pre>				

```

__asm__ __volatile__(
    "mret");

```

之后make qemu,可以看到在第一个时钟到来之前, 就输出了异常处理:

```
Special kernel symbols:
  entry  0x000000008020000c (virtual)
  etext  0x0000000080200a52 (virtual)
  edata  0x0000000080204010 (virtual)
  end    0x0000000080204030 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
Illegal instruction exception written by dyx
Exception program counter (epc): 0x80200050
100 ticks
100 ticks
100 ticks
```

## 知识点总结

### 练习1：内核启动流程

1. 操作系统内核启动流程，从复位地址、到opensbi启动代码、到内核初始化流程。
2. 理解启动代码的初始化操作，如将栈指针设置为栈顶、跳转到内核初始化入口。

### 练习2：处理时钟中断

3. 时钟中断的触发和处理，包括设置中断标志位、计时器设置、中断触发时的处理。
4. 实现每100次时钟中断触发时打印"100 ticks"并在10次打印后关闭操作系统。

### Challenge 1：理解中断流程

5. 中断的处理流程，包括中断触发、中断处理函数的调用。
6. 如何保存和还原寄存器，以及何时需要保存所有寄存器。

### Challenge 2：异常处理

8. 区分中断和异常，了解异常的处理当中栈的操作。

### Challenge 3：异常处理

11. 更深入了解异常处理流程，包括异常触发、异常处理函数调用、异常处理程序的编写。
12. 输出异常信息，如异常类型和异常指令触发地址。