

Lab0.5实验报告

小组成员：钟坤原 邢清画 田晋宇

一、实验目的

实验0.5主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识。

1. 了解内存布局与链接脚本
2. 使用交叉编译生成内核镜像
3. 使用OpenSBI加载内核镜像
4. 使用Qemu模拟器并进行内核调试

二、实验过程

练习1：启动GDB验证启动流程

为了熟悉使用qemu和gdb进行调试工作,使用gdb调试QEMU模拟的RISC-V计算机加电开始运行到执行应用程序的第一条指令（即跳转到0x80200000）这个阶段的执行过程，说明RISC-V硬件加电后的几条指令在哪里？完成了哪些功能？要求在报告中简要写出练习过程和回答。

1.启动GDB并连接到QEMU

通过make debug指令启动QEMU模拟器，随后使用make gdb指令启动GDB并连接到QEMU

我们输入 `make gdb` 来启动gdb调试

```
fzy@ubuntu:~/下载/os/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 (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
Copyright (C) 2020 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:
<https://github.com/sifive/freedom-tools/issues>.
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...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000000100 in ?? ()
(gdb) █
```

可以看到，程序计数器PC首先停留在了0x1000处而不是0x80000000，这是RISC-V计算机上电时，复位造成的上电复位会将系统的各个组件（如处理器、内存和外设）初始化到默认状态。

所谓**复位地址**，指的是CPU在上电的时候，或者按下复位键的时候，PC被赋的初始值。复位代码主要是将计算机系统的各个组件置于初始状态，以确保系统处于可控状态，并且准备好加载和执行操作系统。

这个值的选择会因为厂商的不同而发生变化，例如，80386的复位地址是0xFFF0（因为复位时是16位模式，写成32位时也作0xFFFFF0），MIPS的复位地址是0x00000000。

在QEMU模拟的这款riscv处理器中，将复位向量地址初始化为0x1000，再将PC初始化为该复位地址，因此处理器将从此处开始执行复位代码，复位代码主要是将计算机系统的各个组件（包括处理器、内存、设备等）置于初始状态，并且会启动Bootloader，在这里QEMU的复位代码指定加载Bootloader的位置为0x80000000，Bootloader将加载操作系统内核并启动操作系统的执行。

进入gdb后通过 `x/10i $pc` 查看即将运行的十条指令

```
(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
(gdb)
```

发现硬件加电后的几条指令所在的位置：0x1000、0x10004、0x10008、0x1000c、0x10010。这前五条指令用于初始化处理器的一些配置，读取了mhartid CSR并执行跳转操作，最后跳转到0x80000000启动Bootloader。

下面对这十条指令加上注释

```
0x1000:    auipc    t0,0x0    # 将当前 PC 加上立即数（0），结果存入寄存器 t0。
0x1004:    addi     a1,t0,32    # 将寄存器 t0 的值加上立即数 32，结果存入寄存器 a1。
0x1008:    csrr     a0,mhartid # 读取 mhartid CSR (Machine Hart ID) 的值到寄存器 a0。
0x100c:    ld       t0,24(t0) # 从寄存器 t0 偏移 24 个字节处加载一个双字（64位）到寄存器 t0。
0x1010:    jr       t0          # 跳转到地址0x80000000
0x1014:    unimp                # 未实现指令
0x1016:    unimp
0x1018:    unimp
0x101a:    0x8000                # 直接的立即数
0x101c:    unimp
```

不断单步执行，我们可以发现五步后到达了地址**0x80000000**

```

(gdb) si
0x00000000000001004 in ?? ()
(gdb) si
0x00000000000001008 in ?? ()
(gdb) si
0x0000000000000100c in ?? ()
(gdb) si
0x00000000000001010 in ?? ()
(gdb) si
0x0000000000000000 in ?? ()
(gdb)

```

2. 跳转到OpenSBI，观察内核启动流程

我们可以选择使用两种方式查看0x80000000处的十条命令：

```

x/10i $pc
x/10i 0x80000000

```

可以看到结果是一样的：

```

(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) x/10i 0x80000000
=> 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)

```

我们逐行分析：

0x80000000: csrr a6,mhartid	# 读取mhartid (获取当前硬件线程的ID)的值存到a6中
0x80000004: bgtz a6,0x80000108	# 如果 a6>0, 则跳转到地址0x80000108处执行, 否则继续执行下一条指令。
0x80000008: auipc t0,0x0	# 将当前指令的地址的高 20 位加上一个偏移量 (0x0), 并将结果存储在 t0 寄存器中。
0x8000000c: addi t0,t0,1032	# 将 t0 寄存器的值加上 1032, 并将结果存储在 t0 寄存器中。
0x80000010: auipc t1,0x0	# 将当前指令的地址的高 20 位加上一个偏移量 (0x0), 并将结果存储在 t1 寄存器中。
0x80000014: addi t1,t1,-16	# 将 t1 寄存器的值减去 16, 并将结果存储在 t1 寄存器中。
0x80000018: sd t1,0(t0)	# 将t1的值 (0x80000000) 存储在地址0x80000408处
0x8000001c: auipc t0,0x0	# 将当前指令的地址的高 20 位加上一个偏移量 (0x0), 并将结果存储在 t0 寄存器中。
0x80000020: addi t0,t0,1020	# 将 t0 寄存器的值加上 1020, 并将结果存储在 t0 寄存器中。
0x80000024: ld t0,0(t0)	# 将 t0 寄存器指定地址的内存内容加载到 t0 寄存器。

0x80000000到0x80000024这部分代码是OpenSBI固件启动流程的一部分。这些指令主要完成内核启动前的地址准备工作和多核环境下的调度。代码中的跳转操作会最终引导主核进入实际的操作系统内核加载阶段,即将控制权交给内核启动代码。

3.启动内核镜像

接着输入指令 `break kern_entry`, 在目标函数kern_entry的第一条指令处设置断点, 输出如下:

```
(gdb) break kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
```

为了正确地 and 上一阶段的 OpenSBI 对接, 我们需要保证内核的第一条指令位于物理地址 0x80200000 处, 因为这里的代码是**地址相关的**, 这个地址是由处理器, 即Qemu指定的。地址 0x80200000 由 `kernel.ld` 中定义的 `BASE_ADDRESS` (加载地址) 所决定, 标签 `kern_entry` 是在 `kernel.ld` 中定义的 `ENTRY` (入口点)

输入 `continue` 后, 可以运行到我们打断点的位置

```
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
```

debug输出如下信息

```

Platform Name      : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs  : 8
Current Hart       : 0
Firmware Base      : 0x80000000
Firmware Size      : 112 KB
Runtime SBI Version : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffff (A,R,W,X)

```



```
// kern/init/init.c
#include <stdio.h>
#include <string.h>
//这里include的头文件，并不是c语言的标准库，而是我们自己编写的！

//noreturn 告诉编译器这个函数不会返回
int kern_init(void) __attribute__((noreturn));

int kern_init(void) {
    extern char edata[], end[];
    //这里声明的两个符号，实际上由链接器ld在链接过程中定义，所以加了extern关键字
    memset(edata, 0, end - edata);
    //内核运行的时候并没有c标准库可以使用，memset函数是我们自己在string.h定义的

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message); //cprintf是我们自己定义的格式化输出函数
    while (1)
        ;
}
```

三、练习1回答

1. RISC-V加电后的指令在地址 0x1000 到地址 0x1010。
2. 完成的功能如下：

0x1000:	auipc	t0,0x0	# 将当前 PC 加上立即数 (0)，结果存入寄存器 t0。
0x1004:	addi	a1,t0,32	# 将寄存器 t0 的值加上立即数 32，结果存入寄存器 a1。
0x1008:	csrr	a0,mhartid	# 读取 mhartid CSR (Machine Hart ID) 的值到寄存器 a0。
0x100c:	ld	t0,24(t0)	# 从寄存器 t0 偏移 24 个字节处加载一个双字 (64位) 到寄存器 t0。
0x1010:	jr	t0	# 跳转到地址0x80000000

在QEMU模拟的RISC-V处理器中，复位地址被初始化为0x1000即PC被初始化为0x1000，故处理器加电后从0x1000开始执行复位代码。在此处复位代码将计算机系统初始化后，由于QEMU的复位代码指定加载Bootloader的位置为0x80000000，故将启动Bootloader，之后Bootloader将加载操作系统内核并启动操作系统的执行。这里使用的是QEMU自带的 bootloader: OpenSBI固件。在Qemu开始执行指令之前，作为bootloader的OpenSBI.bin会被加载到物理内存以物理地址0x80000000开头的区域上，内核镜像os.bin被加载到以物理地址0x80200000开头的区域上。

四、本实验重要知识点

1. **RISC-V 复位地址及流程：**RISC-V 处理器在上电时，PC被初始化为复位地址0x1000，程序从此处开始执行，并最终跳转到内核加载地址0x80000000。
复位地址指的是CPU在上电或按下复位键时将PC赋的初始值。复位代码主要是将计算机系统的各个组件置于初始状态，以确保系统处于可控状态，并且准备好加载和执行操作系统。

2. **程序内存布局与链接**：内核的第一条指令必须位于物理地址0x80200000，这通过链接脚本 `kernel.ld` 中定义的 `BASE_ADDRESS` 实现。
3. **OpenSBI接口与内核启动**：OpenSBI提供了与硬件抽象层的接口，内核通过这些接口实现设备的基本管理和初始化。
4. **GDB调试**：通过GDB可以逐步跟踪指令执行的流程，特别是通过设置断点、查看寄存器状态来验证启动流程的正确性。

Lab1 断，都可以断

一、实验目的

实验1 主要讲解的是中断处理机制。通过本章的学习，我们了解了riscv的中断处理机制、相关寄存器与指令。我们知道在中断前后需要恢复上下文环境，用一个名为中断帧（TrapFrame）的结构体存储了要保存的各寄存器，并用了很大篇幅解释如何通过精巧的汇编代码实现上下文环境保存与恢复机制。最终，我们通过处理断点和时钟中断验证了我们正确实现了中断机制。

二、实验过程

练习一：理解内核启动中的程序入口操作

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` 将 `bootstacktop` 这个标签所代表的地址加载给 `sp` 栈顶寄存器，目的是将内核栈的顶部地址设置为 `bootstacktop`，即将栈指针 `sp` 设置为内核栈的顶部，从而实现内存栈的初始化。在操作系统的引导过程中，最初的栈通常是一个非常小的栈，用于执行引导加载程序（`bootloader`）的一些基本操作。`bootstacktop` 就是引导栈的起始地址。
2. `tail kern_init`：尾调用，在函数 `kern_init` 的位置继续执行，目的是进入操作系统的入口，也避免了这一次的函数调用影响 `sp`。使用 "tail" 关键字可以实现函数调用的尾递归优化，即在跳转。该指令通过尾调用的方式跳转执行 `kern_init` 这个函数进行内核的一系列初始化操作。尾调用就是在调用函数后，不会返回到原来的函数调用位置，而是将控制权传递给被调用的函数，使其成为新的执行上下文。这样可以确保在内核初始化过程中，使用的是一个干净的栈空间，避免出现不可预料的错误，比如函数调用时的堆栈溢出。

练习二：完善中断处理（需要编程）

1. 定时器中断处理的函数

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。

```
clock_set_next_event();
ticks++;
// 每100次中断，输出 "100 ticks"
if (ticks % TICK_NUM == 0) {
```



```

        num++;
        cprintf("100 ticks\n");
    }
    // 输出10次 "100 ticks" 后，关机
    if (num == 10) {
        sbi_shutdown(); // 调用 <sbi.h> 中的关机函数
    }
    break;
    /*(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()函数位于/lab/lab1/kern/driver/clock.c文件中，代码为：

```

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

```

2.定时器中断处理流程

注意，我们需要“每隔若干时间就发生一次时钟中断”，但是OpenSBI提供的接口一次只能设置一个时钟中断事件。我们采用的方式是：一开始只设置一个时钟中断，之后每次发生时钟中断的时候，设置下一次的时钟中断。

在clock.c里面初始化时钟并封装一些接口

```

//对64位和32位架构，读取time的方法是不同的
//32位架构下，需要把64位的time寄存器读到两个32位整数里，然后拼起来形成一个64位整数
//64位架构简单的一句rdtime就可以了
//__riscv_xlen是gcc定义的一个宏，可以用来区分是32位还是64位。
static inline uint64_t get_time(void) { //返回当前时间
    #if __riscv_xlen == 64
        uint64_t n;
        __asm__ __volatile__ ("rdtime %0" : "=r"(n));
        return n;
    #else
        uint32_t lo, hi, tmp;
        __asm__ __volatile__(
            "1:\n"
            "rdtimeh %0\n"
            "rdtime %1\n"
            "rdtimeh %2\n"
            "bne %0, %2, 1b"
            : "=&r"(hi), "=&r"(lo), "=&r"(tmp));
        return ((uint64_t)hi << 32) | lo;
    #endif
}

```

```
// Hardcode timebase
static uint64_t timebase = 100000;

void clock_init(void) {
    // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭的
    // 所以我们要在初始化的时候，使能时钟中断
    set_csr(sie, MIP_STIP); // enable timer interrupt in sie
    //设置第一个时钟中断事件
    clock_set_next_event();
    // 初始化一个计数器
    ticks = 0;

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

//设置时钟中断: timer的数值变为当前时间 + timebase 后，触发一次时钟中断
//对于QEMU, timer增加1, 过去了10^-7 s, 也就是100ns
void clock_set_next_event(void) { sbi_set_timer(get_time() + timebase); }
```

然后在trap.c中，每秒100次时钟中断，触发每次时钟中断后设置10ms后触发下一次时钟中断，每触发100次时钟中断（1秒钟）输出一行信息到控制台。

效果如下

```
fzy@ubuntu: ~/下载/OS/riscv64-ucore-labcodes/lab1$
PMPO: 0x0000000080000000-0x000000008001ffff (A)
PMPI: 0x0000000000000000-0xffffffffffff (A,R,W,X)
(THU.CST) os is loading ...

Special kernel symbols:
entry 0x000000008020000a (virtual)
etext 0x0000000080200a28 (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
100 ticks
100 ticks
100 ticks
100 ticks
fzy@ubuntu:~/下载/OS/riscv64-ucore-labcodes/lab1$
```

扩展练习 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。

1. 异常处理的步骤如下：

- 异常产生后，会跳转到寄存器 `stvec` 保存的地址执行指令，由于内核初始化时将该寄存器设置为 `__alltraps`，所以会跳转到 `trapentry.S` 中的 `__alltraps` 标签处执行。
 - 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，判断异常是中断还是异常，分别跳转到对应的处理函数 `interrupt_handler` 或 `expection_handler` 处根据 `cause` 的值执行相应的处理程序。
2. `mov a0, sp` 的目的是将当前栈指针（`sp`）的值存入 `a0` 寄存器。根据 RISC-V 的调用约定，`a0` 是第一个函数参数。当 CPU 跳转到 `trap` 函数时，`a0` 会携带栈指针作为参数传递给 C 语言的 `trap` 函数。`trap` 函数可以通过栈指针来访问保存的上下文（寄存器、状态等）。
 3. `SAVE_ALL` 宏保存寄存器的顺序和位置是固定的，根据寄存器编号和 RISC-V 的寄存器保存规则，寄存器依次被压栈。栈中的位置由以下几个因素确定：
 - **寄存器的顺序**：RISC-V 通用寄存器（如 `ra`, `sp`, `gp`, `tp`, `t0-t6`, `s0-s11`, `a0-a7`）按照一定顺序依次被保存。
 - **栈指针**：保存寄存器的地址由当前的栈指针 `sp` 确定，每保存一个寄存器，`sp` 向下移动，以给下一个寄存器腾出空间。
 4. 需要保存所有的寄存器。因为这些寄存器都将用于函数 `trap` 参数的一部分，如果不保存所有寄存器，函数参数不完整。如果修改 `trap` 的参数结构体的定义，可以不需要存所有的寄存器，比如 `0` 寄存器，因为它永远是 `0`。

扩展练习 Challenge2：理解上下文切换机制

1. `csrrw sscratch, sp; csrrw s0, sscratch, x0` 实现了什么操作，目的是什么？

答：`csrrw sscratch, sp` 指令把原先上文的栈顶指针 `sp` 赋值给 `sscratch` 寄存器。

`csrrw s0, sscratch, x0` 指令先把 `sscratch` 寄存器里存储的上文栈指针的值写入 `s0` 寄存器，用于后续存入内存实现上文的保存；然后把零寄存器 `x0` 复制给 `sscratch`，实现 `sscratch` 的清零，以便后续标识中断前程序处于 `S` 态。

这两句指令实现了上下文的切换，把上文的内容保存在内存之中，然后加载下文，以便安全地执行异常处理，然后在完成后返回原始上下文。这样的做法有助于在处理中断或异常时区分内核态和用户态的执行环境。当 `sscratch` 为 `0` 时，说明处理器当前运行在内核态，这样如果再次发生中断或异常，处理器可以知道它当前处于内核模式。

2. 为什么 `SAVE_ALL` 里保存了 `stval` 和 `scause`，但 `RESTORE_ALL` 不还原它们？`SAVE_ALL` 保存 `stval` 和 `scause` 的意义

1. **`stval` 和 `scause` 用于诊断和调试：**

这些寄存器的值只在中断或异常发生时才有意义，用于分析发生了什么情况以及异常的原因，但它们不会对程序的继续执行产生影响。一旦中断或异常处理完成，这些信息就不再需要。

2. **异常处理时这些寄存器会被覆盖：**

当新的中断或异常发生时，寄存器的值会被新的信息覆盖，因此它们的值是临时的，只对当前处理的异常或中断有意义。恢复原来的程序时，程序不依赖这些寄存器的值。

3. **中断处理的重点是恢复执行状态：**

在中断处理完成后，重点是恢复所有的通用寄存器和控制寄存器，以确保程序能够继续执行。因此，`RESTORE_ALL` 只需恢复执行所需的寄存器，而不需要恢复像 `stval` 和 `scause` 这样仅用于分析的寄存器。

总结来说，不还原那些 `csr`，是因为异常已经由 `trap` 处理过了，没有必要再去还原。它们包含有关导致异常或中断的信息，这些信息在处理异常或中断时可能仍然需要。在异常或中断处理程序中，这些 `csr` 可能需要被读取以确定异常的原因或其他相关信息。这样的意义是将这些状态寄存器作为参数的一部分传递给 `trap` 函数，确保在处理异常或中断时能够保留关键的执行上下文，以供进一步处理或记录异常信息。这种方式允许更灵活地处理异常和中断情况。`stval` 和 `scause` 主要用于异常的分析和诊断，它们在恢复程序的正常运行时没有必要被恢复，因为它们仅提供调试信息。

扩展练习 Challenge3：完善异常中断

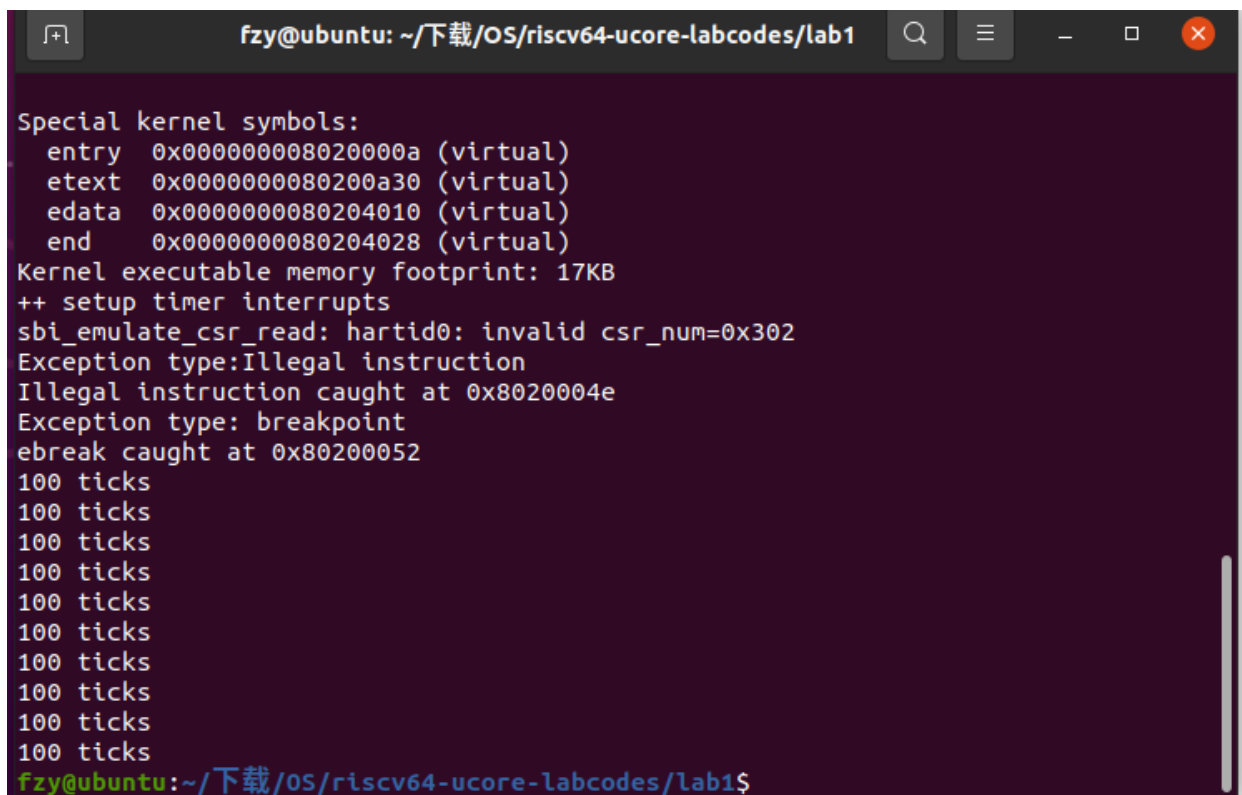
编程实现如下：

```
case CAUSE_ILLEGAL_INSTRUCTION:
    cprintf("Exception type:Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
    tf->epc += 4;
    break;
case CAUSE_BREAKPOINT:
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);
    tf->epc += 2;
    break;
```

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

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

运行后得到结果：



```
fzy@ubuntu: ~/下载/OS/riscv64-ucore-labcodes/lab1
Special kernel symbols:
entry 0x000000008020000a (virtual)
etext 0x0000000080200a30 (virtual)
edata 0x0000000080204010 (virtual)
end 0x0000000080204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
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
fzy@ubuntu:~/下载/OS/riscv64-ucore-labcodes/lab1$
```

三、知识点

1. 内核启动操作

`la sp, bootstacktop`：将内核栈的顶部地址加载到 `sp`（栈指针寄存器），用于初始化内核栈，确保内核启动过程中栈的正确性。

`tail kern_init`：通过尾调用方式跳转到 `kern_init` 函数，进行内核初始化。这种优化方式避免了不必要的函数返回，保持栈的干净状态。

2. 定时器中断处理

在 `trap.c` 文件中编写代码处理时钟中断，每 100 次中断输出 "100 ticks"，并在输出 10 行后通过 `sbi_shutdown` 关机。时钟中断通过 `clock_set_next_event` 函数设置，确保每次中断发生后重新设置下一个中断事件。

3. 上下文切换与 TrapFrame

上下文切换是通过汇编代码 `SAVE_ALL` 宏保存寄存器状态到 `TrapFrame`，并通过 `RESTORE_ALL` 宏恢复寄存器状态。这种机制确保在中断处理前后，任务的状态能被正确保存和恢复。

通过 `csrw sscratch, sp` 和 `csrrw s0, sscratch, x0` 实现了上下文切换的过程，将栈指针保存到 `sscratch`，确保后续的中断处理能够安全进行。

4. 中断向量表与异常处理

在异常处理过程中，程序会根据 `cause` 的值跳转到相应的中断或异常处理程序，并通过 `stval` 和 `scause` 了解导致异常的原因。这些寄存器的状态不需要在中断恢复时还原，因为它们只在异常处理中有用。

5. 扩展练习：处理非法指令和断点**

实现了对非法指令和断点的异常处理，通过捕获相应的异常类型并打印异常信息。同时，调整 `epc`（程序计数器）以确保程序继续执行或进入异常处理后的适当状态。

四、最终检测

我们 `make grade` 后得到结果

