浙江大学实验报告

Lab：`Rinux`内核引导

课程名称： 操作系统 实验类型： 综合

实验项目名称： `Rinux`内核引导

学生姓名： 汪辉 专业： 计算机科学与技术 学号： 3190105609

同组学生姓名： 个人实验 指导老师： 季江民

电子邮件：3190105609

实验地点： 曹西503 实验日期： 2021 年 10 月 21 日

1 实验内容

完善代码

本实验需要完善以下文件：

1. `arch/riscv/kernel/head.S`
2. `lib/Makefile`
3. `arch/riscv/kernel/sbi.c`
4. `lib/print.c`
5. `arch/riscv/include/defs.h`

编写`head.S`

`head.S`完成以下内容即可：

为即将运行的第一个 C 函数设置程序栈（栈的大小可以设置为4KB），并将该栈放置在`.bss.stack`段。
通过RISC-V跳转指令，跳转至 `main.c` 中的 `start_kernel` 函数即可。

```
.extern start_kernel
    .section .text.entry
    .globl _start
_start:
    # ------------------
    la sp,boot_stack_top #加载
    call start_kernel#跳转
    # - your code here -
    # ------------------
    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 0x1000 # <-- change to your stack size
    .globl boot_stack_top
boot_stack_top:
```

编写Makefile

对比同级别目录init下的Makefile文件可以容易地完成lib的Makefile：

```
C_SRC       = $(sort $(wildcard *.c))
OBJ         = $(patsubst %.c,%.o,$(C_SRC))

all:$(OBJ)

%.o:%.c
    ${GCC} ${CFLAG} -c $<
clean:
    $(shell rm *.o 2>/dev/null)
```

编写`sbi.c`

学习并完成内联汇编的C文件`sbi.c`，主要为完成文件内`sbi_ecall`函数的功能，即通过标准输出打印出字符。

```
struct sbiret sbi_ecall(int ext, int fid, uint64 arg0, uint64 arg1, uint64 arg2,
                        uint64 arg3, uint64 arg4, uint64 arg5)
{
    struct sbiret ret;
    __asm__ volatile(
        "mv t4, %[arg4]\n"
        "mv t5, %[arg5]\n"
        "mv a7, %[ext]\n"
        "mv a6, %[fid]\n"
        "mv a0, %[arg0]\n"
        "mv a1, %[arg1]\n"
        "mv a2, %[arg2]\n"
        "mv a3, %[arg3]\n"
        "mv a4, t4\n"
        "mv a5, t5\n"
        "ecall\n"
        "mv %[ret_error], a0\n"
        "mv %[ret_value], a1"
        : [ret_error] "=r" (ret.error) , [ret_value] "=r" (ret.value)
        : [ext] "r" (ext), [fid] "r" (fid), [arg0] "r" (arg0), [arg1] "r"
(arg1), [arg2] "r" (arg2), [arg3] "r" (arg3), [arg4] "r" (arg4), [arg5] "r"
(arg5)
        : "memory"
    ) ;
    return ret ;
}
```

注意由于传入`sbi_ecall`的参数在汇编代码中默认从寄存器`a0`往后存放，因此如果修改了`a7`寄存器的内容`mv a7,%[ext]`之后原本寄存器`a7`的内容`%[arg5]`就会被丢失，再做赋值`mv a5, %[arg5]`时`a5`得到的将是实际上的`%[ext]`。这在传参的逻辑上是不正确的。应用交换寄存器值的逻辑，采用临时寄存器`t4`、`t5`以完成两个寄存器的临时存放是合理的解决方法。

编写`print.c`

`print.c`需要实现两个函数`puts`和`puti`，通过调用`sbi_ecall`来实现单个字符的输出：

```
void puts(char *s) {
    // unimplemented
    for (int i = 0; s[i] ; i++)
    {
        sbi_ecall(0x1,0x0, s[i], 0, 0,0,0,0) ;
        /* code */
    }
}
void recur(int x) {
    if (x>=10)
    {
        recur ( x/10 ) ;
        /* code */
    }
    x = x % 10 ;
    sbi_ecall(0x1,0x0,x+'0',0,0,0,0, 0) ;
}
void puti(int x) {
    // unimplemented
    recur(x) ;
}
```

对于字符串顺序打印内容，对于数字通过递归来逐一打印。

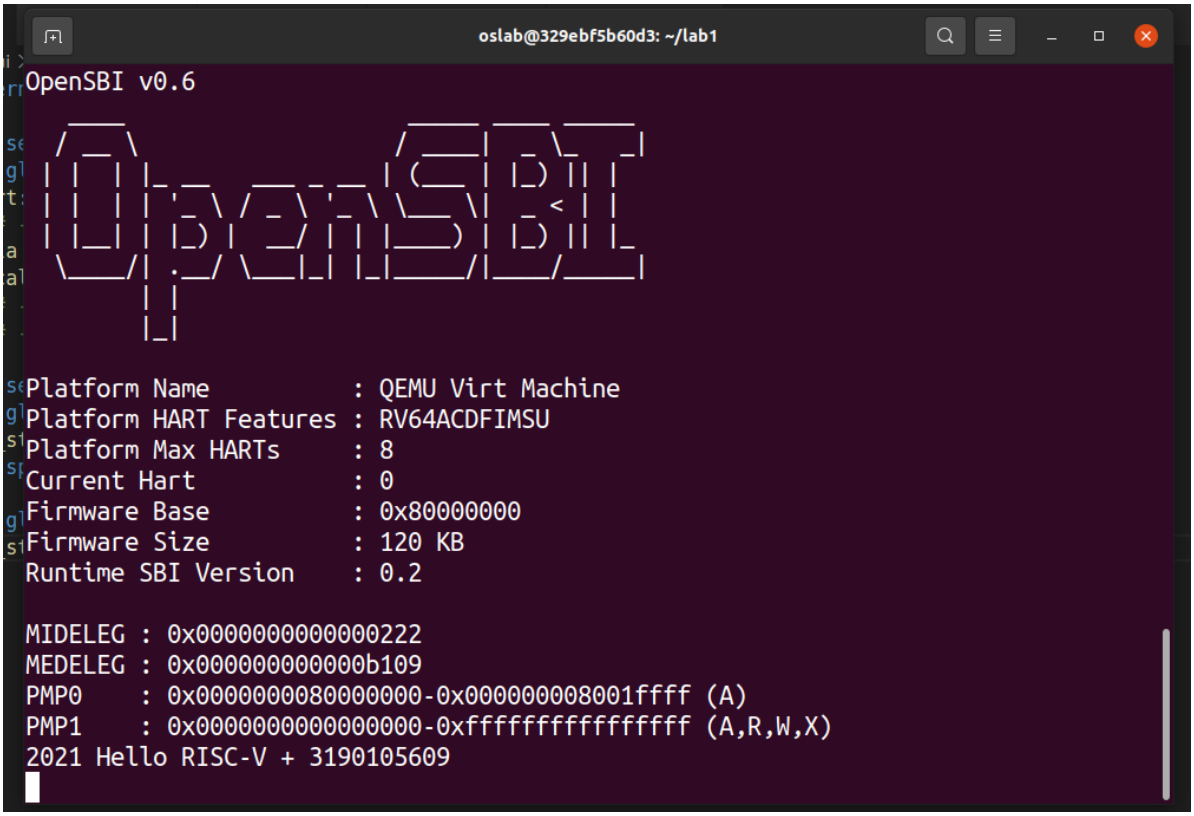
编写`defs.h`

参照宏`csr_write`，完成`csr_read`的宏定义：

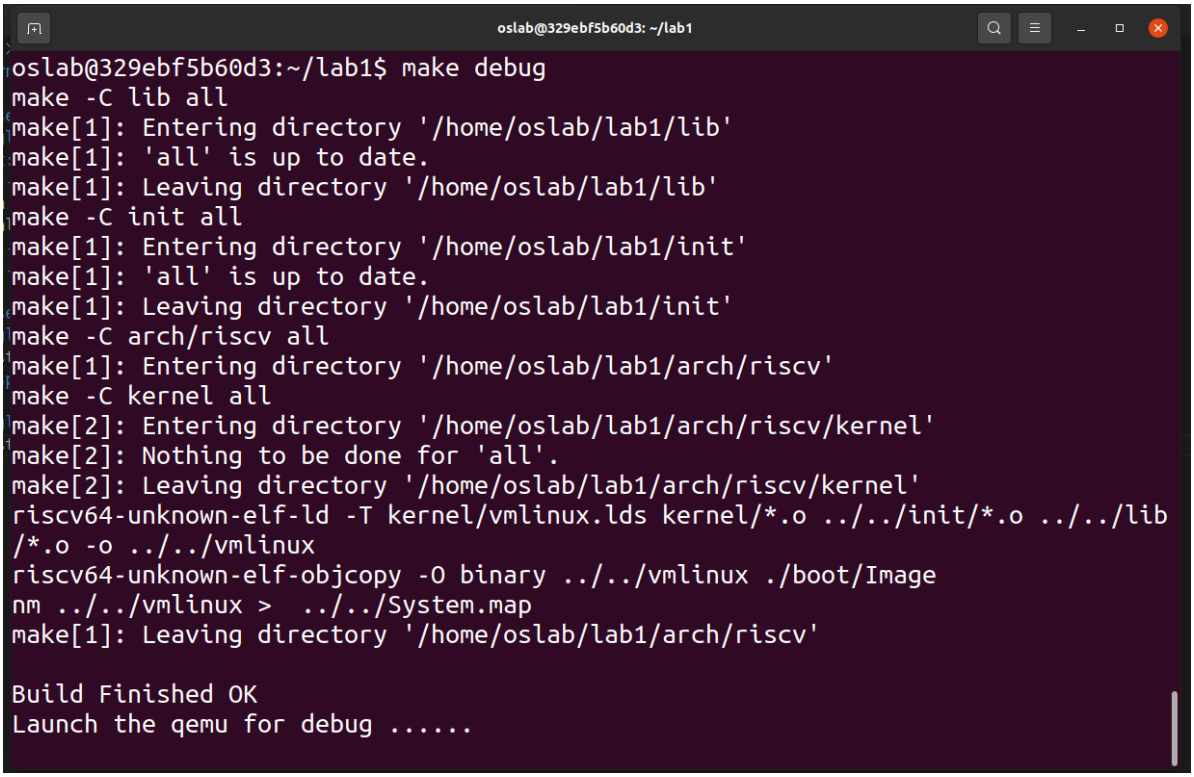
```
#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    /* unimplemented */                     \
    __asm__ volatile ("csrr" ",%0" #csr : "=r" (__v): :"memory" ) ;\
    __v;                                    \
})
```

实验结果

`make run`打印指定内容以及个人学号

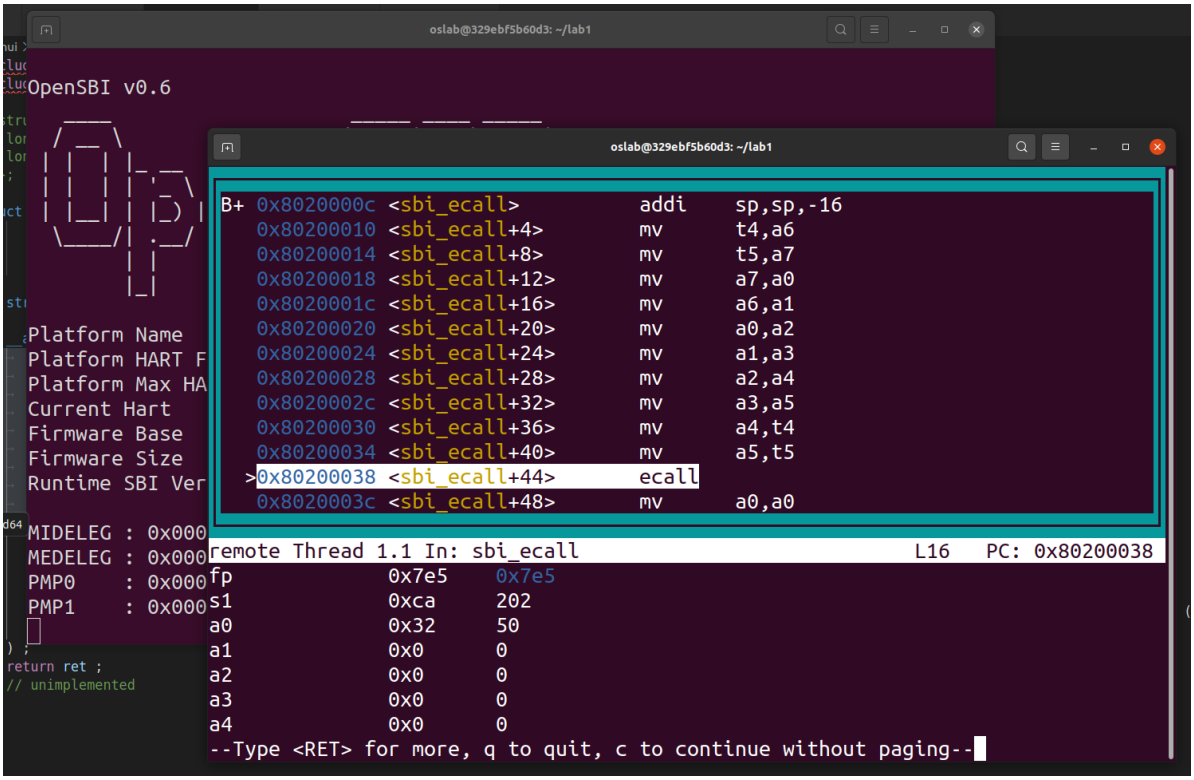


`make debug` **调制运行**



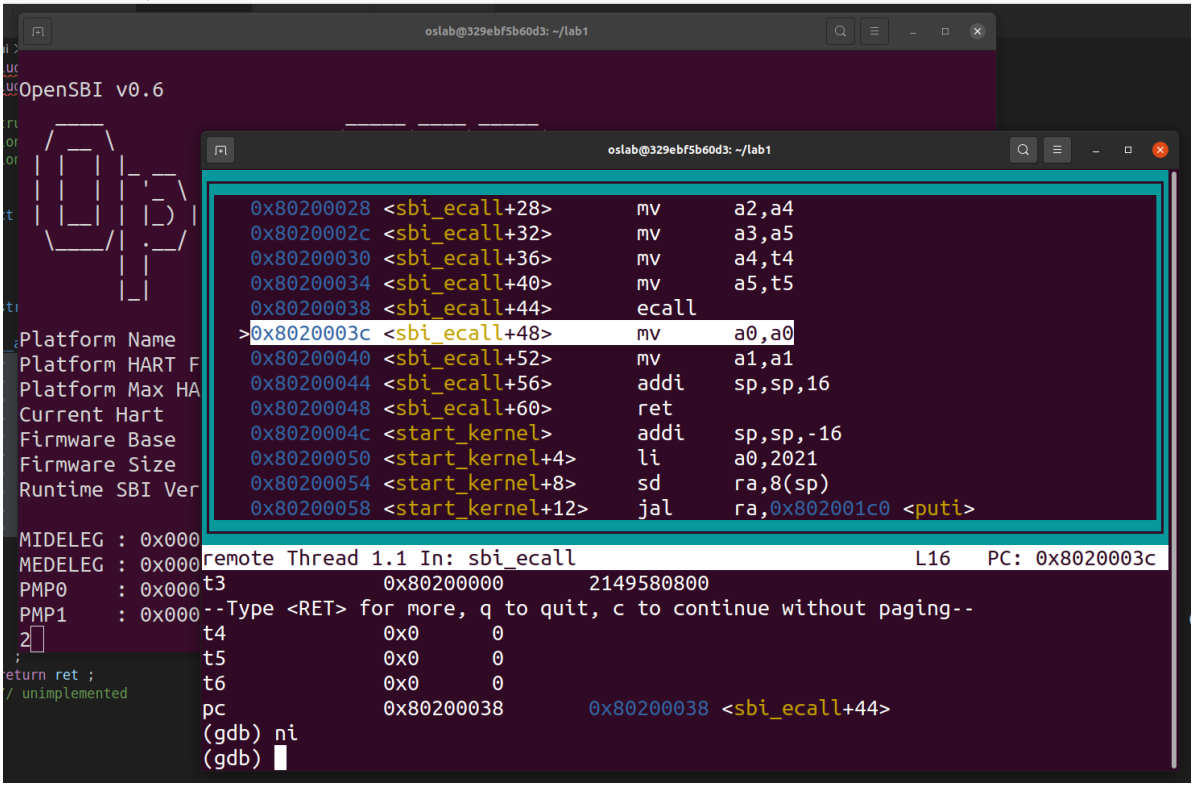
```
Reading symbols from vmlinux...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) b sbi_ecall
Breakpoint 1 at 0x8020000c: file sbi.c, line 16.
(gdb) continue
Continuing.

Breakpoint 1, sbi_ecall (ext=1, fid=0, arg0=50, arg1=0, arg2=0, arg3=0,
    arg4=0, arg5=0) at sbi.c:16
16              __asm__ volatile(
(gdb) s
```

观察栈信息以及寄存器内容：



如图：2将被打印，`'2'`即50被填入寄存器`a0`，当调用`ecall`时字符2就被标准输出打印出来。

下一步：

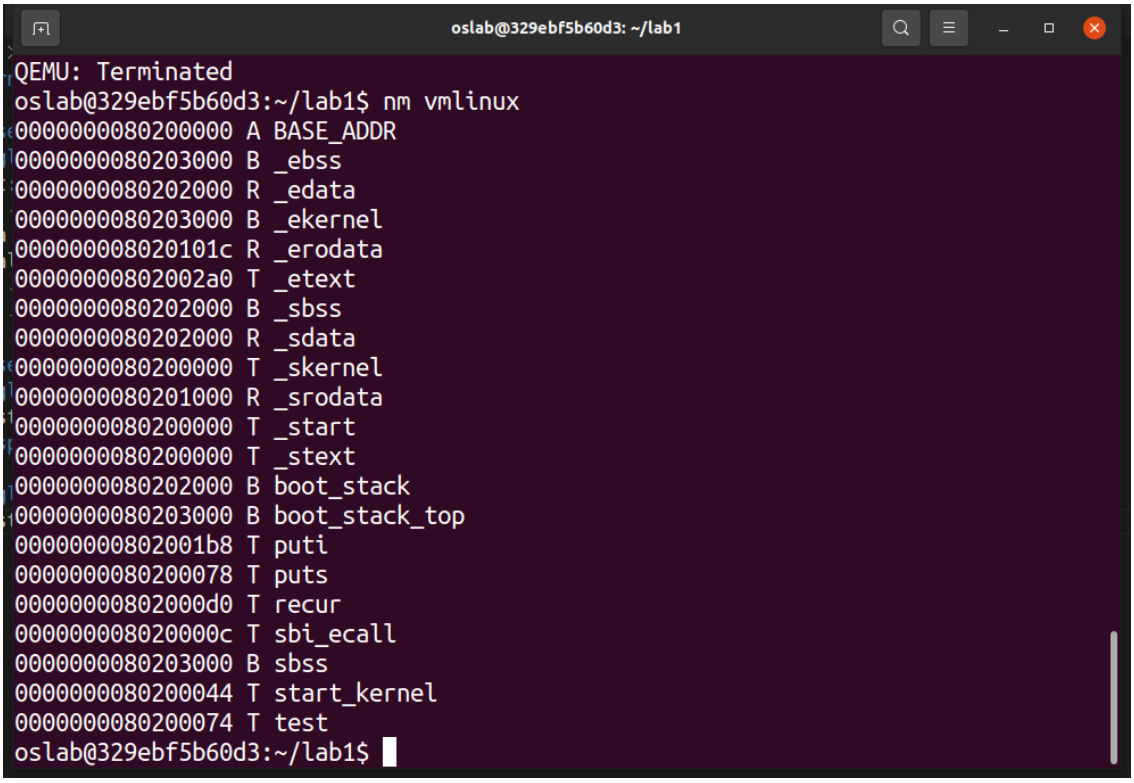


字符2被打印出来。

2 思考题

1. 请总结一下 RISC-V 的 calling convention，并解释 Caller / Callee Saved Register 有什么区别?
   - 在`risc-v`平台上，有8个整形寄存器(`a0`~`a7`)，传入参数（如果就是整形的话）默认从第一个开始存放在`a0`、`a1`.....
   - 函数返回值如果是基本类型或者只包含一两个成员的基本结构体的成员，存放在相应的整形寄存器`a0`和`a1`。
   - 顾名思义，caller saved register就是由调用者保存的寄存器，调用者必须在调用前保存并且调用后恢复这些寄存器，而callee saved register则是由被调用者保存的寄存器，调用者不需要在调用前后保存和恢复。
2. 编译之后，通过`System.map`查看`vmlinux.lds`中自定义符号的值

   查看`System.map`观察各个自定义符号的段地址



可以看到boot_stack_top位于地址80203000，也位于B开头即`.bss`的段内。

3 心得体会

上手完成输出内容的汇编代码，内联汇编的实现使得对系统调用`ecall`有了体会。

简单学习完成了`makefile`的原理和编写，对复杂目录的工程编译有了更细致的理解和体会。

`RiscV`伪指令掌握的不甚深入，应用得就更不够熟练了，对启动代码部分还有些许疑惑。

本次实验代码量不多但是对于操作系统启动的实现有了理解同时需要熟悉各个部分的原理。