Lab 1: RV64 内核引导

学号: 3190105838

姓名: 范钊瑀

4.1 准备工程

从实验仓库pull下来最新的源码，做好准备工作。

- arch/riscv/kernel/head.S
- lib/Makefile
- arch/riscv/kernel/sbi.c
- lib/print.c
- arch/riscv/include/defs.h

4.2 编写head.S

在 `head.S` 中，一方面我们要分配足够的栈空间供程序的stack pointer调用，另一方面在 `head.S` 函数中我们需要跳转到 `main.c` 中的主函数以跳转到内核的启动程序，需要注意的是，在分配完栈空间后我们也需要将栈顶指针 `sp` 加载到刚刚分配空间的栈顶位置处。接下来的程序可以用于将栈顶的地址加载到 `sp` 寄存器中，同时跳转到 `start_kernel` 的启动位置处，需要注意的是 `x0` 寄存器恒为常数0，所以 `jal` 指令将会将返回值加载到 `x0` 寄存器中并无影响。

```
la  sp, boot_stack_top
jal x0, start_kernel
```

同时使用 `.space` 伪指令分配4KB大小的栈空间

```
.space  4096 # <-- stack size
```

4.3 完善 Makefile 脚本

我们需要完善的是 `lib` 目录下的Makefile脚本，不难发现在根目录的Makefile中调用了 `${MAKE} -C lib all`，因此在我们需要完善的Makefile脚本中需要实现 `all` 标签，同时为了方便也需要实现 `clean` 标签以方便我们清理编译结果，仿照 `init` 目录下的Makefile文件，我们可以得到如下脚本：

```
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)
```

4.4 补充 `sbi.c`

在 `sbc.c` 中，我们需要完成相应寄存器的赋值和 `ecall` 函数的调用，这里需要用到内联汇编的语句，即在调用 `ecall` 指令前将ext、fid、arg等参数传入到对应的寄存器中，在调用结束后再将返回到值放入到函数返回到结构体中，用于实现功能的函数代码如下：

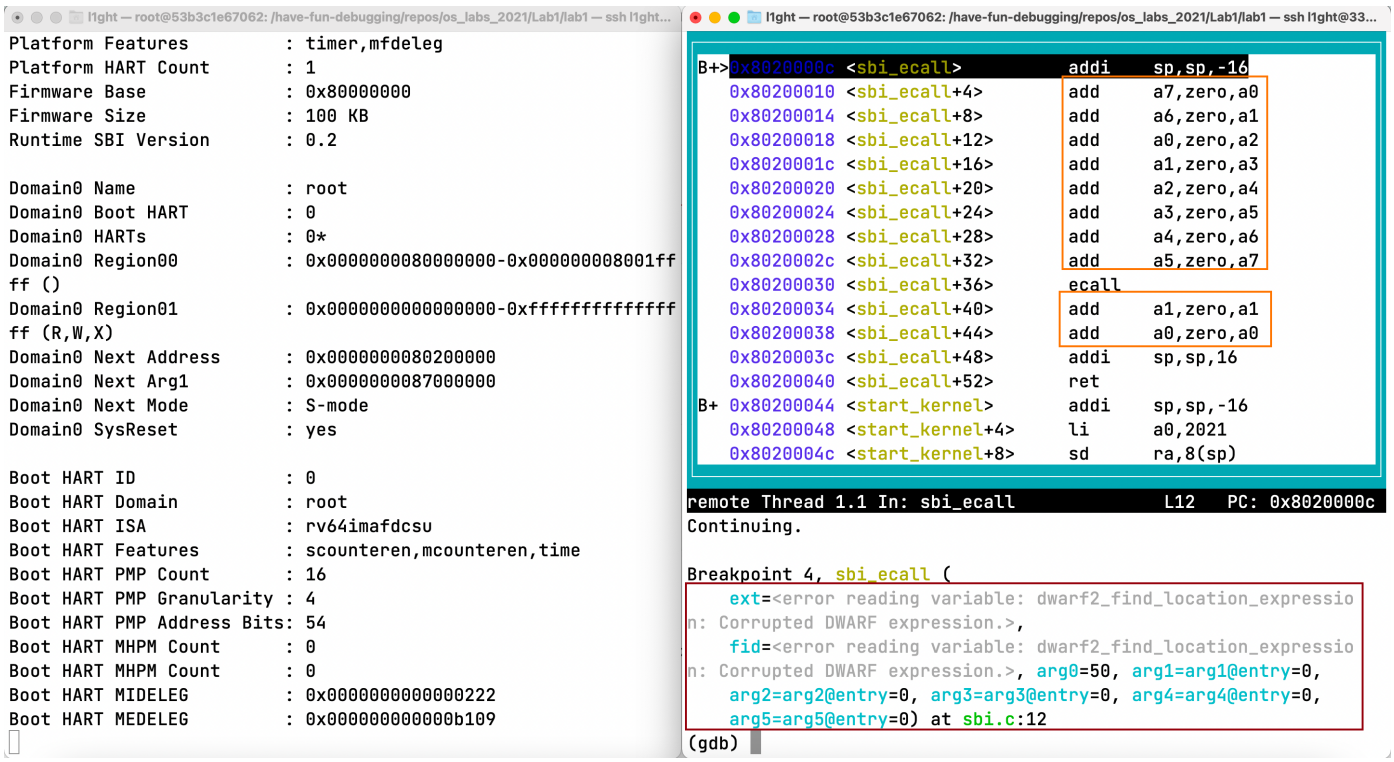
```
struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
                  uint64 arg1, uint64 arg2,
                  uint64 arg3, uint64 arg4,
                  uint64 arg5)
{
  long error, value;
  __asm__ volatile (
    "add a7, x0, %[ext]\n"
    "add a6, x0, %[fid]\n"
    "add a0, x0, %[arg0]\n"
    "add a1, x0, %[arg1]\n"
    "add a2, x0, %[arg2]\n"
    "add a3, x0, %[arg3]\n"
    "add a4, x0, %[arg4]\n"
    "add a5, x0, %[arg5]\n"
    "ecall\n"
```

```
      "add %[value], x0, a1\n"
      "add %[error], x0, a0\n"

      :[error] "=r" (error), [value] "=r" (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"
    );
    struct sbiret _ret;
    _ret.error = error;
    _ret.value = value;
    return _ret;
}
```

下图右侧橙色方框中的是调用 `ecall` 函数前寄存器的赋值过程。右下侧gdb调试窗口中能够看到调用 `sbi_ecall` 函数时传入的参数。

P.S.此处需要注意的是， `ext` 和 `fid` 变量的值并没有被成功被检测到，询问助教后得知是gdb显示的原因，而事实上参数已经正常被传入，故此处对结果并无影响。



4.5 puts() 和 puti()

这一部分的函数实现非常简单，仅需将原本用c实现的的输出字符串和整型变量的函数变成刚刚完成的 `sbi_ecall` 函数的调用即可，需要注意的是调用需满足 `sbi_ecall` 函数的参规范。

如下所示， `puts` 函数将读入字符串的从首地址的值开始，将字符串中每一位的ascii码依次传入到 `sbi_ecall` 函数并实现打印功能，直到我们遇见字符串末尾并停止。

```
void puts(char *s) {
    int i = 0;
    while(s[i]){
        uint64 _output = s[i];
        sbi_ecall(0x1, 0x0, _output, 0, 0, 0, 0, 0);
        i++;
    }
}
```

`puti` 函数用于打印出一个整型变量，在实现中我应用到了两个循环，第一个用于检测传入参数的位数，第二个用于按从左到右的顺序减去最高位并依次输出。当然，需要额外注意的是，负数需要单独判断并处理。
同时需要注意的是，由于int类型的范围是-2147483648到2147483647，-2147483648的相反数不在int的范围内，所以需要单独处理-2147483648这个值的输出。

```
void puti(int x) {
    // implemented
    unsigned long long cyc;
    if(x == -2147483648){
        puts("-2147483648");
        return;
    }
    if(x < 0){
        x = -x;
        sbi_ecall(0x1, 0x0, 0x2D, 0, 0, 0, 0, 0);
    }
    int _x = x;

    int length = 0;
```

```
    cyc = 1;
    while(cyc <= x){
        cyc *= 10;
        length++;
    }
    cyc /= 10;

    while(cyc){
        int current = _x / cyc;
        _x = _x % cyc;
        cyc /= 10;
        sbi_ecall(0x1, 0x0, (current + 0x30), 0, 0, 0, 0, 0);

    }

}
```

4.6 修改 defs

仿照 `csrw` 指令的宏定义方法，定义 `csrr` 指令的宏

```
#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    asm volatile ("csrr " "%0, " #csr       \
                  : : "r" (__v)                     \
                  : "memory");                      \
})
```

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?

查阅risc-v手册得到risc-v的不同寄存器都有其专门的用途，caller saved register表明在函数调用时由调用者将原本的值储存起来，于是在被调用内部函数看来，这些寄存器可以被直接使用。而callee saved register相反，是由被调用者将寄存器的值进行保存，保存后才可使用这些寄存器，因为一个寄存器的值不需要保护两次，RISC-V中约定了Caller / Callee Saved Register如下图所示。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

2. 编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值

   使用 `nm vmlinux` 指令或直接查看编译得到的符号表，包含符号和地址的对应关系，以及该段地址所包含的内容类型

```
root@53b3c1e67062:/have-fun-debugging/repos/os_labs_2021/Lab1/lab1# nm vmlinux
0000000080200000 A BASE_ADDR
0000000080203000 B _ebss
0000000080202000 R _edata
0000000080203000 B _ekernel
000000008020100f R _erodata
0000000080200180 T _etext
0000000080202000 B _sbss
0000000080202000 R _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080202000 B boot_stack
0000000080203000 B boot_stack_top
00000000802000d0 T puti
0000000080200078 T puts
000000008020000c T sbi_ecall
0000000080203000 B sbss
0000000080200044 T start_kernel
0000000080200074 T test
```