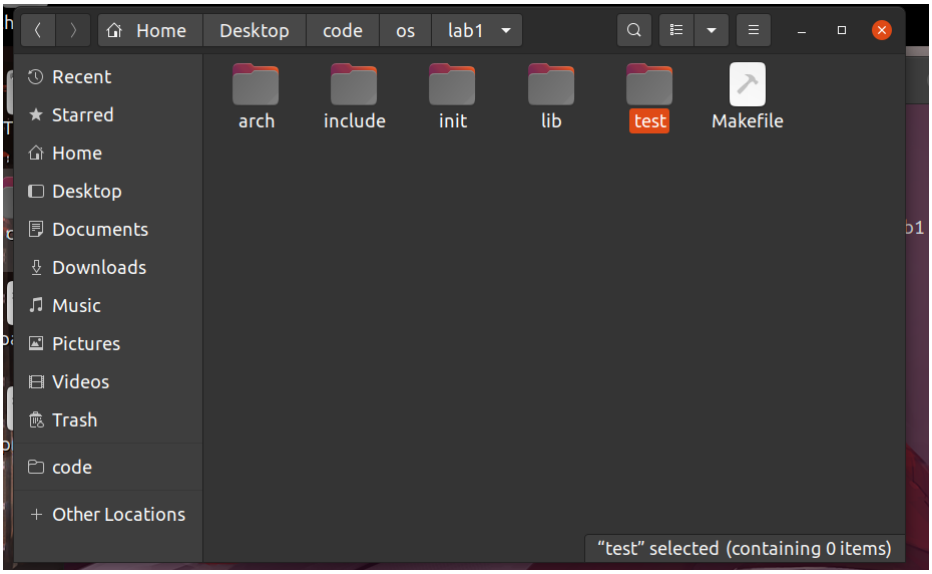
浙江大学实验报告

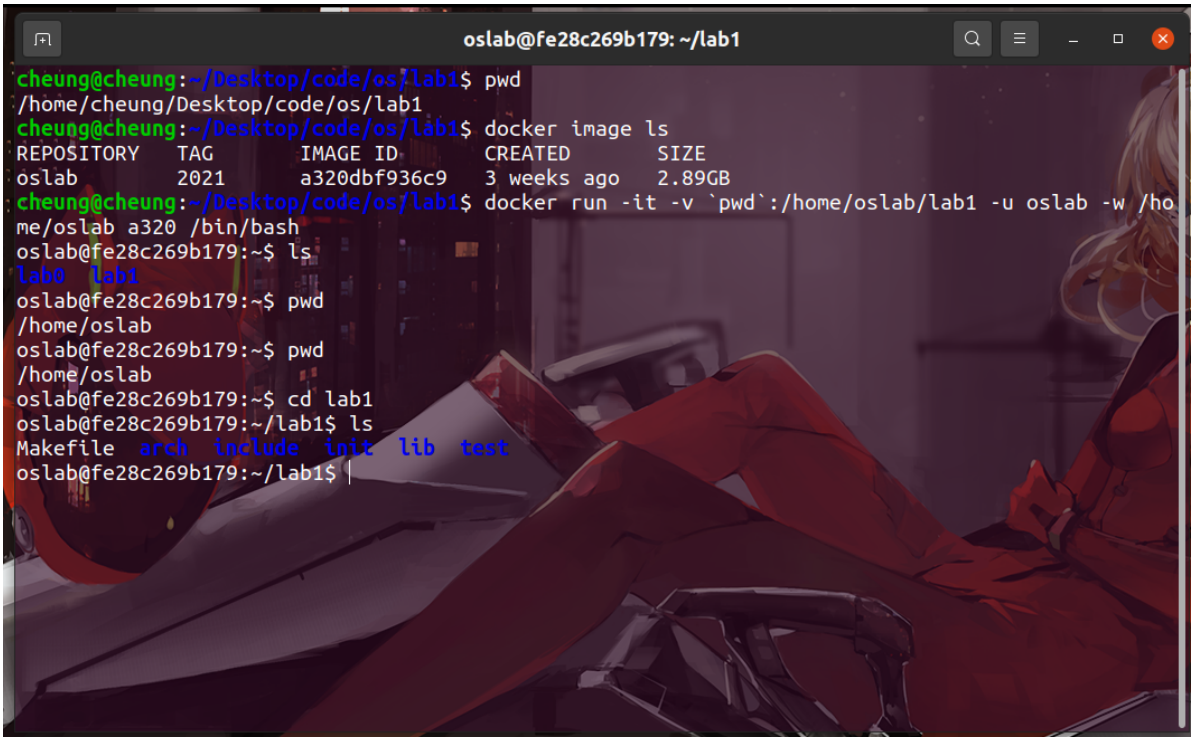
一、实验内容

准备工程

从git仓库中将代码克隆至本地后，根据文档的代码将工程代码映射进容器中，方便在本地开发。建立映射后，如下图在本地新建一个名为test的文件夹。



进入容器，可以看到此时`ls`命令显示容器内同步出现了test文件夹。可知映射建立成功。



编写head.S

查看文档可知，在该文件内需要完成的任务有两个：设置程序栈和跳转。可以看到文件内第一行就已经把跳转的目标函数声明了。因此在文件内我们只需要修改三行：第一行是将预留空间设置为4096；第二行是增加一行，通过`la`命令将栈顶的地址赋值给sp寄存器；第三行是跳转至目标函数。

完整代码如下：

```
.extern start_kernel

    .section .text.entry
    .globl _start
_start:
    la sp, boot_stack_top
    j start_kernel

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096 # <-- change to your stack size

    .globl boot_stack_top
boot_stack_top:
```

完善Makefile脚本

经过研究文件的结构以及Makefile命令，我发现需要填写的Makefile完全可以借用`arch\riscv\kernel\Makefile`的内容。并且，因为`lib`路径下只有c源文件而没有汇编源文件，Makefile中设计汇编源文件的部分可以删去。

代码如下：

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

all:$(OBJ)


%.o:%.S
    ${GCC}  ${CFLAG} -c $<

%.o:%.c
    ${GCC}  ${CFLAG} -c $<

clean:
    $(shell rm *.o 2>/dev/null)
```

补充`sbi.c`

只需要参考内联汇编的写法，清楚`ecall`的参数调用方式即可。主要工作是将参数放入相应的寄存器。

其中内联汇编的第四部分，指明汇编指令会影响的七个寄存器最好不要省略。其原因会在讨论心得部分说明。

```
#include "types.h"
#include "sbi.h"

struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
                        uint64 arg1, uint64 arg2,
                        uint64 arg3, uint64 arg4,
                        uint64 arg5)
{
    // unimplemented
    struct sbiret res;
    __asm__ volatile(
        "addi a0, %[arg0], 0\n"
        "addi a1, %[arg1], 0\n"
        "addi a2, %[arg2], 0\n"
        "addi a3, %[arg3], 0\n"
        "addi a4, %[arg4], 0\n"
        "addi a5, %[arg5], 0\n"
        "addi a6, %[fid], 0\n"
        "addi a7, %[ext], 0\n"
        "ecall\n"
        "addi %[err], a0, 0\n"
        "addi %[val], a1, 0"
        : [err] "=r"(res.error), [val] "=r"(res.value)              // output
        : [ext] "r" (ext), [fid] "r" (fid), [arg0] "r" (arg0), [arg1] "r"
(arg1), [arg2] "r" (arg2), [arg3] "r" (arg3), [arg4] "r" (arg4), [arg5] "r"
(arg5)    // input
        : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
    );
    return res;
}
```

`puts()`和`puti()`

调用上一步完成的`sbi_ecall`函数，只需要按次序、逐个打印字符即可实现字符串的打印以及整数的打印。其中，字符串的打印只需要遍历即可。而整数的打印，则需要先处理整数，将十进制的整数数字逐个打印。

```
#include "print.h"
#include "sbi.h"

void puts(char *s) {
    // unimplemented
    for(int i=0; s[i]; i++){
        sbi_ecall(0x1, 0x0, s[i], 0, 0, 0, 0, 0);
    }
    return ;
}
```

```
void puti(int x) {
    // unimplemented
    int stack[16], top = -1;
    while(x){
        stack[++top] = x%10;
        x/=10;
    }
    for( ; top>-1; top--){
        sbi_ecall(0x1, 0x0, stack[top]+'0', 0, 0, 0, 0, 0);
    }
    return ;
}
```

修改defs

参考函数宏的实现以及内联汇编的写法，简单调用`csrr`命令即可。

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

测试结果

将`main.c`中的`start_kernel`函数修改如下：

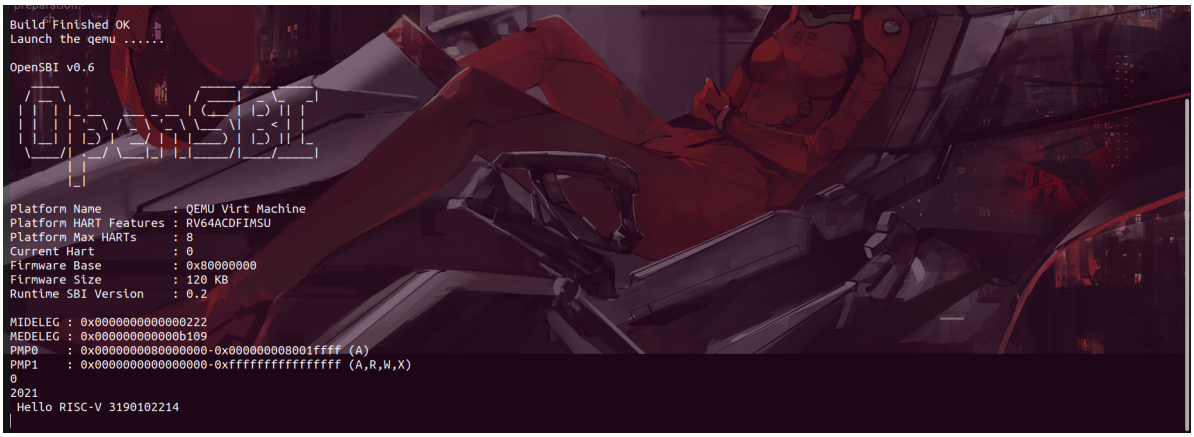
```
int start_kernel() {
    sbi_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0);
    puts("\n");
    puti(2021);
    puts("\n");
    puts(" Hello RISC-V 3190102214\n");

    test(); // DO NOT DELETE !!!

    return 0;
}
```

进入容器，在`lab1`目录下执行`make run`命令，可以看到内核可以正常输出。

在输出的最后，可以看到测试 `sbi_ecall` 的0，测试 `puti` 的2021，以及测试 `puts` 的字符串都被正常打印到了屏幕上。



二、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?

calling convention：

- 把函数参数放到函数可以访问的地方
- 将控制权转移给函数，跳转到函数开始的位置
- 拿到内存中的资源，按需保存寄存器
- 运行函数中的指令
- 将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源
- 返回调用函数的位置

为了加快运行速度，函数传参应优先通过寄存器进行。

Caller / Callee Saved Register区别：

- Caller Saved Register
    - a0-a7，如果需要为callee传参并且在调用callee之后还要使用这些寄存器的值，则需要在调用之前由caller保存旧值
    - ra，返回地址**必须**在调用callee之前保存，否则caller可能不能在执行结束后正常返回
    - t0-t9，如果在调用callee之后还要使用这些寄存器的值，则需要在调用之前由caller保存旧值
- Callee Saved Register
    - 包括s0-s11，callee在使用其中的任意一个寄存器之前都要先保存其旧值

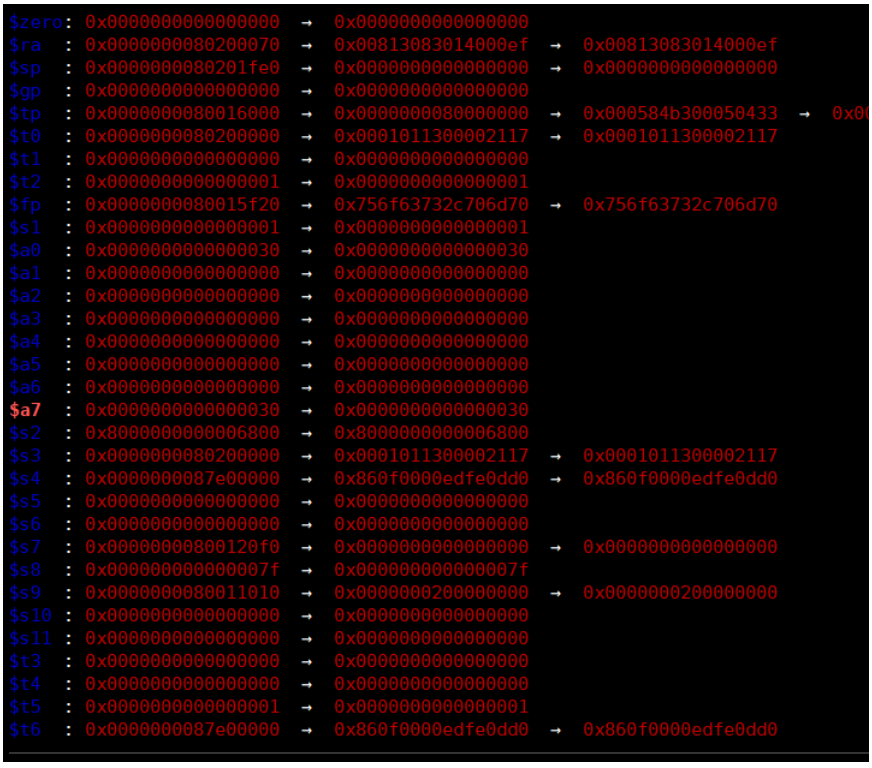
2. 编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值

直接打开System.map，可以看到自定义符号的值。

```
0000000080200000 A BASE_ADDR
0000000080203000 B _ebss
0000000080202000 R _edata
0000000080203000 B _ekernel
000000008020101a R _erodata
000000008020029c 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
000000008020013c T puti
00000000802000e4 T puts
000000008020000c T sbi_ecall
0000000080200074 T start_kernel
00000000802000e0 T test
```

三、讨论心得

本次实验遇到的最主要问题出在写`sbi_ecall`的时候。一开始，我没有在第四部分指明要修改的七个寄存器，并直接进行寄存器传参，发现屏幕上不能正确打印出数字。为此，调试程序，在调用`ecall`之前查看寄存器的值如下：



通过查看值，我猜测是因为出现了几个寄存器之间的交叉赋值。为了验证，在刚进入`sbi_ecall`时查看寄存器的值，可以看到结果如下：

当然，这个结果既是函数参数的顺序，也是内联汇编的传参顺序。经过再次试验，我发现寄存器的值对应函数参数的顺序。为了避免七个寄存器之间的交叉赋值，在这个过程中我使用过的解决方法如下：

1. 在内联汇编的第四部分声明要修改七个寄存器，但是编译器为了实现这个结果会优化汇编指令，程序实际的汇编指令与输入的汇编指令不同
2. 修改函数参数的顺序
3. 使用堆栈，确保各个寄存器都被正确赋值

本次实验使我对操作系统的底层有了更多了解，调试的过程也让我对于gdb的使用更加熟练，还让我学习了内联汇编的用法。