**Lab1：RV64内核引导**

姓名：沈韵沨 专业：软件工程 学号：3200104392 日期：2022年10月06日

**1 实验目的**

* 学习RISC-V汇编，编写head.S实现跳转到内核运行的第一个C函数。
* 学习OpenSBI，理解OpenSBI在实验中所起到的作用，并调用OpenSBI提供的接口完成字符的输出。
* 学习Makefile相关知识，补充项目中的Makefile文件，来完成对整个工程的管理。

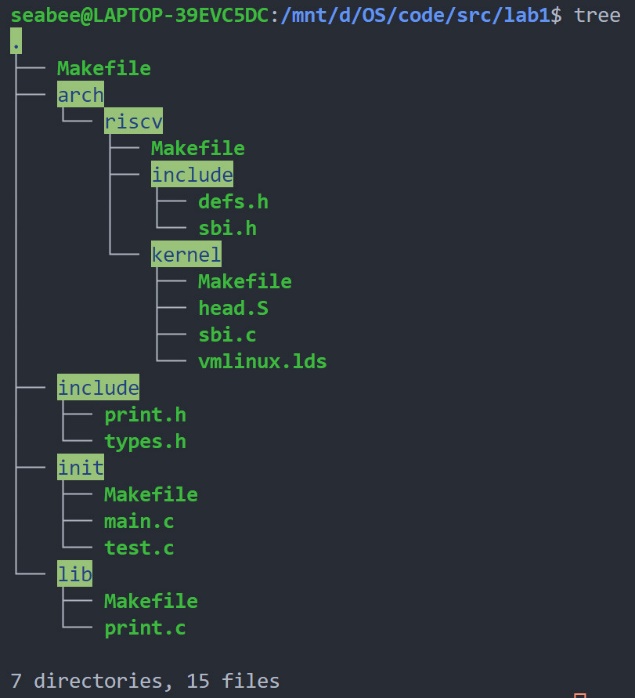
**2 实验环境**

* Ubuntu 22.04.1 LTS Windows Subsystem for Linux 2（同Lab0）

**3 实验步骤**

**3.1 准备工程**

由于在Lab0的实验过程中已完成对代码框架的同步，此处直接展示/mnt/d/OS/code/src/lab1 路径下的目录结构：



**3.2 编写 head.S**

在本节中需要完成对 arch/riscv/kernel/head.S 的编写 。

我们首先为即将运行的第一个 C 函数设置程序栈（大小为4kB），并将该栈放置在.bss.stack 段。此后通过跳转指令，跳转至 main.c 中的 start\_kernel 函数。

编写完成后的 head.S 文件内容如下：

.extern start\_kernel

    .section .text.entry

    .globl \_start

    .equ stack\_size, 4096 # *定义栈大小为4kB*

\_start:

    # *------------------*

    # *my code*

    la sp, boot\_stack\_top # *使栈指针指向栈顶*

    jal start\_kernel  # *跳转至main.c中的start\_kernel函数*

    # *------------------*

    .section .bss.stack

    .globl boot\_stack

boot\_stack:

    .space stack\_size # *将栈大小设置为4kB*

    .globl boot\_stack\_top

boot\_stack\_top:

**3.3 完善 Makeflie 脚本**

通过阅读 [makefile介绍 — 跟我一起写Makefile 1.0 文档 (seisman.github.io)](https://seisman.github.io/how-to-write-makefile/introduction.html) 与工程中的 Makefile 文件，我们成功掌握了 Makefile 的使用规则，并对lib/Makefile文件进行了补充，该文件内容如下：

# *lib/Makefile*

SRC = print.c

OBJ = print.o

all : $(OBJ)

%.o : %.c

    ${GCC} ${CFLAG} -c $(SRC)

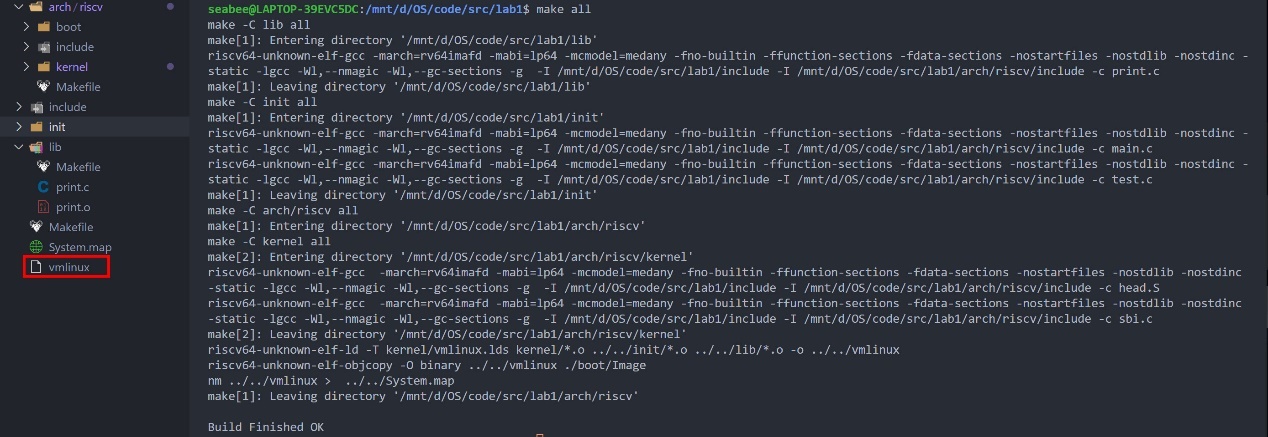
.PHONY : clean

clean:

    -rm $(OBJ)

在该文档，我们定义了变量SRC与OBJ，分别指代该路径下的所有.c文件与.o文件，以便对所有源文件进行编译，并快速清除所有构建产物。

在工程根文件夹lab1下执行make all命令，可以看到工程成功编译出vmlinux。



**3.4 补充 sbi.c**

在sbi\_ecall函数中，我们需要完成以下内容：

1. 将ext (Extension ID) 放入寄存器a7中，fid (Function ID) 放入寄存器a6中，将arg0~arg5放入寄存器a0~a5中。
2. 使用ecall指令使系统进入M模式，之后通过OpenSBI完成相关操作。
3. 将OpenSBI的返回结果分别存放在寄存器a0 (error code)和a1(返回值)中，用sibret接受这两个返回值。

编写完成的arch/riscv/kernel/sbi.c内容如下：

// *arch/riscv/kernel/sbi.c*

#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)

{

    struct sbiret res;// *记录OpenSBI的返回值*

    // *对各寄存器进行赋值*

    register uint64 a0 asm ("a0") = arg0;

    register uint64 a1 asm ("a1") = arg1;

    register uint64 a2 asm ("a2") = arg2;

    register uint64 a3 asm ("a3") = arg3;

    register uint64 a4 asm ("a4") = arg4;

    register uint64 a5 asm ("a5") = arg5;

    uint64 \_fid = (uint64)(fid);

    uint64 \_ext = (uint64)(ext);

    register uint64 a6 asm ("a6") = \_fid;

    register uint64 a7 asm ("a7") = \_ext;

    //*内联汇编部分*

    \_\_asm\_\_ volatile (

        "ecall\n"//*使系统进入M模式*

        :"=r" (a0), "=r" (a1)::

    );

    // *将OpenSBI的返回值存入res*

    res**.**error = a0;

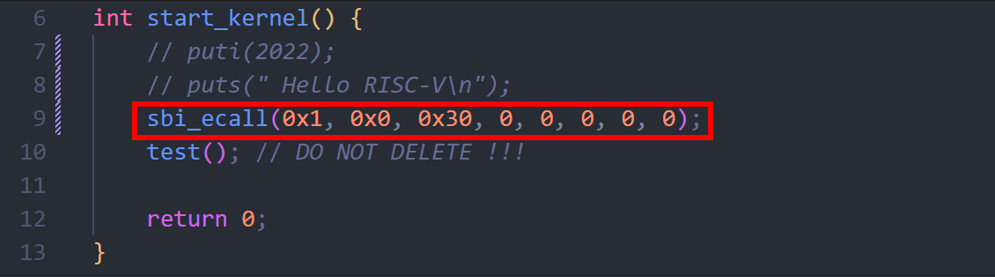
    res**.**value = a1;

    return res;

}

我们首先定义了用于接收OpenSBI返回结果的sbiret结构体res，随后对各个寄存器进行赋值。在内联汇编部分，我们使用ecall命令使系统进入M模式，并使得OpenSBI将返回结果分别写入寄存器a0, a1。最后我们将返回结果赋给sbiret结构体res。

通过在init/main.c中的start\_kernel()函数调用sbi\_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0)，再进行编译运行，发现成功输出字符 ‘0’，编写正确：



测试时start\_kernel()函数的内容



测试输出结果

**3.5 编写函数 puts() 与 puti()**

通过调用在3.4中完成的sbi\_ecall()函数，我们实现了用于打印字符串的puts()函数与用于打印整形变量的puti()函数。

编写完成的lib/print.c内容如下：

// *lib/print.c*

#include "print.h"

#include "sbi.h"

void puts(char \*s) { // *用于打印字符串*

    char idx = 0;

    while(s[idx] != '\0') {

        sbi\_ecall(0x1, 0x0, (int)s[idx], 0, 0, 0, 0, 0);

// *调用3.4中完成的sbi\_ecall函数打印单个字符*

        idx++;

    }

}

void puti(int x) { // *用于打印整形变量*

    if(x) {

        char tmp[11] = "\0";// *用于存储倒序数字串*

        int len = 0;

        while(x) {

            tmp[len++] = x%10 + 0x30;

            x /= 10;

        }

        int i = 0;

        while(i < len) {// *正序输出数字串*

            sbi\_ecall(0x1, 0x0, (int)tmp[len-i-1], 0, 0, 0, 0, 0);

            i++;

        }

    }

    else {//*单独处理x==0 的情况*

        sbi\_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0);

    }

}

**3.6 修改 defs.h**

待完成宏csr\_read涉及的RISC-V指令为csrr，在RISC-V中文手册中查的其用法如下：



参照文件中已完成的宏csr\_write的实现，我们完成了对arch/riscv/include/defs.h的编写，内容如下：

// *arch/riscv/include/defs.h*

#ifndef \_DEFS\_H

#define \_DEFS\_H

#include "types.h"

#define csr\_read(csr)                       \

({                                          \

    register uint64 \_\_v;                    \

    asm volatile ("csrr %0, " #csr        \

                  : "=r" (\_\_v) ::);         \

    \_\_v;                                    \

})

#define csr\_write(csr, val)                         \

({                                                  \

    uint64 \_\_v = (uint64)(val);                     \

    asm volatile ("csrw " #csr ", %0"               \

                    : : "r" (\_\_v)                   \

                    : "memory");                    \

})

#endif

**3.8 编译测试**

在工程根目录Lab1下使用make run vmlinux命令，运行编译产生的vmlinux文件，并成功输出“2022 Hello RISC-V”，证明函数编写正确且执行成功：



**4 思考题**

**4.1 总结RISC-V的calling convention，并解释Caller/Callee Saved Register的区别。**

* RISC-V的calling convention：
  1. 数据类型及对齐：

低精度数据被保存至寄存器中将进行相应扩展（带符号扩展或0扩展），具体位数由编译器决定（RV32将扩展至32位，RV64将扩展至64位），以确保自然对齐。

* 1. 调用约定：

\* 应尽可能使用寄存器来传递参数：a0-a7位整数寄存器，其中a0与a1可用于传递返回值；fa0-fa7为浮点数寄存器，其中fa0与fa1可用于传递返回值（仅当其作为原始参数，或为只包含1/2个浮点值的struct时）。浮点值外的2个指针字长的返回值放入a0与a1，更大的返回值通过存储器返回（此时寄存器由caller分配，并作为第一个隐藏参数传递给callee）。

\* 若参数为结构体，则参数的每个字段按照指针长度对齐，参数寄存器保存结构体头部8个指针字长的数据。

\* 浮点参数通过浮点寄存器fai传递，但属于union或struct的浮点参数将通过整数寄存器传递；没有在参数列表中明确命名的变长浮点参数也将通过整数寄存器传递。

\* 整型参数通过整数寄存器ai传递。

\* 小于1个指针字的参数通过寄存器的最低有效位传递；若通过栈传递，该参数也将出现在指针字的低位。

\* 等于2个指针字的参数通过栈传递时自然对齐；若通过整数寄存器传递，则被置于even-odd寄存器对中（偶数编号寄存器保存最低有效位）。

\* 更长的参数通过reference传递。

\* 栈向下增长，且栈指针16位对齐。

（剩余内容在随后对于Caller/Callee Saved Register的说明中阐明）

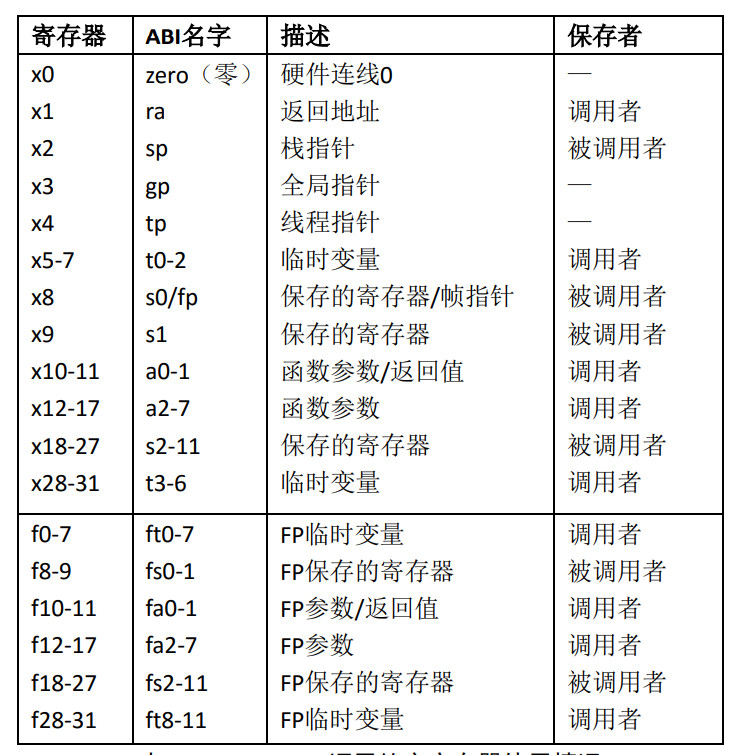
* Caller Saved Register

在被叫函数运行时，这些寄存器的值可能被破坏，但被叫函数无需对其进行保存；

若主叫函数需要在调用结束后恢复该值，则应该在主叫函数中将其压入堆栈或复制到其他位置。

* Callee Saved Register

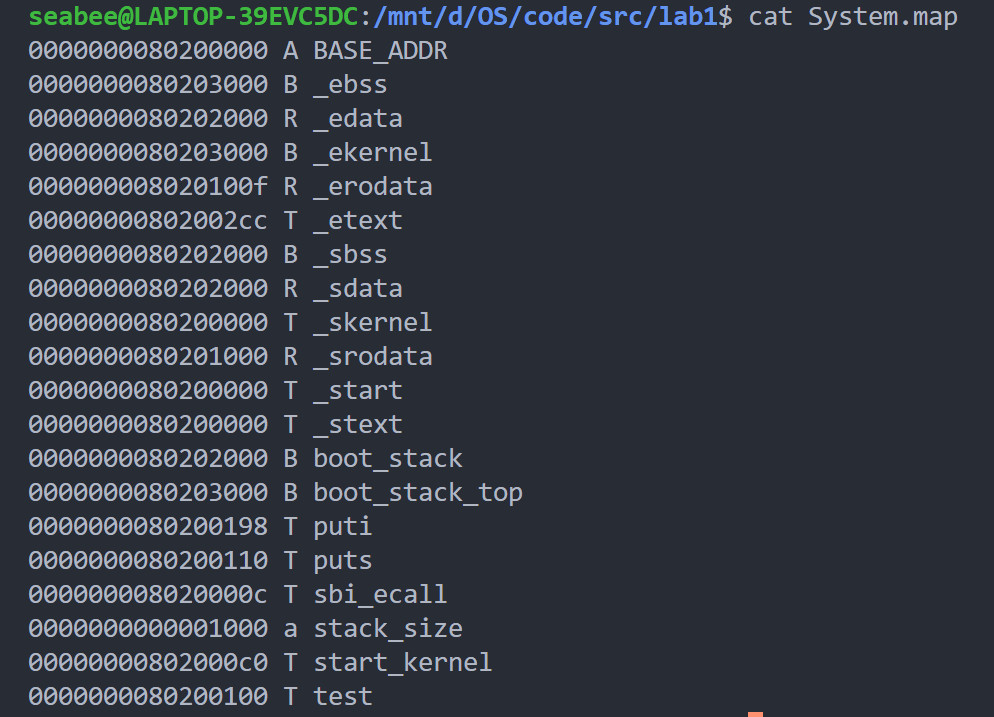
这些寄存器中的值需要在被叫函数运行前后保持一致。若被叫函数需要调用这些寄存器，则应该先对寄存器中的值进行保存，并在结束调用时对其进行恢复。



RISC-V调用约定寄存器使用情况

**4.2 编译之后，通过Sytem.map查看vmlinux.Ids中自定义符号的值。**

编译完成后，在根目录Lab1下使用命令cat System.map，结果如下：



**4.3 用csr\_read宏读取sstatus寄存器的值，对照RISC-V手册解释其含义。**

修改start\_kernel()函数为以下内容：

int start\_kernel() {

    puti(2022);

    puts(" Hello RISC-V\n");

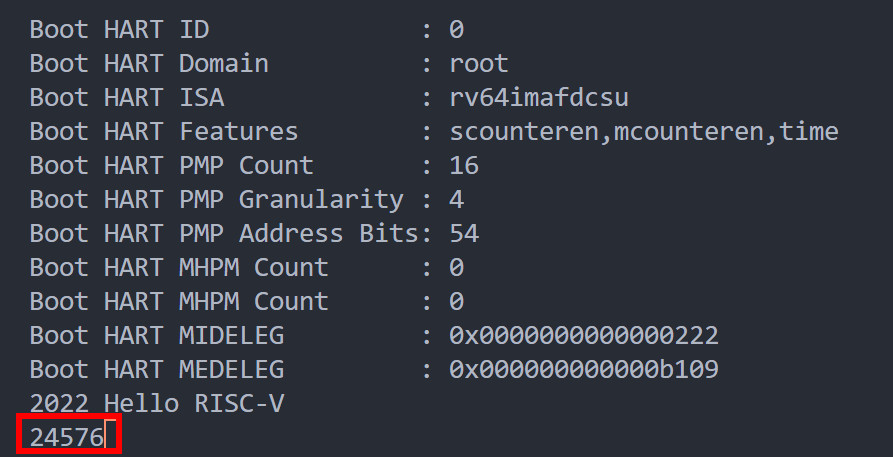
    puti(csr\_read(sstatus));

    test();// *DO NOT DELETE !!!*

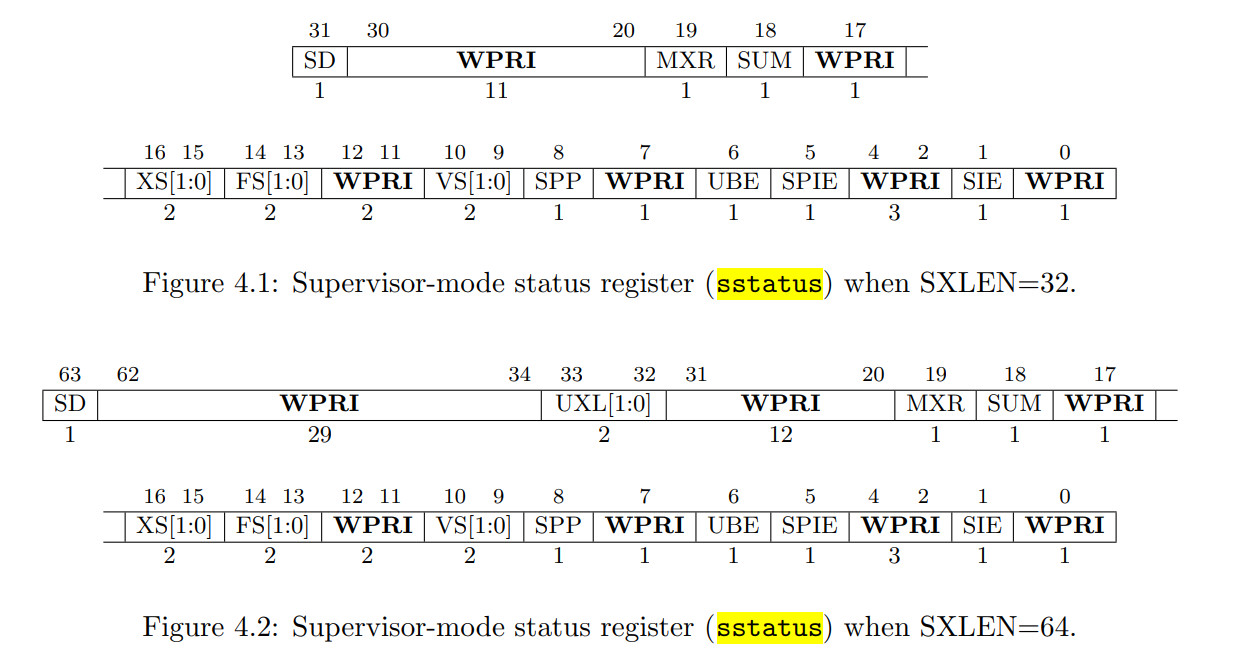
    return 0;

}

如下图，编译运行后输出“24576”（0x6000）：



查询《RISC-V Privileged Architectures V20211203》得到sstatus寄存器的存储结构：



对照可知：

\* SIE = 0：S-mode下禁止中断

\* UBE = 0：U-mode下进行的显示内存访问为小端字节序

\* SPP = 0：之前的mode为U-mode

\* MXR = 0：只有从标记为可读的页面加载才会成功

\* SUM = 0：对U-mode可访问的页面，S-mode访问将会出错

**4.4 用csr\_write宏向sscratch寄存器写入数据，并验证是否写入成功。**

修改start\_kernel()函数为以下内容，通过宏csr\_write向sscratch寄存器写入数据8080，再通过宏csr\_read进行读取、函数puti()进行打印：

int start\_kernel() {

    puti(2022);

    puts(" Hello RISC-V\n");

    csr\_write(sscratch, 8080);

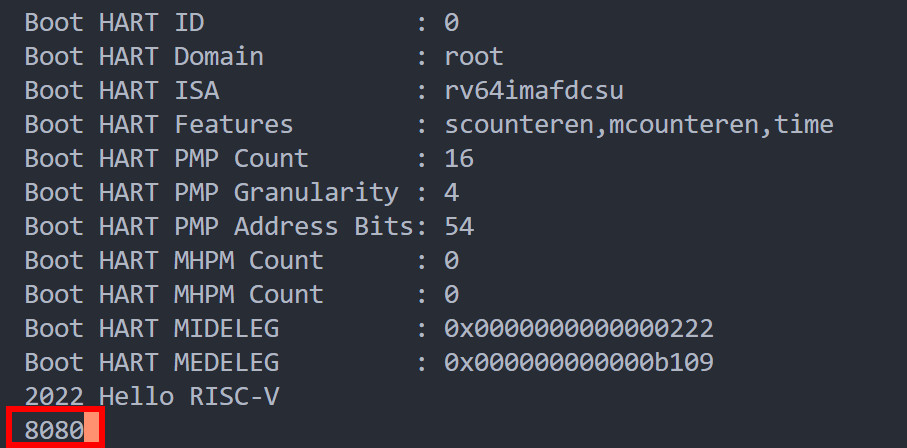
    puti(csr\_read(sscratch));

    test();// *DO NOT DELETE !!!*

    return 0;

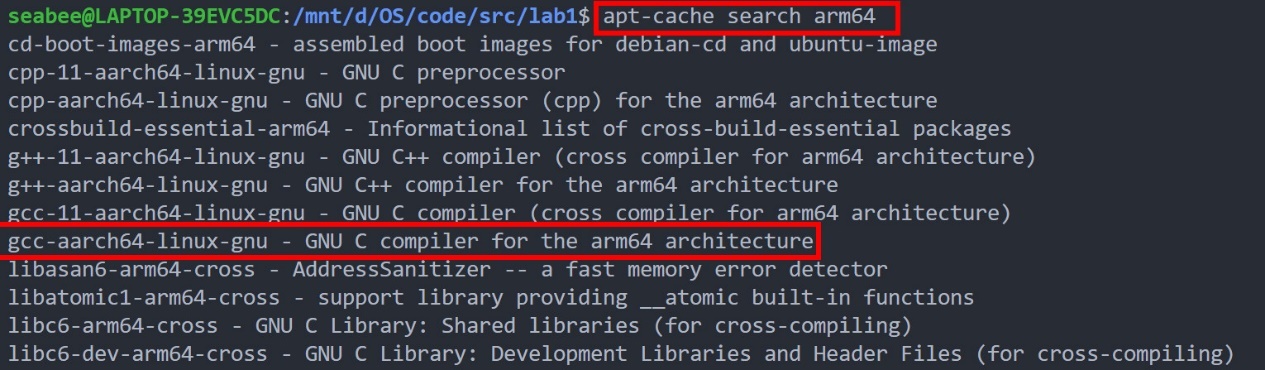
}

实验输出结果如下，写入成功：

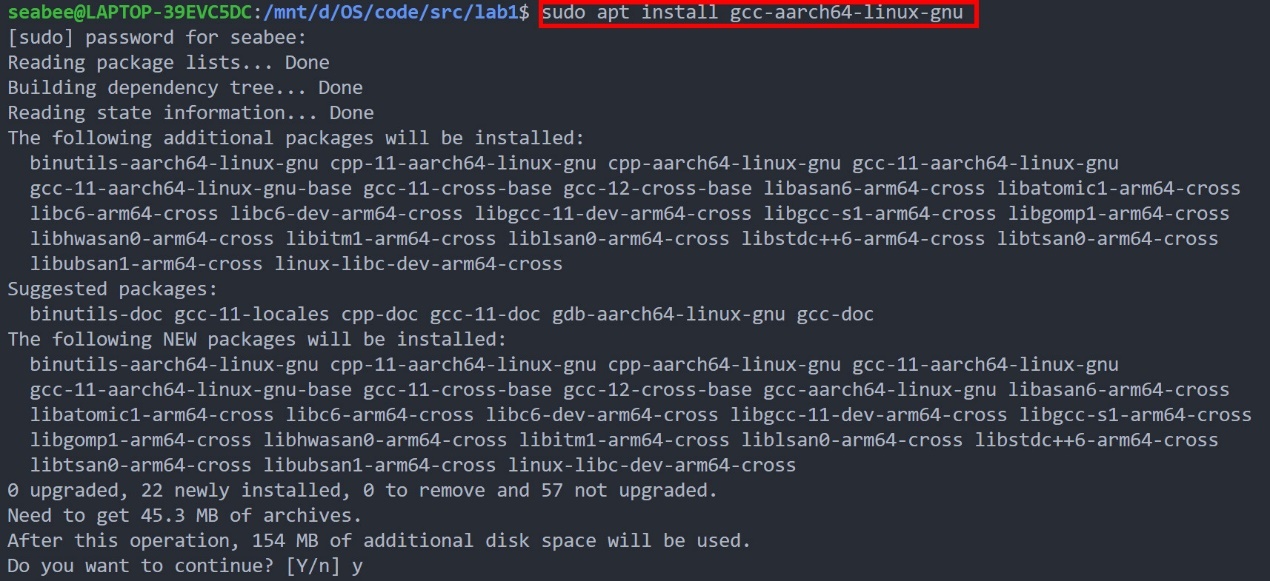


**4.5 Detail your steps about how to get arch/arm64/kernel/sys.i**

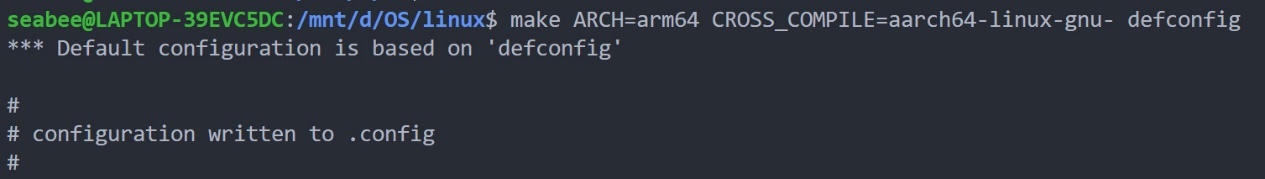
1. 安装交叉编译工具链：  
   首先通过apt-cache search arm64命令搜索适用于arm64架构的交叉编译工具链：



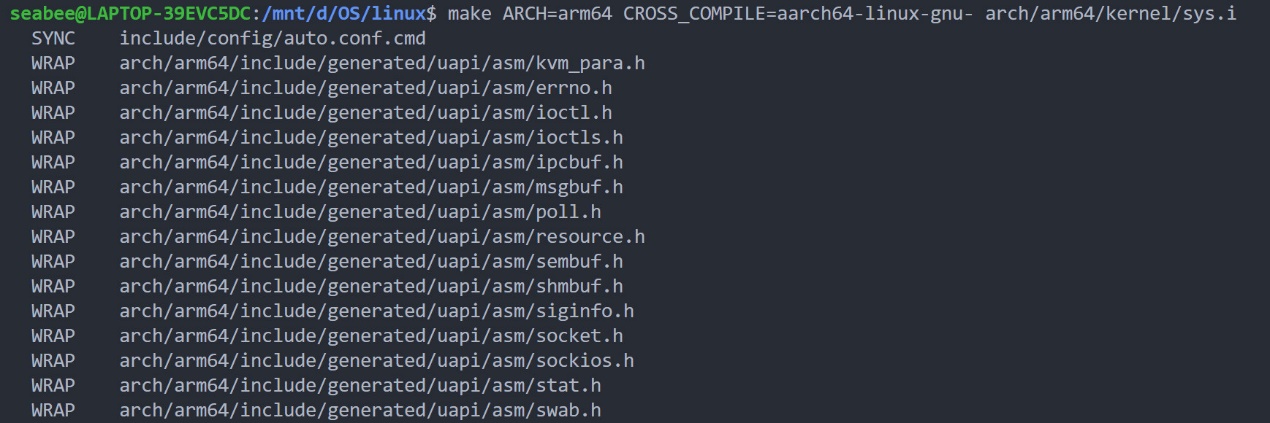
使用sudo apt install gcc-aarch64-linux-gnu命令安装交叉编译工具链：



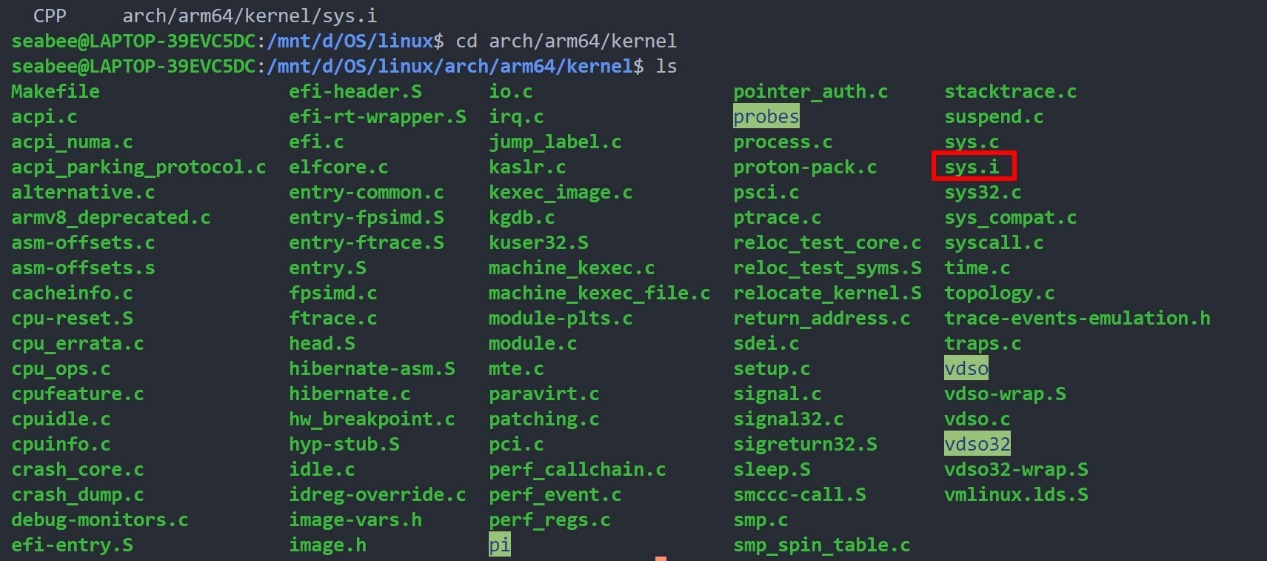
1. 使用默认配置



1. 进入linux根目录后指定要生成的文件sys.i



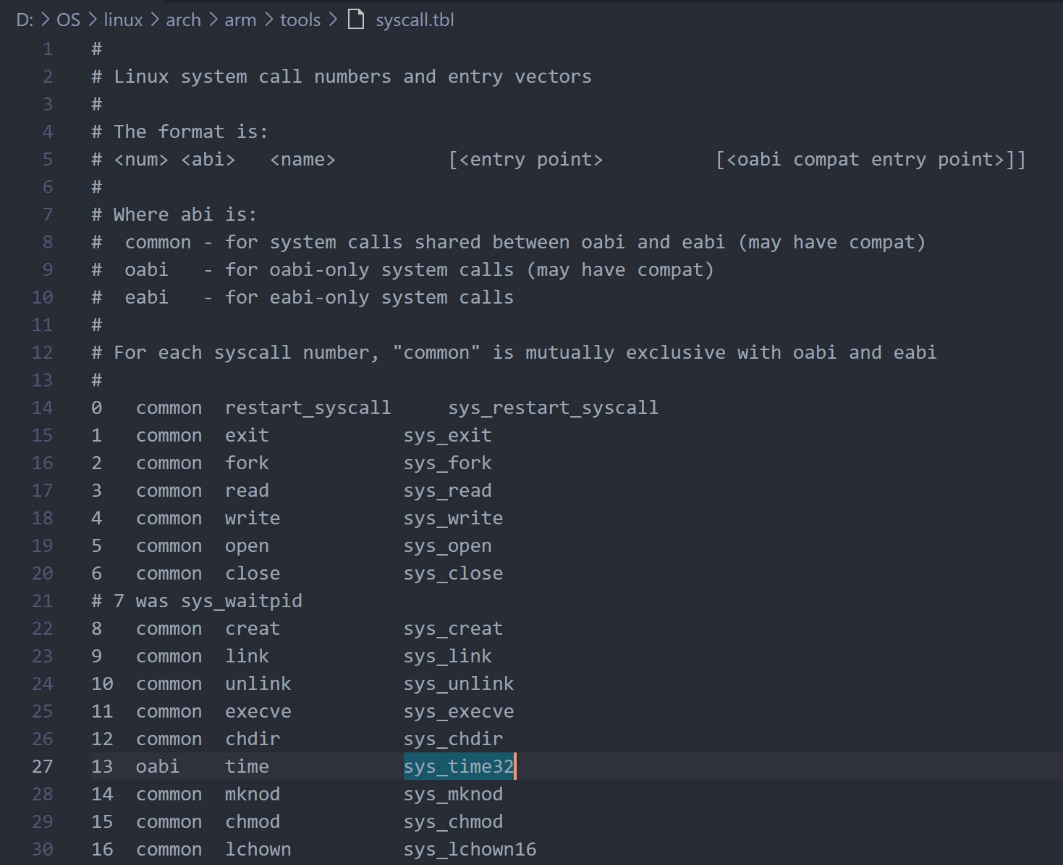
进入arch/arm64/kernel路径，证明确实生成了sys.i文件。



**4.6 Find system call table of Linux v6.0 for ARM32, RISC-V(32 bit), RISC-V(64 bit), x86(32**

**bit), x86\_64. List source code file, the whole system call table with macro expanded, screenshot every step.**

* ARM32：arch/arm/tools/syscall.tbl

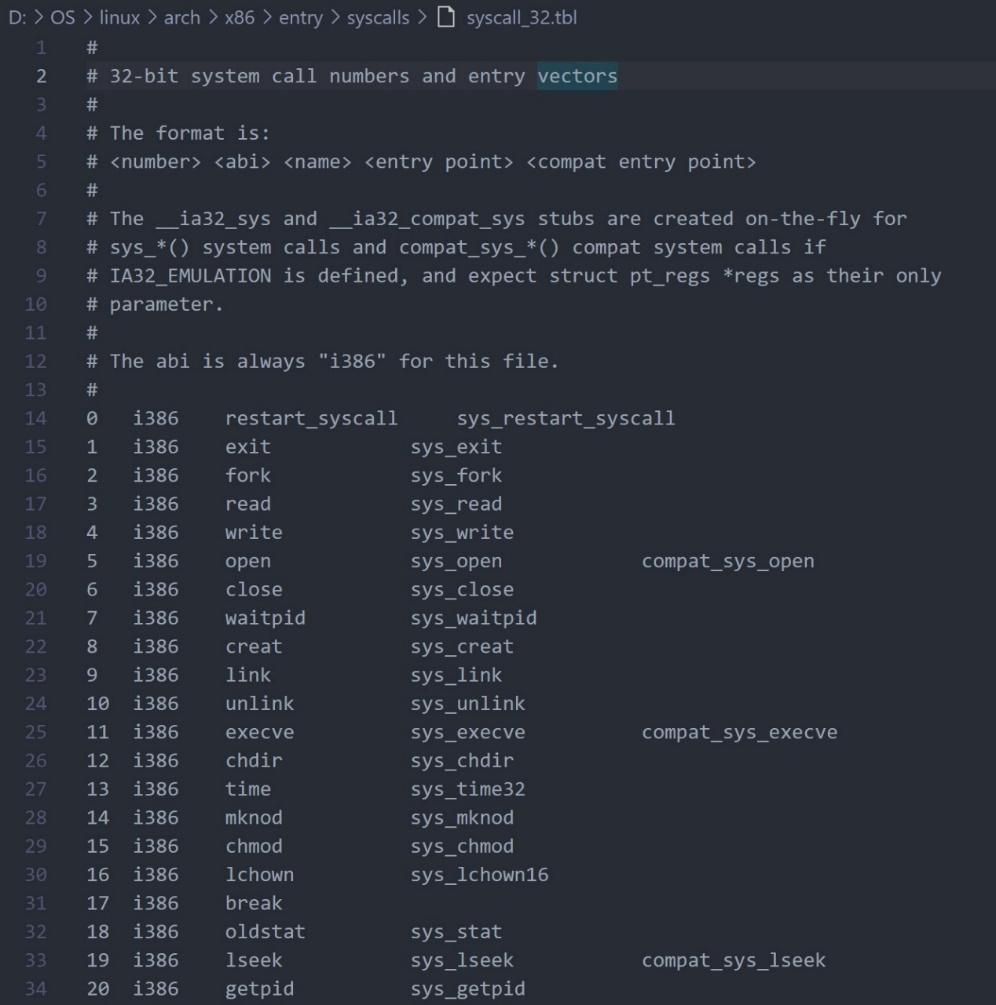


* RISC-V(32 bit) & RISC-V(64 bit)：

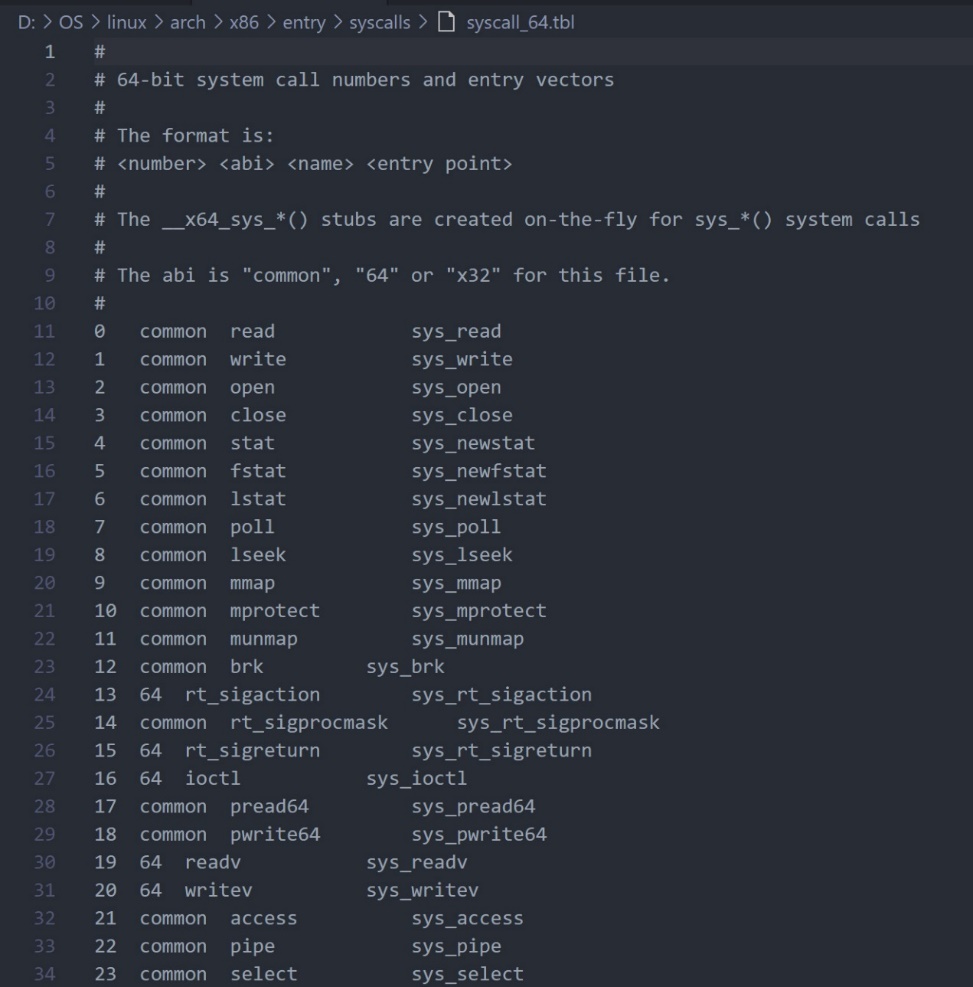
真的找不到了，只找到了arch/riscv/include/asm/syscall.h



* x86(32 bit)：arch/x86/entry/syscalls/syscall\_32.tbl



* x86\_64：arch/x86/entry/syscalls/syscall\_64.tbl

****

**4.7 Explain what is ELF file? Try readelf and objump command on an ELF file, give screenshot of the output. Run an ELF file and cat /proc/PID/maps to give its memory layout.**

ELF文件即Executable and Linkable Format（可执行与可连接格式），由ELF header，Program header table，Section，Section header table四部分构成，后三者的位置与大小信息由ELF header中的各项值决定。在Linux系统中有如下三种文件：

（1）可执行文件.out:

Executable File, 包含二进制代码和数据且均具有固定地址或相对基地址的偏移量，系统可根据地址信息将程序加载到内存执行。

（2）可重定位文件.o:

Relocatable File, 包含二进制代码和数据，但没有绝对地址，适合与其他目标文件链接以创建可执行文件或共享目标文件。

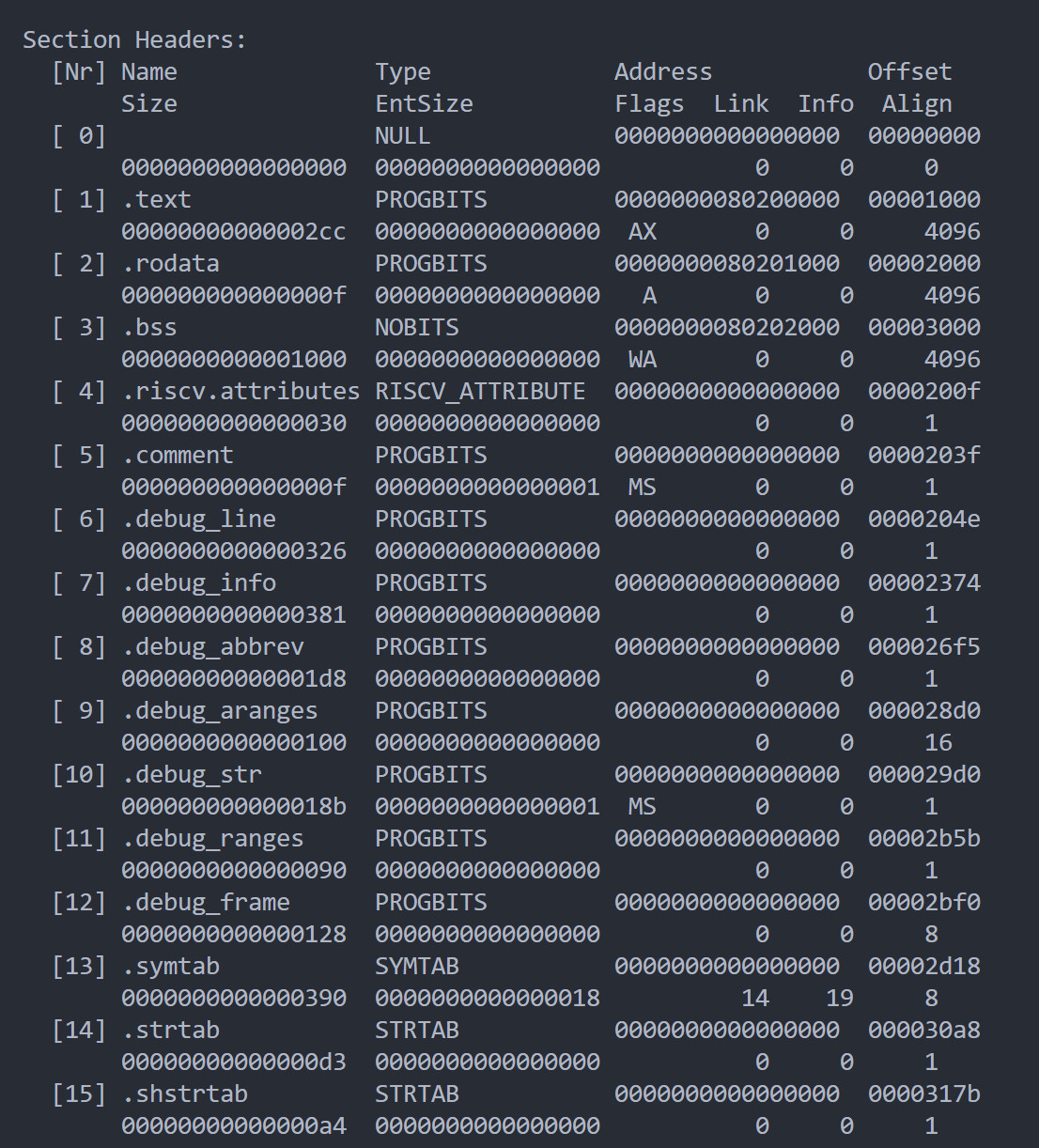
（3）共享目标文件.so:

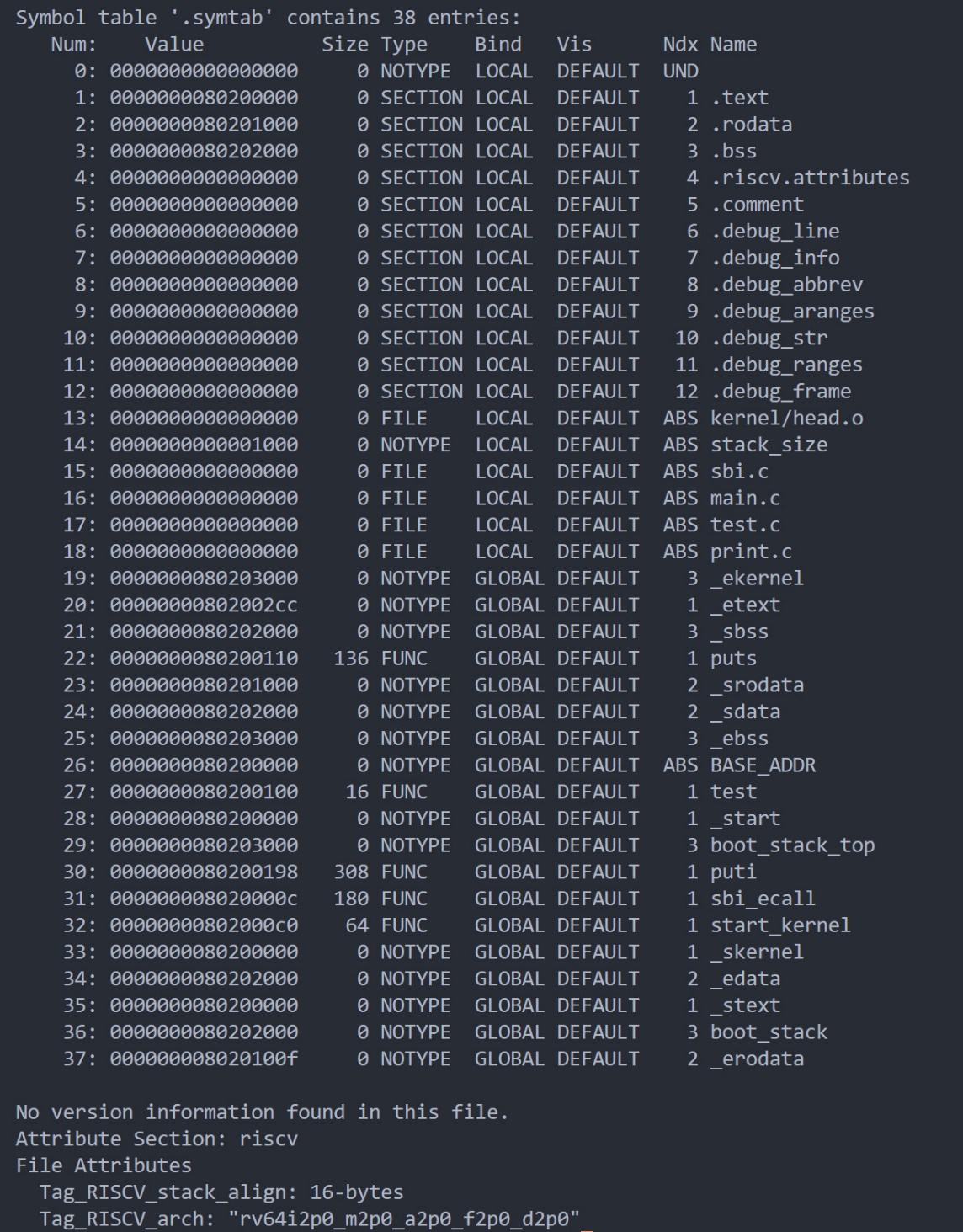
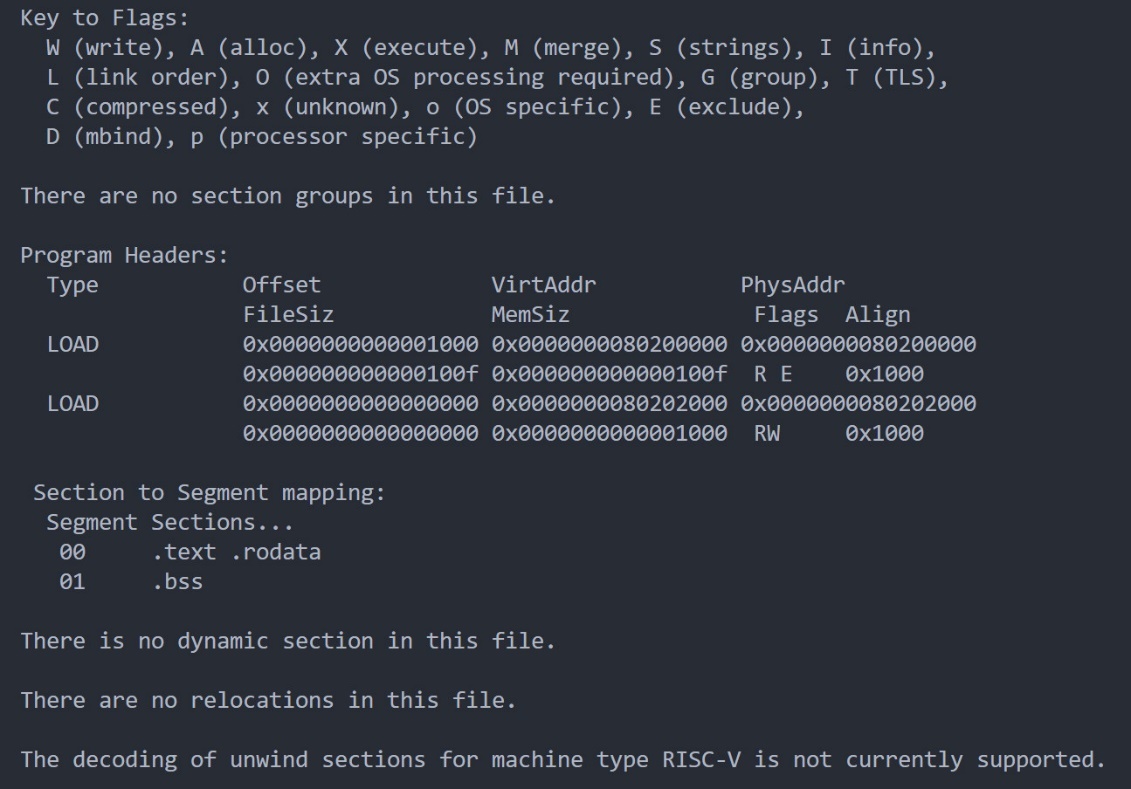
Shared Object File, 包含代码和在链接时被链接器和运行时动态链接器使用的数据，可以被动态加载和链接。

* 使用readelf命令

此处查看之前编译生成的vmlinux文件，通过readelf -a命令查看文件的所有信息：

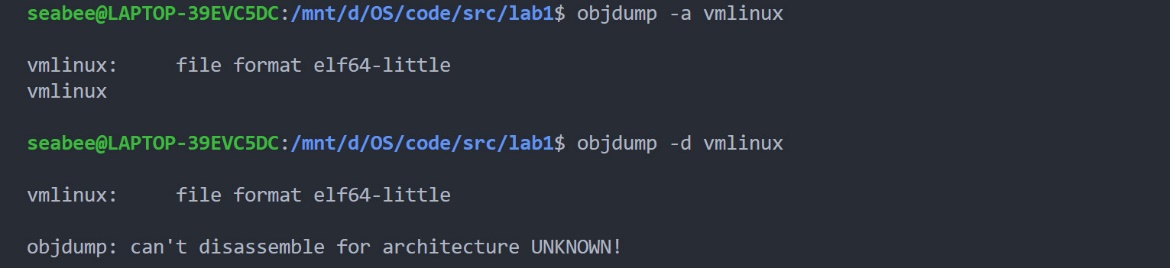




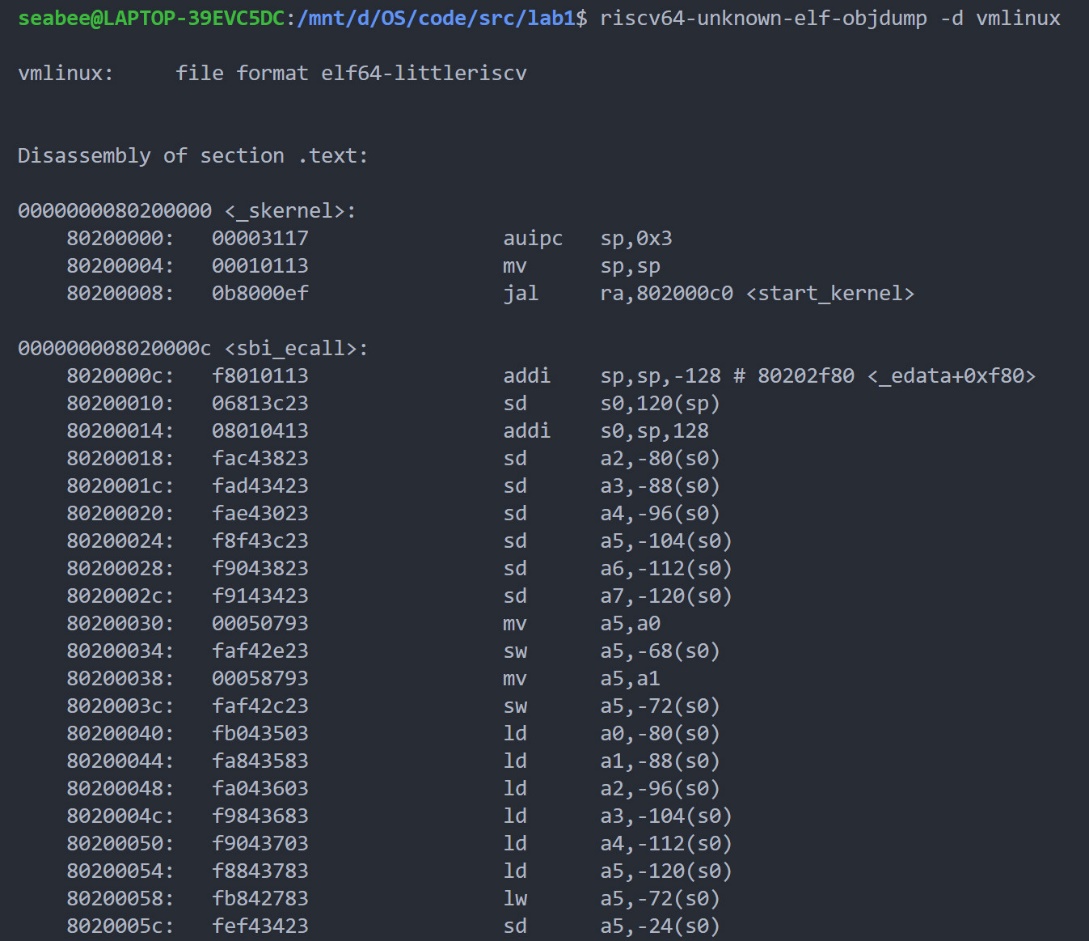


* 使用objump命令

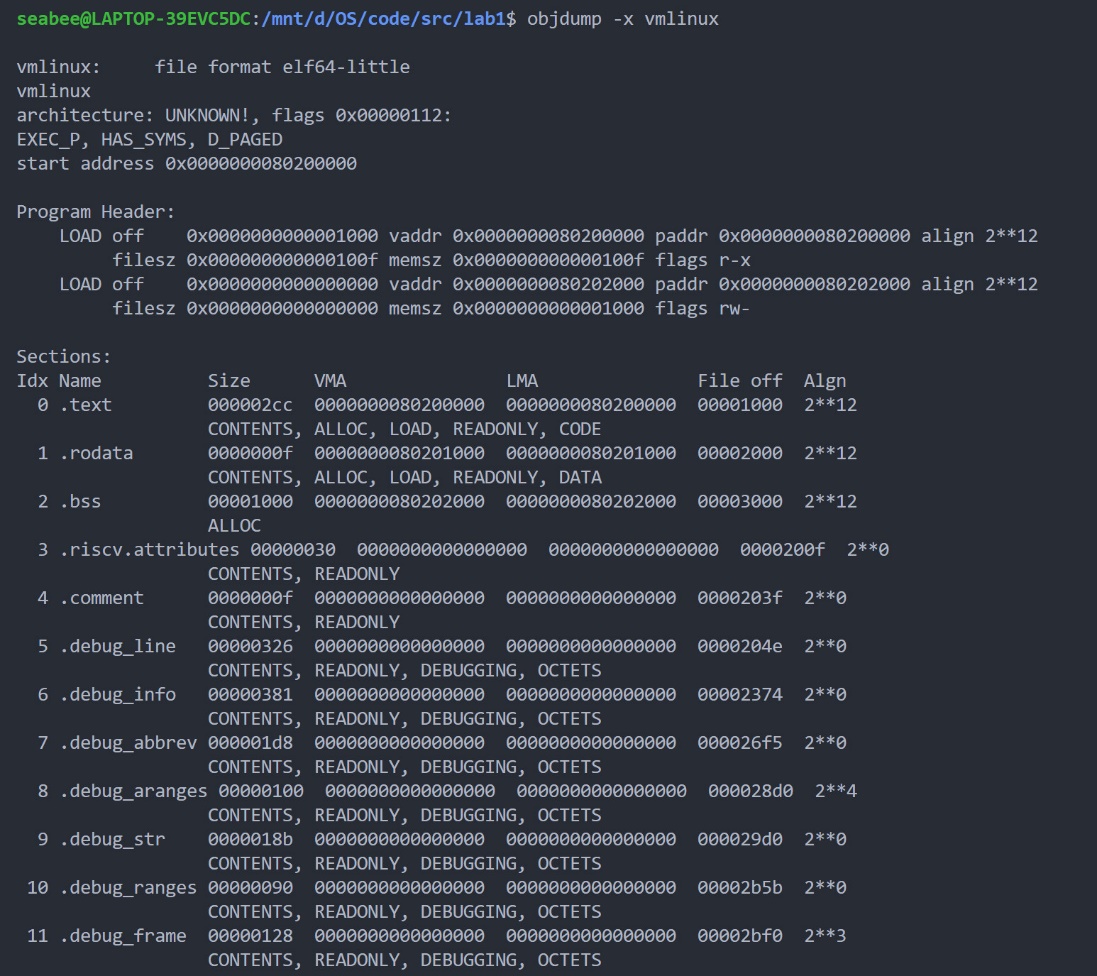
分别使用objdump -a和objdump -d命令查看vmlinux文件的archive header信息，并对executable section进行反汇编（因为架构未知而失败）



重新使用riscv64-unknown-elf-objdump -d命令对vmlinux文件的excutable section部分进行反汇编，执行成功，部分结果如下：



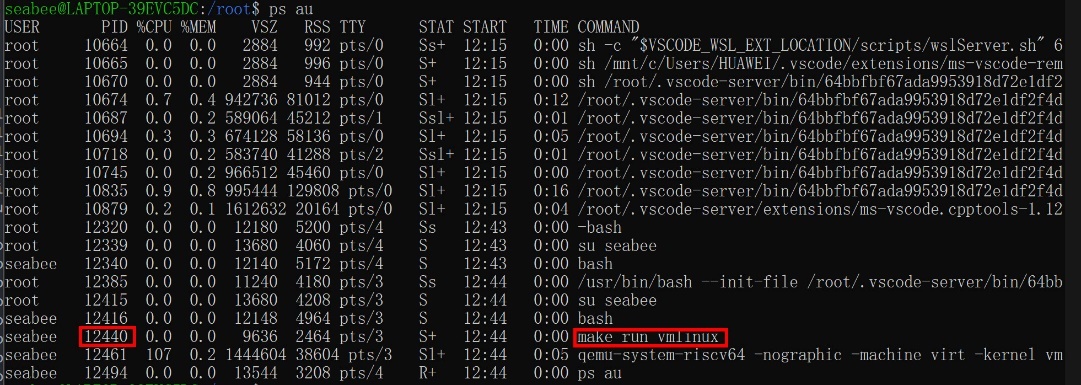
使用objdump -x显示vmlinux的所有header信息，部分结果如下：



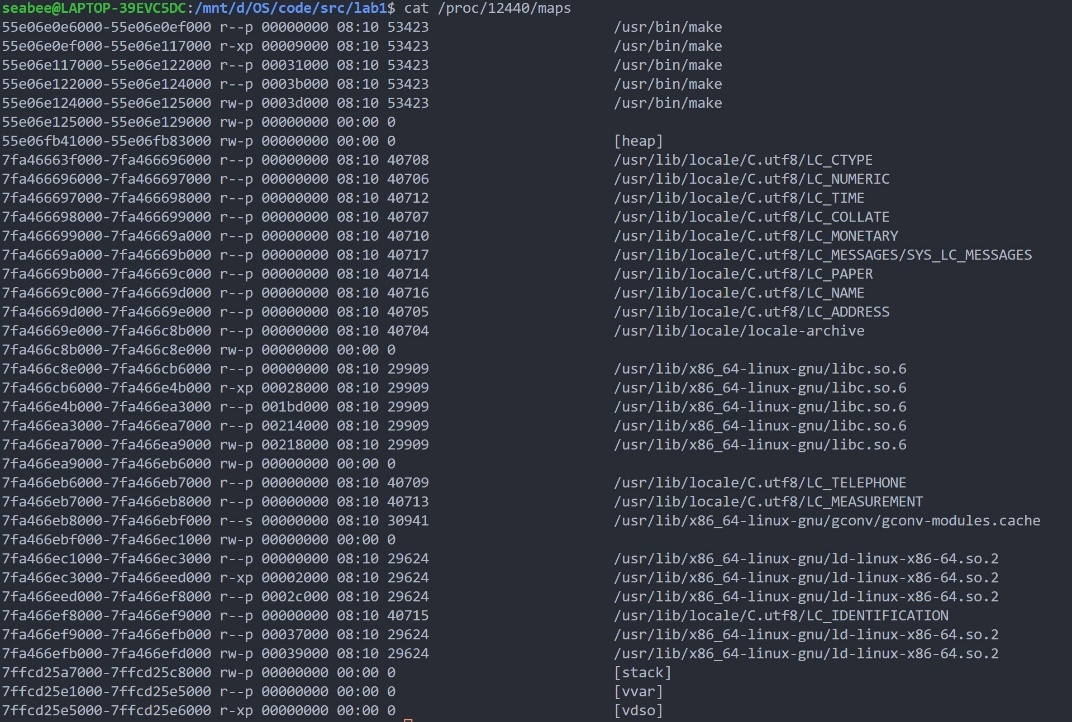
* Run an ELF file and cat /proc/PID/maps to give its memory layout：
  1. 使用make run命令运行ELF文件vmlinux



* 1. 使用ps au命令查看进程id



* 1. 使用cat /proc/PID/maps查看maps信息



**5 心得体会**

在本次实验中，我着重学习了内联汇编的相关知识与Makefile的书写规则，并且感觉十分神奇：因为之前都是使用IDE，或者在命令行手动输入编译命令。按照教程自主编写完Makefile并且成功跑起来的时候感觉自己在某种程度上变强了。但是对比工程中的参考文件还是发现自己写的过于简陋，会在后续的学习过程中进行深入学习和改进。内联汇编的知识让我想起上学期写嵌入式SQL语句的痛苦经历，但在反复查看文档后还是成功编写，也发现了属于嵌入式开发的更多可能性。此外，进行思考题实验的过程也让我更加深入的了解了linux系统下不同文件的本质。