浙江大学

本科实验报告

RV64 内核引导

课程名称：	操作系统
姓　　名：	陈杰伟
学　　院：	地球科学学院
专　　业：	地理信息科学
学　　号：	3200101205
指导老师：	寿黎但

2022 年 10 月 8 日

浙江大学实验报告

专业：地理信息科学
姓名：陈杰伟
学号：3200101205
日期：2022 年 10 月 8 日
地点：玉泉曹光彪-西 503

课程名称：操作系统　　指导老师：寿黎但　　成绩：
实验名称：RV64 内核引导　　实验类型：编程实验　　同组学生姓名：无

一、 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。

二、 实验环境

Ubuntu 20.04

三、 实验步骤

1. 编写 head.S

head.S 为程序分配程序栈的入口，并且跳转至本程序的入口函数（start_kernel）
在 lds 中，链接器已经指明了程序的入口为 start 标签，kernel 代码起始位置为 0x80200000。

程序的入口

```
1    /* 程序入口 */
2    ENTRY( _start )
3
4    /* kernel代码起始位置 */
5    BASE_ADDR = 0x80200000;
```

所以程序开始时，会运行至 start 标签处，在 start 标签处编写 RISC-V 汇编如下，代码意义在注释中说明

设置栈指针并跳转到主函数

```
1    .extern start_kernel #外部引入入口函数start_kernel
2
3    .section .text.entry #指示改部分是text部分的entry
4    .globl _start #定义start为全局标签
5_start:
6    la sp, boot_stack_top #将栈指针寄存器指向栈顶地址
7    j start_kernel #跳跃到函数入口的地址
```

这段汇编代码使栈指针寄存器指向栈顶，并且跳到入口函数的地址
随后为 boot_stack（即 bss 部分的 stack）分配空间

分配栈空间

```
    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 0x4000 #分配栈空间4k

    .globl boot_stack_top
boot_stack_top:
```

2. 完善 Makefile 脚本

完善 lib 文件夹中 makefile 脚本如下

makefile

```
    C_SRC     = $(sort $(wildcard *.c)) #获取当前目录所有c文件写入C_SRC变量，wildcard寻找通配符，
        sort以首字母排序
    OBJ      = $(patsubst %.c,%.o,$(C_SRC)) #将c文件后缀一一替换成o文件后缀写入OBJ变量

    all:$(OBJ) #指定输出目标为OBJ变量文件
    %.o:%.c
        ${GCC} ${CFLAG变量} -c $< #调用GCC编译器编译，中在上一级makefile中GCC编译器写入GCC变量，参
            数写入CFLAG变量
    clean:
        $(shell rm *.o 2>/dev/null) #使用rm命令进行clean
```

编译成功的终端输出如图 1

输出的 vmliunx 如图 2

3. 补充 sbi.c

sbi_ecall 函数的功能是利用 ecall 指令调用 OpenSBI 接口的功能，实现字符输入输出等功能

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。

sbi

```
    register unsigned int a0 = 0, a1 = 0;
  __asm__ volatile(
    "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, %[arg4]\n"
    "mv a5, %[arg5]\n"

    :

    : [ext] "r"(ext),
```

```
jwimd@jwimd-ROG-PC-Linux:~/Study/Operating_System/Lab/Lab1/lab1$ make
make -C lib all
make[1]: 进入目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/lib”
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--
gc-sections -g  -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/include -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/include -c print.c
make[1]: 离开目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/lib”
make -C init all
make[1]: 进入目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/init”
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--
gc-sections -g  -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/include -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/include -c main.c
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--
gc-sections -g  -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/include -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/include -c test.c
make[1]: 离开目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/init”
make -C arch/riscv all
make[1]: 进入目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv”
make -C kernel all
make[2]: 进入目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/kernel”
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,-
-gc-sections -g  -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/include -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/include -c head.S
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,-
-gc-sections -g  -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/include -I /home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/include -c sbi.c
make[2]: 离开目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv/kernel”
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o -o ../../vmlinux
riscv64-linux-gnu-objcopy -O binary ../../vmlinux ./boot/Image
nm ../../vmlinux >  ../../System.map
make[1]: 离开目录“/home/jwimd/Study/Operating_System/Lab/Lab1/lab1/arch/riscv”

Build Finished OK
```

图 1: 编译内核成功

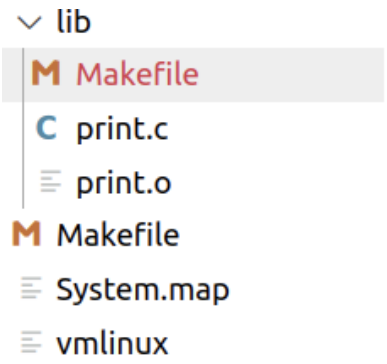

图 2: vmlinux

```
      [fid] "r"(fid),
      [arg0] "r"(arg0),
      [arg1] "r"(arg1),
      [arg2] "r"(arg2),
      [arg3] "r"(arg3),
      [arg4] "r"(arg4),
      [arg5] "r"(arg5)

    : "memory");
  //将变量值写入寄存器
```

a0-a7 是 RISC-V 标注内专用的保存函数形参的寄存器

2. 使用 ecall 指令。ecall 之后系统会进入 M 模式，之后 OpenSBI 根据我们传入的参数调用硬件完成相关的指令，然后将返回值存入寄存器 a0 和 a1 中，其中 a0 为 error code，a1 为返回值，用 sbiret 来接受这两个返回值作为 c 的返回值

ecall

```
    __asm__ volatile(
    "ecall" //调用ecall指令

    : [a0] "=r"(a0), [a1] "=r"(a1) //结果写入变量

    :

    : "memory");

  struct sbiret return_value;
  return_value.error = a0;
  return_value.value = a1;

  return return_value;
```

在 start_kernel 函数中加入

将 rootfs copy 到 Linux 源代码根目录，cd 到该目录然后执行

测试 sbi

```
  sbi_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0) //打印字符0
```

并使用 make run 运行，输出结果如图 3

证明该函数功能正常

4. puts() 和 puti()

puts() 函数的实现思路是反复调用 sbi 打印命令循环打印字符串中每一个字符直到读到 0

puts

```
  void puts(char *s) {
     while(*s!='\0')
     {
         uint64 char_code = *s;                  //获取字符编码
```

```
        sbi_ecall(0x1, 0x0, char_code, 0, 0, 0, 0, 0);//调用打印字符ecall
        s++;
    }

    return;
}
```

puti() 的实现思路是将 int 转化为 char* 然后调用 puts() 打印

puti

```
void puti(int x) {
    char s[int_max] = {0};//分配字符串空间
    int_to_str(x, s);
    puts(s);

    return;
}
```

其中 int_to_str 的实现思路是首先读取正负号然后取绝对值，然后从低位到高位逐步写入字符串，然后将字符串数字部分倒转

int 转 char

```
void int_to_str(int num, char *str)
{
    int i = 0; //指示填充str
    if (num < 0) //如果num为负数，将num变正
    {
        num = -num;
        str[i++] = '-';
    }
    //从低位到高位写入字符串
    do
    {
        str[i++] = num % 10 + code_0;
        num /= 10;
    } while (num);

    str[i] = '\0';

    int j = 0;
    if (str[0] == '-') //如果有负号，负号不用调整
    {
        j = 1; //从第二位开始调整
        ++i;  //由于有负号，所以交换的对称轴也要后移1位
    }
    //将字符串的数字部分倒转
    for (; j < i / 2; j++)
    {
        str[j] = str[j] + str[i - 1 - j];
        str[i - 1 - j] = str[j] - str[i - 1 - j];
        str[j] = str[j] - str[i - 1 - j];
```

```
30        }
31
32        return;
33    }
```

运行结果如图 4

5. 修改 defs

read_csr 的目的是从指定 csr 中读取值，具体实现如下

csr 宏

```
1  #define csr_read(csr)           \
2     ({                           \
3        register uint64 __v;      \
4        asm volatile(             \
5           "csrr %0," #csr        \
6           : "=r"(_v)             \
7           :                      \
8           : "memory");           \
9        __v;                      \
10    })
```

然后在 start_kernel 里加入

验证

```
1     csr_write(sstatus, 24578);
2     puti(csr_read(sstatus));
```

运行后读取值如图 5
写入的值和读取值一致证明 csr_read 宏的功能正确

四、 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?

RISC-V 的 calling convention 简单总结是对 RISC-V 汇编语言中函数调用的寄存器使用规范，简单总结如图

在编写 RISC-V 汇编程序时，一定要遵守该寄存器使用方式和命名，这样程序才有足够的共用性，不会产生数据被覆盖的情况。

可以看到寄存器分为两种 caller-saved 类型和 callee-saved 类型

caller-saved：可以从该类寄存器中读取数据，写入数据，该寄存器所存储的值在函数调用过程中可能发生改变，比如参数寄存器就属于该类型

callee-saved：该类寄存器中存储的值在函数调用过程中不能改变，比如栈指针寄存器

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

如图 7

3. 用 csr_read 宏读取 sstatus 寄存器的值，对照 RISC-V 手册解释其含义（截图）。

将 start_kernel 函数改写如下

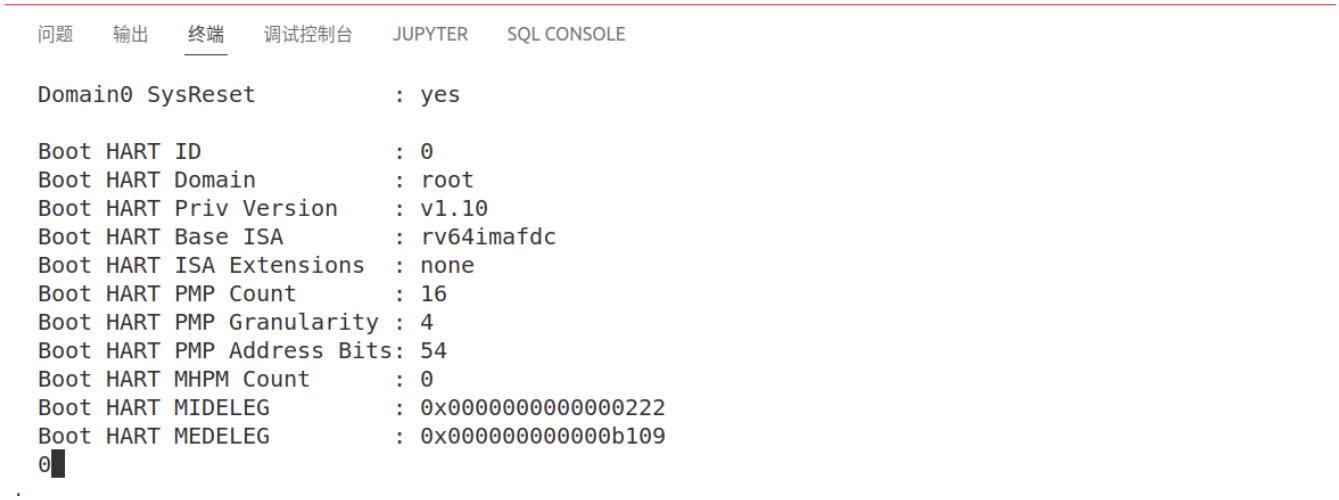

图 3: sbi 运行

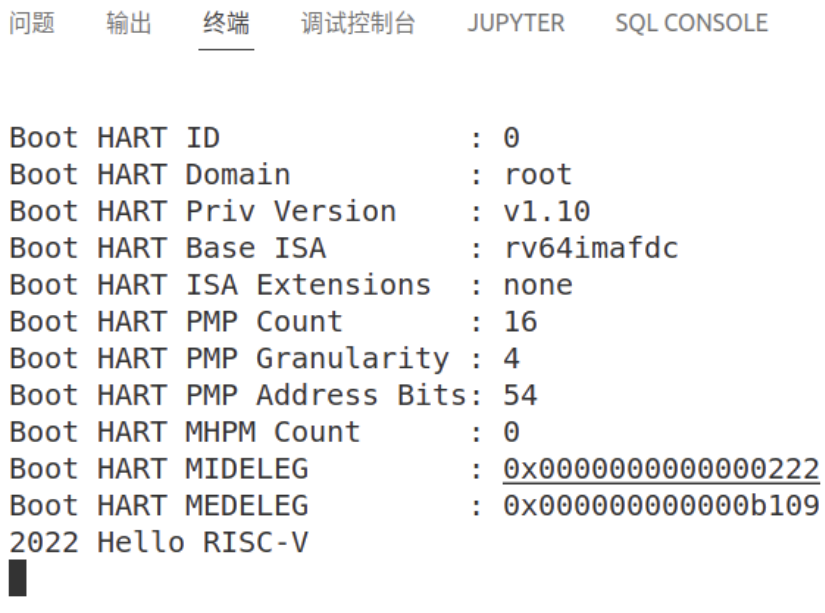

图 4: 打印函数运行结果

```
Boot HART Priv Version    : v1.10
Boot HART Base ISA        : rv64imafdc
Boot HART ISA Extensions  : none
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2022 Hello RISC-V
24578
```

图 5: 读 csr

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

图 6: calling convention

验证

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

    extern void test();

    int start_kernel()
    {

        puti(2022);
        puts("Hello RISC-V\n");

        puti(csr_read(sstatus));

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

        return 0;
    }
```

make run 运行结果如图 8
状态值 24576 转化为 2 进制
0110 0000 0000 0000
翻阅手册获取每一位值的含义如图 9
SPP 位指示进入管理员模式之前执行的特权级别，当前是 0 代表用户模式
SIE 位启用或禁用监管者模式下的所有中断，当前是禁用状态
SPIE 位指示在进入管理模式之前是否启用了管理中断。当前未启用
UIE 位启用或禁用用户模式中断。当前未启用
4. 用 csr_write 宏向 sscratch 寄存器写入数据，并验证是否写入成功（截图）。
编写 start_kernel

验证

```c
    int start_kernel()
    {

        puti(2022);
        puts("Hello RISC-V\n");


        puti(csr_read(sscratch));
        puts("\n");
        csr_write(sscratch, 1);
        puti(csr_read(sscratch));

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

        return 0;
    }
```

```
jwimd@jwimd-ROG-PC-Linux:~/Study/Operating_System/Lab/Lab1/lab1$ nm vmlinux
0000000080200000 A BASE_ADDR
0000000080203000 B boot_stack
0000000080207000 B boot_stack_top
0000000080207000 B _ebss
0000000080202000 D _edata
0000000080207000 B _ekernel
000000008020100f R _erodata
000000008020044c T _etext
0000000080202000 d _GLOBAL_OFFSET_TABLE_
0000000080200220 T int_to_str
00000000802001c4 T puti
0000000080200150 T puts
000000008020000c T sbi_ecall
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200100 T start_kernel
0000000080200000 T _stext
0000000080200140 T test
```

图 7: System.map

问题 输出 终端 调试控制台 JUPYTER SQL CONSOLE

```
Boot HART Priv Version    : v1.10
Boot HART Base ISA        : rv64imafdc
Boot HART ISA Extensions  : none
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2022 Hello RISC-V
24576
```

图 8: 读取 sstatus

运行结果如图 10

写入成功

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

使用如下 bash 命令获得编译中间产物 sys.i（如图 11）

获取 sys.i

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i
```

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.

利用 vscode，查找后缀名为.tbl 的文件，如图 12

鉴于表太长，我将表的 makedown 放置在报告的 syscall 文件夹里

ARM32: arch/arm/tools/syscall.tbl（见 ARM.md)

x86(32 bit):arch/x86/entry/syscalls/syscall_32.tbl（见 x86(32 bit).md)

x86_64:arch/x86/entry/syscalls/syscall_64.tbl（见 x86_64.md)

risc-v 获取 table

```
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-arch/riscv/kernel/syscall_table.i # 64.bit

make ARCH=riscv CROSS_COMPILE=riscv32-linux-gnu-arch/riscv/kernel/syscall_table.i # 32.bit
```

由于 RISC-V 架构的源文件里没有 tbl 文件，我通过编译中间产物能够大概获得 system_call_table，但还没有想出好方法来列表，读取的列表见 RISC-V.md

获取 sys_call_table.i 的方法

sys_call_table.i 附在 syscall 文件夹内

7.Explain what is ELF file? Try readelf and objdump 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（Executable Linkable Format）是一种 Unix-like 系统上的二进制文件格式标准，见图 13

ELF 大致分为图 14 所示的几层

我在 elf 文件夹内编写了一个简单的 RISC-V 汇编程序，执行寄存器加法

add

```
  .text
  .global _start

_start:
  li t1, 1
  li t2, 2
  add t0, t1, t2


stop:
  j stop

  .end
```

XLEN-1	XLEN-2	20	19	18	17	
SD	0		MXR	SUM	0	
1	XLEN-21		1	1	1	

16 15	14 13	12 9	8	7 6	5	4	3 2	1	0
XS[1:0]	FS[1:0]	0	SPP	0	SPIE	UPIE	0	SIE	UIE
2	2	4	1	2	1	1	2	1	1

图 9: sstatus 含义

问题 输出 终端 调试控制台 JUPYTER SQL CONSOLE

```
Boot HART Base ISA          : rv64imafdc
Boot HART ISA Extensions    : none
Boot HART PMP Count         : 16
Boot HART PMP Granularity   : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count        : 0
Boot HART MIDELEG           : 0x0000000000000222
Boot HART MEDELEG           : 0x000000000000b109
2022 Hello RISC-V
0
1
```

图 10: 写入 sscratch

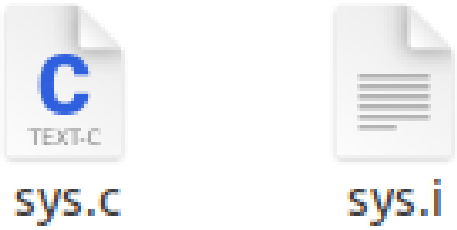

图 11: sys.i

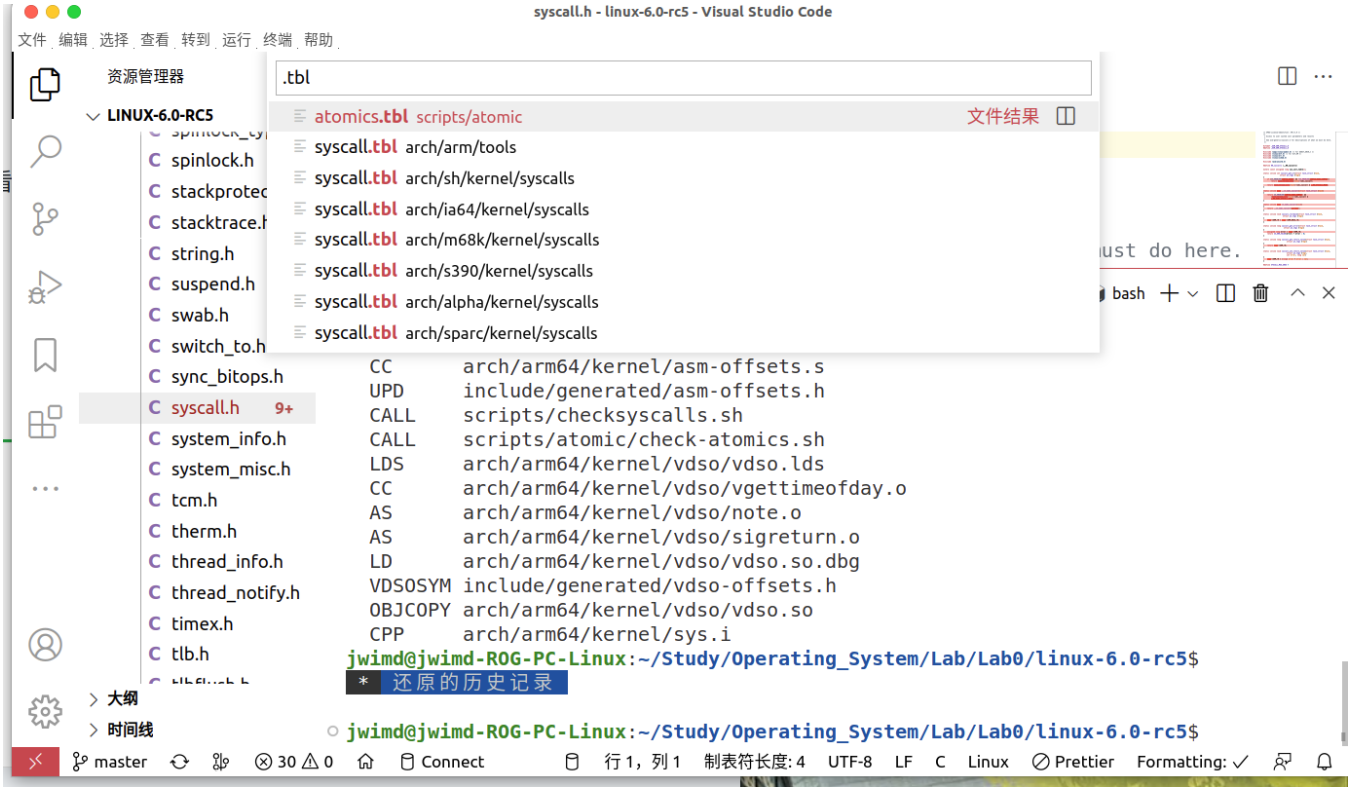

图 12: vscode 查找

ELF 文件类型	说明	实例
可重定位文件（Relocatable File）	内容包含了代码和数据，可以被链接成可执行文件或共享目标文件。	Linux 上的 .o文件
可执行文件（Executable File）	可以直接执行的程序	Linux 上的a.out
共享目标文件（Shared Object File）	内容包含了代码和数据，可以作为链接器的输入，在链接阶段和其他的 Relocatable File或者 Shared Object File一起链接成新的 Object File；或者在运行阶段，作为动态链接器的输入，和 Executable File 结合，作为进程的一部分来运行	Linux 上的 .so
核心转储文件（Core Dump File）	进程意外终止时，系统可以将该进程的部分内容和终止时的其他状态信息保存到该文件中以供调试分析。	Linux 上的core 文件

图 13: elf

然后在同一目录下编写 makefile 文件指定输出为 elf。同时指定 readelf 和 objdump 的伪指令

makefile

```
CROSS_COMPILE = riscv64-linux-gnu-
CFLAGS = -nostdlib

CC = ${CROSS_COMPILE}gcc
OBJDUMP = ${CROSS_COMPILE}objdump
READELF = ${CROSS_COMPILE}readelf

EXEC = test
SRC = ${EXEC}.s

.DEFAULT_GOAL := all
all:
    @${CC} ${CFLAGS} ${SRC} -o ${EXEC}.elf

.PHONY : objdump
objdump: all
    @${OBJDUMP} -S ${EXEC}.elf

.PHONY : readelf
readelf: all
    @${READELF} -S ${EXEC}.elf

.PHONY : clean
clean:
    rm -rf *.o *.elf
```

执行 readelf 结果如图 15
执行 objdump 结果如图 16
使用 QEMU 运行 elf，改写 makefile

makefile

```
CROSS_COMPILE = riscv64-linux-gnu-
CFLAGS = -nostdlib

CC = ${CROSS_COMPILE}gcc
OBJDUMP = ${CROSS_COMPILE}objdump
READELF = ${CROSS_COMPILE}readelf

QEMU = qemu-system-riscv32
QFLAGS = -nographic -smp 1 -machine virt -bios none

EXEC = test
SRC = ${EXEC}.s

.DEFAULT_GOAL := all
all:
    @${CC} ${CFLAGS} ${SRC} -o ${EXEC}.elf
```

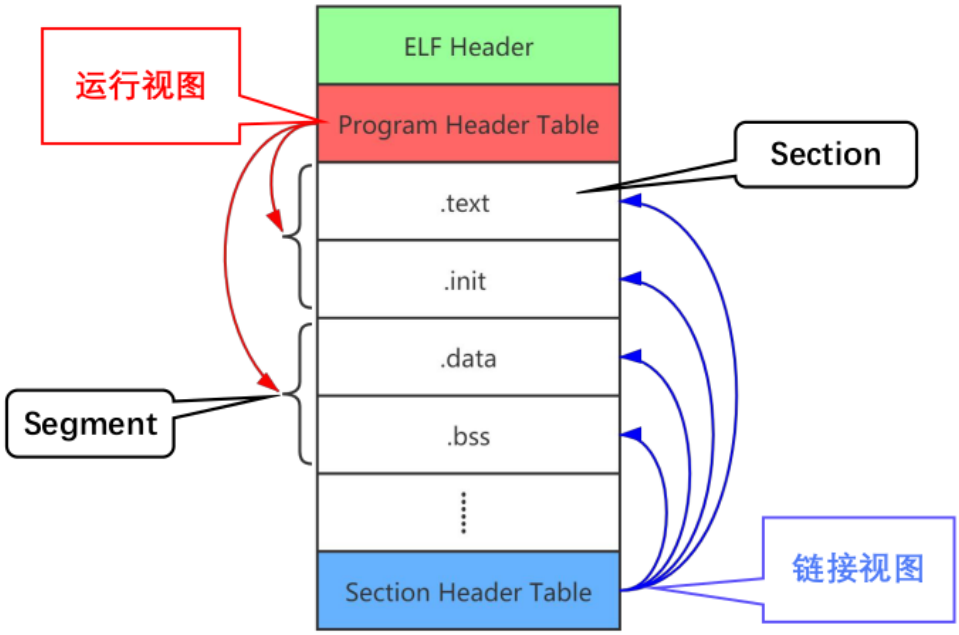

图 14: elf 分层

```
jwimd@jwimd-ROG-PC-Linux:~/Study/Operating_System/Lab/Lab1/Report/elf$ make read
elf
There are 12 section headers, starting at offset 0x1338:

节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         00000000000001c8  000001c8
       0000000000000021  0000000000000000   A       0     0     1
  [ 2] .note.gnu.build-i NOTE             00000000000001ec  000001ec
       0000000000000024  0000000000000000   A       0     0     4
  [ 3] .gnu.hash         GNU_HASH         0000000000000210  00000210
       000000000000001c  0000000000000000   A       4     0     8
  [ 4] .dynsym           DYNSYM           0000000000000230  00000230
       0000000000000030  0000000000000018   A       5     2     8
  [ 5] .dynstr           STRTAB           0000000000000260  00000260
       0000000000000001  0000000000000000   A       0     0     1
  [ 6] .text             PROGBITS         0000000000000262  00000262
       000000000000000a  0000000000000000  AX       0     0     2
  [ 7] .dynamic          DYNAMIC          0000000000001ef0  00000ef0
       0000000000000110  0000000000000010  WA       5     0     8
  [ 8] .got              PROGBITS         0000000000002000  00001000
       0000000000000008  0000000000000008  WA       0     0     8
  [ 9] .symtab           SYMTAB           0000000000000000  00001008
       0000000000000228  0000000000000018          10    15     8
  [10] .strtab           STRTAB           0000000000000000  00001230
       00000000000000a4  0000000000000000           0     0     1
  [11] .shstrtab         STRTAB           0000000000000000  000012d4
       0000000000000064  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

图 15: readelf 结果

```
17
18    .PHONY : objdump
19    objdump: all
20       @${OBJDUMP} -S ${EXEC}.elf
21
22    .PHONY : run
23    run: all
24       @${QEMU} ${QFLAGS} -kernel ./${EXEC}.elf
25
26    .PHONY : readelf
27    readelf: all
28       @${READELF} -S ${EXEC}.elf
29
30    .PHONY : map
31    map: all
32       nm ${EXEC}.elf
33
34    .PHONY : clean
35    clean:
36       rm -rf *.o *.elf
```

执行 make run 可以运行一次寄存器加法，可以使用 gdb 查看
执行 make map 读取内存结构如图 17

```
jwimd@jwimd-ROG-PC-Linux:~/Study/Operating_System/Lab/Lab1/Report/elf$ make objdump

test.elf:     文件格式 elf64-littleriscv


Disassembly of section .text:

0000000000000262 <_start>:
 262:   4305                    li      t1,1
 264:   4389                    li      t2,2
 266:   007302b3                add     t0,t1,t2

000000000000026a <stop>:
 26a:   a001                    j       26a <stop>
```

图 16: objdump 结果

```
jwimd@jwimd-ROG-PC-Linux:~/Study/Operating_System/Lab/Lab1/Report/elf$ make map
nm test.elf
0000000000002008 D __BSS_END__
0000000000002008 D __bss_start
0000000000002000 D __DATA_BEGIN__
00000000000001ef0 a _DYNAMIC
0000000000002008 D _edata
0000000000002008 D _end
0000000000002000 a _GLOBAL_OFFSET_TABLE_
0000000000002800 A __global_pointer$
0000000000000270 a _PROCEDURE_LINKAGE_TABLE_
0000000000002008 D __SDATA_BEGIN__
0000000000000262 T _start
000000000000026a t stop
```

图 17: map