

Lab1 RV64 内核

赵伊蕾 学号：3200104866

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. 实验任务

2.1 完成 head.s 程序

arch/riscv/kernel/head.s 的作用是为C函数设置程序栈(4KB)，并将栈放在 .bss.stack 段，然后通过跳转指令跳转到main.c中的 start_kernel。

```
_start:  
# -----  
# - your code here -  
# set up the stack of 4K  
la sp, boot_stack_top  
call start_kernel  
# jump to start_kernel  
# -----  
  
.section .bss.stack  
.globl boot_stack
```

2.2 完善Makefile 脚本

lib/Makefile 需要填充，用来编译 lib 文件夹下的 print.c 文件

这部分的 makefile 文件参考 init 文件夹下的 Makefile，只不过需要编译的文件不是 main.c 而是 print.c

```
C_SRC      = $(sort $(wildcard *.c))  
OBJ       = $(patsubst %.c,%.o,$(C_SRC))  
  
file = print.o  
all:$ (OBJ)  
  
%.o: %.c  
 ${GCC} ${CFLAGS} -c $<  
clean:  
 $(shell rm *.* 2>/dev/null)
```

2.3 补充sbi.c文件

这里需要写内联汇编，使得最后实现 `sbi_ecall` 函数的实现。

在 `sbi.c` 文件中要完成三件事情：

1. 将函数的参数放入对应的寄存器中
2. 使用ecall指令使系统进入M模式
3. 接收返回值

```
struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
                         uint64 arg1, uint64 arg2,
                         uint64 arg3, uint64 arg4,
                         uint64 arg5)

{
    struct sbiret var;
    //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。ecall 之后系统会进入 M 模式, 之后 OpenSBI 会完成相关操作。
    //3. OpenSBI 的返回结果会存放在寄存器 a0 , a1 中, 其中 a0 为 error code, a1 为返回值, 我们
    用 sbiret 来接受这两个返回值
    register uint64 a0 asm ("a0") = (uint64)(arg0);
    register uint64 a1 asm ("a1") = (uint64)(arg1);
    register uint64 a2 asm ("a2") = (uint64)(arg2);
    register uint64 a3 asm ("a3") = (uint64)(arg3);
    register uint64 a4 asm ("a4") = (uint64)(arg4);
    register uint64 a5 asm ("a5") = (uint64)(arg5);
    register uint64 a6 asm ("a6") = (uint64)(fid);
    register uint64 a7 asm ("a7") = (uint64)(ext);

    asm volatile ("ecall"
                 : "+r" (a0), "+r" (a1)
                 : "r" (a2), "r" (a3), "r" (a4), "r" (a5), "r" (a6), "r" (a7)
                 : "memory");
    var.error = a0;
    var.value = a1;
    return var;
}
```

2.4 完成puts()与puti()函数

这两个函数的思路其实都是直接打印字符串或者数字，但是不是通过调用系统函数 `printf`，而是通过我们这里实现的 `sbi_ecall` 来输出字符。所以两个函数的思路就是一位一位输出字符。

经过测试，这些函数具有正确性，可以处理一些极端的情况。比如 `puti(int x)` 函数可以正确输出0、负数等比较特殊的值。

```
void puts(char *s) {
    // unimplemented
    int i = 0;
```

```

while(s[i++]!='\0')
    sbi_ecall(0x1, 0x0, s[i], 0,0,0,0,0);
//sbi_ecall的第三个参数是输出字符的asc码, 只要字符不是'\0'就可以输出
}

void puti(int x) {
    // unimplemented
    if(x < 0){
        char c = '-';
        sbi_ecall(0x1, 0x0, c, 0,0,0,0,0);
        x = x * (-1);
    }
    int a = x, cnt = 1, num = 1;
    while(x >= cnt*10) {
        cnt *= 10; //整数位数
        num++;
        a /= 10;
    }
    for(int i = 0; i<num; i++){
        int t = x / cnt;
        sbi_ecall(0x1, 0x0, t+48, 0,0,0,0,0);
        x %= cnt;
        cnt /= 10;
    }
}

```

2.5 修改 deft.h 头文件使得能够内联汇编

`csr_read` 宏定义的格式类似于程序中给出的 `csr_write`，不过不同的是 `"=r"` 表示将汇编指令的值放到相应的变量当中，而 `"r"` 是将变量的值放到寄存器中，所以两个宏定义有一点不太一样。另外 `csr_read` 还需要返回值，也就是最后的 `__v`。

```

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

```

如此定义好 `csr_read` 和 `csr_write` 之后，就可以像宏定义一样来调用使用它们，在思考题第三第四题中将展开描述。

3 实验结果

3.1 成功编译

在 lab0 编译好的环境下进行 lab1 的开发，然后通过 make 指令进行编译，终端提示编译通过即可。

```
Build Finished OK
root@c30e77226e10:/os22fall-stu/src/lab1# ls
Makefile System.map arch include init lib vmlinux
```

3.2 运行程序

使用 make run 使刚编译好的程序运行起来，也就是运行我们 main 文件。

我们可以看到 main.c 文件主要是有一个 start_kernel 函数，也就是我们之前通过低级汇编语言跳转的函数，然后函数里调用了两个输出的函数。所以最终实现的效果将会是输出 2021Hello RISC-V

```
extern void test();

int start_kernel() {
    puti(2021);
    puts(" Hello RISC-V\n");
    test(); // DO NOT DELETE !!

    return 0;
}
```

最后一行输出 2021Hello RISC-V，验证之前的猜想，程序完成！

```
Domain0 Name          : root
Domain0 Boot HART      : 0
Domain0 HARTs          : 0*
Domain0 Region00       : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01       : 0x0000000000000000-0xffffffffffff (R,W,X)
Domain0 Next Address   : 0x0000000080200000
Domain0 Next Arg1     : 0x0000000087000000
Domain0 Next Mode      : S-mode
Domain0 SysReset       : yes

Boot HART ID          : 0
Boot HART Domain      : root
Boot HART ISA          : rv64imafdcu
Boot HART Features    : scounteren,mcounteren,time
Boot HART PMP Count    : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count   : 0
Boot HART MHPM Count   : 0
Boot HART MIDELEG      : 0x0000000000000022
Boot HART MEDELEG      : 0x0000000000000b109
2021Hello RISC-V
```

4. 思考题

4.1 请总结一下 RISC-V 的 calling convention，并解释 Caller / Callee Saved Register 有什么区别？

RISC-V的手册中有写道，函数调用过程通常分为 6 个阶段：

1. 将参数存储到被调用的函数可以访问到的位置；
2. 跳转到函数起始位置；
3. 获取函数需要的局部存储资源，按需保存寄存器(callee saved registers)；
4. 执行函数中的指令；
5. 将返回值存储到调用者能够访问到的位置，恢复寄存器(callee saved registers)，释放局部存储资源；
6. 返回调用函数的位置。

Caller saved register是指由用户自己来维护的寄存器，这些寄存器内的数据在函数被调用的时候值可能会被修改，所以用户最好在使用前先把值放到栈中然后再调用寄存器；Callee saved register是在函数调用过程中，系统会自动保存的寄存器，用户可以不需要考虑函数调用之后寄存器的值被修改变不回来的情况。

4.2 使用 nm vmlinux 命令查看 vmlinux.lds 中自定义符号的值

```
Makefile System.map arch include init lib vmlinux
root@c30e77226e10:/os22fall-stu/src/lab1# cd nm vmlinux
bash: cd: too many arguments
root@c30e77226e10:/os22fall-stu/src/lab1# nm vmlinux
0000000080200000 A _BASE_ADDR
0000000080206000 B _ebss
0000000080202000 R _edata
0000000080206000 B _ekernel
000000008020100f R _erodata
000000008020001c8 T _etext
0000000080202000 B _sbss
0000000080202000 R _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080202000 B _boot_stack
0000000080206000 B _boot_stack_top
00000000802000d4 T _puti
0000000080200078 T _puts
000000008020000c T _sbi_ecall
0000000080200044 T _start_kernel
0000000080200074 T _test
```

4.3 用 `read_csr` 宏读取 `sstatus` 寄存器的值

4.4 用 `write_csr` 宏向 `sscratch` 寄存器写入数据

这两个小实验我放在一起完成。在 `start_kernel` 函数里进行 `csr_read` 和 `csr_write` 操作，调用这两个宏定义。`csr_read` 宏可以通过输入寄存器的名字返回寄存器里存的内容。我们通过之前写过的 `puti(int x)` 函数把内容打印出来，得到的结果是：24567。然后再尝试用 `csr_write` 对寄存器 `sscratch` 进行改写，传入 `value = 100`，然后再把 `sscratch` 的值读出来并打印，结果如下图所示。这表明了 `csr_write` 宏定义对寄存器写入成功。

```

int start_kernel() {
    puti(2021);
    puts(" Hello RISC-V\n");

    puti(csr_read(ssstatus));
    puts(" \n");
    csr_write(sscratch,100);
    puti(csr_read(sscratch));

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

return ;
}

```

Boot HART MIDELEG	: 0x0000000000000000222
Boot HART MEDELEG	: 0x0000000000000000b109
2021Hello RISC-V	
24576	
100	

ssstatus: Supervised status register

31	30												20	19	18	17
SD		WPRI											MXR	SUM	WPRI	
1		11											1	1	1	1
16	15	14	13	12	11	10	9	8	7	6	5	4	2	1	0	
XS[1:0]	FS[1:0]	WPRI	VS[1:0]	SPP	WPRI	UBE	SPIE	WPRI	SIE	WPRI						
2	2	2	2	2	1	1	1	1	3	1	1	1	1	1	1	1

Figure 4.1: Supervisor-mode status register (**ssstatus**) when SXLEN=32.

将ssstatus的内容转成二进制：10111111110111，对照上面的表格我们可以得出以下信息

WPRI:1	SIE:1	WPRI:101	SPIE:1	UBE:1	WPRI:1
SSP:1	VS:11	WPRI:11	FS: 01		

SSP为1，表示是Supervisor-mode；SIE为1表示开启中断；SPIE为1表示trap中禁止S-mode的中断；UBE为1表示大端字节序。

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

我们可以利用交叉编译手段来得到kernel中`sys.c`的预处理产物`sys.i`，步骤如下：

```

# 先 config
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig

# 然后指定要生成的文件
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- path/to/file/xxx.i

```

然后就会得到编译完成的`sys.i`文件。

```

root@c30e77226e10:/os22fall-stu/src/lab0/linux# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
root@c30e77226e10:/os22fall-stu/src/lab0/linux# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i
    SYNC      include/config/auto.conf.cmd
  WRAP     arch/arm64/include/generated/uapi/asm/kvm_para.h
  WRAP     arch/arm64/include/generated/uapi/asm/errno.h
  WRAP     arch/arm64/include/generated/uapi/asm/ioctl.h
  WRAP     arch/arm64/include/generated/uapi/asm/ioctls.h
  WRAP     arch/arm64/include/generated/uapi/asm/ipcbuf.h
  WRAP     arch/arm64/include/generated/uapi/asm/msgbuf.h
  WRAP     arch/arm64/include/generated/uapi/asm/poll.h
  WRAP     arch/arm64/include/generated/uapi/asm/resource.h
  WRAP     arch/arm64/include/generated/uapi/asm/sembuf.h
  WRAP     arch/arm64/include/generated/uapi/asm/shmbuf.h
  WRAP     arch/arm64/include/generated/uapi/asm/siginfo.h

          AS      arch/arm64/kernel/vdso/stgreturn.o
          LD      arch/arm64/kernel/vdso/vdso.so.dbg
          VDSOSYM include/generated/vdso-offsets.h
          OBJCOPY arch/arm64/kernel/vdso/vdso.so
          CPP     arch/arm64/kernel/sys.i
root@c30e77226e10:/os22fall-stu/src/lab0/linux#

```

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.

X86_32 system call table

```

root@c30e77226e10:/os22fall-stu/src/lab0/linux/arch/x86/entry/syscalls# ls
Makefile  syscall_32.tbl  syscall_64.tbl
root@c30e77226e10:/os22fall-stu/src/lab0/linux/arch/x86/entry/syscalls# cat syscall_32.tbl
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
# sys_*() system calls and compat_sys_*() compat system calls if
# IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
# parameter.
#
# The abi is always "i386" for this file.
"
```

x86_64 system call table

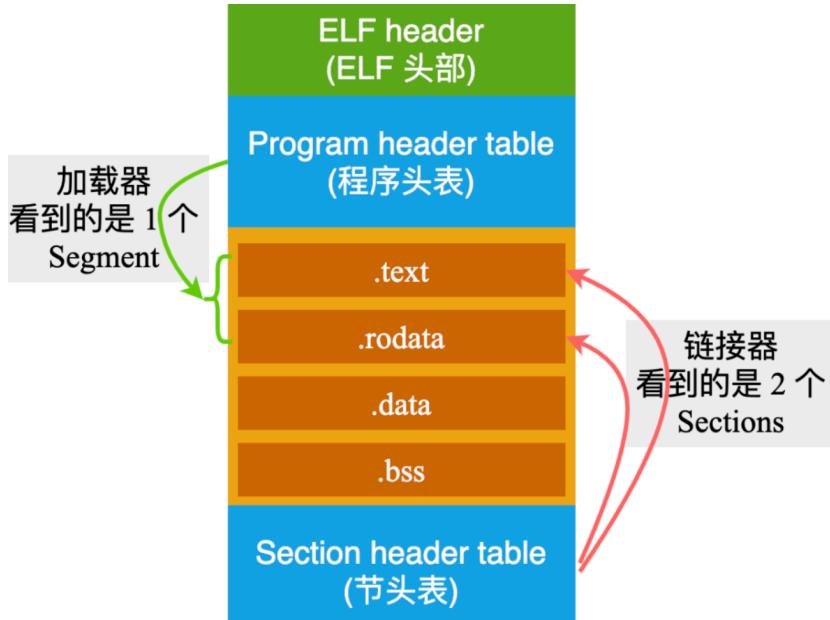
```
root@c30e77226e10:/os22fall-stu/src/lab0/linux/arch/x86/entry/syscalls# cat syscall_64.tbl
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0    common  read           sys_read
1    common  write          sys_write
2    common  open           sys_open
3    common  close          sys_close
4    common  stat           sys_newstat
```

Mips system call table

```
root@c30e77226e10:/os22fall-stu/src/lab0/linux/arch/mips/kernel/syscalls# cat syscall_o32.tbl
# SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
#
# system call numbers and entry vectors for mips
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The <abi> is always "o32" for this file.
#
0    o32    syscall        sys_syscall      sys32_syscall
1    o32    exit           sys_exit
2    o32    fork           __sys_fork
3    o32    read           sys_read
4    o32    write          sys_write
5    o32    open           sys_open         compat_sys_open
6    o32    close          sys_close
7    o32    waitpid        sys_waitpid
8    o32    creat          sys_creat
9    o32    link           sys_link
```

4.7 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全称是Executable and Linkable Format, 文件的形式如下：有ELF header、Program header table、Section header table等。



4.7.1 `readelf` 指令

我进入到 lab1 的 init 文件中，用 `readelf` 指令查看 `main.o` 文件 ELF 头的内容，结果如下：

- `readelf -h main.o` 这个指令是查看文件 ELF 头的内容

```
root@c30e77226e10:/os22fall-stu/src/lab1/init# readelf -h main.o
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: REL (Relocatable file)
  Machine: RISC-V
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 1352 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 64 (bytes)
  Number of section headers: 12
  Section header string table index: 11
```

- `readelf -S main.o` 是查看 `main.o` 文件节头表的内容

```

root@c30e77226e10:/os22fall-stu/src/lab1/init# readelf -S main.o
There are 12 section headers, starting at offset 0x548:

Section Headers:
[Nr] Name          Type      Address      Offset
     Size        EntSize   Flags Link Info Align
[ 0]             NULL       0000000000000000 00000000
                0000000000000000 0000000000000000 0 0 0
[ 1] .text         PROGBITS 0000000000000000 00000040
                0000000000000000 0000000000000000 AX 0 0 4
[ 2] .data         PROGBITS 0000000000000000 0000000000000000 00000040
                0000000000000000 0000000000000000 WA 0 0 1
[ 3] .bss          NOBITS   0000000000000000 0000000000000000 00000040
                0000000000000000 0000000000000000 WA 0 0 1
[ 4] .rodata.start_ker PROGBITS 0000000000000000 0000000000000000 00000040
                0000000000000013 0000000000000001 AMS 0 0 8
[ 5] .text.start_kerne PROGBITS 0000000000000000 0000000000000000 00000054
                0000000000000070 0000000000000000 AX 0 0 4

```

4.7.2 objdump 指令

- objdump -s main.o

-s可以查看代码段、数据段

```

root@c30e77226e10:/os22fall-stu/src/lab1/init# objdump -s main.o

main.o:      file format elf64-little

Contents of section .rodata.start_kernel.str1.8:
0000 2048656c 6c6f2052 4953432d 560a0000  Hello RISC-V...
0010 200a00          ..
Contents of section .text.start_kernel:
0000 130101ff 1305507e 23341100 97000000  .....P~#4.....
0010 e7800000 17050000 13050500 97000000  .....
0020 e7800000 73250010 1b050500 97000000  ....s%.....
0030 e7800000 17050000 13050500 97000000  .....
0040 e7800000 93074006 73900714 73250014  .....@.s...s%..
0050 1b050500 97000000 e7800000 97000000  .....
0060 e7800000 83308100 13010101 67800000  .....0.....g...
Contents of section .comment:
0000 00474343 3a202867 35393634 62356364  .GCC: (g5964b5cd
0010 37323729 2031312e 312e3000          727) 11.1.0.
Contents of section .riscv.attributes:
0000 41330000 00726973 63760001 29000000  A3...riscv...)...
0010 04100572 76363469 3270305f 6d327030  ...rv64i2p0_m2p0
0020 5f613270 305f6632 70305f64 32703000  _a2p0_f2p0_d2p0.
0030 08010a0b          .....
root@c30e77226e10:/os22fall-stu/src/lab1/init# 
```

- objdump -h main.o

该指令可以输出目标文件中节表中所包含的所有节头的信息：

```

root@c30e77226e10:/os22fall-stu/src/lab1/init# objdump -h main.o

main.o:      file format elf64-little

Sections:
Idx Name      Size    VMA          LMA          File off  Algn
 0 .text      00000000  0000000000000000  0000000000000000  00000040  2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .data      00000000  0000000000000000  0000000000000000  00000040  2**0
              CONTENTS, ALLOC, LOAD, DATA
 2 .bss       00000000  0000000000000000  0000000000000000  00000040  2**0
              ALLOC
 3 .rodata.start_kernel.str1.8 00000013  0000000000000000  0000000000000000  00000040  2**3
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .text.start_kernel 00000070  0000000000000000  0000000000000000  00000054  2**2
              CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 5 .comment    0000001c  0000000000000000  0000000000000000  000000c4  2**0
              CONTENTS, READONLY
 6 .riscv.attributes 00000034  0000000000000000  0000000000000000  000000e0  2**0
              CONTENTS, READONLY
root@c30e77226e10:/os22fall-stu/src/lab1/init# 

```

- 之前在lab0中也有使用到 `objdump` 指令，用以反汇编编译产物：

使用 `riscv64-linux-gnu-objdump` 反汇编 1 中得到的编译产物

- 通过 `cat /proc/$$/maps` 指令查看shell 的内存映射情况

可以看到 shell 进程的 `stack` 段的内存范围以及其属性为可读可写但不可执行，`p` 表示该虚拟内存段为私有的(private)，`s` 表示该虚拟内存段为共享的(shared)。

```
cat: /proc/pid/maps: No such file or directory
root@c30e77226e10:/os22fall-stu/src/lab1/init# cat /proc/$$/maps
556138c27000-556138c54000 r--p 00000000 00:36 3410904 /usr/bin/bash
556138c54000-556138d05000 r-xp 0002d000 00:36 3410904 /usr/bin/bash
556138d05000-556138d3c000 r--p 000de000 00:36 3410904 /usr/bin/bash
556138d3c000-556138d40000 r--p 00114000 00:36 3410904 /usr/bin/bash
556138d40000-556138d49000 rw-p 00118000 00:36 3410904 /usr/bin/bash
556138d49000-556138d53000 rw-p 00000000 00:00 0 [heap]
55613993c000-55613999f000 rw-p 00000000 00:00 0
7fa40fd4a000-7fa40fd4d000 r--p 00000000 00:36 3039231 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fa40fd4d000-7fa40fd54000 r-xp 00003000 00:36 3039231 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fa40fd54000-7fa40fd56000 r--p 0000a000 00:36 3039231 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fa40fd56000-7fa40fd57000 r--p 0000b000 00:36 3039231 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fa40fd57000-7fa40fd58000 rw-p 0000c000 00:36 3039231 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fa40fd58000-7fa40fd61000 rw-p 00000000 00:00 0
7fa40fd61000-7fa40fd83000 r--p 00000000 00:36 3037590 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa40fd83000-7fa40fefb000 r-xp 00022000 00:36 3037590 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa40fefb000-7fa40ff49000 r--p 0019a000 00:36 3037590 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa40ff49000-7fa40ff4d000 r--p 001e7000 00:36 3037590 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa40ff4d000-7fa40ff4f000 rw-p 001eb000 00:36 3037590 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa40ff4f000-7fa40ff53000 rw-p 00000000 00:00 0
7fa40ff53000-7fa40ff54000 r--p 00000000 00:36 3037687 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7fa40ff54000-7fa40ff56000 r-xp 00001000 00:36 3037687 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7fa40ff56000-7fa40ff57000 r--p 00003000 00:36 3037687 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7fa40ff57000-7fa40ff58000 r--p 00003000 00:36 3037687 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7ffc546af000-7ffc546d0000 rw-p 00000000 00:00 0 [stack]
7ffc546e9000-7ffc546ed000 r--p 00000000 00:00 0 [vvar]
7ffc546ed000-7ffc546ef000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-ffffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
root@c30e77226e10:/os22fall-stu/src/lab1/init#
```