

Lab 1: RV64 内核引导

1 实验目的

学习RISC-V相关知识，了解OpenSBI平台，实现sbi调用函数，封装打印函数，并利用Makefile来完成对整个工程的管理。

2 实验内容及要求

- 阅读RISC-V中文手册，学习RISC-V相关知识
- 学习Makefile编写规则，补充Makefile文件使得项目成功运行
- 了解OpenSBI的运行原理，编写代码通过sbi调用实现字符串的打印

3 实验步骤

3.1 搭建实验环境

按照以下方法创建新的容器，并建立volume映射，在本地编写代码，并在容器内进行编译检查。

1. 创建容器并建立映射关系

```
oslab@f451e1ace656: ~  
File Edit View Search Terminal Help  
skyyyy@naki:~$ ls  
Desktop Documents Downloads examples.desktop Music Pictures Public Templates Videos  
skyyyy@naki:~$ cd Documents/  
skyyyy@naki:~/Documents$ ls  
a.txt oslab  
skyyyy@naki:~/Documents$ cd oslab  
skyyyy@naki:~/Documents/oslab$ ls  
oslab.tar  
skyyyy@naki:~/Documents/oslab$ mkdir lab1  
skyyyy@naki:~/Documents/oslab$ ls  
lab1 oslab.tar  
skyyyy@naki:~/Documents/oslab$ cd lab1  
skyyyy@naki:~/Documents/oslab/lab1$ pwd  
/home/skyyyy/Documents/oslab/lab1  
skyyyy@naki:~/Documents/oslab/lab1$ docker image ls  
REPOSITORY TAG IMAGE ID CREATED SIZE  
oslab 2020 6753703b0ae8 3 days ago 2.89GB  
skyyyy@naki:~/Documents/oslab/lab1$ docker run -it -v `pwd`: /home/oslab/lab1 -u oslab -w /home/oslab 6753 /bin/bash  
oslab@f451e1ace656:~$
```

2. 测试映射关系

```
oslab@f451e1ace656: ~/lab1
File Edit View Search Terminal Help
skyyyy@maki:~$ ls
Desktop Documents Downloads examples.desktop Music
skyyyy@maki:~$ cd Documents/
skyyyy@maki:~/Documents$ ls
a.txt oslab
skyyyy@maki:~/Documents$ cd oslab
skyyyy@maki:~/Documents/oslab$ ls
oslab.tar
skyyyy@maki:~/Documents/oslab$ mkdir lab1
skyyyy@maki:~/Documents/oslab$ ls
lab1 oslab.tar
skyyyy@maki:~/Documents/oslab$ cd lab1
skyyyy@maki:~/Documents/oslab/lab1$ pwd
/home/skyyyy/Documents/oslab/lab1
skyyyy@maki:~/Documents/oslab/lab1$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
oslab 2020 6753703b0ae8 3 days ago 2.
skyyyy@maki:~/Documents/oslab/lab1$ docker run -it -v
oslab@f451e1ace656:~$ ls
lab0 lab1
oslab@f451e1ace656:~$ cd lab1
oslab@f451e1ace656:~/lab1$ ls
testfile
oslab@f451e1ace656:~/lab1$
```

3.2 了解项目框架，编写MakeFile

Makefile是一种实现关系整个工程编译规则的文件，通过自动化编译极大提高了软件开发的效率。

1. 编写Makefile文件

```
.
├── arch
│   └── riscv
│       ├── boot
│       ├── include
│       │   ├── print.h
│       │   ├── sbi.h
│       │   └── test.h
│       ├── kernel
│       │   ├── head.S
│       │   ├── main.c (需修改数字为学号)
│       │   ├── Makefile
│       │   ├── sbi.c (需通过内联汇编实现sbi调用)
│       │   ├── test.c
│       │   └── vmlinux.lds
│       ├── libs
│       │   ├── MAKEFILE (需编写字符串打印函数及数字打印函数)
│       │   └── print.c
│       └── Makefile
├── cp.bat
├── include
│   └── defs.h
└── Makefile
```

项目代码结构由上图所示，通过最外层的Makefile中的"make run"命令进行编译与运行，其中 `./arch/riscv/libs/Makefile` 文件如下：

```
#lab1/arch/riscv/libs/Makefile
# YOUR MAKEFILE CODE
#$(wildcard *.c)表示工作目录下所有.c文件列表，sort命令的作用是将后面的字符串按照首字母生序排序，并去除重复项
C_SRC      = $(sort $(wildcard *.c))
#$(patsubst 原模式，目标模式，文件列表)
#$(patsubst <pattern>,<replacement>,<text>) 查找<text>中的单词是否符合<pattern>，如果匹配的话，则以<replacement>替换。
OBJ        = $(patsubst %.c,%.o,$(C_SRC))

all: $(OBJ)

#%为匹配符，$< 指第一个依赖文件
%.o:%.c
    ${CC}  ${CFLAG}  -c $<

#2表示标准错误stderr，>为重定向符号
#/dev/null是一个特殊的设备文件，这个文件接收到任何数据都会被丢弃
clean:
    $(shell rm *.o 2>/dev/null)
```

2. 解释Makefile命令

```
${LD} -T arch/riscv/kernel/vmlinux.lds arch/riscv/kernel/*.o arch/riscv/libs/*.o -o vmlinux
```

含义：\${LD} 被展开为 `riscv64-unknown-elf-ld`，即链接命令。-T arch/riscv/kernel/vmlinux.lds 选项表示使用 vmlinux.lds 作为链接器脚本。-o vmlinux 选项指定输出文件的名称为 vmlinux，整行命令的意思是将 arch/riscv/kernel/*.o 和 arch/riscv/libs/*.o 根据 vmlinux.lds 链接形成 vmlinux。

```
${OBJCOPY} -O binary vmlinux arch/riscv/boot/Image
```

含义：\${OBJCOPY} 表示把一种目标文件的内容复制到另一种类型的目标文件中，-O binary 表示输出为二进制文件。vmlinux 为输出文件，arch/riscv/boot/Image 为复制文件。整行命令的意思为将 arch/riscv/boot/Image 中的内容以二进制形式复制到文件 vmlinux 中。

```
${MAKE} -C arch/riscv all
```

含义：\${MAKE} 表示对make命令的递归调用，-C arch/riscv all 表示进入 arch/riscv 目录后对该目录下所有文件执行make。

3.3 学习RISC-V相关知识及特权架构

1. 基础命令

```
#1.加载立即数, t0 = 0x40000;  
li t0,0x40000  
  
#2.写特殊寄存器CSRs, 将t0寄存器的值写入satp寄存器, satp = t0;  
csrw satp, t0  
  
#3.寄存器减法, t0 = t0 - t1;  
sub t0,t0,t1  
  
#4.存储双字, 将x1寄存器的值写入Memory[sp + 8];  
sd x1, 8(sp)  
  
#5.加载地址, 将stack_top的内存地址值加载到sp寄存器中  
la sp, stack_top
```

2. RISC-V特权模式

RISC-V有三个特权模式：U（user）模式、S（supervisor）模式和M（machine）模式。它通过设置不同的特权级别模式来管理系统资源的使用。其中M模式是最高级别，该模式下的操作被认为是安全可信的，主要为对硬件的操作；U模式是最低级别，该模式主要执行用户程序，操作系统中对应于用户态；S模式介于M模式和U模式之间，操作系统中对应于内核态，当用户需要内核资源时，向内核申请，并切换到内核态进行处理。

本实验主要在S模式运行，通过调用运行在M模式的OpenSBI提供的接口操纵硬件。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

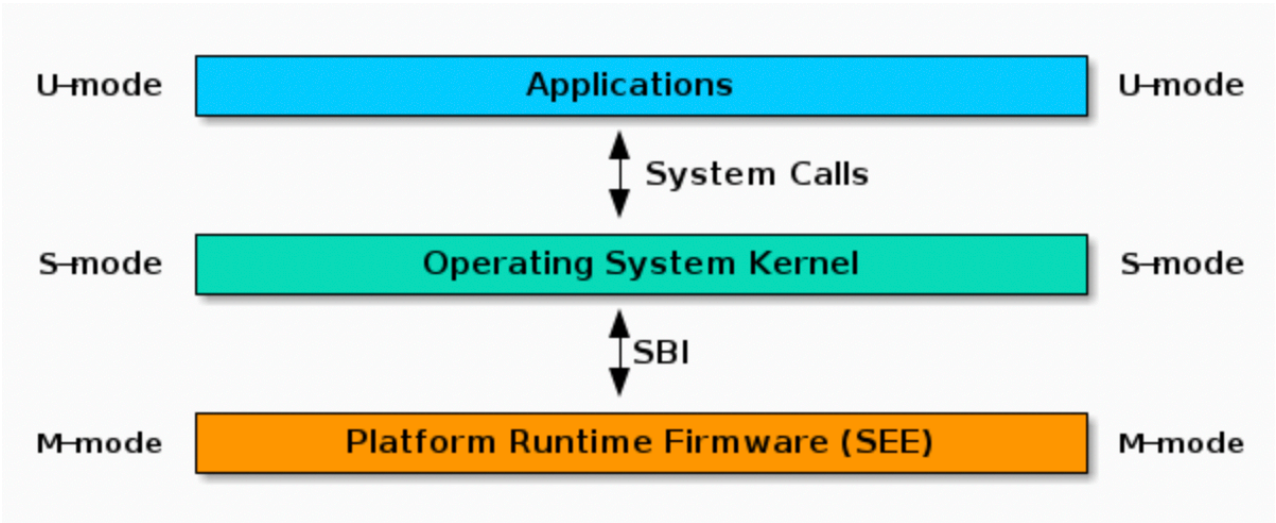
3.4 通过OpenSBI接口实现字符串打印函数

1. 程序执行流介绍

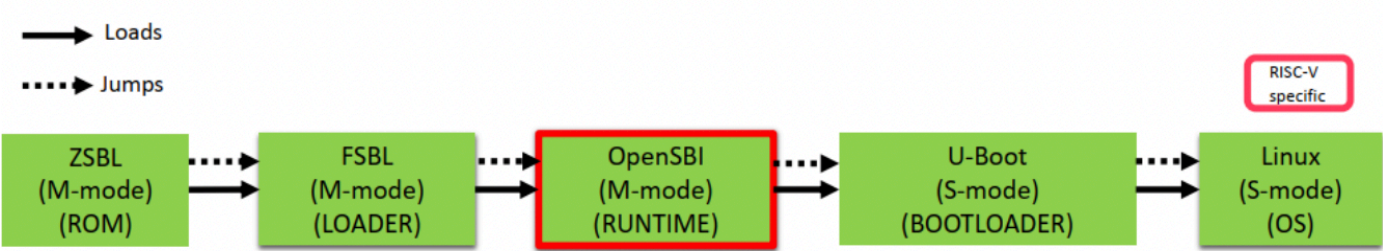
本次实验选择OpenSBI作为bios来进行机器启动时m模式下的硬件初始化与寄存器设置，并使用OpenSBI所提供的接口完成诸如字符串打印等操作。

OpenSBI介绍

SBI (Supervisor Binary Interface)是 S-Mode 的 kernel 和 M-Mode 执行环境之间的标准接口，而OpenSBI项目的目标是为在M模式下执行的平台特定固件提供RISC-V SBI规范的开源参考实现，使运行在s-mode下的内核可以按照标准对这些硬件进行操作。



RISC-V启动过程



- ZSBL(Zeroth Stage Boot Loader): 片上ROM程序，烧录在硬件上，是芯片上电后最先运行的代码。它的作用是加载FSBL到指定位置并运行。
- FSBL(First Stage Boot Loader)：启动PLLs和初始化DDR内存，对硬件进行初始化，加载下一阶段的bootloader。
- OpenSBI: 运行在m模式下的一套软件，提供接口给操作系统内核调用，以操作硬件，实现字符输出及时钟设定等工作。OpenSBI就是一个开源的RISC-V虚拟化二进制接口的通用的规范。
- Bootloader: OpenSBI初始化结束后会通过mret指令将系统特权级切换到s模式，并跳转到操作系统内核的初始化代码。这一阶段，将会完成中断地址设置等一系列操作。之后便进入了操作系统。

从ZSBL到OpenSBI运行这一阶段的工作已通过QEMU模拟器完成。运行QEMU时，我们使用-bios default选项将OpenSBI代码加载到0x80000000起始处。OpenSBI初始化完成后，会跳转到0x80200000处。因此，我们所编译的代码需要放到0x80200000处。

vmlinux & vmlinux.lds

vmlinux 通常指Linux Kernel编译出的可执行文件（Executable and Linkable Format, ELF），特点是未压缩，带调试信息和符号表。本次实验中指代码进行编译，链接后生成的可供QEMU运行的RISC-V 64-bit架构程序。

```
oslab@f451e1ace656:~/lab1$ file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, with debug_info, not stripped
```

vmlinux.lds 指在Linux Kernel中链接脚本。GNU ld即链接器，用于将*.o文件（和库文件）链接成可执行文件。在操作系统开发中，为了指定程序的内存布局，ld使用链接脚本（Linker Script）来控制。

vmlinux.lds 链接脚本指定了程序的内存布局，最先加载的 .text.init 段代码为 head.s 文件的内容，顺序执行调用 main() 函数。main() 函数调用了两个打印函数，通过 sbi_call() 向OpenSBI发起调用，完成字符的打印。

```

vmlinux.lds
~/Documents/oslab/lab1/arch/riscv/kernel

BASE_ADDR = 0x80200000;
SECTIONS
{
    . = BASE_ADDR;
    .text : {
        *(.text.init)
        *(.text.entry)
        *(.text)
        *(.text.*)
    }
    .rodata : {
        *(.rodata)
        *(.rodata.*)
    }
    .data : {
        *(.data)
        *(.data.*)
    }
    bss_start = .;
    .bss : {
        *(.sbss)
        *(.sbss.*)
        *(.bss)
        *(.bss.*)
    }
    bss_end = .;
    . += 0x8000;
    stack_top = .;
    _end = .;
}

```

链接脚本中有 . * 两个重要的符号。单独的 . 在链接脚本代表当前地址，它有赋值、被赋值、自增等操作。而 * 有两种用法，其一是 *() 在大括号中表示将所有文件中符合括号内要求的段放置在当前位置，其二是作为通配符。

SECTIONS部分将程序的各个段按顺序放在各个地址上，本次实验中从0x80200000地址开始放置了.text, .rodata, .data和.bss段。

段名	主要作用
.text	通常存放程序执行代码
.rodata	通常存放常量等只读数据
.data	通常存放已初始化的全局变量、静态变量
.bss	通常存放未初始化的全局变量、静态变量

System.map

System.map是内核符号表（Kernel Symbol Table）文件，是存储了所有内核符号及其地址的一个列表。

```

System.map
~/Documents/oslab/lab1

0000000000200000 A BASE_ADDR
0000000000208237 R _end
0000000000200000 T _start
0000000000200237 R bss_end
0000000000200237 R bss_start
0000000000200010 T main
000000000020012c T put_num
00000000002000cc T puts
0000000000200058 T sbi_call
0000000000208237 R stack_top
00000000002000b4 T test

```

make run命令

```
#make run 实际为执行lab1/Makefile中的该行命令
@gemu-system-riscv64 -nographic --machine virt -bios default -device
loader,file=vmlinux,addr=0x80200000 -D log
```

QEMU模拟器完成从ZSBL到OpenSBI阶段的工作，使用-bios default选项将OpenSBI代码加载到0x80000000起始处。OpenSBI初始化完成后，跳转到0x80200000处。将vmlinux程序这一RISC-V 64-bit架构程序加载至0x80200000处运行。

2. 编写sbi_call()函数

当系统处于m模式时，对指定地址进行写操作便可实现字符的输出。但编写的内核运行在s模式，需要使用OpenSBI提供的接口，让运行在m模式的OpenSBI实现输出。此时，运行在s模式的内核通过ecall发起sbi调用请求，RISC-V CPU 会从s态跳转到m态的OpenSBI固件。

执行ecall前需要指定sbi调用的编号，传递参数。一般而言，a7(x17)为SBI调用编号，a0(x10)、a1(x11)和a2(x12)寄存器为sbi调用参数。

以内联汇编形式实现lab1/arch/riscv/kernel/sbi.c中的sbi_call()函数。

```
#include "defs.h"

uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        //Your code
        "mv a7, %[sbi_type]\n"
        "mv a0, %[arg0]\n"
        "mv a1, %[arg1]\n"
        "mv a2, %[arg2]\n"
        "ecall\n"
        "mv %[ret_val], a0\n"
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2)
        : "memory"
    );
    return ret_val;
}
```

- 将sbi_type放到寄存器a7中，将arg0-arg2放入a0-a2中。
- 使用ecall指令发起sbi调用请求。
- OpenSBI的返回结果会放到a0中，需要将其取出作为sbi_call的返回值。

在本次实验中为了打印字符，需要向sbi_call()函数传入sbi_type=1以调用sbi_console_putchar(int ch)函数，第一个参数arg0需要传入待打印字符的ASCII码，第二、三个参数用0填充。

Type	Function	Function ID
Timer	sbi_set_timer	0
Console	sbi_console_putchar	1
	sbi_console_getchar	2
IPI	sbi_clear_ipi	3
	sbi_send_ipi	4
Memory Model	sbi_remote_fence_i	5
	sbi_remote_sfence_vma	6
	sbi_remote_sfence_vma_as	7
	id	
Shutdown	sbi_shutdown	8

3. 编写字符串打印函数

在 `./arch/riscv/libs/print.c` 文件中通过调用 `sbi_call()` 实现字符串打印函数 `int puts(char* str)` 及数字打印函数 `int put_num(uint64_t n)`，后者可将数字转换为字符串后调用前者执行。

```
#include "defs.h"

extern sbi_call();

int puts(char* str){
    // your code
    do{
        sbi_call(1, (uint64_t)*str, 0, 0);
        str = str + 1;
    }while (*str != '\0');

    return 0;
}

int put_num(uint64_t n){
    // your code
    char str[100];
    char ustr[100];
    int m;
    int i = 0;
    int j = 0;
```



```
do{
    m = n%10;
    n = n/10;
    ustr[i] = '0'+m;
    i++;
}while (n != 0);
for (i = i - 1; i >= 0; i--){
    str[j] = ustr[i];
    j++;
}
str[j] = '\0';
puts(str);
return 0;
}
```

3.5 编译及测试

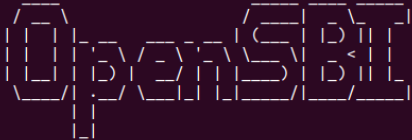
在 lab1/arch/riscv/kernel/main.c 中将 21922192 修改为学号，在项目最外层输入 make run 命令调用 Makefile 文件完成整个工程的编译及执行。

如果编译失败，及时使用 `make clean` 命令清理文件后再重新编译。

程序输出：

```
oslab@f451e1ace656:~/lab1$ make run
make -C arch/riscv all
make[1]: Entering directory '/home/oslab/lab1/arch/riscv'
make -C kernel
make[2]: Entering directory '/home/oslab/lab1/arch/riscv/kernel'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/oslab/lab1/arch/riscv/kernel'
make -C libs
make[2]: Entering directory '/home/oslab/lab1/arch/riscv/libs'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/oslab/lab1/arch/riscv/libs'
make[1]: Leaving directory '/home/oslab/lab1/arch/riscv'
riscv64-unknown-elf-ld -T arch/riscv/kernel/vmlinux.lds arch/riscv/kernel/*.o arch/riscv/libs/*.o -o vmlinux
riscv64-unknown-elf-objcopy -O binary vmlinux arch/riscv/boot/Image
nm vmlinux > System.map

OpensBI v0.6
```



```
Platform Name       : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs   : 8
Current Hart         : 0
Firmware Base        : 0x80000000
Firmware Size        : 120 KB
Runtime SBI Version   : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
Hello riscv!
3190102362
```

4 讨论与心得

本次实验由于详尽的实验指导，在编程方面没有遇到太多问题，整个过程比较顺利。

但这次实验涉及大量扩展知识，尽量理解每一个知识点是本次实验的重难点，尤其是不同风格makefile的理解与书写以及项目中原先写好的文件的作用。riscv汇编语言在本次实验中也没有太多使用，我对它的学习理解还需要加深。