

# lzu\_os lab

---

姓名：张金鹏 学号：320210931971 专业：2021级计算机基地班

## lab1

---

### 1、注释makefile文件。

```
#first step, func $subst(FROM,TO,TEXT) :substitute FROM with TO in TEXT, then
return the substituted text
#second step,func $foreach(VAR,TEXT,COMMAND): for each unit separated by space do
COMMAND,then return the list separated by space after all commands
#third step, func $wildcard( PATTERN...):enable pattern matching,return matched
files
#finally, ifneq judges whether the expression value is 0,then decide to jump to
which branch

#this block is to check whether you have the 'riscv64-linux-gnu tool chain,
#and if doesn't exist, then use 'riscv64-unknown-elf' as tool chain
ifneq ($(wildcard $(foreach p,$(subst :, ,$(PATH)),$(p)/riscv64-linux-gnu-*)),)
    PREFIX := riscv64-linux-gnu
else
    PREFIX := riscv64-unknown-elf
endif

CC := $(PREFIX)-gcc      #declare complier
LD := $(PREFIX)-ld       #linker
AR := $(PREFIX)-ar       #archive library
OBJCOPY := $(PREFIX)-objcopy #copy and convert binary file
OBJDUMP := $(PREFIX)-objdump #disassembler
READELF := $(PREFIX)-readelf #read elf file
NM := $(PREFIX)-nm        #symbol table,shared library
AS := $(PREFIX)-as         #assembler
TMUX := tmux              #divide screen tool
QEMU := qemu-system-riscv64 #riscv64 arch, running on qemu system emulator
GDB := riscv64-unknown-elf-gdb #debugger

KERN_IMG := kernel.img #the kernel iso , include bios
KERN_BIN := kernel.bin #the kernel ios(binary)
KERN_MAP := kernel.map #include symbol table(func,var), and address of these
table
KERN_SYM := kernel.sym #symbol table of kernel
KERN_ASM := kernel.asm #assemble language of kernel

#$1: memeory model = medany(handle medium scale memory)
#$2: enable warning messages
#$3: generate debugging info with level 03
#$4: not use built-in funcs
```

```

#$5: disable stack protector
#$6: disable strict aliasing rules
#$7: not include standard library headers
#$8: refers to including file, and look for headers in current directory
#all these options form CFLAGS(compile flags)
CFLAGS := -mcmodel=medany -Wall -g3 -fno-builtin -fno-stack-protector -fno-
strict-aliasing -nostdinc -I .

#search for all .c files and substitute it with .o file, then generate the target
objects
objects := $(patsubst %.c, %.o, $(wildcard *.c)) entry.o

#these are make command, do not track them
.PHONY : all build run run-gui symbol debug clean disassembly

all: build

build : $(KERN_BIN) #generate kernel.bin, with given kernel.map,kernel.img and
all objects

$(KERN_BIN) : $(objects)
    $(LD) -T linker.ld -Map=$(KERN_MAP) -o $(KERN_BIN) $^
    $(OBJCOPY) $(KERN_BIN) --strip-all -O binary $(KERN_IMG)

#run non-graphic qemu system,use kernel.img as kernel,init address at 0x80200000
run : build
    @$(QEMU) \
        -machine virt \
        -nographic \
        -bios fw_jump.bin \
        -device loader,file=$(KERN_IMG),addr=0x80200000

#run graphic terminal,same as above
run-gui : build
    @$(QEMU) \
        -machine virt \
        -bios fw_jump.bin \
        -device loader,file=$(KERN_IMG),addr=0x80200000

#run debug mode with tmux
debug :
    $(TMUX) new -s debug -d "$(QEMU) \
        -machine virt \
        -s -S \
        -serial mon:stdio \
        -nographic \
        -bios fw_jump.bin \
        -device loader,file=$(KERN_IMG),addr=0x80200000" && $(TMUX)

split-window -h "$(GDB) -q -x gdbinit"
    $(TMUX) attach-session -t debug

#create file kernel.sym which includes kernel symbol infos
symbol : $(KERN_BIN)
    $(NM) -s $(KERN_BIN) > $(KERN_SYM)

#create file kernel.asm which include asm forms of kernel

```

```
disassembly: $(KERN_BIN)
$(OBJDUMP) -d $(KERN_BIN) > $(KERN_ASM)

clean: #clean all the files that make build
-rm -f *.o $(KERN_BIN) $(KERN_IMG) $(KERN_SYM) $(KERN_ASM)
```

## 2、img文件是如何生成的？

首先，当在makefile中直接或者间接执行make build指令时，会生成kernel.bin,这个目标的依赖是所有的.o文件，当有源文件发生改动后，就会重新生成kernel.bin，生成该文件需要需要链接器（\$LD）进行链接，同时使用使用 linker.ld 文件作为链接脚本，这个脚本定义了链接的规则和地址布局。-Map=\$(KERN\_MAP) 用于生成kernel.map，记录了链接器的详细信息。-o \$(KERN\_BIN) 指定链接后生成的可执行文件的名称。\$^ 是一个自动变量，代表所有的依赖文件，也就是 \$(objects) 中的所有对象文件。最后利用工具objcopy将kernel.bin转为二进制文件kernel.img。

核心代码为以下三行：

```
$(KERN_BIN) : $(objects)
$(LD) -T linker.ld -Map=$(KERN_MAP) -o $(KERN_BIN) $^
$(OBJCOPY) $(KERN_BIN) --strip-all -O binary $(KERN_IMG)
```

## 3、通过 cat kernel.map来查看链结器命令稿和内存映射，通过 readelf -a kernel.bin了解kernel的程序段和节，需要加载的段等信息，画出启动后的内存地址空间布局。

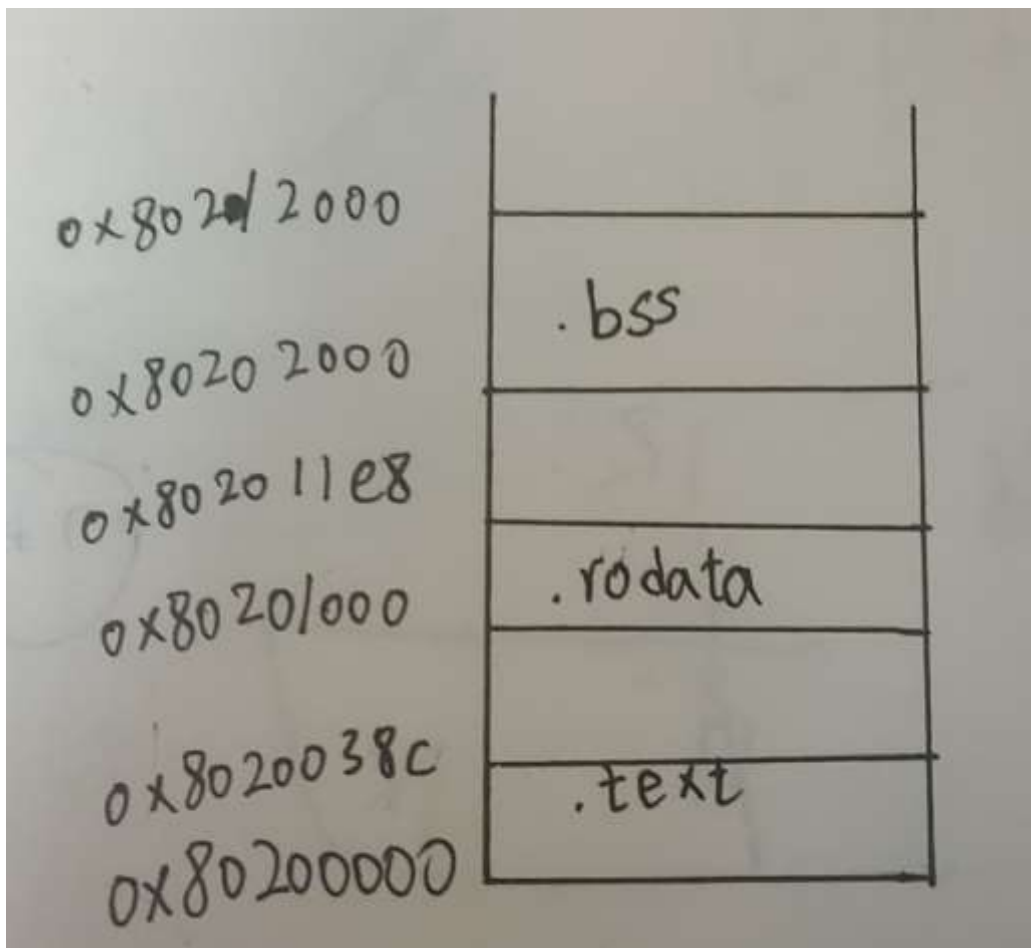
kernel的程序段有：

- 1、RISCV\_ATTRIBUTE段，该段的起始地址为0x0，大小为0x33个字节，标志为R（可读）。
- 2、LOAD段（1），起始地址为0x80200000，大小为0x11e8个字节，标志为RE（可读可执行）。
- 3、LOAD段（2），起始地址为0x80202000，大小为0x10000字节，标志为RW（可读可写）。
- 4、GNU\_STACK段，并没有分配大小，标志位RW（可读可写）。

节信息：

- 1、NULL节，无效的节，不分配空间
- 2、.text节，程序代码，大小为0x38字节，标志为AX（可分配可执行）
- 3、.rodata节，只读数据，大小为0x1e8字节，标志为A（可分配）
- 4、.bss节，未初始化的数据段，大小为0x1000，标志为AW（可写可分配）
- 5、.debug\_\*等节，负责调试信息

内存空间布局：



#### 4、反汇编得到kerne.bin的汇编代码，并与entry.s部分的代码进行比较。gdb调试找到while (1) 的PC值。

kernel.bin的反汇编代码：

Disassembly of section .text:

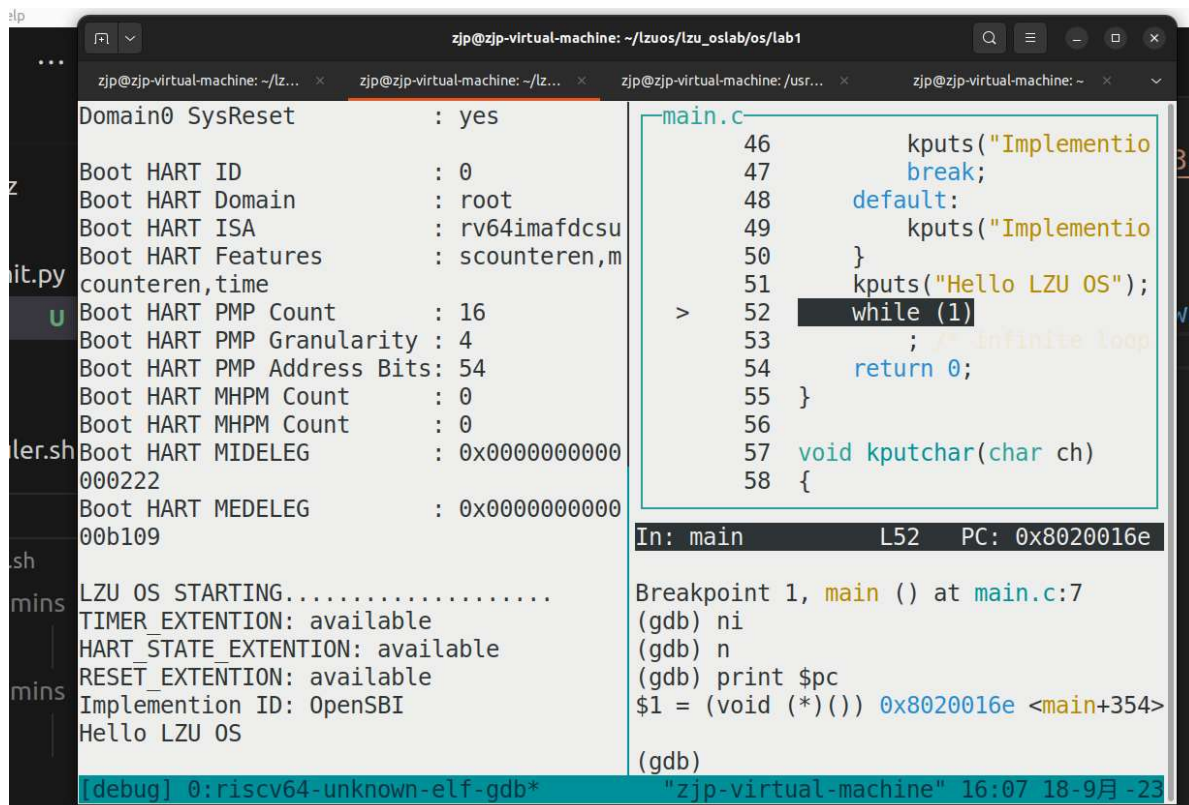
```
0000000080200000 <_start>:
80200000: 00012117          auipc    sp,0x12
80200004: 00010113          mv       sp,sp
80200008: 004000ef          jal      ra,8020000c <main>

000000008020000c <main>:
8020000c: 1101             addi     sp,sp,-32 # 80211fe0 <boot_stack+0xffe0>
8020000e: ec06             sd       ra,24(sp)
80200010: e822             sd       s0,16(sp)
80200012: 1000             addi     s0,sp,32
80200014: 00001517          auipc    a0,0x1
80200018: fec50513          addi     a0,a0,-20 # 80201000 <rodata_start>
8020001c: 176000ef          jal      ra,80200192 <kputs>
80200020: 544957b7          lui      a5,0x54495
80200024: d4578513          addi     a0,a5,-699 # 54494d45 <_start-0x2bd6b2bb>
80200028: 302000ef          jal      ra,8020032a <sbi_probe_extension>
8020002c: 872a             mv       a4,a0
8020002e: 87ae             mv       a5,a1
80200030: fee43023          sd       a4,-32(s0)
80200034: fef43423          sd       a5,-24(s0)
80200038: fe843783          ld       a5,-24(s0)
8020003c: cb81             beqz     a5,8020004c <main+0x40>
8020003e: 00001517          auipc    a0,0x1
80200042: fea50513          addi     a0,a0,-22 # 80201028 <rodata_start+0x28>
80200046: 14c000ef          jal      ra,80200192 <kputs>
8020004a: a039             j        80200058 <main+0x4c>
8020004c: 00001517          auipc    a0,0x1
80200050: ffc50513          addi     a0,a0,-4 # 80201048 <rodata_start+0x48>
80200054: 13e000ef          jal      ra,80200192 <kputs>
```

entry.s代码:

```
zjp@zjp-virtual-machine:~/lzuos/lzu_oslab/os/lab1$ cat entry.s
.globl boot_stack, boot_stack_top, _start
.section .text.entry
_start:
    la sp, boot_stack_top
    call main

.section .bss
boot_stack:
    .space 4096 * 16
boot_stack_top:
```



```
Domain0 SysReset      : yes
Boot HART ID          : 0
Boot HART Domain      : root
Boot HART ISA          : rv64imafdcsu
Boot HART Features     : scounteren,m
counteren,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      : 0x000000000000
000222
Boot HART MEDELEG      : 0x000000000000
00b109
LZU OS STARTING.....
TIMER EXTENTION: available
HART STATE EXTENTION: available
RESET_EXTENTION: available
Implemention ID: OpenSBI
Hello LZU OS

main.c
46     kputs("Implementio
47     break;
48     default:
49     kputs("Implementio
50     }
51     kputs("Hello LZU OS");
> 52     while (1)
53         ; // infinite loop
54     return 0;
55 }
56
57 void kputchar(char ch)
58 {

In: main      L52      PC: 0x8020016e
Breakpoint 1, main () at main.c:7
(gdb) ni
(gdb) n
(gdb) print $pc
$1 = (void (*)(void)) 0x8020016e <main+354>
(gdb)
```

## 5、注释一个sbi封装函数。

以用来将字符输出到控制台的函数sbi\_console\_putchar(char ch)为例:

```
void sbi_console_putchar(char ch)
{
    //define a 64bit var as a0, then associated it with register a0 using
    'asm("a0")'.It is used to store the 'ch',cause register a0 is used to pass
    arguments in system call.
    register uint64_t a0 asm("a0") = ch;

    //define a 64bit var a6 which is assigned to zero, then associated it with
    register a6 using 'asm("a6")'.In RISCv, register a6 is used to keep the system
    call number, and zero stands for the exact number of system call(when it happens
    in Linux, it's called sys_call )
    register uint64_t a6 asm("a6") = 0;
```

//define a 64bit var a7 which is assigned to 1, then associated it with register a7 using 'asm("a7")'. In RISC-V, register a7 is used to keep the system call argument. It's set to 1, indicating the system call to perform a 'write' operation.

```
register uint64_t a7 asm("a7") = 1;
```

//the '\_\_asm\_\_volatile\_\_' key word indicates this is an inline assembly instruction. And \_\_volatile tells the compiler not to optimize this assembly code.

//ecall is a special RISC-V instruction used to trigger a system call.

//the 'empty output list' after colon indicates there is no output list in this case, so it's empty.

//': "r"(a0), "r"(a6), "r"(a7)': these are input arguments, specifying the input register for the ecall. r(a0) stands for the value of register a0, and a6, a7 are the same as a0.

// : "memory": this tells the compiler that this assembly code may modify the memory, so it needs to ensure memory is updated correctly. In programmer's case, they often use it to tell compiler not to switch the sequence in this assembly code (if it has context), it's another way to tell compiler not to optimize the code.

```
__asm__ __volatile__("ecall \n\t"  
    : /* empty output list */  
    : "r"(a0), "r"(a6), "r"(a7)  
    : "memory");  
}
```