

# Lab3: RV64 Kernal

朱熙哲

3220103361

2023 年 11 月 12 日

## 目录

<b>1</b>	<b>代码编写</b>	<b>2</b>
1.1	编写 head.S 与 vmlinux.lds . . . . .	2
1.2	完善 Makefile . . . . .	3
1.3	补充 sbi.c . . . . .	3
1.4	puts() 和 puti() . . . . .	5
1.5	修改 defs . . . . .	6
<b>2</b>	<b>运行</b>	<b>7</b>
<b>3</b>	<b>思考题</b>	<b>8</b>
3.1	查看 system.map 地址 . . . . .	8
3.2	观察程序开始时的特权态和中断信息 . . . . .	9
3.3	查看程序开始时各段内存 . . . . .	10
3.4	从汇编代码给 start_kernel 传参 . . . . .	10
<b>4</b>	<b>Bonus: spike 工具链</b>	<b>12</b>

# 1 代码编写

## 1.1 编写 head.S 与 vmlinux.lds

对于 vmlinux.lds, 在末尾加上 stack 段来方便插入栈空间。

```
/* 记录 kernel 代码的结束地址 */  
. = ALIGN(0x1000);  
_end = .;  
  
.stack : ALIGN(0x1000){  
    _sstack = .;  
  
    *(.stack.entry)  
  
    _estack = .;  
}
```

在 head.S 中使用 .space 设置 4KB 的栈空间, 把 sp 设为栈顶并跳转到 start\_kernel。

```
_start:  
    la sp, stack_top  
    jal start_kernel  
  
.section .stack.entry  
.globl stack_bottom  
stack_bottom:  
    .space 0x1000  
.globl stack_top  
stack_top:
```

## 1.2 完善 Makefile

lib 的 makefile 与 init 当中使用的一致即可。print.c 的编译需要 print.h 与 sbi.h。

```
C_SRC  = $(sort $(wildcard *.c))
OBJ     = $(patsubst %.c,%.o,$(C_SRC))

file = print.o
all:$(OBJ)

%.o:%.c
    ${GCC} ${CFLAG} -c $<
clean:
    $(shell rm *.o 2>/dev/null)
```

## 1.3 补充 sbi.c

依照要求将参数分别传入到寄存器中，ecall 并接收返回值。

```
struct sbiret ecall_ret;
__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"
    "ecall \n"
```

```

    "mv %[ecall_ret_error], a0 \n"
    "mv %[ecall_ret_value], a1 "
    : [ecall_ret_error] "=r" (ecall_ret.error),
      [ecall_ret_value] "=r" (ecall_ret.value)
    : [ext] "r" (ext), [fid] "r" (fid), [arg0] "r" (arg0),
      [arg1] "r" (arg1), [arg2] "r" (arg2), [arg3] "r" (arg3),
      [arg4] "r" (arg4), [arg5] "r" (arg5)
    : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7", "memory"
);
return ecall_ret;

```

将 `ecall_ret_error` 和 `ecall_ret_value` 分别更新到 `ecall_ret.error` 和 `ecall_ret.value` 中。将 `ext`、`fid`、`arg[0 5]` 分别置入对应寄存器。最后声明可能影响的寄存器和内存，避免相互影响。

可以另外将 `sbi_set_timer`、`sbi_console_putchar`、`sbi_console_getchar` 都做个简单实现。

```

void sbi_set_timer(uint64 set_time_value){
    sbi_ecall(0x00, 0, set_time_value, 0, 0, 0, 0, 0);
}

void sbi_console_putchar(char c){
    sbi_ecall(0x01, 0, (int)c, 0, 0, 0, 0, 0);
}

int sbi_console_getchar(){
    struct sbiret ret;
    ret = sbi_ecall(0x02, 0, 0, 0, 0, 0, 0, 0);
    return ret.error;
}

```

## 1.4 puts() 和 puti()

使用 `console_putchar` 将字符一个个打印出，到 0 停止。

```
void puts(char *s) {
    while(*s){
        sbi_console_putchar(*(s++));
    }
}
```

考虑负数情况与零的情况，靠取模与移位将数字逐位转换为字符，再反向输出。注意使用 `dowhile` 来应对数为 0 的情况。

```
void puti(int x) {
    char s[20];
    int i = 0;
    if(x < 0){
        sbi_console_putchar('-');
        x = -x;
    }
    do{
        s[i++] = '0' + x%10;
        x /= 10;
    }while(x);
    while(i){
        sbi_console_putchar(s[--i]);
    }
}
```

## 1.5 修改 defs

调用 csrr 将 \_\_v 读到 csr 中。

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

## 2 运行

```

master | 115 269
make -C lib all
make[1]: 进入目录“/mnt/d/software/git/sys2-fa23/src/lab3/lib”
riscv64-linux-gnu-gcc -g3 -march=rv64imafdc -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -Wl,--nmagic -Wl,--gc-sections -I /mnt/d/software/git/sys2-fa23/src/lab3/include -I /mnt/d/software
make[1]: 离开目录“/mnt/d/software/git/sys2-fa23/src/lab3/lib”
make -C init all
make[1]: 进入目录“/mnt/d/software/git/sys2-fa23/src/lab3/init”
make[1]: “all”已是最新。
make[1]: 离开目录“/mnt/d/software/git/sys2-fa23/src/lab3/init”
make -C arch/riscv all
make[1]: 进入目录“/mnt/d/software/git/sys2-fa23/src/lab3/arch/riscv”
make -C kernel all
make[2]: 进入目录“/mnt/d/software/git/sys2-fa23/src/lab3/arch/riscv/kernel”
make[2]: 对“all”无需做任何事。
make[2]: 离开目录“/mnt/d/software/git/sys2-fa23/src/lab3/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
riscv64-linux-gnu-objdump -S ../../vmlinux > ../../vmlinux.asm
nm ../../vmlinux > ../../System.map
make[1]: 离开目录“/mnt/d/software/git/sys2-fa23/src/lab3/arch/riscv”

Build Finished OK

```

## 3 思考题

### 3.1 查看 system.map 地址

```
/mnt/d/software/git/sys2-fa23/src/lab3 | master !15 ?69 .....
cat System.map | tail -23 | sort
0000000080200000 A _BASE_ADDR
0000000080200000 T _start
0000000080200000 T _stext
0000000080200000 T sbi_ecall
00000000802000f0 T sbi_set_timer
000000008020013c T sbi_console_putchar
000000008020018c T sbi_console_getchar
00000000802001ec T start_kernel
000000008020022c T test
000000008020023c T puts
0000000080200290 T puti
0000000080200370 T _etext
0000000080201000 R _srodata
0000000080201019 R _erodata
0000000080202000 N _sstack
0000000080202000 N stack_bottom
0000000080202000 R _ebss
0000000080202000 R _edata
0000000080202000 R _end
0000000080202000 R _sbss
0000000080202000 R _sdata
0000000080203000 N _destack
0000000080203000 N _stack_top
```

`_start` 的地址在 `0x8020000`，`.text` 段的起始地址 `_stext` 也在 `0x80200000`，之后是实现的函数，然后 `.text` 段结束 (`_etext`)。 `.rodata` 中保存了字符串“ZJU Computer System II\n”。然后是空的 `.data` 段和 `.bss` 段，均对齐到 `0x80202000`。 `_end` 是内核结束地址，然后是分配的 4KB 大小的栈空间。



## 3.2 观察程序开始时的特权态和中断信息

```
0x801ffff4 2byte 0x0
0x801ffff6 2byte 0x0
0x801ffff8 2byte 0x0
+ 0x80200000 <_stext+0>
0x80200004 <_stext+4>
0x80200008 <_stext+8>
0x8020000c <sbi_ecall+0>
0x80200010 <sbi_ecall+4>
0x80200014 <sbi_ecall+8>
3  mv sp, sp
jal ra, 0x802001ec <start_kernel>
addi sp, sp, -128
sd s0, 120(sp)
sd s1, 112(sp)
3  section .text.entry
4  globl _start, _end
5  _start:
6  //Error "Still have unfilled code!"
7  //unimplemented
+ 8  //csrr a0, mstatus
9  jal start_kernel
10
11  section .stack.entry
12  globl _stack_bottom
13  _stack_bottom:
[0] Id 1, stopped 0x80200000 in _stext (), reason: BREAKPOINT
[0] 0x80200000 + _stext ()
get+ i r mstatus
mstatus 0x8000000a00006080 SD:1 VM:0 MXR:0 PUM:0 MPRV:0 XS:0 FS:3 MPP:0 HPP:0 SPP:0 MPIE:1 HPPIE:0 SPIE:0 UPIE:0 MIE:0 HIE:0 SIE:0 UIE:0
get+ i r priv
priv 0x1 prv:1 [Supervisor]
```

开始执行时的特权态为 1，即 Supervisor，并观察存储中断信息的寄存器。

### 3.3 查看程序开始时各段内存

```
gef> x/1s 0x80201000
0x80201000: "ZJU Computer System II\n"
gef> l _stext
3      .section .text.entry
4      .globl _start, _end
5      _start:
6      // #error "Still have unfilled code!"
7      //unimplemented
8      la $sp, stack_top
9      //csrr a0, mstatus
10     jal start_kernel
11
12     .section .stack.entry
gef> x/4xg
0x80201019: 0x0000000000000000 0x0000000000000000
0x80201029: 0x0000000000000000 0x0000000000000000
gef> x/4xg 0x80202000
0x80202000: 0x0000000000000000 0x0000000000000000
0x80202010: 0x0000000000000000 0x0000000000000000
gef> x/4xg 0x80203000
0x80203000: 0x0000000000000000 0x0000000000000000
0x80203010: 0x0000000000000000 0x0000000000000000
```

.text 段存储了指令，此处列出的正为 head.S 中的内容。.rodata 中保存了字符串。.data 段和.bss 段均为空。栈空间尚未使用，也为空。

### 3.4 从汇编代码给 start\_kernel 传参

由于 risc-v 调用函数使用 a0-a7 传参，因此修改 main.c 与 head.S 用以传参。

- main.c

```
int start_kernel(int x) {
    puti(x);
    puts(" ZJU Computer System II\n");
    ...
}
```

- head.S

```
_start:
    la sp, stack_top
    li a0, 2023
    jal start_kernel
    ...
```

```
196 \begin{minted}{asm}
Boot HART Priv Version      : v1.12
197 \start:
Boot HART Base ISA         : rv64imafdc
198 \la sp, stack_top
Boot HART ISA Extensions   : zicntr
199 \li a0, 2023
Boot HART PMP Count        : 16
200 \jal start_kernel
Boot HART PMP Granularity  : 4
201 \...
Boot HART PMP Address Bits : 54
202 \end{minted}
Boot HART MHPM Info        : 0 (0x00000000)
203 \end{itemize}
Boot HART MIDELEG          : 0x00000000000000222
204
Boot HART MEDELEG          : 0x00000000000000b109
205
206 2023 ZJU Computer System II
207 \end{document}
```

运行 make run，启动内核并输出“2023 ZJU Computer System II”。

## 4 Bonus: spike 工具链

先运行 `make install` 下载所有工具。随后修改 `testcase` 为 `lab3` 中产生的 `Image` 的地址，在 `Makefile` 中修改各工具的实际地址，然后 `make run`。

```
make run 编写head.S与v...
spike/build/bin/spike --kernel /mnt/d/software/git/sys2-fa23/src/lab3/arch/riscv/boot/Image fw_jump.elf
OpenSBI v1.3
1.3 补充sbic 234 \end{document}
1.5 修改defs
运行
Platform Name : ucbbbar,spike-bare
Platform Features : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 10000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : htif
Platform Shutdown Device : htif
```

进行 debug。先运行 make debug，然后运行 make bridge 开放端口等待 gdb 连接。

```
/mnt/d/software/git/sys2-fa23/spike_tool master .....
make debug
spike/build/bin/spike -H --rbb-port=9824 --kernel /mnt/d/software/git/sys2-fa23/sr
Listening for remote bitbang connection on port 9824.
```

```
/mnt/d/software/git/sys2-fa23/spike_tool master 116 770 .....
make bridge
openocd -f spike.cfg
Open On-Chip Debugger 0.12.0+dev-03251-gd14b71cd3 (2023-11-12-15:05)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
DEPRECATED! use 'adapter driver' not 'interface'
Info : only one transport option; autoselecting 'jtag'
DEPRECATED! use 'remote_bitbang host' not 'remote_bitbang_host'
DEPRECATED! use 'remote_bitbang port' not 'remote_bitbang_port'
Info : Initializing remote_bitbang driver
Info : Connecting to localhost:9824
Info : remote_bitbang driver initialized
Info : Note: The adapter "remote_bitbang" doesn't support configurable speed
Info : JTAG tap: riscv.cpu tap/device found: 0xdeadbeef (mfg: 0x777 (<unknown>), part: 0xeadb, ver: 0xd)
Warn : JTAG tap: riscv.cpu UNEXPECTED: 0xdeadbeef (mfg: 0x777 (<unknown>), part: 0xeadb, ver: 0xd)
Error: JTAG tap: riscv.cpu expected 1 of 1: 0x10e31913 (mfg: 0x489 (SiFive Inc), part: 0x0e31, ver: 0x1)
Error: Trying to use configured scan chain anyway...
Warn : Bypassing JTAG setup events due to errors
Info : [riscv.cpu] datacount=2 progbufsize=2
Info : [riscv.cpu] Examined RISC-V core; found 1 harts
Info : [riscv.cpu] XLEN=64, misa=0x800000000001411d
[riscv.cpu] Target successfully examined.
Info : starting gdb server for riscv.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

之后用 gdb 连接到 3333 端口，可以开始调试。

```
/mnt/d/software/git/sys2-fa23/src/lab3 | master !16 ?70 出"2023. ZJU. Computer. System. II".
gdb-multiarch vmlinux
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>
For help, type "help".
Type "apropos word" to search for commands related to "word"...
warning: ./gef.py: 没有那个文件或目录
GEF for linux ready, type 'gef' to start, 'gef config' to configure
88 commands loaded and 5 functions added for GDB 12.1 in 0.00ms using Python engine 3.10
Reading symbols from vmlinux...
gef> gef-remote localhost 3333
0x0000000000001000 in ?? ()
[!] Command 'gef-remote' failed to execute properly, reason: Remote I/O error: 函数未实现
gef> b _start
Breakpoint 1 at 0x80200000: file head.S, line 8.
gef> c
Continuing.

Breakpoint 1, s_text() at head.S:8
8      la sp, stack_top
[ Legend: Modified register | Code | Heap | Stack | String ]

$zero: 0x0000000000000000 -> 0x0000000000000000
$ra : 0x8000b4ca
```