Lab3: RV64 Kernal

朱熙哲

3220103361

2023 年 11 月 12 日

目录

1 代码编写

1.1 编写 head.S 与 vmlinux.lds

对于 vimlinux.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_kernal。

```
_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

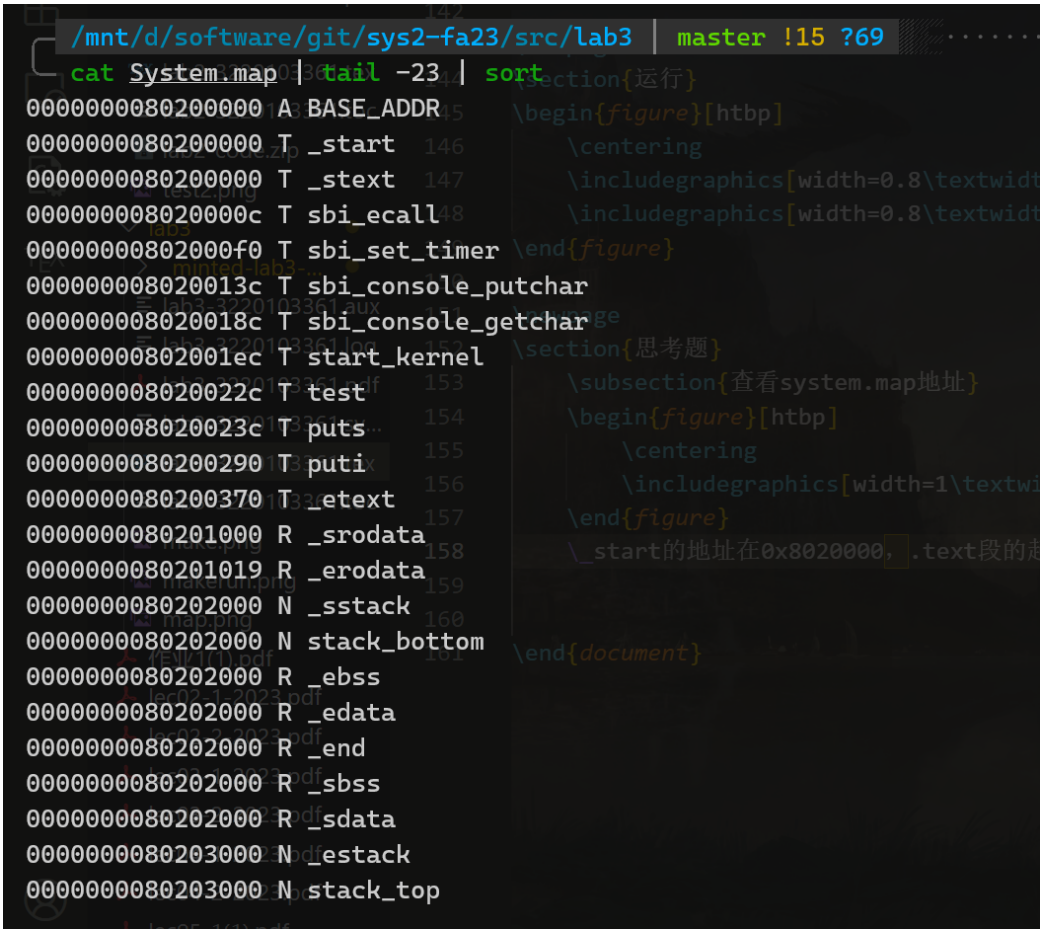
```
/mnt/d/software/git/sys2-fa23/src/lab3 | master !15 ?69
make
make -C lib all
make[1]: 进入目录“/mnt/d/software/git/sys2-fa23/src/lab3/lib”
riscv64-linux-gnu-gcc -g3 -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fd
tic -lgcc -Wl,--nmagic -Wl,--gc-sections -I /mnt/d/software/git/sys2-fa23/src/lab3/include -I /mnt/d/softw
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
```

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.10
Boot HART Base ISA        : rv64imafdc
Boot HART ISA Extensions  : time
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 ZJU Computer System II
```

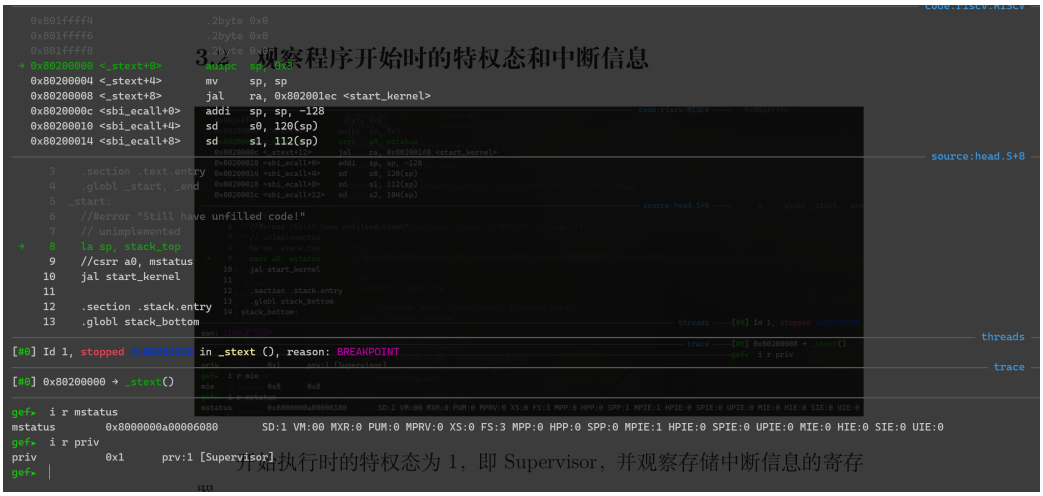
3 思考题

3.1 查看 system.map 地址



_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



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

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

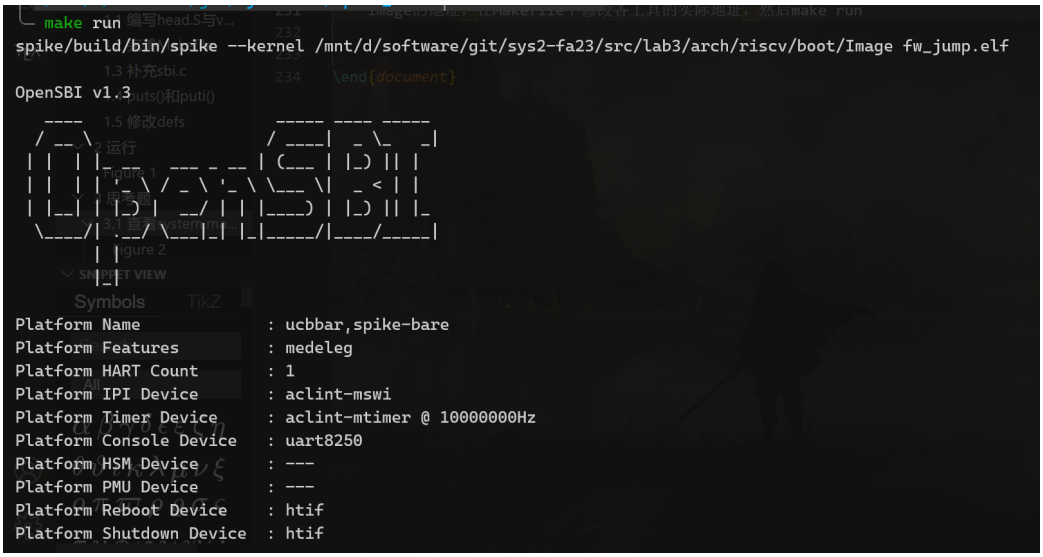
```
_start:
    la sp, stack_top
    li a0, 2023
    jal start_kernel
    ...
```

```
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdc
Boot HART ISA Extensions  : zicntr
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Info       : 0 (0x00000000)
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2023 ZJU Computer System II
```

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。



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
lf
Listening for remote bitbang connection on port 9824.
```

```
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=0x800000000014112d
[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。