CS302 Assignment 2 Report

12011702 张镇涛

Q1

-machine virt

运行帮助程序（结合grep使用）：

```
os12011702@vmos-tony:~/oslab/lab3/lab$ qemu-system-riscv64 -machine help
Supported machines are:
none                 empty machine
sifive_e             RISC-V Board compatible with SiFive E SDK
sifive_u             RISC-V Board compatible with SiFive U SDK
spike                RISC-V Spike Board (default)
spike_v1.10          RISC-V Spike Board (Privileged ISA v1.10)
spike_v1.9.1         RISC-V Spike Board (Privileged ISA v1.9.1)
virt                 RISC-V VirtIO board
```

由此可知 -machine virt 的意思是选取仿真器为 RISC-V VirtIO board

-nographic

```
os12011702@vmos-tony:~/oslab/lab3/lab$ qemu-system-riscv64 -help | grep
nographic
-nographic      disable graphical output and redirect serial I/Os to console
```

由此可知 -nographic 的意思是禁用图像输出， 重定向通过串行输入输出设备到控制台

-bios default

```
os12011702@vmos-tony:~/oslab/lab3/lab$ qemu-system-riscv64 -help | grep bios
                'sp_name': the file's name that would be passed to bios as logo
picture, if menu=on
-bios file      set the filename for the BIOS
```

由此可知， 该参数的意思是将BIOS的文件名设为 default

-device loader,file=bin/ucore.bin,addr=0x80200000

```
os12011702@vmos-tony:~/oslab/lab3/lab$ qemu-system-riscv64 -device loader,help
loader options:
  addr=<uint64>          -  (default: 0)
  cpu-num=<uint32>       -  (default: 4294967295)
  data-be=<bool>         -  (default: false)
  data-len=<uint8>       -  (default: 0)
  data=<uint64>          -  (default: 0)
  file=<str>
  force-raw=<bool>       -  (default: false)
```

`bin/ucore.bin`，这是装有我们最小化操作系统内核的二进制文件, 然后此条参数的意思就是将此程序加载至地址0x80200000运行

Q2

```
SECTIONS
{
  sections-command
  sections-command
  …
}
```

SECTIONS描述了内存布局，里面的内容告诉链接器怎样把输入段映射到输出段，以及怎样在内存里存放输出段。

```
. = BASE_ADDRESS;
```

这句话的意思是设置当前段的地址`.`为BASE_ADDRESS.

```
.text : {
    *(.text.kern_entry)
    *(.text .stub .text.* .gnu.linkonce.t.*)
}
```

.text 表示代码段，后面的冒号是语法要求，花括号里填上要链接到".text"这个输出段里面的所有输入段的名称，`*`表示通配符，可以匹配符合的任意文件。

因此，在此句代码中，工程中所有目标文件的输入段`.text.kern_entry`，`.text`，`.stub`，`.text.*`，`.gnu.linkonce.t.*`都放置到此输出段.text中。

```
PROVIDE(etext = .); /* Define the 'etext' symbol to this value */
```

PROVIDE关键字可以用来定义一个符号，比如'*etext*'，只有当它被引用但没有被定义时才使用。即：如果程序定义了该符号，链接器会默认使用程序中的定义。如果程序引用了它但没有定义它，链接器将使用链接器脚本中的定义。这句话将*etext*符号赋值为`.`。

```
.rodata : {
   *(.rodata .rodata.* .gnu.linkonce.r.*)
}
```

.rodata 输出段定义只读数据，符号规则与前文所述.text相同，因此此句表示工程中所有目标文件的输入段`.rodata`，`.rodata.*`，`.gnu.linkonce.r.*`都放置到此输出段.rodata中。

```
 . = ALIGN(0x1000);
```

重设当前段的地址`.`，`ALIGN（align）`返回位置计数器(`.`)或任意表达式对齐到下一个*align*指定边界的值，必须为2的整数次幂。

因此这句话的意思是重设当前段地址为到下一个0x1000字节对齐边界的值，也就是下一个内存页的起始处。

Q3

`memset(edata, 0, end - edata)`

注意到kernel.ld中的这段代码：

```
PROVIDE(edata = .);
.bss : {
    *(.bss)
    *(.bss.*)
    *(.sbss*)
}

PROVIDE(end = .);
```

我们可以发现`edata`是`bss`段的开始地址,而`end`是`bss`段的结束地址，因此这句话就是将整个`bss`段的数据内容全部初始化为0。

Q4

```
int cputs(const char *str) {
    int cnt = 0;
    char c;
    while ((c = *str++) != '\0') {
        cputch(c, &cnt);
    }
    cputch('\n', &cnt);
    return cnt;
}
```

上述代码是在kern/libs/stdio.c中的cputs实现，可以发现，这段代码的输入为char*，通过while循环指针位置按字符一个一个调用`cputch()`进行单字符输出。

```
static void cputch(int c, int *cnt) {
    cons_putc(c);
    (*cnt)++;
}
```

在该文件下找到`cputch()`,发现其将该单字符输入交给另一个函数`cons_putc()`， 因此需要找到该函数。

```
void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }
```

此方法调用了sbi_console_putchar()，在libs/sbi.c中

```
uint64_t SBI_SET_TIMER = 0;
uint64_t SBI_CONSOLE_PUTCHAR = 1;
uint64_t SBI_CONSOLE_GETCHAR = 2;
uint64_t SBI_CLEAR_IPI = 3;
uint64_t SBI_SEND_IPI = 4;
uint64_t SBI_REMOTE_FENCE_I = 5;
uint64_t SBI_REMOTE_SFENCE_VMA = 6;
uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;
uint64_t SBI_SHUTDOWN = 8;

uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t
arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n"
        "ecall\n"
        "mv %[ret_val], x10"
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1),
[arg2] "r" (arg2)
        : "memory"
    );
    return ret_val;
}

void sbi_console_putchar(unsigned char ch) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
}
```

可以看到，此方法将ch隐式类型转换为int64_t后作为输入，调用sbi_call(), 并且使用了**内联汇编**，使用mv命令将参数的值存放在寄存器中，再通过命令ecall进行调用实现单字符输出。

Q5

代码修改：

在sbi.c 中增加sbi_shutdown()函数，此函数调用sbi_call，并设置参数为SBI_SHUTDOWN

init.c × console.c × console.h × sbi.c ×

```c
uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;
uint64_t SBI_SHUTDOWN = 8;

uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n"
        "ecall\n"
        "mv %[ret_val], x10"
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2)
        : "memory"
    );
    return ret_val;
}

void sbi_shutdown(void){
    sbi_call(SBI_SHUTDOWN,0,0,0);
}

```

在sbi.h中增加此函数声明

init.c × console.c × console.h × sbi.c × sbi.h ×

```c
#ifndef _ASM_RISCV_SBI_H
#define _ASM_RISCV_SBI_H

typedef struct {
  unsigned long base;
  unsigned long size;
  unsigned long node_id;
} memory_block_info;

unsigned long sbi_query_memory(unsigned long id, memory_block_info *p);

unsigned long sbi_hart_id(void);
unsigned long sbi_num_harts(void);
unsigned long sbi_timebase(void);
void sbi_set_timer(unsigned long long stime_value);
void sbi_send_ipi(unsigned long hart_id);
unsigned long sbi_clear_ipi(void);
void sbi_shutdown(void);

void sbi_console_putchar(unsigned char ch);
int sbi_console_getchar(void);
```

对此函数在console.c中进行包装，包装为shutdown()函数供init.c直接调用

init.c | console.c

```
#include <sbi.h>
#include <console.h>

/* kbd_intr - try to feed input characters from k
void kbd_intr(void) {}

/* serial_intr - try to feed input characters fro
void serial_intr(void) {}

/* cons_init - initializes the console devices */
void cons_init(void) {}

/* cons_putc - print a single character @c to con
void cons_putc(int c) { sbi_console_putchar((unsi

void shutdown(void){
    sbi_shutdown();
}

/* *
 * cons_getc - return the next input character fr
 * or 0 if none waiting.
 * */
int cons_getc(void) {
    int c = 0;
    c = sbi_console_getchar();
```

C ▾ Tab Width: 8

同样在console.h中进行函数声明

init.c | console.c | console.h

```
#ifndef __KERN_DRIVER_CONSOLE_H__
#define __KERN_DRIVER_CONSOLE_H__

void cons_init(void);
void cons_putc(int c);
int cons_getc(void);
void serial_intr(void);
void kbd_intr(void);
void shutdown(void);

#endif /* !__KERN_DRIVER_CONSOLE_H__ */
```

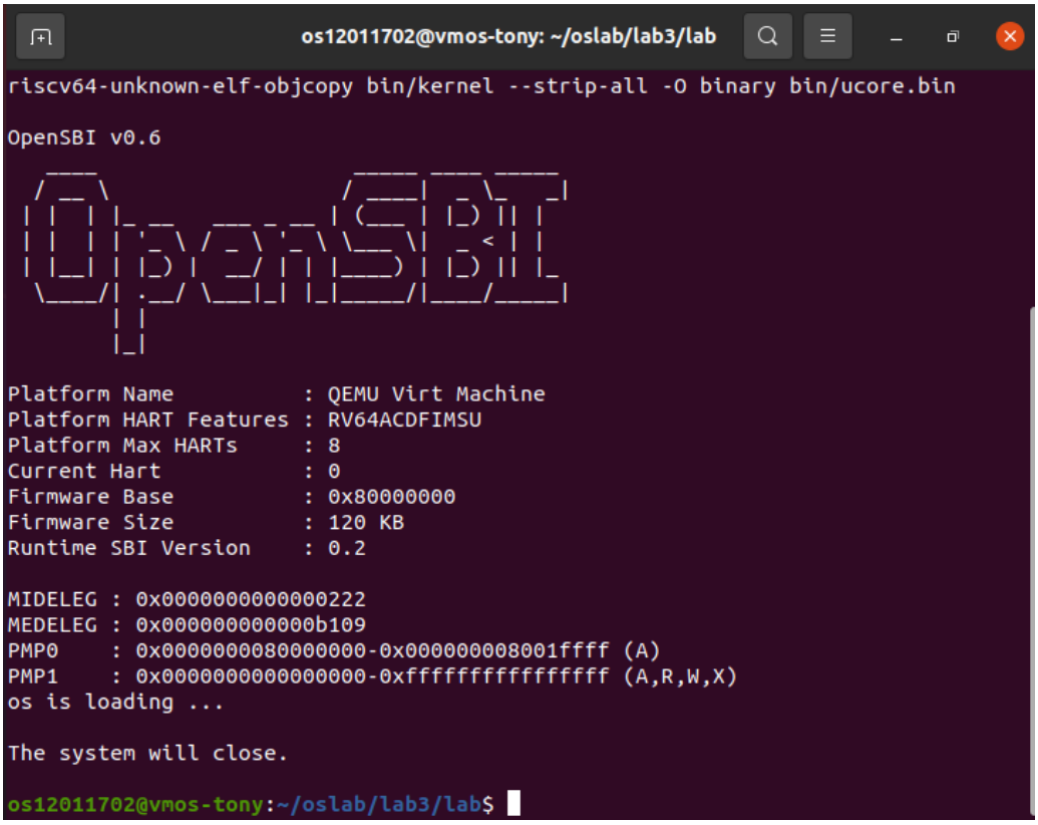
最后修改init.c,直接调用该函数

init.c | console.c

```
#include <riscv.h>
#include <stdio.h>
#include <string.h>
#include <trap.h>

int kern_init(void) __attribute__((noreturn));
void grade_backtrace(void);
static void lab1_switch_test(void);

int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    const char *message = "os is loading ...\n";
    cputs(message);
    cputs("The system will close.\n");
    shutdown();

    // clock_init();
    // ----------start------------

    // -----------end-------------

    while (1)
        ;
}
```

C ▾ Tab Width: 8 ▾

运行结果截图：


os12011702@vmos-tony: ~/oslab/lab3/lab
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin
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)
os is loading ...
The system will close.
os12011702@vmos-tony:~/oslab/lab3/lab$