

# CS334 Assignment1

SID: 12012919

Name: 廖铭騫

1) [20pts] `make qemu` 指令将指向makefile对应的label, 该指令对应于:

```
qemu-system-riscv64 \  
-machine virt \  
-nographic \  
-bios default \  
-device loader,file=bin/ucore.bin,addr=0x80200000
```

请解释以上指令中每个参数的作用

`-machine virt` 指定用来仿真模拟的机器: RISC-V VirtIO board 仿真板

`-nographic` 禁用图像输出, 指定是使用非图形化界面。且将串行IO重定向输入输出到控制台。

`-bios default` 加载指定的文件, 将OpenSBI 代码加载到 0x80000000 起始处

`-device loader, file=bin/ucore.bin, addr=0x80200000` 基于一种 Misc 设备加载器添加设备, 使用 bin/ucore.bin 作为直接运行程序, 且加载程序的程序入口地址位于 0x80200000

2) [20pts] 请查阅资料, 理解并解释/lab/tools/kernel.ld文件中每一行的作用 (<https://sourceware.org/binutils/docs/Ld/Scripts.html>)

```
/* Simple linker script for the ucore kernel.  
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */  
  
OUTPUT_ARCH(riscv) /*Specify a particular output machine architecture.  
In this case, the machine architecture is riscv*/  
ENTRY(kern_entry) /*set the entry point. In this case, the entry point  
is set as kern_entry*/  
BASE_ADDRESS = 0x80200000; /* assign the 0x80200000 to BASE_ADDRESS */  
  
SECTIONS /*A keyword followed by a series of symbol assignments and  
output section descriptions enclosed in curly braces*/  
{  
    /* Load the kernel at this address: "." means the current address  
    */  
    . = BASE_ADDRESS; /* "." as the location counter, here assign  
0x80200000 to it*/
```

```

/* combine all the .text.kern_entry, .text .stub, .text.*,
.gnu.linkonce.t.* input file into a .text section */
.text : { /* programming code output sections, followed by the
input files which should be placed into the output section*/

    *(.text.kern_entry) /* The input section that should be placed
into the .text output section*/
    *(.text .stub .text.* .gnu.linkonce.t.*) /* The input section
that should be placed into the .text output section*/
}

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

/*combine all the .rodata .rodata.* .gnu.linkonce.r.* input files into
a .rodata output section*/
.rodata : { /* read-only data output sections, followed by the
input files which should be placed into the output section*/
    *(.rodata .rodata.* .gnu.linkonce.r.*) /* The input section
that should be placed into the .text output section*/
}

. = ALIGN(0x1000); /* Adjust the address for the data segment to
the next page */

/* The read-write initialized data segment, the usage is similar
to .text */
.data : {
    /* The input section that should be placed into the .data output
section*/
    *(.data)
    *(.data.*)
}

/* The sdata segment*/
.sdata : {
    /* The input section that should be placed into the .sdata output
section*/
    *(.sdata)
    *(.sdata.*)
}

PROVIDE(edata = .);/* Define the 'edata' symbol to the location of
current value of location counter */

/* The read-write zero initialized data segment, the usage is
similar to .text, followed by the input files which should be placed
into the output .bss section */
.bss : {

```

```

    /* The input sections that should be placed into the .bss output
    section*/
    *(.bss)
    *(.bss.*)
    *(.sbss*)
}

PROVIDE(end = .);/* Define the 'end' symbol to the location of
current value of location counter */

/DISCARD/ : { /* Any input sections which are assigned to an output
section named '/DISCARD/' are not included in the output file.*/
    *(.eh_frame .note.GNU-stack) /*All ".eh_frame" and ".note.GNU-
stack" input section willnot included in the output file*/
}
}

```

**3) [10pts]** 请解释 /lab/kern/init/init.c 中 main函数中 `memset(edata, 0, end - edata);` 的参数及语句作用。（需要读到的代码有init.c, kernel.ld）

#### 参数：

edata, 标志着 .bss output section 的起始位置

0, 标志着将要给对应内存块填充的内容, 初始化的值

end - edata, 通过阅读 kernel.ld 代码, 发现 end 代表着 .bss output section 的结束位置, 而 end - edata 恰好就是 .bss section 的大小

#### 语句作用

该语句通过显式地规定需要填充字节块的大小, 将对应的内存区域 (.bss output section) 全部填充为 0, 进行区域的初始化操作。

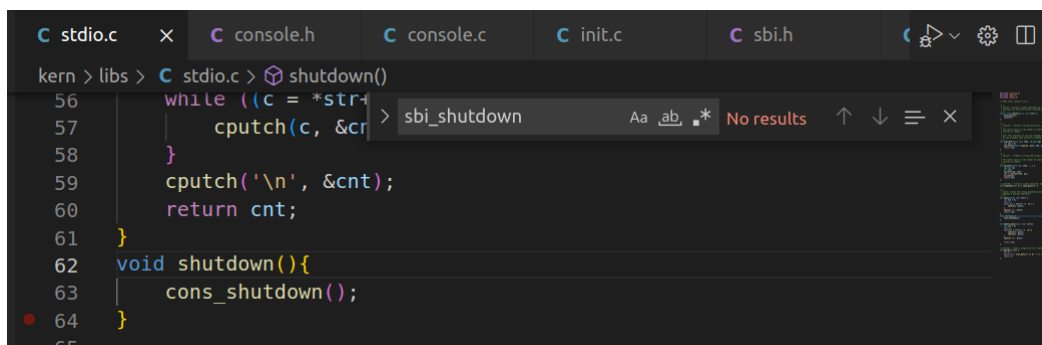
### 4) [20pts] 请描述cputs()指令是如何通过sbi打印字符的。

cputs() 先是通过调用cputch(), 输出字符串中的单字符直到遇到'\0', 最后输出'\n', cputch() 的作用是将单个字符写入标准输出流。之后 cputch() 通过调用 cons\_putc(), 而 cons\_putc() 又调用sbi\_console\_putchar(), 通过传入需要在控制台中打印的字符来进一步传给sbi\_call(), 在 sbi\_call 当中传入 SBI\_CONSOLE\_PUTCHAR 作为 sbi\_type 指示指令为打印字符, 剩余参数还有要打印的字符, 以及两个0 进行 sbi\_call() 的调用。

在 sbi\_call() 当中, 通过使用内联汇编, 将传入参数指定的寄存器当中的值存入相应的寄存器 (sbi\_type 的值存入 x17寄存器, 传入的字符 arg0 的值存入 x10寄存器), 之后使用 `ecall` 进行系统调用, 交给 OpenSBI 执行, 将传入的字符在屏幕中打印显示出来, 最后再将 x10 寄存器的值, 也就是 arg0 作为返回值返回。

5) [30pts] 编程题 请理解使用ecall打印字符的原理，实现一个 shutdown() 关机函数。（所有修改到的代码请截图和运行结果截图一起放在报告中）

为了能在 init.c 当中使用 shutdown() 函数，先在 stdio.h 当中声明该方法，之后在 stdio.c 当中实现。经过查看 cputch() 方法，发现是通过调用 console.h 当中的 cons\_putc() 来实现，所以在 shutdown() 函数当中，我们也通过进一步调用 console.h 当中的，将要实现的 cons\_shutdown() 来进一步实现，截图如下。

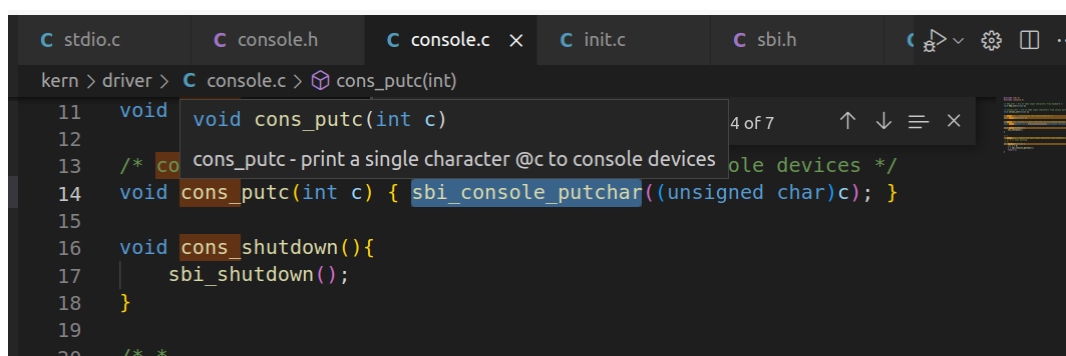


```
stdio.c x console.h console.c init.c sbi.h
kern > libs > C stdio.c > shutdown()
56 while ((c = *str)
57     cputch(c, &cnt);
58 }
59 cputch('\n', &cnt);
60 return cnt;
61 }
62 void shutdown(){
63     cons_shutdown();
64 }
65
```

之后，我们关注于 cons\_shutdown() 函数的实现。

通过对 cons\_putc() 实现的查看，发现它进一步调用了 sbi.h 当中的 sbi\_console\_putchar() 方法进行系统调用实现。

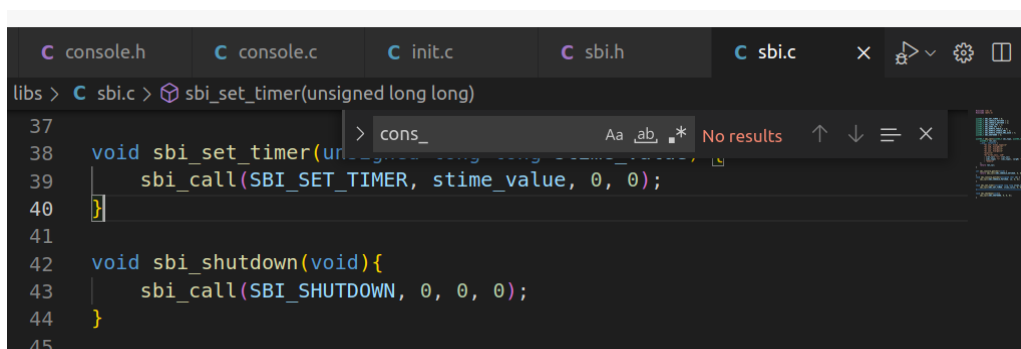
于是我们参考这样的实现方法，在 cons\_shutdown() 当中调用 sbi\_shutdown()方法



```
stdio.c console.h console.c x init.c sbi.h
kern > driver > C console.c > cons_putc(int)
11 void cons_putc(int c)
12
13 /* cons_putc - print a single character @c to console devices */
14 void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }
15
16 void cons_shutdown(){
17     sbi_shutdown();
18 }
19
20 /* *
```

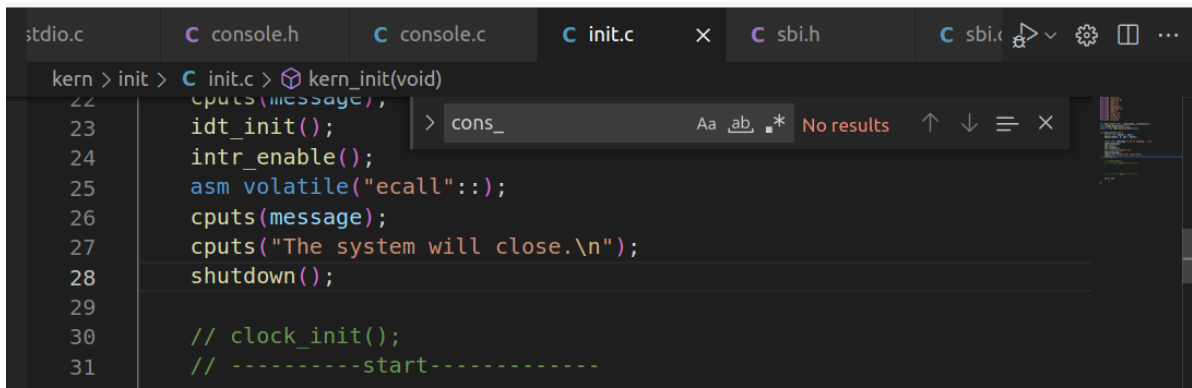
之后发现在 sbi.c 当中，并没有实现该方法，通过查看 sbi\_console\_putchar() 方法，发现是通过调用 sbi\_call() 进行实现。

最终，我们通过阅读 sbi\_call() 方法的参数作用，实现了 sbi\_shutdown() 方法



```
console.h console.c init.c sbi.h sbi.c x
libs > C sbi.c > sbi_set_timer(unsigned long long)
37
38 void sbi_set_timer(unsigned long long time_value) {
39     sbi_call(SBI_SET_TIMER, time_value, 0, 0);
40 }
41
42 void sbi_shutdown(void){
43     sbi_call(SBI_SHUTDOWN, 0, 0, 0);
44 }
45
```

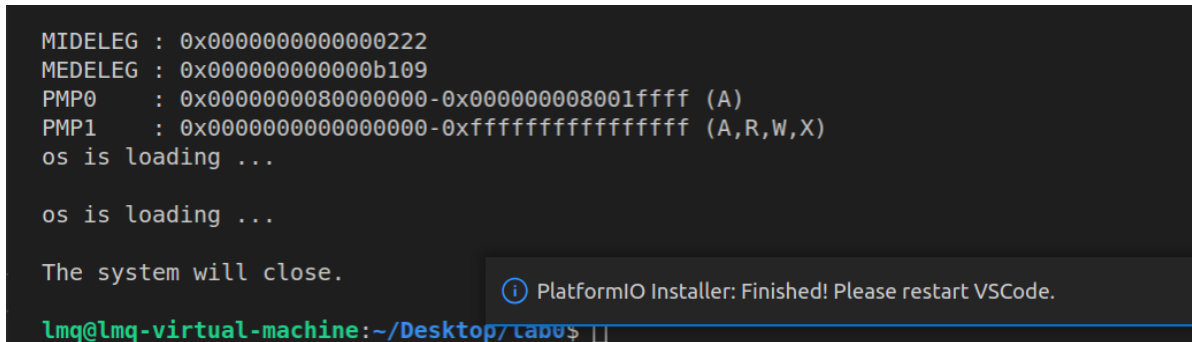
最终运行截图如下：



The screenshot shows the VS Code editor interface with several tabs open: `stdio.c`, `console.h`, `console.c`, `init.c`, `sbi.h`, and `sbi.c`. The `init.c` file is active, showing the following code:

```
kern > init > C init.c > kern_init(void)
22  cputs(message);
23  idt_init();
24  intr_enable();
25  asm volatile("ecall");
26  cputs(message);
27  cputs("The system will close.\n");
28  shutdown();
29
30  // clock_init();
31  // -----start-----
```

A search bar is visible above the code, showing the search term `cons_` and the result `No results`.



The screenshot shows a terminal window with the following output:

```
MIDELEG : 0x0000000000000222
MEDELEG : 0x0000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffff (A,R,W,X)
os is loading ...

os is loading ...

The system will close.
```

A notification banner at the bottom of the terminal reads: `PlatformIO Installer: Finished! Please restart VSCode.`

The terminal prompt is `lmq@lmq-virtual-machine:~/Desktop/lab0$`.