Homework #1

周雨扬
2000013061

1 Challanges

本次作业完成了所有的代码补全任务，没有做 Challange。

2 Exercise 3

在文件 `boot/boot.S, Line 59` 之后进入了 32 位模式。在 `boot/boot.S, Line 28 42` 中设置了 A20 信号，使得大于 1MB 的内存不再被自动设置为 0.

`obj/boot/boot.asm, Line 306, 0x7d6b: call *0x10018`. `obj/kern/kernel.asm, Line 20, 0xf010000c, movw $0x1234,0x472`

`kern/entry.S, line 44, movw $0x1234,0x472`.

由于内核的 ELF 头文件部分包含了每一个 sector 的包含长度的信息，因此加载器可以对每一个 sector 的长度简单相加，获得代码长度。

3 Exercise 6

加载 kernel 前 0x100000 处全部都是 0，加载 kernel 后 `0x100000` 处存储了内核的代码。

4 Exercise 7

存储在 `0x100000` 处的代码被拷贝到了虚拟内存 `0xf0100000` 处。

如果注释掉，之后第一个被执行的指令是 `add %al,(%eax)`, 而其机器码均为 0.

5 Exercise 8

`console.c` 提供了一个接口，在该接口下能够在不同的模式下输出相同的字符：(包括了在终端下，在操作系统程序下，以及在并行化的接口下等)。`print.c` 通过调用 `console.c` 来完成单个字符的输出工作。

在这种情况下，如果界面上已经占满了字符，则我们需要删除最上面一行，并将剩余的字符上移一行，来空出新的字符输出所需要的空间。

`fmt` 指向 texttt"x %d, y %x, z %d" 的第一个字符，`ap`指向 x 的指针.(事实上是 ___VA_ARGS___ 多变量的指针)。

代码实现

代码实现在 `kern/printfmt.c` 中。观察得知在标准 c++ 中 8 进制数字均采用无符号模式输出，因此使用函数 `getuint` 获取参数中对应的数字。之后观察上面的输出发现我们只需要定义进制，就

可以直接使用现成的代码，因此直接 goto number 即可。

6 Exercise 9

初始化栈的代码在 kern/entry.S, line 69 ∼ 80. 同时栈所占用的部分内存在该文件末尾被定义，kern/entry.S, line 86 ∼ 95.

初始的时候指针 %esp 指向栈空间内存最大的地方，代码中定义为 %esp = 0xf0110000。

7 Exercise 10

每一次递归调用 backtrace 的时候会向栈里压入 8 个元素，其中包括了被调用程序刚进入时的栈指针 %ebp，程序返回的地址，调用 backtrace 函数的参数，若干空数字的占位符等。

8 Exercise 11 & 12

代码实现在 kern/moniter.c 和 kern/kdebug.c 中。观察栈结构可以知道每一次进入子程序前栈顶会存储 call 记录的返回地址，同时在进入该子程序的时候会向栈中压入调用其的子程序进入是的栈指针 (mov %esp,%ebp 存储)。

同时观察发现，该内核利用栈来传输子程序参数，且参数按照从前向后的顺序从栈顶到站地排列，因此我们可以递归通过记录 %ebp 的值，来一层一层的剥离函数调用的过程。据此我们可以直接利用一次 while 循环解决该问题。这部分代码可以见 kern/moniter.c, Line 58 ∼ 95.

同时观察 kern/moniter.c，发现命令行交互的时候会查询 commands 里的指令。因此在 commands 中直接加入 backtrace 对应的信息即可，这部分见 kern/moniter.c, Line 27.

同时我们也需要在给定 eip 的时候，查询其在代码中的位置。这里我们直接调用 kern/kdebug.c 中的相关实现，利用其返回值即可获取这些信息。kern/kdebug.c 需要我们写在给定行区间的情况下，查询当前地址对应 stab 中的哪一个信息。这里我们仍然可以使用提供的 stab_binsearch 的实现进行查找，只需要检查返回的行是否存在即可。这部分见 kern/kdebug.c, Line 182 ∼ 186.