



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

操作系统实验报告

Lab1 print stackframe

2014074 费泽锬

年级：2020 级

专业：信息安全

指导教师：宫晓利

2022 年 10 月 6 日

摘要

Lab1 主要包括 5 个练习，练习 1 为对实验环境，实验工具以及实验内容的熟悉。练习 2 到练习 4 主要是阅读代码的练习，要理解从 BIOS 到 bootloader 的逻辑，了解从 8086 到 80386 中间的转换，实模式到保护模式以及 A20 的启用等，练习 5 则是主要编程内容，需要手动实现打印函数调用栈的具体 ebp, eip 和 args 信息等。

关键字：makefile, BIOS, bootloader, stack_frame

目录

| | |
|-------------------|----------|
| 一、 实验内容 | 1 |
| (一) 练习一 | 1 |
| (二) 练习二 | 1 |
| (三) 练习三 | 3 |
| (四) 练习四 | 4 |
| (五) 练习五 | 5 |
| 二、 总结 | 7 |

一、 实验内容

(一) 练习一

练习一的主要内容是探究操作系统镜像文件 ucore.img 的生成过程，了解 Makefile 中每一条相关命令和命令参数的含义。并探究被系统认为是符合规范的硬盘主引导扇区的特征是什么。

在本个练习中遇到的最大的问题就是首次使用 qemu 和 gdb 的各种环境和操作问题。在 Ubuntu22.04 上如果只使用 apt-get 下载 qemu 的话，在 bash 中输入 qemu 会显示“无该指令”的 error，在 /usr/bin 中查看后，发现 apt-get 会下载适配各个不同硬件的版本，这时我们需要将 qemu 与 qemu-system-i386 建立软连接，再使用 qemu 命令即可。

指令建立软连接：ln -s /usr/bin/qemu-system-i386 /usr/bin/qemu

makefile 的编译指令如下图：

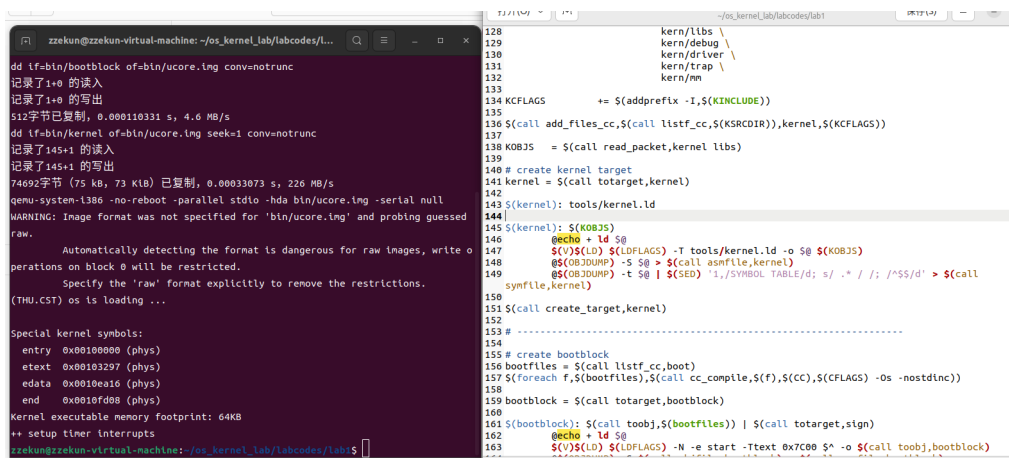


图 1: makefile 中的指令对照

在观察练习 1 的实验结果的时候，也遇到了一些问题，那就是为什么使用 make "V=" 的命令之后显示只有一条 qemu 指令呢，后来把 ucore.img 删掉之后，又多了些链接文件的命令，但是和 makefile 的内容还是大相径庭。

经过研究，发现原来是因为我再 clone 库之后运行了 make qemu 命令，导致原来的.c 文件已经被编译成.o 文件了，ucore.img 镜像也生成好了，所以我们只需要把库重新 clone 一遍，在第一次运行时直接使用 make "V=" 命令就可以看到完整 makefile 中的内容了，包含了前一部分的对.c 文件编译的命令过程。

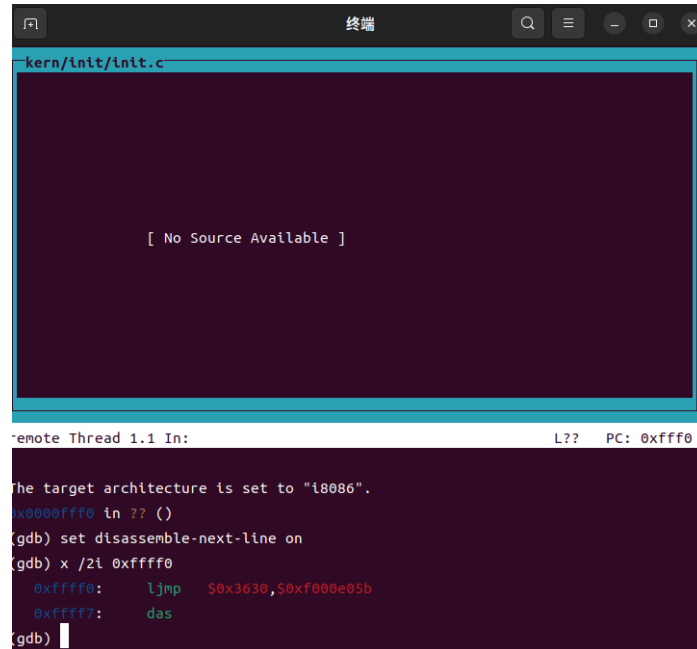
主引导扇区大小为 512 字节；多余的空间填 0，与初始化有关；第 510 个（倒数第二个）字节是 0x55；第 511 个（倒数第一个）字节是 0xAA。

(二) 练习二

练习 2 的主要内容是为了熟悉 qemu 和 gdb 的使用，以及 BIOS 的部分内容，需要对 BIOS 的内容以及 bootloader 的部分使用 gdb 进行单步调试。

经过对实验手册的认真研读，我们可以发现如果在 /Lab1 的目录下使用 make debug 命令应该就能够进入 gdb 并且能停止在第一条指令处即 0xffff0 处（只包含了偏移），但是在实验的过程中使用 make debug 命令，gdb 竟然直接停止在了 0x10000 处。

经过深刻的研究，发现竟然是因为 tools 目录下的 gdbinit 文件导致的，在文件中竟然提前设置好了一个断点设置在 kernel init 处，所以单纯的使用 make debug 命令是看不到第一条指令的结果的，我们只需要按照实验指导书上描述的，将反汇编的功能打开设置成 x/ni (n 取决于想要查看多少条指令)，将原来的断点设置删除即可，就能看到成功的结果如下：



```

kern/init/init.c

[ No Source Available ]

remote Thread 1.1 In: L?? PC: 0xffff0

The target architecture is set to "i8086".
0x0000ffff in ?? ()
(gdb) set disassemble-next-line on
(gdb) x /2i 0xffff0
0xffff0: jmp $0x3630,$0xf000e05b
0xffff7: das
(gdb)

```

图 2: 第一条指令

需要注意的是，如果想要查看第一条指令的汇编代码，一定要展示的是 0xffff0 处的代码，而不是 0xff0 处的代码，因为有个原始的偏移量是 0xf0000，这样才能看到希望看到的 jmp 长跳转指令。

当解决这些问题之后，这个练习也就迎刃而解了，实验结果如下：



```

0x0000ffff in ?? ()
The target architecture is set to "i8086".
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00: cli
0x7c01: cld
(gdb)

```

图 3: 0x7c00

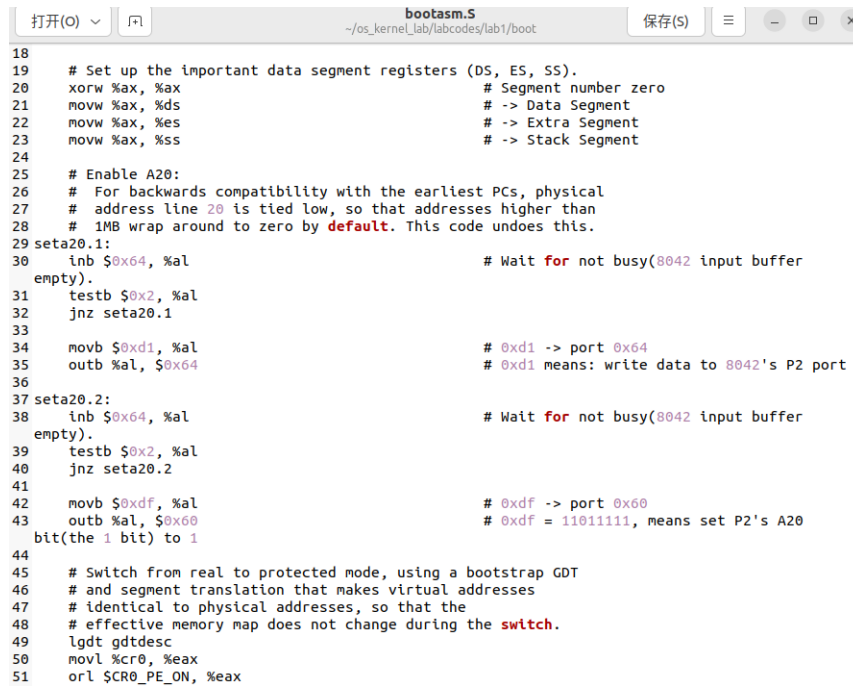
在加载完 BIOS 的部分之后，bootloader 的第一条指令即位于 0x7c00 的位置，我们可以看到 cli 和紧邻的 cld 指令，cli 指令禁止中断发生，cld 指令将方向标志位清零。为接下来的初始化描述符和堆栈作准备。

实验结果与 bootasm.S 和 bootblock.asm 进行比较发现其汇编指令是相同的。

(三) 练习三

BIOS 将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行 bootloder。练习 3 主要分析 bootloder 是如何完成从实模式进入保护模式的。

在练习三中并没有出现什么大问题，那就稍微分析一下 bootloder 中的第一部分内容。先展示一下 lab1/boot/bootasm.S 中的部分源码：



```

18
19 # Set up the important data segment registers (DS, ES, SS).
20 xorw %ax, %ax          # Segment number zero
21 movw %ax, %ds          # -> Data Segment
22 movw %ax, %es          # -> Extra Segment
23 movw %ax, %ss          # -> Stack Segment
24
25 # Enable A20:
26 # For backwards compatibility with the earliest PCs, physical
27 # address line 20 is tied low, so that addresses higher than
28 # 1MB wrap around to zero by default. This code undoes this.
29 seta20.1:
30 inb $0x64, %al          # Wait for not busy(8042 input buffer
   empty).
31 testb $0x2, %al
32 jnz seta20.1
33
34 movb $0xd1, %al          # 0xd1 -> port 0x64
35 outb %al, $0x64          # 0xd1 means: write data to 8042's P2 port
36
37 seta20.2:
38 inb $0x64, %al          # Wait for not busy(8042 input buffer
   empty).
39 testb $0x2, %al
40 jnz seta20.2
41
42 movb $0xdf, %al          # 0xdf -> port 0x60
43 outb %al, $0x60          # 0xdf = 11011111, means set P2's A20
   bit(the 1 bit) to 1
44
45 # Switch from real to protected mode, using a bootstrap GDT
46 # and segment translation that makes virtual addresses
47 # identical to physical addresses, so that the
48 # effective memory map does not change during the switch.
49 lgdt gdtdesc
50 movl %cr0, %eax
51 orl $CR0_PE_ON, %eax

```

图 4: Enable A20

上图虽然主要展示了启用 A20 的部分（即通过 0x64 和 0x60 端口等待输入信号启动 A20）但是也能够看到 bootasm.S 中包含了从实模式到保护模式的三个操作，第一个操作就是为了向下兼容到 8086 实模式的“回卷”，而开启 A20 功能，使得能够使用 32 位地址。

第二个操作就是将 CR0 寄存器的第 0 位置 1 表示处于保护模式。

第三个操作则是设置段寄存器，初始 GDT 表并且建立初始化堆栈。如下图所示：

```

50 movl %cr0, %eax
51 orl $CR0_PE_ON, %eax
52 movl %eax, %cr0
53
54 # Jump to next instruction, but in 32-bit code segment.
55 # Switches processor into 32-bit mode.
56 ljmp $PROT_MODE_CSEG, $protcseg
57
58 .code32                                # Assemble for 32-bit mode
59 protcseg:
60 # Set up the protected-mode data segment registers
61 movw $PROT_MODE_DSEG, %ax            # Our data segment selector
62 movw %ax, %ds                        # -> DS: Data Segment
63 movw %ax, %es                        # -> ES: Extra Segment
64 movw %ax, %fs                        # -> FS
65 movw %ax, %gs                        # -> GS
66 movw %ax, %ss                        # -> SS: Stack Segment
67
68 # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
69 movl $0x0, %ebp
70 movl $start, %esp
71 call bootmain
72
73 # If bootmain returns (it shouldn't), loop.
74 spin:
75 jmp spin
76
77 # Bootstrap GDT
78 .p2align 2                            # force 4 byte alignment
79 gdt:
80 SEG_NULLASM                          # null seg
81 SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for bootloader and kernel
82 SEG_ASM(STA_W, 0x0, 0xffffffff)      # data seg for bootloader and kernel
83
84 gdtdesc:
85 .word 0x17                          # sizeof(gdt) - 1
86 .long gdt                          # address gdt

```

图 5: GDT 初始化

我们可以看到很有意思的一句代码：.word 0x17

.word 的作用可以理解为一个大小为 word 数据，那么这里的 0x17=23，应该是指 GDT 描述表（共 48 位）的后 16 位，后 16 位代表着这个 GDT 表的大小限制（大小为 $8*N-1$ ，N 指的是段描述符的数量），所以对于 Lab1 的 ucore 而言段描述符应该有 3 个 GDT Entry，这一点在后面的代码中正好能够验证。

(四) 练习四

通过阅读 bootmain.c，了解 bootloader 如何加载 ELF 文件。通过分析源代码和通过 qemu 来运行并调试 bootloader & OS，探究 bootloader 如何读取硬盘扇区的以及是如何加载 ELF 格式 OS 的。

这一部分是 bootloader 的后半部分，前半部分在完成了从实模式向保护模式的转换之后，这一步则是加载 ucore 操作系统，并将指挥权交给操作系统。

话不多说，直接先上源码：

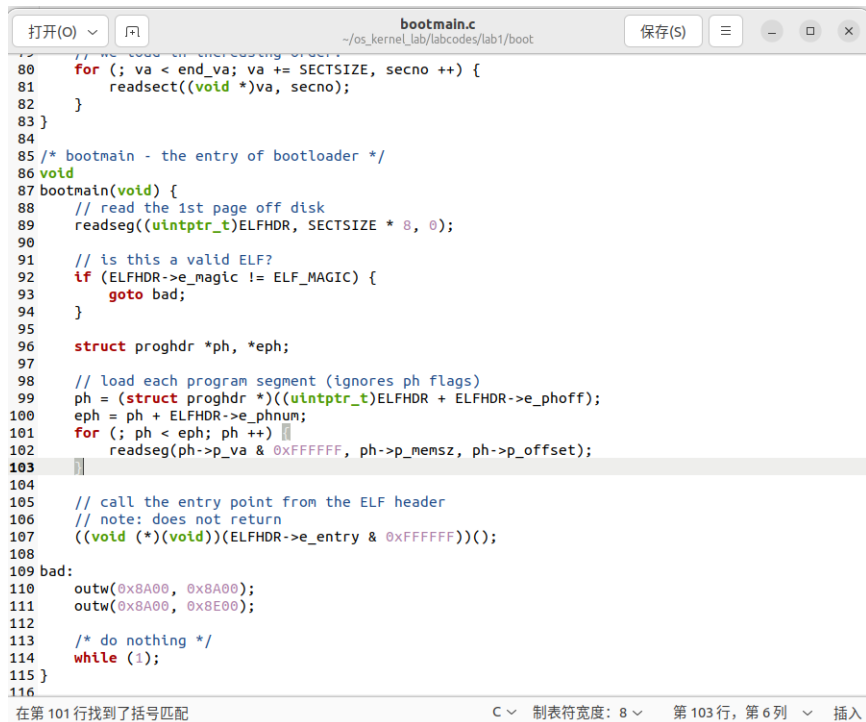


图 6: bootmain

对于读取硬盘中的操作系统程序而言，需先从硬盘中读取程序，实验中将程序放到了硬盘前部。读取硬盘的流程如下：

1. 等待磁盘准备好；
2. 发出读取扇区的命令；
3. 等待磁盘准备好；
4. 把磁盘扇区数据读到指定内存；

可以看到 waitdisc 等函数完成了这些工作，接下来通过 readsect 函数读取扇区，本实验为了完整性，源码中直接读取了 8 个扇区（一页）的内容到内存中。（但是不明白为什么 8 个扇区被加载到了 0x10000 处，也没在源代码中看出来这一点...）

接下来就是将 ELF 格式的操作系统文件进行加载，校验 e_magic 字段，在 ELF header 中读取程序头表的偏移地址，再结合 program header 中的段的入口数目和位置偏移等数值，将段和程序加载到内存中，最后跳转到 entry 处执行操作系统，完成 bootloader 的工作。

（五）练习五

练习 5 主要是补充 kdebug.c 中的 print_stack_frame 函数，在 make qemu 命令后跟踪具体的函数调用栈的相关信息。这里对于函数调用栈的结构就不在赘述了。

我们可以根据 kdebug.c 中的纯英文提示信息，一步一步地完成代码的编写。应当注意的是，上文提示中让我们使用的是 readebp() 函数，该函数的返回值是 uint_32 类型的变量，所以该变量只是 ebp 的数值而不是指向 ebp 的指针，刚开始使用的是数组寻址的方式，那么就会得到结果如下：

```

zzekun@zzekun-virtual-machine: ~/os_kernel_lab/labcodes/L...
zzekun@zzekun-virtual-machine:~/os_kernel_lab/labcodes/lab1$ make qemu
+ cc kern/debug/kdebug.c
kern/debug/kdebug.c: In function 'print_stackframe':
kern/debug/kdebug.c:314:38: error: subscripted value is neither array nor pointer nor vector
    314 |         args[j] = (debug_ebp + 2)[j];
        |                             ^
kern/debug/kdebug.c:318:30: error: subscripted value is neither array nor pointer nor vector
    318 |         debug_ebp = debug_ebp[0];
        |                             ^
kern/debug/kdebug.c:319:30: error: subscripted value is neither array nor pointer nor vector
    319 |         debug_eip = debug_ebp[1];
        |                             ^
make: *** [Makefile:137: obj/kern/debug/kdebug.o] 错误 1
zzekun@zzekun-virtual-machine:~/os_kernel_lab/labcodes/lab1$

```

图 7: 寄

然后可以机智地每次循环设一个 `ebp_addr` 指针变量来更新 `ebp` 和 `eip` 的地址，来一步步向下寻找调用关系即可。

本次实验的代码纯自己手写了一版，因为直接将 `readebp()` 的值赋值给 `uint_32*` 类型的变量，还有 `warning` (看了答案之后，发现这里强制类型转换或者直接全部强制类型转换就 okk 了)。最终的实现结果如下：

```

zzekun@zzekun-virtual-machine: ~/os_kernel_lab/labcodes/L...
args0: -- 0x00000000 --
args1: -- 0x00000000 --
args2: -- 0x00000000 --
args3: -- 0x00000000 --
<unknown>: -- 0xf00ff52 --

ebp: -- 0x00000000 --
eip: -- 0x00000000 --
args0: -- 0xf00e2c3 --
args1: -- 0xf00ff53 --
args2: -- 0xf00ff53 --
args3: -- 0xf00ff54 --
<unknown>: -- 0xffffffff --

ebp: -- 0xf00ff53 --
eip: -- 0xf00ff53 --
args0: -- 0x00000000 --
args1: -- 0x00000000 --
args2: -- 0x00000000 --
args3: -- 0x00000000 --
<unknown>: -- 0xf00ff52 --

++ setup timer interrupts
zzekun@zzekun-virtual-machine:~/os_kernel_lab/labcodes/lab1$

```

图 8: stack frame

Github: <https://github.com/FZaKK/OS-Work>

可以直接点击该超链接 [Github](https://github.com/FZaKK/OS-Work)

二、 总结

本次实验通过五个练习，初步体验了 OS 的实验环境和各实验工具，如 qemu 和 gdb 的使用。了解了操作系统从上电到启用的过程，学习到了不同的细节，如实模式到保护模式的转换以及 BIOS 和 bootloader 的作用等内容，初步感受了 OS 实验级别的代码调试和 debug，也了解到了从 8086 到 80386 的历史过程。