

中山大学计算机学院本科生实验报告

(2021 学年春季学期)

课程名称：操作系统

批改人：

实验	Lab1	专业（方向）	计算机科学与技术
学号	18329015	姓名	郝裕玮
Email	935087919@qq.com	完成日期	2021. 4. 2

1. 实验目的

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。lab1 提供了一个非常小的 bootloader 和 ucore OS，整个 bootloader 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。

通过分析和实现这个 bootloader 和 ucore OS，读者可以了解到：

计算机原理

CPU 的编址与寻址：基于分段机制的内存管理

CPU 的中断机制

外设：串口/并口/CGA，时钟，硬盘

Bootloader 软件

编译运行 bootloader 的过程

调试 bootloader 的方法

PC 启动 bootloader 的过程

ELF 执行文件的格式和加载

外设访问：读硬盘，在 CGA 上显示字符串

ucore OS 软件

编译运行 ucore OS 的过程

ucore OS 的启动过程

调试 ucore OS 的方法

函数调用关系：在汇编级了解函数调用栈的结构和处理过程

中断管理：与软件相关的中断处理

外设管理：时钟

2. 实验过程和核心代码及代码解释&实验结果

练习 1: 理解通过 make 生成执行文件的过程

1. 操作系统镜像文件 ucore.img 是如何一步一步生成的?

答：首先对 Lab1 文件执行 make "V=" 操作，执行结果如下图所示：

```
+ cc kern/init/int.c  
gcc -Ikern/lib/-fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init/int.c -o obj/kern/init/int.o  
kern/init/int.c:95:16: warning: "labs_switch_test" defined but not used [-Wunused-function]  
95 | labs_switch_test(void) {  
    | ~~~~~^~  
  
+ cc kern/lbs/stdio.c  
gcc -Ikern/lbs/-fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/lbs/stdio.c -o obj/kern/lbs/stdio.o  
+ cc kern/lbs/readline.c  
gcc -Ikern/lbs/-fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/lbs/readline.c -o obj/kern/lbs/readline.o  
+ cc kern/debug/panic.c  
gcc -Ikern/debug/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/panic.c -o obj/kern/debug/panic.o  
+ cc kern/debug/kdebug.c  
gcc -Ikern/debug/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o  
+ cc kern/debug/kmonitor.c  
gcc -Ikern/debug/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o  
+ cc kern/driver/clock.c  
gcc -Ikern/driver/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o  
+ cc kern/driver/console.c  
gcc -Ikern/driver/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o  
+ cc kern/driver/plcrq.c  
gcc -Ikern/driver/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/plcrq.c -o obj/kern/driver/plcrq.o  
+ cc kern/driver/intr.c  
gcc -Ikern/driver/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o  
+ cc kern/trap/trap.c  
gcc -Ikern/trap/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o  
kern/trap/trap.c: In function 'print_trapframe':  
kern/trap/trap.c:115:16: warning: taking address of packed member of 'struct trapframe' may result in an unaligned pointer value [-Waddress-of-packed-member]  
115 | print_regs(&tf->regs);  
    |         ^~~~  
  
+ cc kern/trap/vectors.S  
gcc -Ikern/trap/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o  
+ cc kern/trap/trapentry.S  
gcc -Ikern/trap/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o  
+ cc kern/mm/pmnc.  
gcc -Ikern/mm/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/mm/pmnc.c -o obj/kern/mm/pmnc.o  
+ cc libs/string.c  
cc -llibs/-fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o  
+ cc lbs/printfmt.c  
gcc -Ilbs/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilbs/ -c lbs/printfmt.c -o obj/lbs/printfmt.o  
+ ld bin/kernel  
ld -T i386-nostdlib-T tools/kernel.ld -o bin/kernel obj/kern/init/int.o obj/kern/lbs/stdio.o obj/kern/lbs/readline.o obj/kern/debug/panic.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/plcrq.o obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmnc.o obj/lbs/string.o obj/lbs/printfmt.o  
+ cc boot/bootasm.S  
gcc -Iboot/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilbs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o  
+ cc boot/bootmain.c  
gcc -Iboot/ -fno-built-in -Wall -ggdb -n32 -gstabs -nostdinc -fno-stack-protector -Ilbs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o  
+ cc tools/sign.c  
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o  
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign  
+ ld bin/bootblock
```

之后我们打开 Makefile 文件进行查看：

生成 ucore.img 的代码块如下所示:

[illegible]

其中：dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换。

if=文件名：输入文件名，缺省为标准输入。即指定源文件。< if=input file >

of=文件名：输出文件名，缺省为标准输出。即指定目的文件。< of=output file >

count=blocks：仅拷贝 blocks 个块，块大小等于 ibs 指定的字节数。

conv=conversion：用指定的参数转换文件。

conv=notrunc:不截短输出文件

所以由上述代码易知生成 ucore.img 需要先生成 kernel 和 bootblock，之后再创建一个 10000 字节的块，并将 bootblock 和 kernel 复制过去，最后即可生成 ucore.img。

以下为生成 kernel 的代码部分：

```
# create kernel target
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)
@echo + ld $@
$(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
@$(OBJDUMP) -S $@ > $(call asmfile,kernel)
@$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call
symfile,kernel)

$(call create_target,kernel)

# -----
```

查看 Makefile 文件，可知需用 gcc 对 kernel 目录下的.c 文件进行编译：

```
+ cc kern/init/init.c
gcc -Ikern/lib/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
kern/init/init.c:95:1: warning: 'lab1_switch_test' defined but not used [-Wunused-function]
 95 | lab1_switch_test(void) {
    | ~~~~~
+ cc kern/lib/stdio.c
gcc -Ikern/lib/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/lib/stdio.c -o obj/kern/lib/stdio.o
+ cc kern/lib/readline.c
gcc -Ikern/lib/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/lib/readline.c -o obj/kern/lib/readline.o
+ cc kern/debug/panic.c
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/panic.c -o obj/kern/debug/panic.o
+ cc kern/debug/kdebug.c
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
+ cc kern/driver/clock.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o
+ cc kern/driver/console.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o
+ cc kern/driver/picirq.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/picirq.c -o obj/kern/driver/picirq.o
+ cc kern/driver/intr.c
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o
+ cc kern/trap/trap.c
gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o
kern/trap/trap.c:115:16: warning: taking address of packed member of 'struct trapframe' may result in an unaligned pointer value [-Waddress-of-packed-member]
 115 |     print_regs(&tf->tf_regs);
    |                ^
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o
+ cc kern/trap/trapentry.S
gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc lib/string.c
gcc -Ilb/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -c lib/string.c -o obj/lib/string.o
+ cc lib/printfmt.c
gcc -Ilb/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilb/ -c lib/printfmt.c -o obj/lib/printfmt.o
+ ld bin/kernel
ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/lib/stdio.o obj/kern/lib/readline.o obj/kern/debug/panic.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o
obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/picirq.o obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmm.o obj/lib/string.o
obj/lib/printfmt.o
```

上图中的最后一行是用 ld 命令生成链接程序 kernel.ld，并将图中所有生成的.o 文件链接成 bin/kernel。

以下为生成 bootblock 和 sign 的代码部分：

```
# create bootblock
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

bootblock = $(call totarget,bootblock)

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
    @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
    @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
    @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

$(call create_target,bootblock)

# -----

# create 'sign' tools
$(call add_files_host,tools/sign.c,sign,sign)
$(call create_target_host,sign,sign)

# -----
```

查看 Makefile 文件，同理可知需用 gcc 对 boot 目录下的.c 和.s 文件以及 tools 目录下的 sign.c 文件进行编译：

```
+ cc boot/bootasm.S
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
+ cc tools/sign.c
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
+ ld bin/bootblock
ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 492 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
```

上图中的第 7 行是根据 sign 规范生成 bootblock。

上图中的第 8 行是用 ld 命令将图中所有生成的.o 文件链接成 obj/bootblock.o。

以下为最后生成 ucore.img 的 Makefile 部分：

```
dd if=/dev/zero of=bin/ucore.img count=10000
记录了10000+0 的读入
记录了10000+0 的写出
5120000字节 (5.1 MB, 4.9 MiB) 已复制, 0.0242629 s, 211 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节已复制, 0.000102826 s, 5.0 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
记录了163+1 的读入
记录了163+1 的写出
83936字节 (84 kB, 82 KiB) 已复制, 0.00042267 s, 199 MB/s
```

该部分体现的是：创建大小为 10000 个块的 ucore.img，初始化为 0，每个块为 512 字节。

并使用拷贝命令 dd 将 bin/bootblock 写入 bin/ucore.img 的第 1 个块，之后从 bin/ucore.img 的第 2 个块的位置开始将 bin/kernel 按顺序写入。

2. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

答：首先我们先观察 tools/sign.c 中的部分代码：

```
22     char buf[512];
23     memset(buf, 0, sizeof(buf));
24     FILE *ifp = fopen(argv[1], "rb");
25     int size = fread(buf, 1, st.st_size, ifp);
26     if (size != st.st_size) {
27         fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
28         return -1;
29     }
30     fclose(ifp);
31     buf[510] = 0x55;
32     buf[511] = 0xAA;
33     FILE *ofp = fopen(argv[2], "wb+");
34     size = fwrite(buf, 1, 512, ofp);
35     if (size != 512) {
36         fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);
37         return -1;
38     }
39     fclose(ofp);
40     printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
41     return 0;
42 }
```

(1) 由 35-37 行可知，磁盘主引导扇区大小为 512 字节；

(2) 由 31-32 行可知，磁盘主引导扇区的最后两个字节的的信息是固定的，第 510 个字节为 0x55，第 511 个字节为 0xAA。

练习 2: 使用 qemu 执行并调试 lab1 中的软件

1. 从 CPU 加电后执行的第一条指令开始, 单步跟踪 BIOS 的执行。

答: 根据附录可知, 我们在 tools/gdbinit 中添加 5 行代码:

```
set arch i8086
target remote :1234
define hook-stop
x/i $pc
end
```

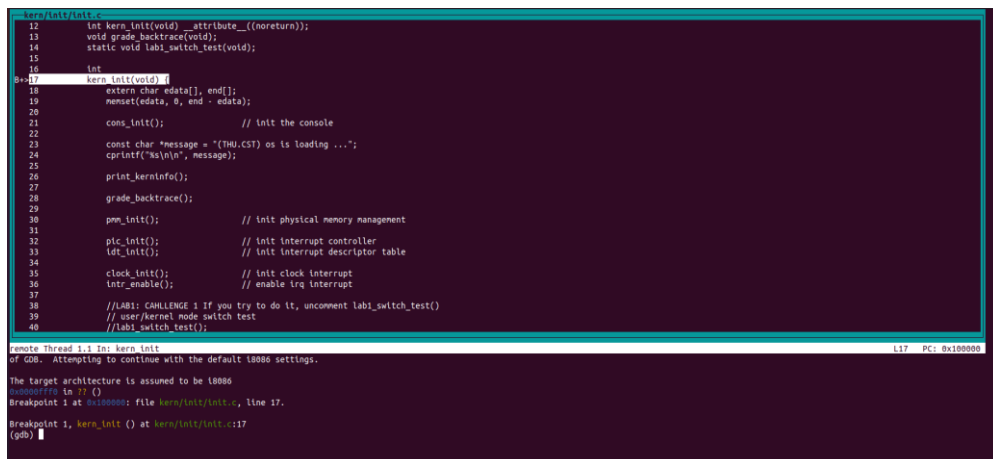
即修改为:



```
gdbinit
~/桌面/lab1-code/tools

1 file bin/kernel
2 set arch i8086
3 target remote :1234
4 break kern_init
5 continue
6 define hook-stop
7 x/i $pc
8 end
```

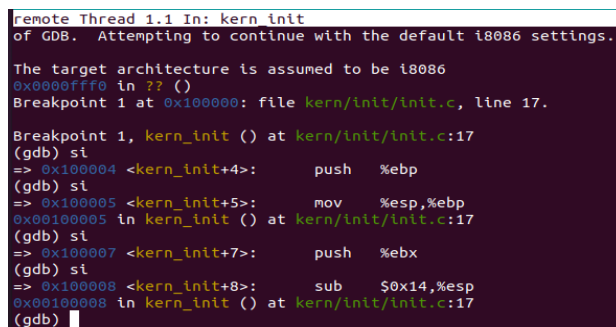
之后在 lab1 目录下执行 make debug:



```
12 int kern_init(void) __attribute__((noreturn));
13 void grade_backtrace(void);
14 static void lab1_switch_test(void);
15
16 int
17 kern_init(void)
18 {
19     extern char edata[], end[];
20     memset(edata, 0, end - edata);
21     cons_init(); // init the console
22     const char *message = "(TMU.CST) os is loading ...";
23     cprintf("%s\n", message);
24     print_kerninfo();
25     grade_backtrace();
26     pmm_init(); // init physical memory management
27     pic_init(); // init interrupt controller
28     idt_init(); // init interrupt descriptor table
29     clock_init(); // init clock interrupt
30     intr_enable(); // enable irq interrupt
31     //LAB1: CHALLENGE 1 If you try to do it, uncomment lab1_switch_test()
32     // user/kernel mode switch test
33     //lab1_switch_test();
34 }
35
36 remote Thread 1.1 In: kern_init
37 of GDB. Attempting to continue with the default i8086 settings.
38
39 The target architecture is assumed to be i8086
40 0x00000000 in ?? ()
41 Breakpoint 1 at 0x1000000: file kern/init/init.c, line 17.
42 Breakpoint 1, kern_init () at kern/init/init.c:17
43 (gdb)
```

可见程序断点停在了 kern_init 函数入口处, 也即 CPU 加电后执行的第一条指令。

输入 si 进行单步调试:



```
remote Thread 1.1 In: kern_init
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x00000000 in ?? ()
Breakpoint 1 at 0x1000000: file kern/init/init.c, line 17.

Breakpoint 1, kern_init () at kern/init/init.c:17
(gdb) si
=> 0x1000004 <kern_init+4>: push %ebp
(gdb) si
=> 0x1000005 <kern_init+5>: mov %esp,%ebp
0x00100005 in kern_init () at kern/init/init.c:17
(gdb) si
=> 0x1000007 <kern_init+7>: push %ebx
(gdb) si
=> 0x1000008 <kern_init+8>: sub $0x14,%esp
0x00100008 in kern_init () at kern/init/init.c:17
(gdb)
```

易知单步调试运行正常。

2.在初始化位置 0x7c00 设置实地址断点,测试断点正常。

答：将 tools/gdbinit 的内容修改为：

```
打开(O)  ▾  [🔍]  gdbinit
~/桌面/lab1-code/tools

1 file bin/kernel
2 set arch i8086
3 target remote :1234
4 b *0x7c00
5 continue
6 define hook-stop
7 x/i $pc
8 end
```

再次在 lab1 目录下执行 make debug:

```
remote Thread 1.1 In:
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x0000ffff in ?? ()
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) █
```

由上图可知，程序停在了 0x7c00 处，所以实地址断点测试正常。

3. 从 0x7c00 开始跟踪代码运行,将单步跟踪反汇编得到的代码与 bootasm.S 和 bootblock.asm 进行比较。

答：单步追踪断点 0x7c00 后的五条指令：

```
remote Thread 1.1 In:
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x0000ffff in ?? ()
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
=> 0x7c01:    cld
0x00007c01 in ?? ()
(gdb) si
=> 0x7c02:    xor     %eax,%eax
0x00007c02 in ?? ()
(gdb) si
=> 0x7c04:    mov     %eax,%ds
0x00007c04 in ?? ()
(gdb) si
=> 0x7c06:    mov     %eax,%es
0x00007c06 in ?? ()
(gdb) si
=> 0x7c08:    mov     %eax,%ss
0x00007c08 in ?? ()
(gdb) █
```

结果如上图所示，发现单步跟踪反汇编得到的代码确实与 bootasm.S 和 bootblock.asm 相同。

The top screenshot shows the assembly file `bootasm.S`. It includes comments about switching to 32-bit protected mode and setting up segment registers. A red box highlights the instructions for setting up the data segment registers (DS, ES, SS):

```

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

```

The bottom screenshot shows the disassembled version in `bootblock.asm`. It includes the same instructions with memory addresses and hex values. A red box highlights the corresponding instructions:

```

19 # Set up the important data segment registers (DS, ES, SS).
20 7c02: 31 c0          xor    %eax,%eax    # Segment number zero
21 7c04: 8e d8          mov     %eax,%ds    # -> Data Segment
22 7c06: 8e c0          mov     %eax,%es    # -> Extra Segment
23 7c08: 8e d0          mov     %eax,%ss    # -> Stack Segment

```

4. 自己找一个 bootloader 或内核中的代码位置，设置断点并进行测试。

答：设置断点为 `0x7c0a`，并单步追踪接下来 5 条指令，并将调试结果与 `bootblock.asm` 和 `bootasm.S` 进行对比：

The screenshot shows a GDB terminal window with the following output:

```

remote Thread 1.1 In:
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x00000fff0 in ?? ()
Breakpoint 1 at 0x7c0a

Breakpoint 1, 0x00007c0a in ?? ()
(gdb) si
=> 0x7c0c: test    $0x2,%al
0x00007c0c in ?? ()
(gdb) si
=> 0x7c0e: jne     0x7c0a
0x00007c0e in ?? ()
(gdb) si
=> 0x7c10: mov     $0xd1,%al
0x00007c10 in ?? ()
(gdb) si
=> 0x7c12: out     %al,$0x64
0x00007c12 in ?? ()
(gdb) si
=> 0x7c14: in      $0x64,%al
0x00007c14 in ?? ()
(gdb)

```



```

bootblock.asm
~/桌面/lab1-code/obj

bootblock.asm
lab1init

26 7c08: 8e d0 mov %eax,%ss
27
28 00007c0a <seta20.1>:
29 # Enable A20:
30 # For backwards compatibility with the earliest PCs, physical
31 # address line 20 is tied low, so that addresses higher than
32 # 1MB wrap around to zero by default. This code undoes this.
33 seta20.1:
34 inb $0x64, %al # Wait for not busy(8042 input buffer empty).
35 7c0a: e4 64 in $0x64,%al
36 testb $0x2, %al
37 7c0c: a8 02 test $0x2,%al
38 jnz seta20.1
39 7c0e: 75 fa jne 7c0a <seta20.1>
40
41 movb $0xd1, %al # 0xd1 -> port 0x64
42 7c10: b0 d1 mov $0xd1,%al
43 outb %al, $0x64 # 0xd1 means: write data to 8042's P2 port
44 7c12: e6 64 out %al,$0x64
45
46 00007c14 <seta20.2>:
47
48 seta20.2:
49 inb $0x64, %al # Wait for not busy(8042 input buffer empty).
50 7c14: e4 64 in $0x64,%al
51 testb $0x2, %al
52 7c16: a8 02 test $0x2,%al
53 jnz seta20.2
54 7c18: 75 fa jne 7c14 <seta20.2>
55

```

```

bootasm.S
~/桌面/lab1-code/boot

bootblock.asm
lab1init
bootasm.S

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 SCR0 PE_ON, %eax

```

对比可知该断点测试正常。

练习 3：分析 bootloader 进入保护模式的过程

答：bootloader 进入保护模式的过程可查看 bootasm.S 的源码（源码中的英文注释已翻译成中文并在某些部分添加了自己对于该部分代码的理解）：

① 宏定义：

```
.set PROT_MODE_CSEG,      0x8          # 内核代码段选择器
.set PROT_MODE_DSEG,      0x10        # 内核数据段选择器
.set CR0_PE_ON,           0x1         # 保护模式使能标志
```

② 关闭中断，并将各个段寄存器置 0（初始化）：

```
# 起始地址应该是0:7c00，在实模式下，运行引导加载程序的起始地址
.globl start
start:
.code16                                # 组装为16位模式
    cli                                # 关闭中断
    cld                                # 设置字符串传送方向为从低地址到高地址

    # 设置重要数据段寄存器(DS, ES, SS)。
    xorw %ax, %ax                      # 置0
    movw %ax, %ds                      # -> 数据段寄存器
    movw %ax, %es                      # -> 附加段寄存器
    movw %ax, %ss                      # -> 堆栈段寄存器
```

③ 开启 A20

开启 A20 的原因：当 A20 地址线关闭时，所有程序等同于在 8086 中运行。且 8086 只有 20 根地址总线，所以在实模式下可访问的物理内存空间不能超过 1MB。又因为 bootloader 需要从实模式切换至保护模式，而在保护模式下需要访问更多，更大的地址空间，所以我们需要将键盘控制器上的 A20 线置于高电位，使得全部的 32 条地址线可用，这样就使得内存空间从之前的 1M 变成了 4G。

```
# 开启 A20:
# 为了向后兼容最早的PC机，物理地址行20被固定在较低的位置，因此高于1MB的地址默认为0。这段代码会撤销它。
seta20.1:
    inb $0x64, %al                    # 等到8042输入缓冲区为空
    testb $0x2, %al                   # 读取al寄存器中的数据的第2位是否为0（为0则代表输入缓存为空）
    jnz seta20.1                     # 若al寄存器第2位不为0则跳回seta20.1继续循环，直至为0

    movb $0xd1, %al                   # 往0x64写入0xd1命令，表示修改8042的P2 port
    outb %al, $0x64                   # 将al中的数据写入到8042的P2端口中（即端口0x64）

seta20.2:
    inb $0x64, %al                    # 等到8042输入缓冲区为空
    testb $0x2, %al                   # 读取al寄存器中的数据的第2位是否为0（为0则代表输入缓存为空）
    jnz seta20.2                     # 若al寄存器第2位不为0则跳回seta20.2继续循环，直至为0

    movb $0xdf, %al                   # 往0x60端口写入0xdf命令，表示将P2 port的第二个位（A20）选通置为1；
    outb %al, $0x60                   # 0xdf = 11011111，表示将P2 port的第二个位（A20）选通置为1；
```

④ 初始化 GDT 表:

查看 bootasm.S 可知代码中已经静态的描述了一个简单的 GDT, 如下图所示:

```
# Bootstrap GDT
.p2align 2                # 强制4字节对齐
gdt:
    SEG_NULLASM            # NULL段
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # bootloader and kernel的代码段
    SEG_ASM(STA_W, 0x0, 0xffffffff)      # bootloader and kernel的数据段
gdtdesc:
    .word 0x17             # sizeof(gdt) - 1
    .long gdt              # address gdt
```

所以开启 A20 之后, 我们调用命令 lgdt gdtdesc 即可载入全局描述符表。

```
# 从真实模式切换到保护模式, 使用引导GDT和段转换, 使虚拟地址与物理地址相同, 这样有效的内存映射在切换期间不会改变。
lgdt gdtdesc                # 载入到全局描述符表寄存器
```

⑤ 进入保护模式

(1) 将 cr0 置 1 开启保护模式 (CR0_PE_ON 在宏定义中已被预赋值为 1)

```
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
```

(2) 使用长跳转指令 ljmp, 将 cs 修改为 32 位段寄存器, 并跳转到 protcseg 这一 32 位代码入口处, 此时 CPU 进入 32 位模式:

```
# 跳转到下一个指令, 但是是在32位代码段。
# 将处理器切换到32位模式。
ljmp $PROT_MODE_CSEG, $protcseg
```

(3) 重置保护模式下的段寄存器并建立堆栈, 最后调用 bootmain.c, 进行操作系统内核的加载。至此, bootloader 已经完成了从实模式进入到保护模式的任务。

```
.code32                # 32位模式组装
protcseg:
    # 将PROT_MODE_DSEG的值0x10赋值给我们的数据段选择器ax
    movw $PROT_MODE_DSEG, %ax                # 将PROT_MODE_DSEG的值0x10赋值给我们的数据段选择器ax
    movw %ax, %ds                            # 重置DS: 数据段寄存器
    movw %ax, %es                            # 重置ES: 附加段寄存器
    movw %ax, %fs                            # 重置FS
    movw %ax, %gs                            # 重置GS
    movw %ax, %ss                            # 重置SS: 堆栈段寄存器

    # 设置堆栈指针并调用c, 堆栈区域从0到0x7c00
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain                #调用bootmain.c
```

练习 4: 分析 bootloader 加载 ELF 格式的 OS 的过程

(1) bootloader 如何读取硬盘扇区的?

答: bootmain 函数中调用 readseg 函数读取硬盘扇区

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // 从磁盘读取第一页
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
}
```

因为 readseg 函数中调用了 readsect 函数, 而 readsect 函数又调用了 waitdisk 函数。所以我们先按照 waitdisk, readsect, readseg 的顺序对这三个函数进行解释:

对于 waitdisk 函数:

```
/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}
```

该函数功能为: 不断查询读 0x1F7 寄存器的最高两位, 直到最高位为 0、次高位为 1 (即磁盘空闲) 才返回。

对于 readsect 函数:

```
/* readsect-从@secno读取单个扇区到@dst */
static void
readsect(void *dst, uint32_t secno) {

    waitdisk();                // 等待磁盘空闲

    outb(0x1F2, 1);            // 往0x1F2地址中写入要读取的扇区数, 这里是读入1个扇区
    //读取的扇区起始编号共28位, 分成4部分依次放在0x1F3~0x1F6寄存器中。
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);         // 向磁盘发出读命令0x20

    waitdisk();                // 等待磁盘空闲

    insl(0x1F0, dst, SECTSIZE / 4); // 读取一个扇区
}
```

该函数功能为: 读取一个磁盘扇区。

对于 readseg 函数：

```
/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // 四舍五入到区域边界
    va -= offset % SECTSIZE;

    // 从字节转换到扇区;内核从扇区1开始
    uint32_t secno = (offset / SECTSIZE) + 1;

    // 如果这太慢,我们可以一次读取很多扇区。
    // 我们会向内存写入比请求的更多的内容,但这没关系——我们是按递增顺序加载的。
    for (; va < end_va; va += SECTSIZE, secno++) {
        readsect((void *)va, secno);
    }
}
```

该函数功能为：从扇区 1 起的第 offset 个位置处，读取 count 个字节到指定内存中。

此外，`va -= offset % SECTSIZE` 表示：就算读取了整个扇区，这样操作就可以使得读到的数据在内存中的起始位置仍然是参数 va。

(2) bootloader 是如何加载 ELF 格式的 OS?

答：ELF 简介：ELF(Executable and linking format)文件格式是 Linux 系统下的一种常用目标文件(object file)格式，有三种主要类型：

① 用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载的内存执行。

这也是本实验的 OS 文件类型。

② 用于连接的可重定位文件(relocatable file)，可与其它目标文件一起创建可执行文件和共享目标文件。

③ 共享目标文件(shared object file)，连接器可将它与其它可重定位文件和共享目标文件连接成其它的目标文件，动态连接器又可将它与可执行文件和其它共享目标文件结合起来创建一个进程映像。

ELF 结构定义如下代码所示：

```
struct elfhdr {
uint32_t e_magic; // 判断读出来的 ELF 格式的文件是否为正确的格式
uint8_t e_elf[12];
uint16_t e_type; // 1=可重定位, 2=可执行, 3=共享对象, 4=核心映像
uint16_t e_machine; // 3=x86, 4=68K 等.
uint32_t e_version; // 文件版本, 总是 1
uint32_t e_entry; // 程序入口所对应的虚拟地址。
uint32_t e_phoff; // 程序头表的位置偏移
uint32_t e_shoff; // 区段标题或 0 的文件位置
uint32_t e_flags; // 特定于体系结构的标志, 通常为 0
uint16_t e_ehsize; // 这个 elf 头的大小
uint16_t e_phentsize; // 程序头中条目的大小
uint16_t e_phnum; // 程序头表中的入口数目
uint16_t e_shentsize; // 节标题中条目的大小
uint16_t e_shnum; // 节标题中的条目数或 0
uint16_t e_shstrndx; // 包含节名称字符串的节号。
};
```

具体加载过程详见如下 bootmain 函数代码注释：

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // 从磁盘读取第一页
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 检验ELF文件是否合法?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // 加载每个程序段(忽略ph值标志)
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff); // 根据ELF头表中的描述, 将ELF文件加载到内存中的相应位置并保存到ph中
    eph = ph + ELFHDR->e_phnum; // ph为ELF段表首地址, eph为ELF段表末地址

    // 将ELF文件的数据从段表首地址到末地址全部载入内存
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_nsize, ph->p_offset); // 调用readseg函数
    }

    // 从ELF头文件中调用入口点
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))(); // 从ELF头表中找到内核的入口地址, 再使用函数调用的方式跳转到该地址上并开始加载运行ELF格式的os

bad:
    outw(0xBA00, 0xBA00);
    outw(0xBA00, 0xBE00);

    /* do nothing */
    while (1);
}
```


练习 5：实现函数调用堆栈跟踪函数

答：根据 kdebug.c 中 print_stackframe 函数部分的注释即可写出该部分代码（对于补充代码的解释与分析已放在每一行的注释中）：

```
void
print_stackframe(void) {
    uint32_t ebp=read_ebp();//获取当前ebp的值，对应注释：(1) call read_ebp() to get the value of ebp. the type is (uint32_t);
    uint32_t eip=read_eip();//获取当前eip的值，对应注释：(2) call read_eip() to get the value of eip. the type is (uint32_t);
    int i,j;
    for(i=0;i<=STACKFRAME_DEPTH-1&&ebp!=0;i++){//循环打印ebp和eip直至ebp地址为0或（即达到栈底）或长度超过STACKFRAME_DEPTH。对应注释：(3) from 0 .. STACKFRAME_DEPTH
        printf("ebp:0x%08x eip:0x%08x args:",ebp,eip);//打印ebp和eip。对应注释：(3.1) printf value of ebp, eip
        uint32_t *args=(uint32_t *)ebp+2;////接收ebp+2的地址，获得参数。对应注释：(3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2 [0..4]
        for(j=0;j<=3;j++){
            printf("0x%08x ",args[j]);
        }
        printf("\n");//对应注释：(3.3) cprintf("\n");
        print_debuginfo(eip-1);//打印当前调用函数的信息。对应注释：(3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
        eip=((uint32_t *)ebp+1);//调用函数的返回地址。对应注释：the calling function's return addr eip = ss:[ebp+4]
        ebp=((uint32_t *)ebp);//上一个函数的栈针。对应注释：the calling function's ebp = ss:[ebp]
    }
    /* LAB1 YOUR CODE : STEP 1 */
    /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
    * (2) call read_eip() to get the value of eip. the type is (uint32_t);
    * (3) from 0 .. STACKFRAME_DEPTH
    * (3.1) printf value of ebp, eip
    * (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2 [0..4]
    * (3.3) cprintf("\n");
    * (3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
    * (3.5) popup a calling stackframe
    * NOTICE: the calling function's return addr eip = ss:[ebp+4]
    * the calling function's ebp = ss:[ebp]
    */
}
```

运行结果如下所示：

```
consthall@consthall-Lenovo-XiaoXin-CHA07000-13:~/桌面/labs$ make qemu
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
entry 0x00100000 (phys)
etext 0x001032bc (phys)
edata 0x0010ea16 (phys)
end 0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
kern/debug/kdebug.c:294: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 args:0x00000000 0x00000000 0x00007b88
kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032dc 0x0000130a 0x00000000
kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00100094
kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d67 --
++ setup timer interrupts
consthall@consthall-Lenovo-XiaoXin-CHA07000-13:~/桌面/labs$
```

最后一行各个数值的含义：

ebp:0x00007bf8 ：最后一行的 ebp 的值是 kern_init 函数的栈顶地址；

eip:0x00007d68：最后一行的 eip 的值是 kern_init 函数的返回地址，即 bootmain 函数调用 kern_init 对应指令的下一条指令的地址（详见下图）。

```

// 从ELF头文件中调用入口点
// note: does not return
((void (*)(void))(ELFHDR->e_entry & 0xFFFFF))(); // 从ELF头表中找到内核的入口地址，再使用函数调用的方式跳转到该地址上并开始加载运行ELF格式的OS
7d5c: a1 18 00 01 00 mov 0x10018,%eax
7d61: 25 ff ff ff 00 and $0xffffffff,%eax
7d66: ff d0 call *%eax
asm volatile ("outb %0, %1" :: "a" (data), "d" (port));
}

static inline void
outw(uint16_t port, uint16_t data) {
asm volatile ("outw %0, %1" :: "a" (data), "d" (port));
7d68: b8 00 8a ff ff mov $0xffff8a00,%eax
7d6d: 89 c2 mov %eax,%edx
7d6f: 66 ef out %ax,(&dx)
7d71: b8 00 8e ff ff mov $0xffff8e00,%eax
7d76: 66 ef out %ax,(&dx)
7d78: eb fe jmp 7d78 <bootmain+0xa7>

```

args: 通常情况下，args 表示被调用函数的参数。但是这里不同（因为 bootmain 函数不需要参数），在经过与 bootblock.asm 的对比之后可发现 args 的四个值对应着 bootloader 指令的前 16 个字节（详见下图对比）：

```

# 起始地址应该是0:7c00，在实模式下，运行引导加载程序的起始地址
.globl start
start:
.code16 # 组装为16位模式
cli # 关闭中断
7c00: fa cli
cld # 设置字节块传送方向为从低地址到高地址
7c01: fc cld

# 设置重要数据段寄存器(DS, ES, SS)。
xorw %ax, %ax # 置0
7c02: 31 c0 xor %eax,%eax
movw %ax, %ds # -> 数据段寄存器
7c04: 8e d8 mov %eax,%ds
movw %ax, %es # -> 附加段寄存器
7c06: 8e c0 mov %eax,%es
movw %ax, %ss # -> 堆栈段寄存器
7c08: 8e d0 mov %eax,%ss

00007c0a <seta20.1>:

# 开启 A20:
# 为了向后兼容最早的PC机，物理地址行20被固定在较低的位置，因此高于1MB的地址默认为0。这段代码会撤销它。
seta20.1:
inb $0x64, %al # 等到8042输入缓冲区为空
7c0a: e4 64 in $0x64,%al
testb $0x2, %al # 读取al寄存器中的数据第2位是否为0（为0则代表输入缓存为空）
7c0c: a8 02 test $0x2,%al
jnz seta20.1 # 若al寄存器第2位不为0则跳回seta20.1继续循环，直至为0
7c0e: 75 fa jne 7c0a <seta20.1>

```

<unknow> - - 0x00007d67 - -: bootmain 函数内调用 OS kernel 入口函数的该指令的地址

练习 6：完善中断初始化和处理（需要编程）

(1) 中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？



答：IDT 中一个表项占 8 个字节。其中 0-15 位是偏移量 offset 的低 16 位，16-31 位是处理代码入口地址的段选择子，48-63 位是偏移量 offset 的高 16 位。0-15 位和 48-63 位结合即可得到段内偏移量，再由段选择子得到段基址，最后将段基址和段内偏移量相加即可得中断处理代码的入口。

(2) 请编程完善 kern/trap/trap.c 中对中断向量表进行初始化的函数 idt_init。在 idt_init 函数中，依次对所有中断入口进行初始化。使用 mmu.h 中的 SETGATE 宏，填充 idt 数组内容。每个中断的入口由 tools/vectors.c 生成，使用 trap.c 中声明的 vectors 数组即可。

答：根据 idt_init 中的注释即可补全代码，代码及其注释详见下图：

```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */
void
idt_init(void) {
    /* LAB1 YOUR CODE : STEP 2 */
    /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
     * All ISR's entry addrs are stored in __vectors. where is uintptr_t __vectors[] ?
     * __vectors[] is in kern/trap/vector.S which is produced by tools/vector.c
     * (try "make" command in lab1, then you will find vector.S in kern/trap DIR)
     * You can use "extern uintptr_t __vectors[);" to define this extern variable which will be used later.
     * (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).
     * Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to setup each item of IDT
     * (3) After setup the contents of IDT, you will let CPU know where is the IDT by using 'lidt' instruction.
     * You don't know the meaning of this instruction? just google it! and check the libs/x86.h to know more.
     * Notice: the argument of lidt is idt_pd. try to find it!
     */
    extern uintptr_t __vectors[]; //声明_vectors[],其中存放着中断服务程序的入口地址
    int i=0;
    for(i=0;i<(sizeof(idt)/sizeof(struct gatedesc));i++){
        SETGATE(idt[i],0,GD_KTEXT,__vectors[i],DPL_KERNEL); //除了系统调用中断 (T_SYSCALL)，其他中断均设置为内核态权限
    }
    SETGATE(idt[T_SYSCALL],0,GD_KTEXT,__vectors[T_SYSCALL],DPL_USER); //系统调用中断T_SYSCALL设置为用户态权限
    lidt(&idt_pd); //使用lidt指令加载中断描述符表idt，让CPU知道idt在哪
}
```

(3) 请编程完善 trap.c 中的中断处理函数 trap，在对时钟中断进行处理的部分填写 trap 函数中处理时钟中断的部分，使操作系统每遇到 500 次时钟中断后，调用 print_ticks 子程序，向屏幕上打印一行文字“500 ticks”。

答：根据 trap_dispatch 函数中的 case IRQ_OFFSET + IRQ_TIMER 中的注释即可补全代码，代码及其注释详见下图（其中 TICK_NUM 已在宏定义中声明为 500）：

```
switch (tf->tf_trapno) {
case IRQ_OFFSET + IRQ_TIMER:
/* LAB1 YOUR CODE : STEP 3 */
/* handle the timer interrupt */
/* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in kern/driver/clock.c
 * (2) Every TICK_NUM cycle, you can print some info using a function, such as print_ticks().
 * (3) Too Simple? Yes, I think so!
 */
ticks++;
if(ticks==TICK_NUM){
    ticks-=TICK_NUM;//ticks归0, 重新开始计数
    print_ticks();
}
break;
```

运行结果见下图：

```
ebp:0x00007b58 eip:0x001000bb arg:0x00000000 0xffff0000 0x00007b84 0x00000029
kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 arg:0x00000000 0x00100000 0xffff0000 0x0000001d
kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe arg:0x0010351c 0x00103500 0x0000130a 0x00000000
kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 arg:0x00000000 0x00000000 0x00000000 0x00010094
kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 arg:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d67 --
++ setup timer interrupts
500 ticks
500 ticks
kbd [097] a
kbd [000]
500 ticks
kbd [098] b
kbd [000]
500 ticks
kbd [099] c
kbd [000]
kbd [100] d
kbd [000]
500 ticks
```

如结果所示每 5 秒出现一次 500 ticks，且在运行过程中按下 A、B、C、D 键也会显示在屏幕上。

3. 实验感想

本次实验是操作系统的第一次实验，一开始我还是很不适应的，因为感觉到实验中的内容相较于理论课更加深入，感觉到无从下手。

我在参考了网上的几篇相关实验报告之后，才大致有了思路和实验方向。虽然有一部分参考了他人的成果，但我都是在学习过后自己独立写出实验报告和相关代码，并且在代码当中补充上了很多自己的注释，以表明自己实实在在的弄懂了这些问题。

本次实验主要有以下一些问题：

① 对于 makefile 的原理和 qemu 的调试不熟，经过对老师所发放的参考资料的学习和针对自己遇到的问题在网上进行查阅之后均得到解决。

② 对于练习 3, 4 这一类原理解释性题目束手无措，所幸 bootasm.S 和 bootmain.c 的书写语言是汇编和 C，文件中的大部分代码还是以解释性语言为主，所以结合注释也能大致看懂代码的作用。在仔细阅读源码和对比参考资料之后这两个练习圆满完成。

③ 该问题在实验完成后仍未解决，网上查阅相关资料也未找到答案，最后在咨询过黄聘老师后得到解决：

由于之前我提前安装了双系统和最新版的 qemu，所以此次实验我并没有安装虚拟机，直接从超算习堂上下载 lab1 的相关文件后就开始进行实验。

在前 4 个练习中并没有出现异样，但是从第 5 个练习开始，我的运行结果和使用虚拟机的同学出现了偏差。

对于练习 5，我在补充完 kdebug.c 代码部分之后在终端运行 `make qemu`，发现最后一行没有出现 `++ setup timer interrupts`（即下图结果）（实验报告图中的结果有，结果截图的来源会在后面补充），但是当时我并没有意识到这一问题，继续进行练习 6 的实验。

```

consthall@consthall-Lenovo-XiaoXin-CHA07000-13:~/桌面/lab1hyw$ make qemu
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x00103b05 (phys)
  edata  0x00110950 (phys)
  end    0x00111dc0 (phys)
Kernel executable memory footprint: 72KB
ebp:0x00007b38 eip:0x00100c26 args:0x00010094 0x00110950 0x00007b68 0x001000aa
  kern/debug/kdebug.c:294: print_stackframe+37
ebp:0x00007b48 eip:0x00100fbf args:0x00000000 0x00000000 0x00100095
  kern/debug/kmonitor.c:125: mon_backtrace+27
ebp:0x00007b68 eip:0x001000aa args:0x00000000 0x00007b90 0xffff0000 0x00007b94
  kern/init/init.c:48: grade_backtrace2+36
ebp:0x00007b88 eip:0x001000dd args:0x00000000 0xffff0000 0x00007bb4 0x001000f5
  kern/init/init.c:53: grade_backtrace1+41
ebp:0x00007ba8 eip:0x00100108 args:0x00000000 0x00100000 0xffff0000 0x0010011d
  kern/init/init.c:58: grade_backtrace0+33
ebp:0x00007bc8 eip:0x00100138 args:0x00000000 0x00000000 0x00103b08
  kern/init/init.c:63: grade_backtrace+41
ebp:0x00007be8 eip:0x0010006a args:0x00000000 0x00000000 0x00007c4f
  kern/init/init.c:28: kern_init+105
ebp:0x00007bf8 eip:0x00007d71 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d70 --
consthall@consthall-Lenovo-XiaoXin-CHA07000-13:~/桌面/lab1hyw$

```

对于练习 6，我在补充完 trap.c 代码部分之后在终端运行 make qemu，发现结果和练习 5 保持一致，没有任何变化，在多次修改代码并且和网上代码一一对比之后，结果仍没有改变。此时我同学把他虚拟机上已经 make 过的整个 lab1 文件夹发给了我，我运行了一下之后发现结果正确，于是我以为可能是老师在超算习堂上上传的文件相对于虚拟机里的文件有差别，就将文件一一替换之后重新 make，发现结果又回到了和练习 5 一样。

在排查之后我发现不是文件问题，即如果我直接运行我同学发来的文件，不 make clean 且重新 make，那么结果就是正确的，反之结果就错误。为了验证我的想法，我把我的 lab1 文件发给了我同学，让他在虚拟机上 make clean 再 make，make qemu 之后发现我的文件在虚拟机上结果运行正确。

我在网上查询了双系统和虚拟机在 make 上的区别，并没有查到有用的回答。所以对于练习 5, 6，我只能将我的文件在我同学的虚拟机上 make 运行之后再重新发回到双系统上直接 make qemu 查看结果。

在咨询过黄聃老师后，黄聃老师给出的回答是：应该是我的 ubuntu 和 gcc 版本和虚拟机不一致，因为 ucore 这个代码是 2015 年以前的代码，如果想要 ubuntu 的编译运行结果和虚拟机保持一致，则需要保证操作系统，kernel 版本，gcc 版本，和虚拟机中保持一致。

所以为了方便起见，我决定从下次实验开始用虚拟机进行实验，从而避免不必要的麻烦。