

OS_Lab1_Experimental report

湖南大学信息科学与工程学院

计科 210X 甘晴void (学号 202108010XXX)

为了实现 lab1 的目标，lab1 提供了 6 个基本练习和 1 个扩展练习，要求完成实验报告。

对实验报告的要求：

基于 markdown 格式来完成，以文本方式为主。

填写各个基本练习中要求完成的报告内容。

完成实验后，请分析 ucore_lab 中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别。

列出你认为本实验中重要的知识点，以及与对应的 OS 原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）。

列出你认为 OS 原理中很重要，但在实验中没有对应上的知识点。

目录

实验内容	3
实验环境	3
练习 1: 理解通过 <code>make</code> 生成执行文件的过程	3
(1) 操作系统镜像文件 <code>ucore.img</code> 是如何一步一步生成的? (需要比较详细地解释 <code>Makefile</code> 中每一条相关命令和命令参数的含义, 以及说明命令导致的结果)	3
(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?	8
练习 2: 使用 <code>qemu</code> 执行并调试 lab1 中的软件	8
(1) 从 CPU 加电后执行的第一条指令开始, 单步跟踪 BIOS 的执行。	8
(2) 在初始化位置 <code>0x7c00</code> 设置实地址断点, 测试断点正常。	9
(3) 从 <code>0x7c00</code> 开始跟踪代码运行, 将单步跟踪反汇编得到的代码与 <code>bootasm.S</code> 和 <code>bootblock.asm</code> 进行比较。	10
(4) 自己找一个 <code>bootloader</code> 或内核中的代码位置, 设置断点并进行测试。	11
练习 3: 分析 <code>bootloader</code> 进入保护模式的过程	12
(1) 一些准备工作	12
(2) 为何要开启 <code>A20</code>	12
(3) 如何开启 <code>A20</code>	13
(4) 初始化 <code>GDT</code> 表	14
(5) 如何使能与进入保护模式	15
(6) 进入 <code>32</code> 位模式	15
(7) 调用 <code>bootmain</code> 函数	15
练习 4: 分析 <code>bootloader</code> 加载 <code>ELF</code> 格式的 <code>OS</code> 的过程	15
(1) <code>bootloader</code> 如何读取硬盘扇区	16
(2) <code>bootloader</code> 是如何加载 <code>ELF</code> 格式的 <code>OS</code>	16
(3) <code>bootmain.c</code> 代码解读	17
练习 5: 实现函数调用堆栈跟踪函数	19
(1) 函数调用堆栈相关知识	20
(2) 实现函数 <code>print_stackframe</code>	20
(3) 函数 <code>print_stackframe</code> 解析	21
(4) 最后一行结果的解析	22
练习 6: 完善中断初始化和处理	22
(1) 中断描述符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?	22
(2) 请编程完善 <code>kern/trap/trap.c</code> 中对中断向量表进行初始化的函数 <code>idt_init</code>	23
(3) 请编程完善 <code>trap.c</code> 中的中断处理函数 <code>trap</code> , 在对时钟中断进行处理的部分填写 <code>trap</code> 函数	24
(4) 执行程序	24
扩展练习 Challenge1	25
(1) 中断及特权保护机制	26
(2) <code>Ucore</code> 如何处理中断	26
(3) 补全代码	27
(4) 执行程序, 验证结果	29
扩展练习 Challenge2	29
参考答案对比	32
重要知识点和对应原理	32

实验内容

阅读 uCore 实验项目开始文档 (uCore Lab 0)，准备实验平台，熟悉实验工具。

uCore Lab 1: 系统软件启动过程

- 编译运行 uCore Lab 1 的工程代码;
- 完成 uCore Lab 1 练习 1-4 的实验报告;
- 尝试实现 uCore Lab 1 练习 5-6 的编程作业;
- 思考如何实现 uCore Lab 1 扩展练习 1-2。

实验环境

- 架构: Intel x86_64 (虚拟机)
- 操作系统: Ubuntu 20.04
- 汇编器: gas (GNU Assembler) in AT&T mode
- 编译器: gcc

练习 1: 理解通过 make 生成执行文件的过程

列出本实验各练习中对应的 OS 原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。

在此练习中，大家需要通过静态分析代码来了解：

(1) 操作系统镜像文件 ucore.img 是如何一步一步生成的？(需要比较详细地解释 Makefile 中每一条相关命令和命令参数的含义，以及说明命令导致的结果)

在 labcodes/lab1/Makefile 文件中，我们可以找到创建 ucore.img 的代码段

```
# create ucore.img
UCOREIMG := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel) $(bootblock)
    $(V)dd if=/dev/zero of=$@ count=10000
    $(V)dd if=$(bootblock) of=$@ conv=notrunc
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create_target,ucore.img)
```

下面将逐句解释其含义。

①UCOREIMG := \$(call totarget,ucore.img)

UCOREIMG := \$(call totarget,ucore.img)表示调用 call 函数生成 UCOREIMG，

其中 call 为调用 call 函数的标记，

其中 totarget 可以在 tools/function.mk 中找到，它被定义为 otarget = \$(addprefix \$(BINDIR)\$(SLASH),\$(1))。在这之中，addprefix 代表在前面加上，\$(BINDIR)代表 bin，\$(SLASH)代表/。

综上所述，totarget,ucore.img 的意思就是在 ucore.img 前面加上 bin/，调用 call 函数生成的 UCOREIMG 即为 bin/ucore.img。

②\$(UCOREIMG): \$(kernel) \$(bootblock)

这一行表示 UCOREIMG 生成所需的依赖文件为 kernel 和 bootblock 这两个文件，我们将分别解读这两个文件。

③kernel 文件

➤ 代码理解

在 makefile 文件中找到注释为 kernel 的代码段

```
# kernel

KINCLUDE    += kern/debug/ \
              kern/driver/ \
              kern/trap/ \
              kern/mm/
KSRCDIR     += kern/init \
              kern/libs \
              kern/debug \
              kern/driver \
              kern/trap \
              kern/mm
KCFLAGS     += $(addprefix -I,$(KINCLUDE))
$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))
KOBJS      = $(call read_packet,kernel libs)
```

- ✓ 一开始的 KINCLUDE 和 KSRCDIR 处的代码将 kern 目录的前缀定义为 kinclude 和 ksrcdir
- ✓ KCFLAGS += \$(addprefix -I,\$(KINCLUDE))表示将 kinclude 的目录前缀加上-I 选项，提供交互模式
- ✓ \$(call add_files_cc,\$(call listf_cc,\$(KSRCDIR)),kernel,\$(KCFLAGS))生成 kern 目录下的.o 文件，这些.o 文件生成时使用的具体命令的参数和方式都差不多。
- ✓ KOBJS = \$(call read_packet,kernel libs)表示使用 call 函数链接 read_packet 和 kernel libs 给 KOBJS

```
# 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)
```

- ✓ kernel = \$(call totarget,kernel)代表表示调用 call 函数生成 kernel，实际为文件 bin/kernel。
- ✓ 接下来的两行\$(kernel)表示生成 kernel 文件需要依赖的三个文件 tools、kernel.ld 链接配置文件、KOBJS 文件。

- ✓ `@echo + ld $@` 中的 `echo` 表示显示内容, `ld` 代表链接, `$@` 代表目标文件, 语句代表将下面的文件和目标文件链接起来, 同时打印 `kernel` 目标文件名
- ✓ `(V)(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)` 代表使用 `kernel.ld` 作为连接器脚本, 链接的文件有 `obj/libs/` 和 `obj/kernel/` 下的所有的 `obj` 文件生成 `kernel` 文件, 关键参数为 `-T <scriptfile>`, 代表让连接器使用指定的脚本, 这里即使用 `kernel.ld` 这个脚本。
- ✓ `$(OBJDUMP) -S $@ > $(call asmfile,kernel)` 代表使用 `objdump` 工具对 `kernel` 文件进行反汇编, 便于调试, `-S` 选项为交替显示 C 源码和汇编代码。
- ✓ `$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/./ /; /^$$/d' > $(call symfile,kernel)` 代表使用 `objdump` 工具通过解析 `kernel` 文件从而能得到符号表。
- ✓ `$(call create_target,kernel)` 生成 `kernel` 直接返回

➤ 查看 `make` 执行的命令

以上为对代码的逐字理解, 我们还可以输入 `make "V="`, 较为清晰地查看生成 `kernel` 文件的具体过程:

```
+ ld bin/kernel
ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/stdio.o obj/kern/libs/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/libs/string.o obj/libs/printfmt.o
```

可见, 要得到一个 `kernel` 文件, 需要链接以下这些文件: `kernel.ld` `init.o` `stdio.o` `readline.o` `panic.o` `kdebug.o` `kmonitor.o` `clock.o` `console.o` `picirq.o` `intr.o` `trap.o` `vectors.o` `trapentry.o` `pmm.o` `string.o` `printfmt.o`

其中 `kernel.ld` 已经存在, 而生成 `kernel` 时, `makefile` 中带 `@` 的前缀的指令都不是必需的, 编译选项中:

- ◆ `ld` 表示链接
- ◆ `-m` 表示模拟指定的连接器
- ◆ `-nostdlib` 表示不使用标准库
- ◆ `-T` 表示让连接器使用指定的脚本
- ◆ `tools/kernel.ld` 是指定连接器脚本
- ◆ `-o` 表示指定输出文件的名称

不难发现, 依赖的 `.o` 文件生成时使用的具体命令的参数和方式都差不多。

以 `pmm.o`, `string.o`, `printf.o` 这三个为例。

```
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc libs/string.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/printfmt.c -o obj/libs/printfmt.o
```

可以总结出以下要点:

- `-I<dir>` 如 `-Ikern/mm/`、`-Ikern/debug/` 等表示给搜索头文件添加路径
- `-march=i686` 表示指定 CPU 架构为 `i686`
- `-fno-builtin` 表示除非使用 `__builtin_` 前缀, 否则不优化 `builtin` 函数
- `-fno-PIC` 表示生成位置无关代码
- `-Wall` 表示开启所有警告
- `-ggdb` 表示生成 `gdb` 可以使用的调试信息, 便于使用 `qemu` 和 `gdb` 来进行调试

- -m32 表示生成在 32 位环境下适用的代码，因为 ucore 是 32 位的软件
- -gstabs 表示生成 stabs 格式的调试信息，便于 monitor 显示函数调用栈信息
- -nostdinc 表示不使用标准库，因为 OS 内核是提供服务的，不依赖其它服务
- -fno-stack-protector 表示不生成检测缓冲区溢出部分的代码

④查看 bootblock 文件

- 代码理解

在 makefile 文件中找到注释为 bootblock 的代码段

```
# 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)
```

- ✓ bootfiles = \$(call listf_cc,boot)中使用 call 调用 listf_cc 函数过滤对应目录下的.c 和.S 文件，用 boot 替换 listf_cc 里面的变量，将 listf_cc 的返回值赋给 bootfiles
- ✓ \$(foreach f,\$(bootfiles),\$(call cc_compile,\$(f),\$(CC),\$(CFLAGS) -Os -nostdinc))编译 bootfiles 生成.o 文件，其中-Os 参数表示为减小代码大小而进行优化
- ✓ 上面两行代码用来生成 bootasm.o，bootmain.o，实际的代码是由宏批量生成。
- ✓ bootblock = \$(call totarget,bootblock)表示 bootblock 实际为文件 bin/bootblock
- ✓ \$(bootblock): \$(call toobj,\$(bootfiles)) | \$(call totarget,sign)其中的 toobj 表示给输出参数加上前缀 obj/，文件后缀名改为.o，语句表示 bootblock 依赖于 obj/boot/*.o 与 bin/sign 文件
- ✓ @echo + ld \$@代表将下面的文件和目标文件链接起来，同时打印 kernel 目标文件名
- ✓ \$(V)\$(LD) \$(LDFLAGS) -N -e start -Ttext 0x7C00 \$^ -o \$(call toobj,bootblock)表示链接所有.o 文件生成 obj/bootblock.o 文件，其中-N 代表设置代码段和数据段均可读写，-e start 代表指定入口为 start，-Ttext 0x7C00 代表代码段开始位置为 0x7C00
- ✓ @\$(OBJDUMP) -S \$(call objfile,bootblock) > \$(call asmfile,bootblock)表示使用 objdump 工具对 obj/bootblock.o 文件进行反汇编得到 obj/bootblock.asm 文件，便于调试，-S 选项为交替显示 C 源码和汇编代码。
- ✓ @\$(OBJCOPY) -S -O binary \$(call objfile,bootblock) \$(call outfile,bootblock) 表示使用 objcopy 工具将 obj/bootblock.o 拷贝到 obj/bootblock.out 文件，其中-S 选项代表移除所有符号和重定位信息，-O binary 选项代表指定输出格式为二进制
- ✓ @\$(call totarget,sign) \$(call outfile,bootblock) \$(bootblock)表示使用 bin/sign 工具将之前的

obj/bootblock.out 用来生成 bin/bootblock 目标文件

✓ \$(call create_target,bootblock)直接返回

➤ 查看 make 执行的命令

输入 make "V=", 查看生成 bootblock 文件的具体过程:

```
+ 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
```

相关参数含义在之前已经展示过, 这里不再赘述。这里仅仅补充之前未出现过的。

➤ -N 代表设置代码段和数据段均可读写

➤ -e <entry>代表指定入口, 这里是 start

➤ -Ttext 代表代码段开始位置, 这里是 0x7C00

从这里我们可以看出, 生成 bootblock 文件所需要的依赖文件是 bootasm.o bootmain.o sign 这三个文件, 我们可以分别查看生成它们三个文件的具体过程。

```
+ cc boot/bootasm.S
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
```

这里特别出现了一个新的参数。

➤ -Os 参数表示为减小代码大小而进行优化, 因为主引导扇区只有 512 字节, 其中最后两位已被占用, 最后写出的 bootloader 不能大于 510 字节。

我们可以查看 sign.o 是怎么生成的。

```
+ 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
```

➤ -g 代表在编译的时候加入调试信息

➤ -O2 代表开启 O2 编译优化

⑤dd 指令

使用 make V=可以看到后续 dd 指令的具体操作

```
'obj/bootblock.out' size: 496 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
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, 0.0168944 s, 303 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, 0.000113326 s, 4.5 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
记录了154+1 的读入
记录了154+1 的写出
78916字节 (79 kB, 77 KiB) 已复制, 0.0002177 s, 362 MB/s
```

dd 指令的作用是使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 参数指定了源文件, of 参数指定了目标文件; 要从源文件拷贝到目标文件。

①dd if=/dev/zero of=bin/ucore.img count=10000

这条命令的作用是拷贝 10000 个全部为 0 的块, 存放到 bin 目录下新建的 ucore.img 文件之中; dev/zero 提供一个零设备, 用于提供无数个零。

②dd if=bin/bootblock of=bin/ucore.img conv=notrunc

这条命令的作用是将 bootblock 中的内容写入 ucore.img 之中, 从第一个块开始写。

③dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc

这条命令的作用是将 kernel 中的内容写入 ucore.img 之中，由于 seek 为 1，跳过了第一个块，从第二个块开始写。由于 bootblock 占用的空间为 512 个字节，可以想见，ucore.img 的 10000 个块中，第一个块存放 bootblock 程序，剩下的存放 kernel 程序。

⑥磁盘镜像相关原理

磁盘镜像是一个模拟的磁盘，计算机启动时需要从这里读取数据。首先，需要执行 BIOS 程序，这个程序会对 CPU 进行一定程度的初始化，并从磁盘的第一个块（也就是主引导扇区）里加载 bootblock 进入内存；bootblock 程序的作用是修改 CPU 从实模式变为保护模式，同时加载磁盘中剩余块里的 kernel 内核代码。bootblock 对应的是所谓加载程序，没有它就无法从磁盘中获取实现操作系统功能的内核代码；kernel 是真正的操作系统内核程序，bootblock 将它加载入内存后，就将控制权转移给它并开始运行操作系统。

（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？

在 sign.c 文件中，我们可以找到这一段核心代码

```
char buf[512];
memset(buf, 0, sizeof(buf));
FILE *ifp = fopen(argv[1], "rb");
int size = fread(buf, 1, st.st_size, ifp);
if (size != st.st_size) {
    fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
    return -1;
}
fclose(ifp);
buf[510] = 0x55;
buf[511] = 0xAA;
FILE *ofp = fopen(argv[2], "wb+");
size = fwrite(buf, 1, 512, ofp);
if (size != 512) {
    fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);
    return -1;
}
fclose(ofp);
printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
return 0;
```

可以看到，代码中 char buf[512]，buf[510] = 0x55，buf[511] = 0xAA，说明一个被系统认为是符合规范的硬盘主引导扇区的特征是：

- 一共 512 个字节
- 倒数第二个字节是 0x55，倒数第一个字节是 0xAA

练习 2：使用 qemu 执行并调试 lab1 中的软件

为了熟悉使用 qemu 和 gdb 进行的调试工作，我们进行如下的小练习：

1. 从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。
2. 在初始化位置 0x7c00 设置实地址断点，测试断点正常。
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。
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。

下面我将逐步操作

（1）从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。

根据附录所示，将 lab1/tools/gdbinit 替换为如下代码，

file bin/kernel

set architecture i8086

target remote :1234

break kern_init

continue

实际上就是加入了 set architecture i8086 这一行。

以上 5 行代码的含义实际上是

①首先进行gdb bin/kernel，加载内核程序（但是还不会执行）

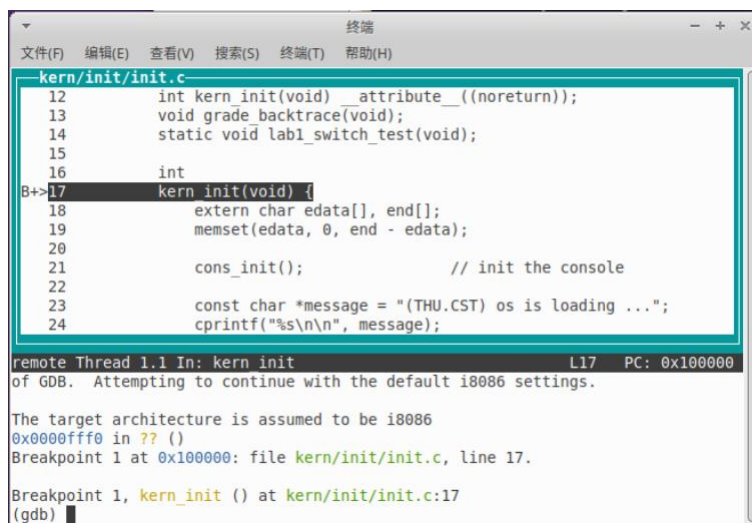
②然后建立与 qemu 的连接，指定 i8086 架构。

③随后用 b *0x7c00 指定了断点。这个位置是 bootloader 引导加载程序的第一条指令的地址。

④设置断点后执行continue，就会运行程序并停在 0x7c00 处。

⑤然后用 x/2i \$pc，查看内存中从当前 pc 指令寄存器中开始的两条指令。

然后在 lab1 的目录下输入 make debug，出现 gdb 调试界面之后，输入 si 单步跟踪 BIOS 的执行，通过语句 x/2i \$pc 可以显示当前 eip 处的汇编指令，查看 BIOS 的代码。



```

kern/init/init.c
12  int kern_init(void) __attribute__((noreturn));
13  void grade_backtrace(void);
14  static void lab1_switch_test(void);
15
16  int
B+>17  kern_init(void) {
18      extern char edata[], end[];
19      memset(edata, 0, end - edata);
20
21      cons_init();          // init the console
22
23      const char *message = "(THU.CST) os is loading ...";
24      cprintf("%s\n\n", message);

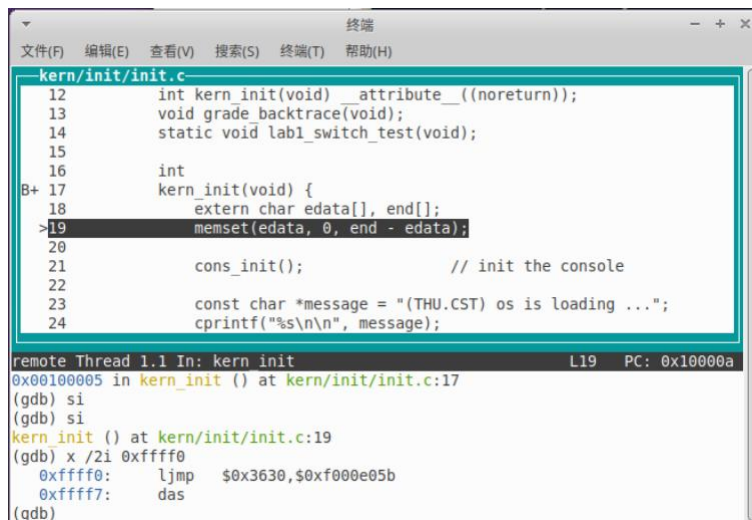
remote Thread 1.1 In: kern_init          L17  PC: 0x100000
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x0000ffff in ?? ()
Breakpoint 1 at 0x100000: file kern/init/init.c, line 17.

Breakpoint 1, kern_init () at kern/init/init.c:17
(gdb) 
```

可以看到，这时 gdb 停在 BIOS 的第一条指令处。此时如果输入 si，就可以看到 gdb 跳转到下一地址处，按照这种方式就可以单步跟踪 BIOS 了。

输入 x/2i \$pc 会显示当前 eip 处的汇编指令。例如输入 x/2i 0xffff0 即可查看 0xffff0 处以及往下的一行代码。



```

kern/init/init.c
12  int kern_init(void) __attribute__((noreturn));
13  void grade_backtrace(void);
14  static void lab1_switch_test(void);
15
16  int
B+ 17  kern_init(void) {
18      extern char edata[], end[];
>19      memset(edata, 0, end - edata);
20
21      cons_init();          // init the console
22
23      const char *message = "(THU.CST) os is loading ...";
24      cprintf("%s\n\n", message);

remote Thread 1.1 In: kern_init          L19  PC: 0x10000a
0x00100005 in kern_init () at kern/init/init.c:17
(gdb) si
(gdb) si
kern_init () at kern/init/init.c:19
(gdb) x/2i 0xffff0
0xffff0:    jmp     $0x3630,$0xf000e05b
0xffff7:    das
(gdb) 
```

(2) 在初始化位置 0x7c00 设置实地址断点,测试断点正常。

设置断点有多种方式，在 lab1/tools/gdbinit 文件中加入 `b *0x7c00` 或在 gdb 输入框输入 `b *0x7c00`，都可以在 0x7c00 设置断点。我这里选择第二种方式。

可以发现，输入 `c` 使程序继续运行后，程序在 0x7c00 处停下，断点正常。

(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。

可以单步跟踪，但这里我采取直接查看地址的方式。

```
(gdb) b *0x7c00
Breakpoint 2 at 0x7c00
(gdb) c
Continuing.

Breakpoint 2, 0x00007c00 in ?? ()
(gdb) x /10i $pc
```

可以查看到从 0x7c00 开始的 10 行汇编代码

```
=> 0x7c00: cli
0x7c01: cld
0x7c02: xor  %eax,%eax
0x7c04: mov  %eax,%ds
0x7c06: mov  %eax,%es
0x7c08: mov  %eax,%ss
0x7c0a: in   $0x64,%al

0x7c0c: test $0x2,%al
0x7c0e: jne  0x7c0a
0x7c10: mov  $0xd1,%al
```

以下为 bootasm.S

```

.globl start
start:
.code16                                # Assemble for 16-bit mode
cli                                  # Disable interrupts
cld                                  # String operations increment

# Set up the important data segment registers (DS, ES, SS).
xorw %ax, %ax                        # Segment number zero
movw %ax, %ds                        # -> Data Segment
movw %ax, %es                        # -> Extra Segment
movw %ax, %ss                        # -> Stack Segment

# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
inb $0x64, %al                      # Wait for not busy(8042 input buffer empty).
testb $0x2, %al
jnz seta20.1

movb $0xd1, %al                     # 0xd1 -> port 0x64
outb %al, $0x64                     # 0xd1 means: write data to 8042's P2 port

seta20.2:
inb $0x64, %al                      # Wait for not busy(8042 input buffer empty).
testb $0x2, %al
jnz seta20.2

movb $0xdf, %al                     # 0xdf -> port 0x60
outb %al, $0x60                     # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1

```

以下为 bootblock.asm

```

.code16                                # Assemble for 16-bit mode
cli                                  # Disable interrupts
cld                                  # String operations increment

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

00007c0a <seta20.1>:
# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
inb $0x64, %al                      # Wait for not busy(8042 input buffer empty).
7c0a: e4 64                          in      $0x64,%al
testb $0x2, %al
7c0c: a8 02                          test    $0x2,%al
jnz seta20.1
7c0e: 75 fa                          jne     7c0a <seta20.1>

movb $0xd1, %al                     # 0xd1 -> port 0x64
7c10: b0 d1                          mov     $0xd1,%al
outb %al, $0x64                     # 0xd1 means: write data to 8042's P2 port
7c12: e6 64                          out     %al,$0x64

00007c14 <seta20.2>:
seta20.2:
inb $0x64, %al                      # Wait for not busy(8042 input buffer empty).
7c14: e4 64                          in      $0x64,%al
testb $0x2, %al
7c16: a8 02                          test    $0x2,%al
jnz seta20.2
7c18: 75 fa                          jne     7c14 <seta20.2>

movb $0xdf, %al                     # 0xdf -> port 0x60
7c1a: b0 df                          mov     $0xdf,%al
outb %al, $0x60                     # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1
7c1c: e6 60                          out     %al,$0x60

```

可以发现，反汇编得到的代码与 bootasm.S 和 bootblock.asm 基本相同。

(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。

我在 0x7c04,0x7c06,0x7c08 三处设置断点，进行测试。

```
(gdb) b *0x7c04
Breakpoint 3 at 0x7c04
(gdb) b* 0x7c06
Breakpoint 4 at 0x7c06
(gdb) b* 0x7c08
Breakpoint 5 at 0x7c08
(gdb) █
```

可以看到，这三处断点都能成功。

```
remote Thread 1.1 In: L?? PC: 0x7c08
Breakpoint 3, 0x00007c04 in ?? ()
(gdb) si

Breakpoint 4, 0x00007c06 in ?? ()
(gdb) si

Breakpoint 5, 0x00007c08 in ?? ()
(gdb) █
```

练习 3：分析 bootloader 进入保护模式的过程

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 **bootloader**。请分析 **bootloader** 是如何完成从实模式进入保护模式的。

提示：需要阅读小节“保护模式和分段机制”和 `lab1/boot/bootasm.S` 源码，了解如何从实模式切换到保护模式，需要了解：

- 为何开启 A20，以及如何开启 A20
- 如何初始化 GDT 表
- 如何使能和进入保护模式

在结合提示以及阅读了 `lab1/boot/bootasm.S` 源码以及注释后。我们大概能够知道 **bootloader** 是通过修改 A20 地址线来完成从实模式进入保护模式的。接下来我将具体地呈现这一过程以及解答上述的一些疑惑。

(1) 一些准备工作

```
# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.
```

开头有一段注释，其意思为：BIOS 将此代码从硬盘的第一个扇区加载到物理地址为 `0x7c00` 的内存中，并开始以实模式在 `cs=0 ip=7c00` 执行。

```
.set PROT_MODE_CSEG, 0x8      # kernel code segment selector
.set PROT_MODE_DSEG, 0x10     # kernel data segment selector
.set CR0_PE_ON, 0x1           # protected mode enable flag
```

这一段代码设置内核代码段选择子、内核数据段选择子、保护模式使能标志置为 1

```
.code16                        # Assemble for 16-bit mode
cli                            # Disable interrupts
cld                            # String operations increment

# Set up the important data segment registers (DS, ES, SS).
xorw %ax, %ax                 # Segment number zero
movw %ax, %ds                 # -> Data Segment
movw %ax, %es                 # -> Extra Segment
movw %ax, %ss                 # -> Stack Segment
```

这一段代码设置 16 位模式，清理环境，关闭中断将 `flag` 置 0 并设置字符串操作是递增方向，将寄存器 `ax`、`ds`、`es`、`ss` 置 0

(2) 为何要开启 A20

或许这个问题我们可以从附录中窥见一点答案。

Intel 早期的 8086 CPU 提供了 20 根地址线,可寻址空间范围即 $0 \sim 2^{20}$ (00000H~FFFFFH) 的 1MB 内存空间。但 8086 的数据处理位宽位 16 位,无法直接寻址 1MB 内存空间,所以 8086 提供了段地址加偏移地址的地址转换机制。

PC 机的寻址结构超过了 20 位地址线的物理寻址能力。所以当寻址到超过 1MB 的内存时,会发生“回卷”(不会发生异常)。但下一代的基于 Intel 80286 CPU 的 PC AT 计算机系统提供了 24 根地址线,这样 CPU 的寻址范围变为 $2^{24}=16\text{M}$,同时也提供了保护模式,可以访问到 1MB 以上的内存了,此时如果遇到“寻址超过 1MB”的情况,系统不会再“回卷”了,这就造成了向下不兼容。

为了保持完全的向下兼容性,IBM 决定在 PC AT 计算机系统上加个硬件逻辑,来模仿以上的回绕特征,于是出现了 A20 Gate。他们的方法就是把 A20 地址线控制和键盘控制器的一个输出进行 AND 操作,这样来控制 A20 地址线的打开(使能)和关闭(屏蔽/禁止)。

一开始时 A20 地址线控制是被屏蔽的(总为 0),直到系统软件通过一定的 IO 操作去打开它(参看 bootasm.S)。

很显然,在实模式下要访问高端内存区,这个开关必须打开,在保护模式下,由于使用 32 位地址线,如果 A20 恒等于 0,那么系统只能访问奇数兆的内存,即只能访问 0--1M、2-3M、4-5M.....,这样无法有效访问所有可用内存。所以在保护模式下,这个开关也必须打开。

为了与最早的 PC 机向后兼容,物理地址行 20 被限制在低位,因此高于 1MB 的地址默认为零。此代码将撤消此操作,通过打开 A20,将键盘控制器上的 A20 线置于高电位,就能使全部 32 条地址线可用,可以访问 4G 的内存空间。

总结来说,如果不打开 A20,就会保留回卷机制,禁止访问大于 1MB 的空间,从而实现向下兼容,保留在实模式;而打开 A20,就会撤销回卷机制,允许访问大于 1MB 的空间。

为了能访问更多的空间,打开 A20 是一个必须的操作。

(3) 如何开启 A20

在这之前,我们需要了解 8042 芯片的一些属性。

8042 键盘控制器的 IO 端口是 0x60~0x6f,实际上 IBM PC/AT 使用的只有 0x60 和 0x64 两个端口(0x61、0x62 和 0x63 用于与 XT 兼容目的)。8042 通过这些端口给键盘控制器或键盘发送命令或读取状态。输出端口 P2 用于特定目的。位 0(P20 引脚)用于实现 CPU 复位操作,位 1(P21 引脚)用户控制 A20 信号线的开启与否。系统向输入缓冲(端口 0x64)写入一个字节,即发送一个键盘控制器命令。可以带一个参数。参数是通过 0x60 端口发送的。命令的返回值也从端口 0x60 去读。

8042 有 2 个端口地址与 4 个功能

- ✧ 读 60h 端口,读 output buffer
- ✧ 写 60h 端口,写 input buffer
- ✧ 读 64h 端口,读 Status Register
- ✧ 操作 Control Register,首先要向 64h 端口写一个命令(20h 为读命令,60h 为写命令),然后根据命令从 60h 端口读出 Control Register 的数据或者向 60h 端口写入 Control Register 的数据(64h 端口还可以接受许多其它的命令)。

8042 有 4 个寄存器

- ✧ 1 个 8-bit 长的 Input buffer; Write-Only;
- ✧ 1 个 8-bit 长的 Output buffer; Read-Only;
- ✧ 1 个 8-bit 长的 Status Register; Read-Only;
- ✧ 1 个 8-bit 长的 Control Register; Read/Write

程序可通过 60h 和 64h 端口操作寄存器。

- ✧ 直接读 60h 端口，可以获得 output buffer 寄存器中的内容；
- ✧ 直接写 60h 端口，可以写入 input buffer 寄存器内容。
- ✧ 直接读 64h 端口，可以读 Status Register 寄存器的内容。

对 Output Port 的操作及端口定义

- ✧ 读 Output Port: 向 64h 发送 0d0h 命令，然后从 60h 读取 Output Port 的内容
- ✧ 写 Output Port: 向 64h 发送 0d1h 命令，然后向 60h 写入 Output Port 的数据
- ✧ 禁止键盘操作命令: 向 64h 发送 0adh
- ✧ 打开键盘操作命令: 向 64h 发送 0aeh

理论上讲，我们只要操作 8042 芯片的输出端口（64h）的 bit 1，就可以控制 A20 Gate，但实际上，当你准备向 8042 的输入缓冲区里写数据时，可能里面还有其它数据没有处理，所以，我们要首先禁止键盘操作，同时等待数据缓冲区中没有数据以后，才能真正地去操作 8042 打开或者关闭 A20 Gate。打开 A20 Gate 的具体步骤大致如下（参考 bootasm.S）：

- 等待 8042 Input buffer 为空；
- 发送 Write 8042 Output Port （P2）命令到 8042 Input buffer；
- 等待 8042 Input buffer 为空；
- 将 8042 Output Port （P2）得到字节的第 2 位置 1，然后写入 8042 Input buffer；

下面的代码分为两部分，两部分代码都要通过读 0x64 端口的第 2 位确保 8042 的输入缓冲区为空后再进行操作。

```
seta20.1:
    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al               # 0xd1 -> port 0x64
    outb %al, $0x64               # 0xd1 means: write data to 8042's P2 port

seta20.2:
    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al               # 0xdf -> port 0x60
    outb %al, $0x60               # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1
```

在 seta20.1 中，首先把数据 0xd1 写入端口 0x64，发送消息给 CPU 准备往 8042 芯片的 P2 端口写数据；

在 seta20.2 中，首先把数据 0xdf 写入端口 0x60，从而将 8042 芯片的 P2 端口的 A20 地址线设置为 1。

（4）初始化 GDT 表

在上一步，我们开启了 A20 并切换了保护模式，接下来需要启动分段机制。

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to physical addresses, so that the
# effective memory map does not change during the switch.
lgdt gdt_desc
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
```

在 kern/mm/pmm.c 文件中可以找到 gdt 的初始化函数，通过这段代码完成 gdt 的初始化。


```

/* gdt_init - initialize the default GDT and TSS */
static void
gdt_init(void) {
    // Setup a TSS so that we can get the right stack when we trap from
    // user to the kernel. But not safe here, it's only a temporary value,
    // it will be set to KSTACKTOP in lab2.
    ts.ts_esp0 = (uint32_t)&stack0 + sizeof(stack0);
    ts.ts_ss0 = KERNEL_DS;

    // initialize the TSS filed of the gdt
    gdt[SEG_TSS] = SEG16(STS_T32A, (uint32_t)&ts, sizeof(ts), DPL_KERNEL);
    gdt[SEG_TSS].sd_s = 0;

    // reload all segment registers
    lgdt(&gdt_pd);

    // load the TSS
    ltr(GD_TSS);
}

/* pmm_init - initialize the physical memory management */
void
pmm_init(void) {
    gdt_init();
}

```

而在 bootasm.S 文件中，可以看到。

```

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
    SEG_NULLASM                           # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for bootloader and kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)       # data seg for bootloader and kernel

gdt_desc:
    .word 0x17                            # sizeof(gdt) - 1
    .long gdt                             # address gdt

```

其中 SEG_ASM 可以在 asm.h 中找到，

```

/* Normal segment */
#define SEG_NULLASM \
    .word 0, 0; \
    .byte 0, 0, 0, 0

#define SEG_ASM(type, base, lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
    (0xc0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xf)

```

可以看到，SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)和 SEG_ASM(STA_W, 0x0, 0xffffffff)把数据段和代码段的 base 设为 0，lim 即 limit 设置为 4G，数据段可读可执行，代码段可写，这样就可以使逻辑地址对应于线性地址。

因为一个简单的 GDT 表和其描述符已经静态储存在引导区中，所以直接使用 lgdt 命令初始化后，将 gdt 的 desc 段表示内容加载到 gdt 就行。也就是代码 lgdt gdt_desc

(5) 如何使能与进入保护模式

将 cr0 寄存器的 PE 位置即最低位设置为 1，就可以打开使能位，进入保护模式。

(6) 进入 32 位模式

接着，通过长跳转使 cs 的基地址得到更新，将 cs 修改为 32 位段寄存器，此时 CPU 进入 32 位模式。设置段寄存器 ds、es、fs、gs、ss，并建立堆栈的帧指针和栈指针。

```

.code32                                # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax         # Our data segment selector
    movw %ax, %ds                     # -> DS: Data Segment
    movw %ax, %es                     # -> ES: Extra Segment
    movw %ax, %fs                     # -> FS
    movw %ax, %gs                     # -> GS
    movw %ax, %ss                     # -> SS: Stack Segment

```

(7) 调用 bootmain 函数

调用 bootmain 函数，bootloader 从实模式进入保护模式。bootmain 函数将从磁盘中读取 kernel 内核代码，进行下一步的操作。

练习 4：分析 bootloader 加载 ELF 格式的 OS 的过程

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，

- bootloader 如何读取硬盘扇区的？
- bootloader 是如何加载 ELF 格式的 OS？

提示：可阅读“硬盘访问概述”，“ELF 执行文件格式概述”这两小节。

由提示，我们不难发现，其实 bootladder 加载 ELF 格式的 OS 的过程大致应该分为读取磁盘扇区、ELF 格式加载这两个过程。我们将逐步渐进讨论。

(1) bootloader 如何读取硬盘扇区

bootloader 让 CPU 进入保护模式后，下一步的工作就是从硬盘上加载并运行 OS。考虑到实现的简单性，bootloader 的访问硬盘都是 LBA 模式的 PIO（Program IO）方式，即所有的 IO 操作是通过 CPU 访问硬盘的 IO 地址寄存器完成。

一般主板有 2 个 IDE 通道，每个通道可以接 2 个 IDE 硬盘。访问第一个硬盘的扇区可设置 IO 地址寄存器 0x1f0-0x1f7 实现的，具体参数见下表。一般第一个 IDE 通道通过访问 IO 地址 0x1f0-0x1f7 来实现，第二个 IDE 通道通过访问 0x170-0x17f 实现。每个通道的主从盘的选择通过第 6 个 IO 偏移地址寄存器来设置。

注意：第 6 位：为 1=LBA 模式；0 = CHS 模式 第 7 位和第 5 位必须为 1
以下为磁盘 IO 地址和对应功能

IO 地址	功能
0x1f0	读数据，当 0x1f7 不为忙状态时，可以读。
0x1f2	要读写的扇区数，每次读写前，你需要表明你要读写几个扇区。最小是 1 个扇区
0x1f3	如果是 LBA 模式，就是 LBA 参数的 0-7 位
0x1f4	如果是 LBA 模式，就是 LBA 参数的 8-15 位
0x1f5	如果是 LBA 模式，就是 LBA 参数的 16-23 位
0x1f6	第 0~3 位：如果是 LBA 模式就是 24-27 位 第 4 位：为 0 主盘；为 1 从盘
0x1f7	状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从 0x1f0 端口读数据

当前 硬盘数据是储存到硬盘扇区中，一个扇区大小为 512 字节。读一个扇区的流程（可参看 boot/bootmain.c 中的 readsect 函数实现）大致如下：

- 等待磁盘准备好
- 发出读取扇区的命令
- 等待磁盘准备好
- 把磁盘扇区数据读到指定内存

(2) bootloader 是如何加载 ELF 格式的 OS

在阅读材料“ELF 执行文件格式概述”中，表明了 bootloader 是如何加载 ELF 格式的 OS。ELF header 在文件开始处描述了整个文件的组织。ELF 的文件头包含整个执行文件的控制结构，其定义在 elf.h 中：

```

struct elfhdr {
    uint magic; // must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint entry; // 程序入口的虚拟地址
    uint phoff; // program header 表的位置偏移
    uint shoff;
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum; //program header表中的入口数目
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};

```

program header 描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。可执行文件的程序头部是一个 **program header** 结构的数组，每个结构描述了一个段或者系统准备程序执行所必需的其它信息。

目标文件的“段”包含一个或者多个“节区”（**section**），也就是“段内容（**Segment Contents**）”。程序头部仅对于可执行文件和共享目标文件有意义。可执行目标文件在 **ELF** 头部的 **e_phentsize** 和 **e_phnum** 成员中给出其自身程序头部的大小。程序头部的数据结构如下表所示：

```

struct proghdr {
    uint type; // 段类型
    uint offset; // 段相对文件头的偏移值
    uint va; // 段的第一个字节将被放到内存中的虚拟地址
    uint pa;
    uint filesz;
    uint memsz; // 段在内存映像中占用的字节数
    uint flags;
    uint align;
};

```

根据 **elfhdr** 和 **proghdr** 的结构描述，**bootloader** 就可以完成对 **ELF** 格式的 **ucore** 操作系统的加载过程（参见 **boot/bootmain.c** 中的 **bootmain** 函数）。

（3）bootmain.c 代码解读

①宏定义

```

#define SECTSIZE      512
#define ELFHDR        ((struct elfhdr *)0x10000) // scratch space

```

②waitdisk()函数：等待磁盘就绪

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

这段代码实际上不断查询 0x1F7 寄存器的最高两位，当最高两位为 01，即磁盘空闲时，才允许程序继续运行。

③readsect 函数

```
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1); // count = 1
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20); // cmd 0x20 - read sectors

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}
```

这个函数的作用是从设备的第 secno 个扇区的文章读取数据到 dst 内存中。

④readseg 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;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for (; va < end_va; va += SECTSIZE, secno++) {
        readsect((void *)va, secno);
    }
}
```

调用 readsect 函数实现从设备中读取任意长度内容。

⑤bootmain 函数

```

/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

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

```

该函数主要完成以下任务

1、首先从磁盘的第一个扇区中将 ELF 文件 bin/kernel 的内容读取出来

代码: `readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);`

2、接下来检验 ELF 头部的 `e_magic` 变量判断是不是 ELF 文件

```

if (ELFHDR->e_magic != ELF_MAGIC) {
    goto bad; 若不是，跳转至后面处理
}

```

3、读取 ELF 头部的 `e_phoff` 变量得到描述表的头地址。表示 ELF 文件应该加载到内存的什么位置

代码: `struct proghdr *ph, *eph;`

```

ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);

```

4、读取 ELF 头部的 `e_phnum` 变量，得到描述表的元素数目。

代码: `eph = ph + ELFHDR->e_phnum;`

5、按照描述表将 ELF 文件中数据按照偏移、虚拟地址、长度等信息载入内存

```

for (; ph < eph; ph++) {
    readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
}

```

6、通过 ELF 头部的 `e_entry` 变量储存的入口信息，找到内核的入口地址，并开始执行内核代码

```

((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();

```

⑥总结

总结来说，bootloader 加载 ELF 格式的 OS 的大致过程是先等待磁盘准备就绪，然后先读取 ELF 的头部判断是否合法，接着读取 ELF 内存位置的描述表，然后按照描述表的内容，将 ELF 文件中的数据载入内存，根据 ELF 头部的入口信息找到内核入口执行内核代码。

练习 5：实现函数调用堆栈跟踪函数

这一题我们需要先复习函数调用时栈的变化，并结合该部分知识完成代码补全

(1) 函数调用堆栈相关知识

为了实现函数堆栈跟踪函数，首先需要比较清楚地掌握函数调用时栈相关的变化。

(这一部分的知识可以参考 CS 计算机系统)

栈帧寄存器是 `ebp`，栈顶寄存器是 `esp`。栈是向下增长的，每次放入数据，`esp` 里的值都会减小。

①调用

假设在 `main` 函数里通过 `call` 指令调用 `add` 函数，则 `main` 会先在自己的栈帧中保存即将传递给过程的参数以及返回地址（`call` 指令负责将返回地址保存到栈之中）。

在 `call` 之后，`add` 会首先保存 `main` 的 `ebp` 值，其地址比栈中保存返回地址的位置更小；然后把此时 `esp` 的值赋值给 `ebp`，这样一来 `ebp` 就变成了 `add` 函数的栈帧，它直接指向 `main` 函数的旧 `ebp` 值，它加 4 的结果就是返回地址，再往上就是函数参数。

接下来把 `esp` 进行自减并对齐，就可以开辟属于 `add` 函数的栈空间。

②返回

等函数执行完毕，先把返回值保存在 `eax` 寄存器之中，再给 `esp` 赋值此时 `ebp` 的值使得 `add` 函数的栈顶指针和栈帧指针一致（即将 `esp` 指针撤回至 `ebp` 的位置）；接着弹出栈顶的旧 `ebp` 值给 `ebp` 寄存器，恢复 `main` 函数的栈帧；

之后，`ret` 指令把新的栈顶元素也就是保存的返回地址交给 `eip`，从而转交回控制权。

③地址关系

对于被调用者而言，`[ebp]` 处为保存的调用者的旧 `ebp` 值；`[ebp+4]` 处为返回地址，`[ebp+8]` 处为第一个参数值(最后一个入栈的参数值，此处假设其占用 4 字节内存)。

由于 `ebp` 中的地址处总是“上一层函数调用时的 `ebp` 值”，而在每一层函数调用中，都能通过当时的 `ebp` 值“向上(栈底方向)”能获取返回地址、参数值，“向下(栈顶方向)”能获取函数局部变量值。如此形成递归，直至到达栈底。

(2) 实现函数 `print_stackframe`

我们需要在 `lab1` 中完成 `kdebug.c` 中函数 `print_stackframe` 的实现，可以通过函数 `print_stackframe` 来跟踪函数调用堆栈中记录的返回地址。

打开 `/labcodes/lab1/kern/debug/kdebug.c`，关注到待补全部分。按照注释信息，比较容易地完成待补全部分的代码。

```
void
print_stackframe(void) {
    /* LAB1 YOUR CODE : STEP 1 */
    /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
       uint32_t ebp_val=read_ebp();
    /* (2) call read_eip() to get the value of eip. the type is (uint32_t);
       uint32_t eip_val=read_eip();
    /* (3) from 0 .. STACKFRAME_DEPTH
       for (int i=0; ebp_val != 0 && i<STACKFRAME_DEPTH; i++){
    /* (3.1) printf value of ebp, eip
       printf("ebp:0x%08x eip:0x%08x args:", ebp_val, eip_val);
    /* (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp + 2 [0..4]
       uint32_t *call_args = (uint32_t *)ebp_val + 2;
       printf("0x%08x 0x%08x 0x%08x 0x%08x", call_args[0], call_args[1], call_args[2], call_args[3]);
    /* (3.3) printf("\n");
       printf("\n");
    /* (3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
       print_debuginfo(eip_val - 1);
    /* (3.5) pop up a calling stackframe
       NOTICE: the calling function's return addr eip = ss:[ebp+4]
       eip_val = *((uint32_t *) (ebp_val + 4));
       the calling function's ebp = ss:[ebp]
       ebp_val = *((uint32_t *) ebp_val);
    /*
    }
}
```


根据附录指示,如果能够正确实现此函数,可在 lab1 中执行 “make qemu”后,在 qemu 模拟器中得到类似如下的输出:

```
.....
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xfffff000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xfffff000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xfffff000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c53
    kern/init/init.c:28: kern_init+88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d72 -
.....
```

在本地进行 make qemu 后,得到如下所示输出。

```
Special kernel symbols:
entry 0x00100000 (phys)
etext 0x00103428 (phys)
edata 0x0010fa16 (phys)
end 0x00110d20 (phys)
Kernel executable memory footprint: 68KB
ebp:0x00007b28 eip:0x00100ab4 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:297: print_stackframe+26
ebp:0x00007b38 eip:0x00100dc0 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+14
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xfffff000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+37
ebp:0x00007b78 eip:0x001000c4 args:0x00000000 0xfffff000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+42
ebp:0x00007b98 eip:0x001000e7 args:0x00000000 0x00100000 0xfffff000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+27
ebp:0x00007bb8 eip:0x00100111 args:0x0010345c 0x00103440 0x0000130a 0x00000000
    kern/init/init.c:63: grade_backtrace+38
ebp:0x00007be8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d74 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d73 --
++ setup timer interrupts
```

可以发现,其输出与给定参考大致一致。则代码补全正确。

(3) 函数 print_stackframe 解析

从注释中我们可以发现,内联函数 read_ebp()可以告诉我们当前 ebp 的值。而非内联函数 read_eip()也很有用,它可以读取当前 eip 的值,因为在调用此函数时,read_eip()可以轻松地从堆栈中读取调用方的 eip。因此前两行的作用就是读取初始的 ebp 与 eip。

接着我们从第 1 层遍历到 STACKFRAME_DEPTH 层,STACKFRAME_DEPTH 这表示我们需要回溯的调用嵌套个数。

对于每一次遍历,我们首先打印该层的 ebp 与 eip 的值,然后通过地址来寻找调用者向本层传递的参数值信息。

这里需要注意,ebp 指针指向的位置是旧的 ebp 值,(ebp)+4 指向的位置是返回值,(ebp)+8 指向的位置开始才是真正的参数。

由于 `uint32_t*` 类型指针占据 4 个字节,所以 `(uint32_t *)ebp+2` 对应的地址值就是 `(ebp)+8`,这里涉及到指针的加法,指针自增加上的是该单个类型数据的字节数。

通过 `(uint32_t *)ebp+2` 将 `call_args` 指针指向第一个参数的地址。从这个地址开始以及往后的三个,也就是 `call_args[0]`, `call_args[1]`, `call_args[2]`, `call_args[3]` 这四个分别表示 `(ebp)+8`, `(ebp)+12`, `(ebp)+16`, `(ebp)+20`,这就是来自调用者传递的 4 个参数。

接着按照注释要求,调用 `print_debuginfo(eip-1)`,打印相关信息。

最后是回溯一步,将此时的 ebp 值与 eip 值回溯到本层的调用者,相当于是上溯一层。这样可以实现递归的具体实现过程。

(4) 最后一行结果的解析

附录要求我们解释最后一行的输出结果。

```
ebp:0x00007bf8 eip:0x00007d74 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
```

```
<unknown>: -- 0x00007d73 --
```

可以看到,最后一行的 ebp 值最大,意味着它是最初的函数,是刚刚一切函数嵌套调用的源头。结合前面的 `bootloader` 相关问题的探讨,若不考虑 BIOS 程序段,在保护模式中第一个执行的函数是 `bootloader`。`bootloader` 程序是从 `0x7c00` 开始的,在 `0x7d70` 处使用 `call` 指令进行了第一次嵌套调用,`call` 指令将下一条指令的地址也就是 `0x7d72` 保存在栈中。而这恰好就是我们这里最后一行所示调用的源头。

练习 6: 完善中断初始化和处理

请完成编码工作和回答如下问题:

- 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?
- 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。
- 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 100 次时钟中断后,调用 `print_ticks` 子程序,向屏幕上打印一行文字“100 ticks”。

【注意】

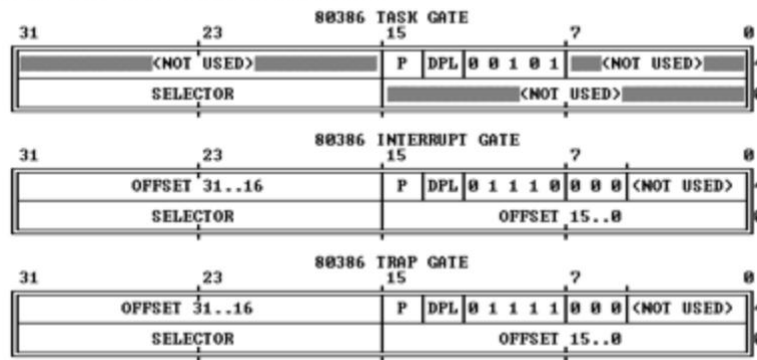
除了系统调用中断(`T_SYSCALL`)使用陷阱门描述符且权限为用户态权限以外,其它中断均使用特权级(DPL)为 0 的中断门描述符,权限为内核态权限;而 `ucore` 的应用程序处于特权级 3,需要采用 `int 0x80` 指令操作(这种方式称为软中断,软件中断,Trap 中断,在 lab5 会碰到)来发出系统调用请求,并要能实现从特权级 3 到特权级 0 的转换,所以系统调用中断(`T_SYSCALL`)所对应的中断门描述符中的特权级(DPL)需要设置为 3。

提示:可阅读小节“中断与异常”。

(1) 中断描述符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?

在中断向量表中,一个表项会占 8 个字节,其中第 0-1 和第 6-7 字节组合在一起表示偏移量,第 2~3 字节表示段选择的编号,在选择的段中,计算偏移量后得到的位置,就是中断处理代码的入口。

Figure 9-3. 80386 IDT Gate Descriptors



此外，可以通过查看下列文件中 lab1/kern/mm/mmu.h 比较清晰地看出来。

```
/* Gate descriptors for interrupts and traps */
struct gatedesc {
    unsigned gd_off_15_0 : 16;    // low 16 bits of offset in segment
    unsigned gd_ss : 16;           // segment selector
    unsigned gd_args : 5;          // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;           // reserved (should be zero I guess)
    unsigned gd_type : 4;           // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;             // must be 0 (system)
    unsigned gd_dpl : 2;           // descriptor(meaning new) privilege level
    unsigned gd_p : 1;             // Present
    unsigned gd_off_31_16 : 16;    // high bits of offset in segment
};
```

(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 根据注释完成代码。

首先关注到第一段注释：

“每个中断服务例程（ISR）的入口地址在哪里？（只有找到中断地址，才能初始化 IDT 表）所有 ISR 的地址都存储在 `_vectors` 中。 `uintptr_t __vectors[]` 在哪里？ `__vectors[]` 位于 kern/trap/vector.S 中，由 tools/vector.c 生成（在 lab1 中尝试 “make” 命令，然后在 kern/trap/DIR 中找到 vector.S）您可以使用 “`extern uintptr_t __vectors[];`” 来定义此 extern 变量（外部变量，意味着这个数组是其他文件夹里的，只不过本文件中的函数需要使用它来初始化 idt 表。），该变量将在后头用到。”

从这一段注释里，我们可以看到首先我们需要定义一个 `extern uintptr_t` 类型变量 `__vectors[]`，用来存放 256 个在 vector.S 定义的中断处理例程的入口地址。

接下来关注到第二段注释：

“现在您应该在中断描述符表（IDT）中设置 ISR（各个中断门）条目。你能在这个文件中看到 `idt[256]` 吗？是的，这个数组就是 IDT 中断描述符表（只要给这个数组赋值就可以初始化 IDT 表了）！您可以使用 SETGATE 宏设置 IDT 的各个条目。”

使用 SETGATE 宏，通过循环语句对中断描述符表中的每一个表项进行设置，其中 SETGATE 宏可以在 mmu.h 中找到： `#define SETGATE(gate, istrap, sel, off, dpl)`

其参数含义如下

- 宏的参数 `gate` 代表选择的 idt 数组的项，是处理函数的入口地址
- 参数 `istrap` 为 1 时代表系统段，为 0 时代表中断门
- 参数 `sel` 是中断处理函数的段选择子，GD_KTEXT 代表是 .text 段
- 参数 `off` 是 `__vectors` 数组内容，在 vector.S 中，有 256 个中断处理例程
- 参数 `dpl` 是优先级，宏定义 DPL_KERNEL 是 0 代表内核级，宏定义 DPL_USER 是 3 代表用户级。

- 宏定义 `T_SWITCH_TOK` 是用于用户态切换到内核态的中断号。
故这里一一对应即可完成补全。

最后是第三段注释：

“设置 IDT 的内容后，您将使用“`lidt`”指令让 CPU 知道 IDT 在哪里。你不知道这个说明的意思吗？只需谷歌一下！并查看 `libs/x86.h` 以了解更多信息。注意：`lidt` 的参数是 `idt_pd`。试着找到它！”

由于 `lidt` 的参数为指针，故这里需要以指针的形式传入。

最终完成的代码如下。

```
/* 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.*/
    extern uintptr_t __vectors[];
    /* (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*/
    int idt_size = sizeof(idt) / sizeof(struct gatedesc);
    for (int i = 0; i < idt_size; ++i) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    /* (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!*/
    lidt(&idt_pd);
    /*/
}
```

其中，较为关键的是最后一行的 `DPL_USER`，表示在用户态下就可以完成对于内核态的访问。

(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

这一问主要是向终端输出一些信息。

先看注释：

“（1）计时器中断后，应使用全局变量（增加它）记录此事件，如 `kern/driver/clock.c` 中的 `ticks` *（2）每个 `TICK_NUM` 周期，您都可以使用一个函数打印一些信息，例如使用 `print_ticks()` 函数”

故只要写一个 `ticks` 变量自增以及满 100 输出一次即可。

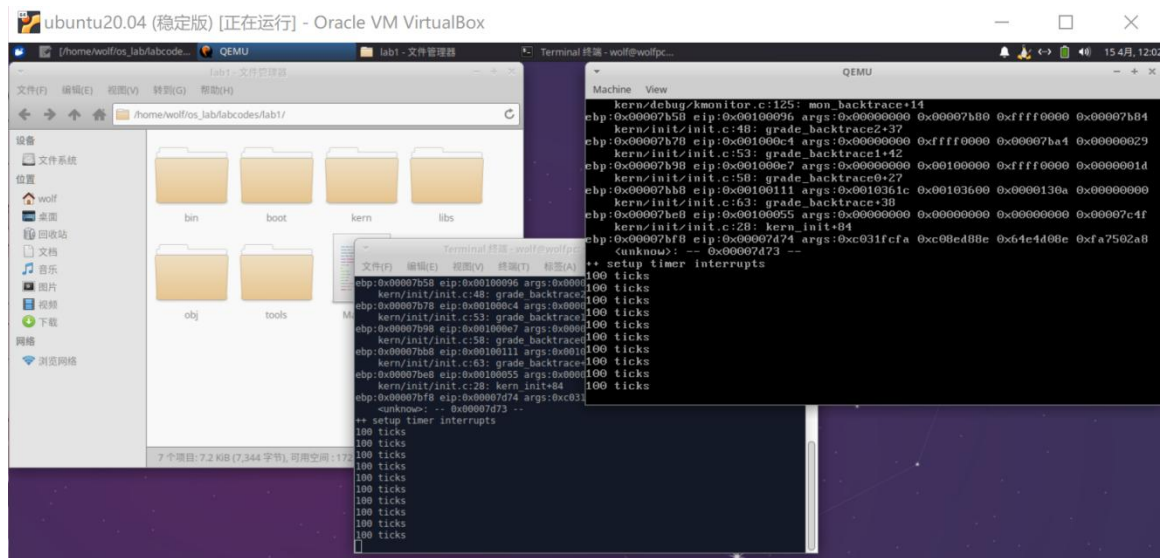
代码如下。

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;

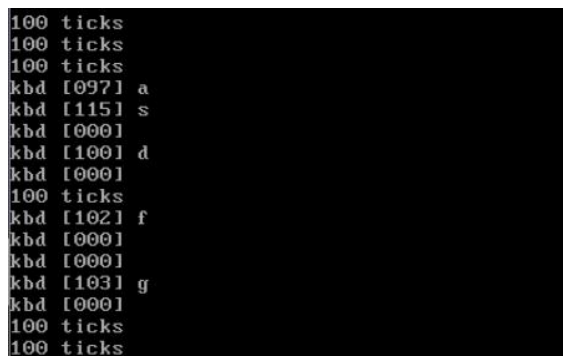
    switch (tf->tf_trapno) {
        case IRQ_OFFSET + IRQ_TIMER:
            /* LAB1 YOUR CODE : STEP 3 */
            /* handle the timer interrupt */
            /* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in kern/driver/clock.c*/
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

(4) 执行程序

在完成（2）（3）段代码之后，在 `lab1` 目录下执行程序，即可看到每 100 次时钟中断终端上输出一行。执行截图如下。



如图是我快速按下 `asdfg` 的终端显示



可以看到，大概每 1 秒输出一行“100 ticks”文字，而且按下的键也会在屏幕上显示。

扩展练习 Challenge1

扩展 `proj4`, 增加 `syscall` 功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务。

提示：

规范一下 `challenge` 的流程。

`kern_init` 调用 `switch_test`，该函数如下：

```
static void
switch_test(void) {
    print_cur_status();           // print 当前 cs/ss/ds 等寄存器状态
    cprintf("+++ switch to user mode +++\n");
    switch_to_user();             // switch to user mode
    print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    switch_to_kernel();           // switch to kernel mode
    print_cur_status();
}
```


`switchto*` 函数建议通过 中断处理的方式实现。主要要完成的代码是在 `trap` 里面处理 `T_SWITCH_TO*` 中断，并设置好返回的状态。

在 `lab1` 里面完成代码以后，执行 `make grade` 应该能够评测结果是否正确。

（1）中断及特权保护机制

中断发生后，首先获取中断向量；然后，以中断向量为索引，去中断描述符表 IDT（IDT 的位置和大小存放在寄存器 IDTR 中，可以从这个寄存器器读出 IDT 处于什么地址）中获得中断描述符（中断描述符分为**段选择子**、**段偏移量**两个部分，其中段选择子部分含有一个 RPL 请求特权级。注意，当前执行代码有一个 CPL 当前程序特权级，存放在特定寄存器中），依据中断描述符的段选择子找到 GDT 中的段描述符（里面有段的访问特权级 DPL），而后可以根据段基址和段偏移量，获得中断服务例程的位置，并跳转到那里执行。

特权级的实现：

特权级在 `ucore` 中分为 0 和 3，3 表示用户态的特权级，0 表示内核态的特权级；只能被内核态访问的数据不能被用户态的特权级访问，内核态特权级可以访问任意特权级的数据，因为数字越小特权级的级别越高。

每一个段都有自己专有的特权级称为 DPL，它被保存在 GDT 段表的段描述符中，表示访问这个段所需的最低特权级（只有数字值小于等于它的、特权级更高的特权级才能访问）；

每一段正在执行的代码有自己的特权级 CPL，保存在 CS 和 SS 的第 0 位和第 1 位上，表示代码所在的特权级——当程序转移到不同特权级的代码段时，CPL 将被改变，表示程序本身的特权级在用户态和内核态之间切换。

中断描述符的段选择子中含有的特权级 RPL 是请求特权级，表示当前代码段发出了一个特定特权级的请求。比如说，一个 CPL 为 3 的用户态的程序，需要执行软中断，所以发出了一个内核态的请求，则其 RPL 就将是 0。一般，访问段时，会取 CPL 和 DPL 中较低的特权级用来与 DPL 进行比对，从而判断能否访问特定段。

但在系统调用中，系统调用的过程发生了特权级的变化，为了执行系统调用，CPL 需要从用户态升级为内核态，数字由 3 变成 0。

内核态和用户态使用不同的栈来执操作，因此特权级切换的本质就是使用内核态的栈而不是用户态的栈，同时保存用户态栈的相关信息便于在中断完成后恢复到用户。

CPU 会根据 CPL 和 DPL 判断需要进行特权级转换（从用户态升级为内核态）。一旦发生，需要从 TR 寄存器中获得当前程序的 TSS 信息的地址，从 TSS 信息中获得内核栈地址，然后将用户态的栈地址信息（SS 和 ESP 值）保存到内核栈中。中断打断了用户态的程序，所以还要把用户态程序的 `eflags`、`cs`、`eip`，乃至可能存在的 `errorCode` 信息都压入内核栈之中。

中断结束后，使用 `iret` 指令可以弹出内核栈中的 `eflags`；如果中断时有特权级切换，说明内核栈中还有用户栈的 SS 和 ESP 信息，`iret` 会把它们也弹出，以便恢复到用户态。

（2）Ucore 如何处理中断

在 `lab1` 的代码中，`vector.S`文件中保存了各个中断向量（中断号）所对应的中断服务例程入口。以部分中断向量（实际上这里有很多，我们只截取部分示例）的相应处理示意如下：


```
# handler
.text
.globl __alltraps
.globl vector0
vector0:
    pushl $0
    pushl $0
    jmp __alltraps
.globl vector1
vector1:
    pushl $0
    pushl $1
    jmp __alltraps
.globl vector2
vector2:
    pushl $0
    pushl $2
    jmp __alltraps
.globl vector3
vector3:
    pushl $0
    pushl $3
    jmp __alltraps
.globl vector4
```

可以看出，上述中断向量对应的操作，是向栈中压入0，然后压入中断号，接着跳转到 `__alltraps`（位于 `trapentry.S` 文件中）中断处理函数，其内容如下：

```
# vectors.S sends all traps here.
.text
.globl __alltraps
__alltraps:
    # push registers to build a trap frame
    # therefore make the stack look like a struct trapframe
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal

    # load GD_KDATA into %ds and %es to set up data segments for kernel
    movl $GD_KDATA, %eax
    movw %ax, %ds
    movw %ax, %es

    # push %esp to pass a pointer to the trapframe as an argument to trap()
    pushl %esp

    # call trap(tf), where tf=%esp
    call trap
```

从注释可以看出，它专门处理从 `vector.S` 发出的中断请求。为了执行中断，他会将诸多寄存器保存在栈中，如 `ds` 到 `gs`；接着，直接调用 `call` 指令，使用 `trap` 函数继续进行处理。

`trap` 函数进一步调用 `trap_dispatch` 函数，它在练习 6 中被补充，从而可以对时钟 `ticks` 计数和输出相应信息。实际上，`trap_dispatch` 函数会根据 `tf` 中保存的中断信息（特别是中断号）来进行真正的中断服务。

中断服务结束后，会依赖 `tp` 中保存的环境信息恢复到中断前的状态。因此，实现扩展练习 1 的方式就是在中断响应函数 `trap_dispatch` 中响应关于申请状态切换的中断请求，响应方式就是修改 `tp` 信息，也就是修改中断结束后需要赖以恢复的状态信息。

操作系统加载后，会首先执行 `kern_init` 函数进行内核初始化。使这个函数得以调用 `switch_test`。这个函数的作用，是先执行 `switch_to_user()`；切换到用户态，然后执行 `switch_to_kernel()`；切换到内核态。实现这两个函数，本质上就是让它们发出中断申请——分别用于申请切换为用户态和申请切换为内核态。

（3）补全代码

首先，根据注释，在 `init.c` 文件的 `kern_init()` 函数里面，将原先被注释掉的代码 `lab1_switch_test()` 去掉注释，变成可以执行的语句。这一段十分重要，如果没有去掉注释，会造成最后的 `make grade` 不通过。

关注到下面的这两个函数是待实现的。函数及截图如下。

`static void lab1_switch_to_user(void)`和 `static void lab1_switch_to_kernel(void)`

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
}

static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 : TODO
}
```

接下来我将分别讲解。

①static void lab1_switch_to_user(void)函数

对于 static void lab1_switch_to_user(void)，这个函数的功能是从内核态返回到用户态，需要调用 T_SWITCH_TOU 中断，在函数中使用内联汇编实现：

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile (
        "pushl %%ss\n"
        "pushl %%esp\n"
        "int %0\n"
        "movl %%ebp, %%esp"
        :
        : "i"(T_SWITCH_TOU)
    );
}
```

在调用中断之前首先需要使用语句"pushl %%ss\n"和 pushl %%esp\n"提前将 ss、esp 压入栈，因为当切换优先级时，中断返回时 iret 指令会额外弹出 ss 和 esp 两位，但使用"int %0\n"语句调用 T_SWITCH_TOU 中断时并不会产生特权级的切换，因此不用压入 ss 和 esp，所以要先将栈压两位，预先留出空间，在中断返回后使用"movl %%ebp, %%esp" :: "i"(T_SWITCH_TOU)语句恢复栈指针，修复 esp。

②static void lab1_switch_to_kernel(void)函数

对于函数 static void lab1_switch_to_kernel(void)，实现的功能是从用户态切换回内核态，需要调用 T_SWITCH_TOK 中断，在函数中使用内联汇编实现：

```
static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile (
        "int %0\n"
        "movl %%ebp, %%esp\n"
        :
        : "i"(T_SWITCH_TOK)
    );
}
```

从用户态切换到内核态时，由于用户态使用"int %0\n"语句调用 T_SWITCH_TOU 中断时会自动切换到内核态，不会另外弹出 ss、esp 两位，中断返回时，esp 仍在堆栈中，在中断返回后要使用 "movl %%ebp, %%esp\n" :: "i"(T_SWITCH_TOK)语句恢复栈指针，修复 esp。

③case T_SWITCH_TOU 与 case T_SWITCH_TOK

在 trap.c 文件中，找到 trap_dispatch()函数中等待完成的 case T_SWITCH_TOU 和 case T_SWITCH_TOK，先定义一个 struct trapframe 类型的变量 switchktou 和一个 struct trapframe * 类型的指针变量 switchutok。

```

//LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
//case T_SWITCH_TOU:
case T_SWITCH_TOU:
// 如果原先保存在trapframe中的cs不是代表用户态的USER_CS
if (tf->tf_cs != USER_CS) {
    // 将保存在trapframe中的cs改成代表用户态的USER_CS
    tf->tf_cs = USER_CS;
    // 将其它的段选择子都修改为代表用户态的USER_DS, 保证中断返回之后可以正常访问数据
    tf->tf_ds = USER_DS;
    tf->tf_es = USER_DS;
    tf->tf_ss = USER_DS;
    // 为了程序在CPL较低的情况下也能使用IO, 需要将对应的IOPL位置改成用户态
    tf->tf_eflags |= FL_IOPL_MASK;
}
break;

//case T_SWITCH_TOK:
case T_SWITCH_TOK:
// 如果原先保存在trapframe中的cs不是代表内核态的KERNEL_CS
if (tf->tf_cs != KERNEL_CS) {
    // 将保存在trapframe中的cs改成代表内核态的KERNEL_CS
    tf->tf_cs = KERNEL_CS;
    // 将其它的段选择子都修改为代表内核态的KERNEL_DS, 保证中断返回之后可以正常访问数据
    tf->tf_ds = KERNEL_DS;
    tf->tf_es = KERNEL_DS;
    // 将调用IO所需权限降低, 才能输出文本
    tf->tf_eflags |= 0x3000;
}
break;

```

(4) 执行程序, 验证结果

实现后, 在终端使用 `make grade` 检验实现效果, 得分如下。

```

wolf@wolfpc:~/os_lab/labcodes/lab1$ make grade
Check Output: (2.2s)
-check ring 0: OK
-check switch to ring 3: OK
-check switch to ring 0: OK
-check ticks: OK
Total Score: 40/40

```

扩展练习 Challenge2

用键盘实现用户模式内核模式切换。具体目标是: “键盘输入 3 时切换到用户模式, 键盘输入 0 时切换到内核模式”。基本思路是借鉴软中断(syscall 功能)的代码, 并且把 `trap.c` 中软中断处理的设置语句拿过来。

注意:

1.关于调试工具, 不建议用 `lab1_print_cur_status()`来显示, 要注意到寄存器的值要在中断完成后 `tranentry.S` 里面 `iret` 结束的时候才写回, 所以在 `trap.c` 里面不好观察, 建议用 `print_trapframe(tf)`

2.关于内联汇编, 最开始调试的时候, 参数容易出现错误, 可能的错误代码如下

```

asm volatile ( "sub $0x8, %%esp \n"
               "int %0 \n"
               "movl %%ebp, %%esp"
               : )

```

要去掉参数 `int %0 \n` 这一行

3.软中断是利用了临时栈来处理的, 所以有压栈和出栈的汇编语句。硬件中断本身就在内核态了, 直接处理就可以了。

首先在 `trap.c` 文件中找到与键盘中断返回有关的代码, 即 `case IRQ_OFFSET + IRQ_KBD`, 在其中加入一个感知键盘输入数组的条件判断语句, 如果输入是 3 则进入用户模式, 如果输

入是 0 则进入内核模式。因为在内核态进入到用户态的过程中，iret 指令中断返回时会额外弹出两位，所以为了保护堆栈上的信息，可以将 **trapframe** 的地址保存到一个变量中，当键盘输入 3 准备从内核模式切换到用户模式时，可以从这个变量中获取正确的 **trapframe** 的地址，恢复栈指针，修复 **esp**。

而因为用户态进入到内核态的过程中，因为 iret 指令调用中断时是系统默认的从权限较低的模式转换到权限较高的模式，所以中断时会自动切换到内核态，堆栈不会再弹出另外的两位，所以当键盘输入 0 准备从用户模式切换到内核模式，实现中断返回时，原来的 **esp** 还在堆栈中，所以需要把 **ebp** 的值传送给 **esp**，恢复栈指针，修复 **esp**。

代码实现如下。

```
/* case IRQ_OFFSET + IRQ_KBD:
   c = cons_getc();
   cprintf("kbd [%03d] %c\n", c, c);
   break;*/

//Challenge 2:用此段替换上面一段即可
case IRQ_OFFSET + IRQ_KBD:
c = cons_getc();
cprintf("kbd [%03d] %c\n", c, c);
if(c=='0') //切换到内核态
{
    if (tf->tf_cs != KERNEL_CS) {
        cprintf("try to be kernel mode\n");
        tf->tf_cs = KERNEL_CS;
        tf->tf_ds = KERNEL_DS;
        tf->tf_es = KERNEL_DS;
        print_trapframe(tf);
    }
    else{
        cprintf("already in kernel mode\n");
    }
}
else if(c=='3') //切换到用户态
{
    if (tf->tf_cs != USER_CS) {
        cprintf("try to be user mode\n");
        tf->tf_cs = USER_CS;
        tf->tf_ds = USER_DS;
        tf->tf_es = USER_DS;
        tf->tf_ss = USER_DS;
        tf->tf_eflags |= 0x3000;
        print_trapframe(tf);
    }
    else{
        cprintf("already in user mode\n");
    }
}
break;
```

下面进行验证。

在内核态下输入“3”。可以发现程序执行尝试返回用户态。

参考答案对比

练习 1:

在实验报告中采取的思路是从生成 `ucore.img` 的 `makefile` 命令倒推实现过程,逐步展示,条理和层次不像参考答案中那么清晰;分别从 `makefile` 依赖的两个关键的文件 `kernal` 与 `bootblock` 分析了生成文件的指令部分,并对 `dd` 指令与参数部分进行了更加详细一些的分析。

练习 2:

本题与参考答案的实现区别不是很大,从以下四个角度进行分析。从 CPU 加电后执行的第一条指令开始,单步跟踪 BIOS 的执行。在初始化位置 `0x7c00` 设置实地址断点,测试断点正常。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。自己找一个 `bootloader` 或内核中的代码位置,设置断点并进行测试。

练习 3:

介绍思路和顺序和答案基本一致,但是补充了一些对应的原理知识点,并对各条指令的内涵做出了自己能基本理解的详细解释。如为什么要开启 `A20` 以及如何开启 `A20` 的部分。再如调用 `bootmain` 函数的部分,都有较为详细的说明。

练习 4:

结合实验指导书提供的资料、具体的代码注释信息,对每一部分的原理和实现机制做出了比参考答案详细得多的说明。如 `bootloader` 如何读取硬盘扇区以及 ELF 格式的 OS,再如 `bootmain.c` 的代码的解读和其中的多个函数 (`waitdisk`、`readsect`、`readseg`、`bootmain` 等)的实现都有更为详细的说明。

练习 5:

实现机制和参考答案思路基本一致。此外,对栈机制的使用做出了自己的解释和详细说明。特别是对于 `c` 指针加法的说明以及传递变量地址的说明。

练习 6:

进行中断初始化的方法和参考答案基本一致,主要增加了自己对相应中断原理的理解并做出注释。对于编程完善的部分也有较为独到的见解。

扩展练习 1

基本和参考答案一致,通过设置 `tf` 信息来完成状态切换,同时用自学的相关知识进行了理解和剖析,参考了学堂在线教学视频的指导。在补全代码的部分有较为清晰的说明。

扩展练习 2

参考答案未提供参考,依靠自学交流以及 CSDN 学习完成。由于较为简单,没有过多的说明与解释。

重要知识点和对应原理

1、实验中的重要知识点

- `makefile` 书写格式和实现方法
- GCC 基本内联汇编语法
- linux gdb 调试基础
- BIOS 启动过程
- `bootloader` 启动过程
- 关于保护模式的硬件实现
- 分段机制的实现原理 (含特权级)

- 硬盘访问实现机制
- ELF文件格式概述
- 操作系统基本启动方式
- 函数堆栈调用的内部原理
- 中断机制的实现（含特权级）

2、对应的 OS 原理知识点

- 虚拟内存的实现
- 分段保护机制
- 过程调用与中断实现
- 内存访问机制

3、二者关系：

- 本实验设计的知识是对 OS 原理的具体实现，在细节上很复杂。

4、未对应的知识点

- 进程调度管理
- 分页管理与页调度
- 并发实现机制