

ucore Lab1

21307265 童齐嘉

实验要求

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

练习1

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）

- 创建出ucore.img的makefile代码为：

```
186 # create ucore.img
187 UCOREIMG      := $(call totarget,ucore.img)
188
189 $(UCOREIMG): $(kernel) $(bootblock)
190     $(V)dd if=/dev/zero of=$@ count=10000
191     $(V)dd if=$(bootblock) of=$@ conv=notrunc
192     $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
193
194 $(call create_target,ucore.img)
```

为了看懂这个代码，我们需要先回忆makefile的规则：

```
target ... : prerequisites ...
    command
    ...
```

prerequisites是需要生成target所需的文件或者目标，command则是所需要执行的命令。

这段代码告诉我们，想要创建出ucore.img，就要先生成kernel和bootblock.

生成kernel的makefile代码为：

```

148 # create kernel target
149 kernel = $(call totarget,kernel)
150
151 $(kernel): tools/kernel.ld
152
153 $(kernel): $(KOBJS)
154     @echo + ld $@
155     $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
156     @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
157     @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call symfile,kernel)
158
159 $(call create_target,kernel)

```

代码解释如下:

- 第149行, 定义了一个变量kernel, 它的值是使用totarget函数生成的kernel文件名。
- 第151行, 指定了kernel的依赖文件, 即tools/kernel.ld, 它是一个链接脚本文件, 用于指定内核镜像的布局 and 地址1。
- 第153行, 指定了kernel的依赖文件, 即\$(KOBJS), 它是一个变量, 表示所有内核相关的.o文件。
- 第155行, 使用ld链接器, 将所有.o文件链接成一个可执行文件。链接时使用-T选项, 表示指定链接脚本为tools/kernel.ld。
- 第156行, 使用objdump工具, 将可执行文件反汇编成汇编代码, 并输出到一个.asm文件中。
- 第157行, 使用objdump工具, 将可执行文件的符号表输出到一个.sym文件中, 并使用sed工具进行一些格式化处理。
- 第159行, 使用create_target函数, 创建一个伪目标kernel

我们先看看生成kernel需要哪些文件

先输入命令 `make clean`, 再输入命令 `make "V="`, 查看makefile执行了哪些命令, 在其中找到 `ld bin/kernel`

```

+ 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/de
bug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o obj/kern/driver/con
sole.o obj/kern/driver/picirq.o obj/kern/driver/intr.o obj/kern/trap/trap.o obj/ke
rn/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmm.o obj/libs/string.o o
bj/libs/printfmt.o

```

```

+ 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

```

对于 `ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel`, 这条命令中的参数的含义如下:

- `-m elf_i386` 表示指定输出文件的格式为ELF 32位。
- `-nostdlib` 表示不使用标准库和启动文件。
- `-T tools/kernel.ld` 表示使用 `tools/kernel.ld` 文件作为链接器脚本。
- `-o bin/kernel` 表示指定输出文件的名称为 `bin/kernel`。

命令结果：将输入文件链接成一个ELF 32位的可执行文件`bin/kernel`，并使用`tools/kernel.ld`文件中的指令来控制链接过程。使得可以通过该文件启动一个操作系统的内核

下面将总结上述所有相关命令中命令参数的含义

- `cc xxx` 表示编译xxx文件
- `gcc -xxx` 表示使用GCC编译器，并在xxx目录下查找头文件。
- `-fno-builtin` 表示不使用GCC内置的函数。(内核当然不能用标准库)
- `-Wall` 表示开启所有的警告信息。
- `-ggdb` 表示生成GDB调试信息。
- `-m32` 表示生成32位的目标代码。
- `-gstabs` 表示生成stabs (symbol table strings) 格式的调试信息。
- `-nostdinc` 表示不使用标准的头文件目录。
- `-fno-stack-protector` 表示不使用栈保护机制。
- `-Ilibs/ -xxx -yyy -zzz` 表示在xxx,yyy,zzz这些目录中查找头文件
- `-c` 表示只编译不链接
- `-o xxx` 表示将输出文件命名为xxx 上述大多命令都是在避免gcc调用标准库/标准头文件等等，保证内核能在内核空间正常实现，另一些命令，例如 `-Ilibs/` 则是在调用自己的文件，为内核的运行提供服务

生成bootblock的makefile代码为：

```
163 # create bootblock
164 bootfiles = $(call listf_cc,boot)
165 $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))
166
167 bootblock = $(call totarget,bootblock)
168
169 $(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
170     @echo + ld $@
171     $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
172     @$ (OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
173     @$ (OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
174     @$ (call totarget,sign) $(call outfile,bootblock) $(bootblock)
175
176 $(call create_target,bootblock)
```

代码解释如下：

- 第164行，定义了一个变量`bootfiles`，它的值是使用`listf_cc`函数列出的所有`boot`目录下的.c文件。
- 第165行，对每个`bootfiles`中的文件，使用`cc_compile`函数调用gcc编译器，生成.o文件。编译时使用`-Os`选项，表示优化代码大小，使用`-nostdinc`选项，表示不使用标准头文件。
- 第167行，定义了一个目标`bootblock`，它的值是使用`totarget`函数生成的`bootblock`文件名。
- 第169行，指定了`bootblock`的依赖文件，即所有`bootfiles`中的.o文件。同时，在这一行末尾使用`|`符号，表示还有一个隐含依赖文件`sign`，它是一个用于给启动块添加签名的程序。
- 第170行，输出一条信息，表示正在链接`bootblock`。
- 第171行，使用`ld`链接器，将所有.o文件链接成一个可执行文件。链接时使用`-N`选项，表示不对代码段和数据段进行页对齐，使用`-e`选项，表示指定入口地址为`start`标签处，使用`-Ttext`选项，表示指定代码段的起始地址为`0x7C00`。
- 第172行，使用`objdump`工具，将可执行文件反汇编成汇编代码，并输出到一个.asm文件中。

- 第173行, 使用objcopy工具, 将可执行文件转换成二进制格式, 并输出到一个.out文件中。
- 第174行, 调用sign程序, 给二进制文件添加一个签名, 并将结果写入到bootblock文件中。
- 第176行, 使用create_target函数, 创建一个伪目标bootblock。

生成代码:

```
+ 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
```

```
+ 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
```

可以看到 bin/bootblock由bootasm和bootmain生成。

但事实上, 我们知道它还调用了sign程序 174 @\$ (call totarget,sign) \$(call outfile,bootblock) \$(bootblock)

- `ld xxx`: 把链接后的输出文件命名为xxx
- `-m elf_i386`: 指定输出文件的格式为ELF 32-bit格式
- `-N`: 设置所有节的读写属性为可读也可写
- `-e start`: 设置程序的入口点为start符号
- `-Ttext 0x7C00`: 设置代码段的起始地址为0x7C00
- `-o xxx`: 把链接后的输出对象文件命名为xxx.

Ps: 输出文件 与 输出对象文件 的区别:

链接后的输出文件是一个可执行的文件, 它包含了所有的机器码、符号表和重定位信息, 可以直接运行。

链接后的输出对象文件是一个中间的文件, 它包含了部分的机器码、符号表和重定位信息, 但还需要和其他的对象文件或库文件一起链接才能生成可执行的文件。

通常, 链接后的输出文件没有扩展名, 或者是.exe或.out, 而链接后的输出对象文件通常是.o。

这是bootasm和bootmain的生成代码:

```
+ cc boot/bootasm.S
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protect
or -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-protect
or -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
```

sign

```
180 # create 'sign' tools
181 $(call add_files_host,tools/sign.c,sign,sign)
182 $(call create_target_host,sign,sign)
```

在makefile里的代码为:

sign的生成代码

```
+ cc tools/sign.c
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
为: gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
```

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?

- 它位于硬盘的第一个扇区, 即柱面0, 磁头0, 扇区1的位置。
- 它占用512字节的空间, 其中包含了主引导程序, 分区表和结束标志。
- 它的最后两个字节是55AAH, 表示它是一个有效的主引导扇区。
- 它的主引导程序能够读取分区表, 找到活动分区, 并将控制权转交给活动分区的引导扇区。

练习2

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。

按照提示，将tools/gdbinit内的内容设置为：

```
target remote :1234
set architecture i8086
```

修改 debug代码如下：

```
219 debug: $(UCOREIMG)
220     $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel
    l stdio -hda $< -serial null"
221     $(V)sleep 2
222     $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
223
```

这样

就可以将调试信息存入q.log中

先使用 `make clean` 确保不受干扰，再输入 `make debug` 进入调试程序，此时输入 `si` 或者 `next` 即可单步调试

2. 在初始化位置0x7c00设置实地址断点,测试断点正常。

```
(gdb) b* 0x7c00
Note: breakpoint 1 also set at pc 0x7c00.
Breakpoint 2 at 0x7c00
```

设置一个断点

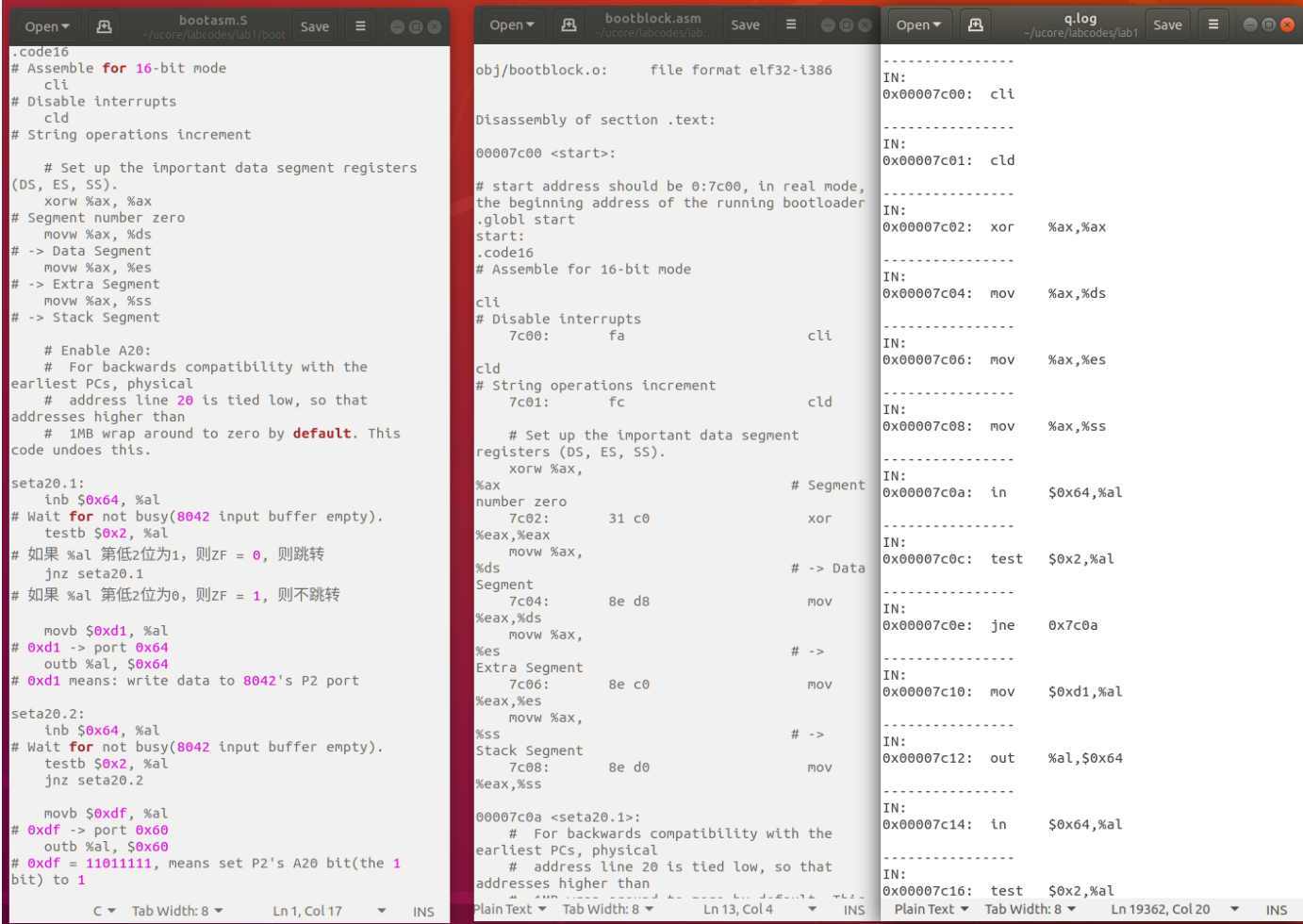
输入 `x /7i $pc`

```
=> 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
(gdb) █
```

断点正常

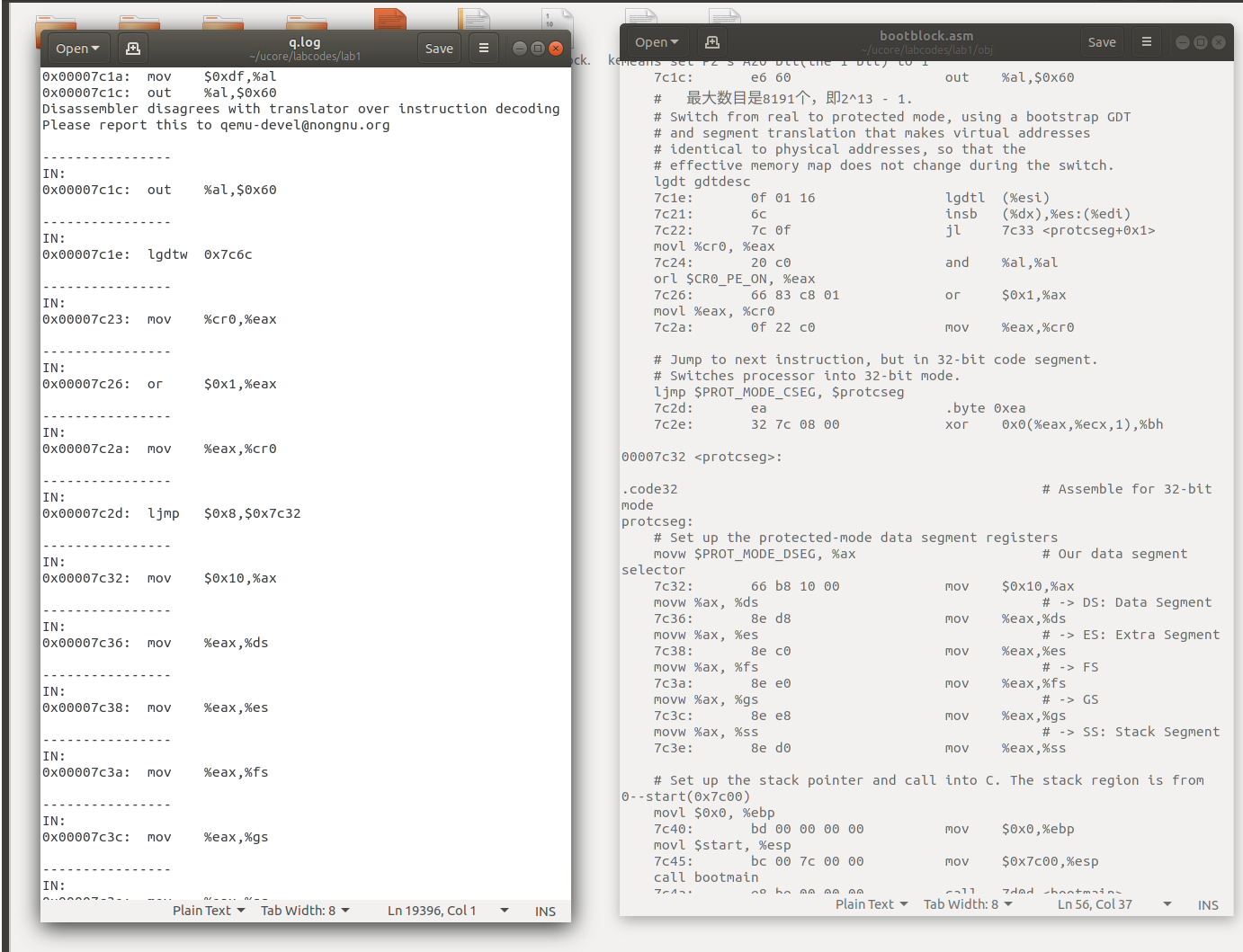
3. 从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和bootblock.asm进行比较。

执行一次后打开bootasm.S和bootblock.asm，发现从0x00007c00后，两边的汇编代码完全一致



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。

设置一个断点 `b* 0x7c1c` 执行完毕后，经对比确认，代码一致



练习3

分析bootloader进入保护模式的过程。（要求在报告中写出分析）

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：

1. 为何开启A20，以及如何开启A20

答：

A20是指地址总线上的第21根线，它可以访问超过1MB的内存空间。

在8086处理器中，由于地址总线只有20根，所以当地址超过1MB时，会发生回绕，即访问0x00000和0x100000的结果会一样。

这样会导致一些程序出现错误。为了保持兼容性，当IBM设计PC AT机器时，使用了80286处理器，它可以访问16MB的内存空间。但是在实模式下，它必须模拟8086的行为，所以它不能直接访问超过1MB的内存空间。为了解决这个问题，IBM在主板上设计了一个开关，叫做A20门，它可以控制A20线的状态。当A20门关闭时，A20线恒为0，实现了回绕效果。当A20门打开时，A20线可以正常工作，实现了扩展内存访问。

如果A20 Gate被打开，则当程序员给出100000H-10FFEFH之间的地址的时候，系统将真正访问这块内存区域；如果A20Gate被禁止，则当程序员给出100000H-10FFEFH之间的地址的时候，系统仍然使用8086/8088的方式。

bootasm.S中采用了一种名为键盘控制器的打开A20门的方法

```
seta20.1:
    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al              # 如果 %al 第低2位为1, 则ZF = 0, 则跳转
    jnz seta20.1                 # 如果 %al 第低2位为0, 则ZF = 1, 则不跳转

    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              # 如果 %al 第低2位为1, 则ZF = 0, 则跳转
    jnz seta20.2                 # 如果 %al 第低2位为0, 则ZF = 1, 则不跳转

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

下面是一些解释：

- 键盘控制器是一个芯片，它有两个端口：\$0x60和\$0x64, \$0x60是数据端口，用于读写键盘或鼠标的数据。\$0x64是命令端口，用于发送命令给键盘控制器或读取键盘控制器的状态。
- inb和outb是输入输出指令，用于从端口读取或写入一个字节。
- \$0x2是一个位数，表示键盘控制器的状态寄存器的第二位，用于检测是否忙碌。
- \$0xd1是一个命令字，表示要写数据到键盘控制器的输出端口。
- \$0xdf是一个数据字，表示要将输出端口的A20位设置为1，从而打开A20门。

所以，打开A20的方法为：

等待端口\$0x64为空，通过发送\$0xd1来告知键盘控制器将要写入数据。

接下来通过inb读取\$0x64的状态，检查键盘的输入缓冲区是否为空，再将\$0xdf写入数据端口\$0x60，这样就成功打开了A20

2. 如何初始化GDT表

使用代码： `lgdt gdt_desc`

这行代码的作用是将全局描述符表（GDT）的地址和大小加载到GDTR寄存器中。

3. 如何使能和进入保护模式

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

CR0寄存器的第0位（PE位）是保护模式使能位，我们将其置1，即告知CPU开启了保护模式

此后，我们需要通过长跳转指令（ljmp）跳转到一个代码段选择子和一个偏移量，从而刷新CS寄存器和EIP寄存器，开始执行保护模式下的代码。

进入保护模式的完整流程

1. 关闭终端，清理好数据寄存器
2. 开启A20
3. 初始化GDT表
4. 将cr0寄存器PE置1，开启保护模式
5. 通过长跳转更新cs的基地址

6. 设置段寄存器，建好堆栈
7. 转入完成，通过 `call bootmain` 进入保护模式

练习4:

分析bootloader加载ELF格式的OS的过程。（要求在报告中写出分析）

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，提示：可阅读“硬盘访问概述”，“ELF执行文件格式概述”这两小节。

1. bootloader如何读取硬盘扇区的？

对于readsect():

```
/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // 等待磁盘准备好
    waitdisk();
    // 发出读取扇区的命令
    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

    // 等待磁盘准备好
    waitdisk();

    // 把磁盘扇区数据读到指定内存
    insl(0x1F0, dst, SECTSIZE / 4);
}
```

对于readreg():

```
/* *
 * 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) {
    // 计算出要读取的数据的结束地址end_va，等于虚拟地址加上字节数。
    uintptr_t end_va = va + count;

    // 将虚拟地址向下取整到扇区边界，扇区大小为SECTSIZE
    va -= offset % SECTSIZE;

    // 将偏移量转换为扇区号secno，加上1是因为内核从第一个扇区开始。
    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.
    // 循环直到虚拟地址达到或超过结束地址为止。
    for (; va < end_va; va += SECTSIZE, secno++) {
        readsect((void *)va, secno);
    }
}
```

该程序功能为：从内核中读取一段数据到虚拟地址空间。

2. bootloader是如何加载ELF格式的OS?

```
/* 从ELF文件中加载OS到内存中 */
void
bootmain(void) {
    // 从ELF头部读取
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 检查是否是有效的ELF文件
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // 找到ELF有关内存位置的描述表
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++) {
        // 从文件中读取段到内存中
        readseg(ph->p_va & 0xFFFFF, ph->p_memsz, ph->p_offset);
    }

    // 跳转到入口地址执行OS代码
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFF))();

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

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

1. 读取ELF头部，检查是否是有效的ELF文件，是否是可执行文件，是否是支持的机器类型等。
2. 找到ELF有关内存位置的描述表，并按这个描述表将数据载入内存
3. 读取ELF头部中指定的入口地址，跳转到该地址开始执行OS代码

练习5

实现函数调用堆栈跟踪函数

(1) 调用函数，获取ebp和eip的值

```
uint32_t ebp = read_ebp();
uint32_t eip = read_eip();
```

(2) 执行循环，从0到STACKFRAME_DEPTH。

但由于内核边界为ebp = 0,所以需要加入限制条件 `ebp != 0`

```
for (int i = 0; i < STACKFRAME_DEPTH && ebp != 0; i++){
    // other code
}
```

(2.1) 输出ebp 与 eip的十六进制值

```
cprintf("ebp:0x%08x eip:0x%08x", ebp, eip);
```

cprintf可以简单理解为内核空间的printf函数，printf函数只能服务于用户态，不能运用于此

(2.2) 使ebp的值加2，获取其地址到一个变量中

也即使ebp位移8位

```
uint32_t *arg = (uint32_t *)(ebp + 8);
```

(2.3) 打印地址的值

```
cprintf("arg:");  
for(int j = 0; j < 4; j++){  
    cprintf("0x%08x ", arg[j]);  
}  
cprintf("\n");
```

(2.4) 调用print_debuginfo(eip - 1)打印C调用的函数名称和行号

```
print_debuginfo(eip - 1);
```

(2.5) 弹出调用堆栈帧

```
eip = ((uint32_t *)ebp)[1];  
ebp = ((uint32_t *)ebp)[0];
```

NOTICE:

- the calling function's return addr eip = ss:[ebp+4]
- the calling function's ebp = ss:[ebp]

最终函数实现为:

```
void  
print_stackframe(void) {  
    uint32_t ebp = read_ebp();  
    uint32_t eip = read_eip();  
    for (int i = 0; i < STACKFRAME_DEPTH && ebp != 0; i++){  
        cprintf("ebp:0x%08x eip:0x%08x", ebp, eip);  
        uint32_t *arg = (uint32_t *)(ebp + 8);  
        cprintf("arg:");  
        for(int j = 0; j < 4; j++){  
            cprintf("0x%08x ", arg[j]);  
        }  
        cprintf("\n");  
        print_debuginfo(eip - 1);  
        eip = ((uint32_t *)ebp)[1];  
        ebp = ((uint32_t *)ebp)[0];  
    }  
}
```

运行截图:

```
Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x001032b3 (phys)
  edata  0x0010ea16 (phys)
  end    0x0010fd20 (phys)
KeHelp: executable memory footprint: 64KB
ebp:0x00007b28 eip:0x00100a63arg:0x00010094 0x00010094 0x00007b58 0x00100092
  kern/debug/kdebug.c:294: print_stackframe+21
ebp:0x00007b38 eip:0x00100d59arg:0x00000000 0x00000000 0x00000000 0x00007ba8
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100092arg:0x00000000 0x00007b80 0xffff0000 0x00007b84
  kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bcarg:0x00000000 0xffff0000 0x00007ba4 0x00000029
  kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000dbarg:0x00000000 0x00100000 0xffff0000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100101arg:0x001032dc 0x001032c0 0x0000130a 0x00000000
  kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100055arg:0x00000000 0x00000000 0x00000000 0x00007c4f
  kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d72arg:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknow>: -- 0x00007d71 --
```

练习6

完善中断初始化和处理

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？

表项的实现代码如下：

```
/* 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
};
```

相加可知，一共占用了64bits，也即8字节。其中，前2个字节和后两个字节，也即0，1，6，7字节拼接后形成了中断处理代码的入口的地址

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。

我们需要先对 #define SETGATE(gate, istrap, sel, off, dpl) 有一定的了解

- gate: 一个指向中断描述符表项的指针；
- istrap: 一个标志位，表示这个表项是一个陷阱门[1] (trap gate) 还是一个中断门[0] (interrupt gate) ；
- sel: 一个段选择子 (segment selector)，表示中断处理程序所在的代码段；一般为GD_KTEXT
- off: 一个偏移量 (offset)，表示中断处理程序在代码段中的地址；

- dpl: 一个特权级 (descriptor privilege level) , 表示允许访问这个表项的最低特权级; 在内核空间中为 DPL_KERNEL, 在用户空间中为 DPL_USER

(1) 声明中断入口

```
extern uintptr_t __vectors[];
```

(2) 遍历设置每一个表项

```
for (int i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++){
    SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
}
```

(3) 非硬件中断需要和用户态沟通, 还需要设置用户态权限的接口

```
SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
```

(4) 使用lidt指令, 告知CPU现在的IDT在哪里

```
lidt(&idt_pd);
```

最终代码实现为:

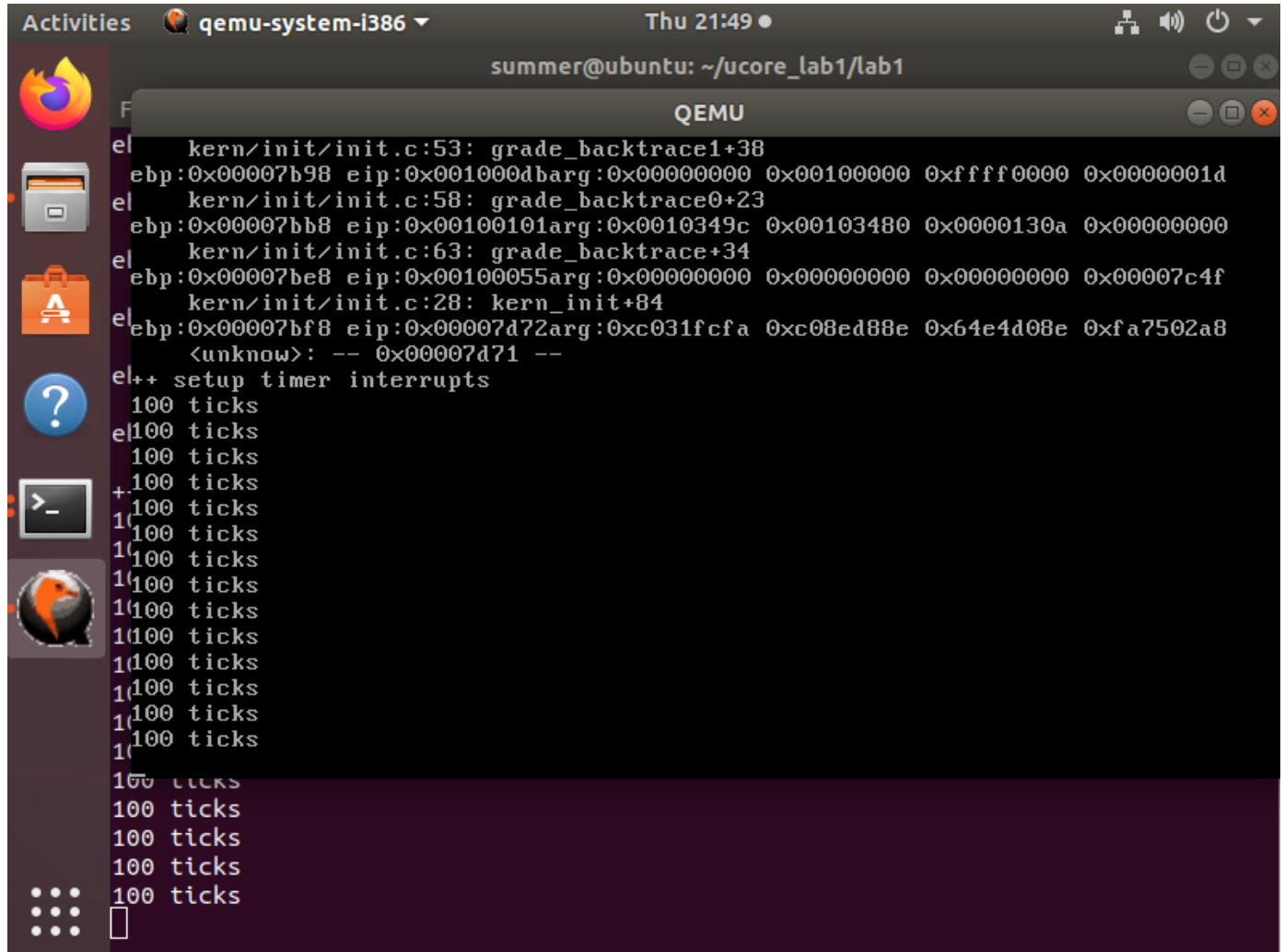
```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    for (int i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++){
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    lidt(&idt_pd);
}
```

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" 。

在相应位置写入:

```
ticks++;
if (ticks % TICK_NUM == 0) {
    print_ticks();
}
```

运行截图：



扩展练习Challenge 1

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。

先根据指引，取消掉 `lab1_switch_test();` 的注释

我们在kern/init中找到了这样一个函数

```
static void
lab1_switch_test(void) {
    lab1_print_cur_status(); // print 当前 cs/ss/ds 等寄存器状态
    cprintf("+++ switch to user mode +++\n");
    lab1_switch_to_user();   // switch to user mode
    lab1_print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    lab1_switch_to_kernel(); // switch to kernel mode
    lab1_print_cur_status();
}
```

而我们需要实现的是：

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



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

在kern/trap中，我们找到了：

```
//LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
case T_SWITCH_TOU:
case T_SWITCH_TOK:
```

我们需要实现这两个中断触发时要做的事情

但到这里，我们还一头雾水，这时候就需要去了解一下两个形态转换时相关的知识了：

- 中断的处理过程

trap函数（定义在trap.c中）是对中断进行处理的过程，所有的中断在经过中断入口函数_alltraps预处理后（定义在trapasm.S中），都会跳转到这里。在处理过程中，根据不同的中断类型，进行相应的处理。在相应的处理过程结束以后，trap将会返回，被中断的程序会继续运行。整个中断处理流程大致如下：

1. 中断产生后，CPU跳转到相应的中断处理入口vectors，此时，如果**特权级发生变化**，就必须将当前的ss(存放栈段的段选择子)和esp(存放栈顶的偏移地址)压栈，保证能够正常返回后，按照预定顺序压入数据
2. 在栈中保存当前被打断程序的 trapframe结构(参见过trapasm.S，结构在trap.h)。设置 kernel的数据段寄存器，最后压入 esp，作为 trap 函数参数 (`struct trapframe* tf`) 并跳转到中断处理函数 trap 处：
3. 结束 trap 函数的执行后，通过 ret 指令返回到 alltraps 执行过程。从栈中恢复所有寄存器的值。调整 esp 的值：跳过栈中的 trap_no 与error_code，使esp指向中断返回 eip，通过 iret 调用恢复 cs、eflag以及 eip，继续执行。

但需要特殊注意的是：

- 从内核态到用户态时，属于高特权级向低特权级转化，此时不会自动切换栈，也就没有保存ss和esp

因此，从内核态到用户态的过程，我们需要手动实现压入ss和esp的操作，相反的，在从用户态到内核态的过程，我们不需要手动实现这个过程

现在，我们先实现 `lab1_switch_to_kernel(void)`：

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

同理，实现 `lab1_switch_to_user(void)`：

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

接下来移步kern/trap.h，我们需要对两种情况中中断帧的数据进行修改，我们先阅读中断帧的结构。

```
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_gs;
    uint16_t tf_padding0;
    uint16_t tf_fs;
    uint16_t tf_padding1;
    uint16_t tf_es;
    uint16_t tf_padding2;
    uint16_t tf_ds;
    uint16_t tf_padding3;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

其每一项的解释如下：

- tf_regs: 一个结构体，用来保存通用寄存器的值，如eax、ebx、ecx、edx等。
- tf_gs、tf_fs、tf_es、tf_ds: 分别用来保存段寄存器gs、fs、es、ds的值。
- tf_padding0~5: 分别用来填充对应的段寄存器，使其占用4个字节，方便对齐。
- tf_trapno: 用来保存中断号，即引发中断的类型。
- tf_err: 用来保存错误码，即中断发生时的一些附加信息，如发生在哪个段上等。
- tf_eip: 用来保存指令指针寄存器eip的值，即中断发生时的下一条指令地址。
- tf_cs: 用来保存代码段寄存器cs的值，即中断发生时的代码段选择子。**也就是用于存放当前正在运行的程序代码所在的段的段基址，其最低两位为访问该段时请求的特权级。**特权级0代表内核级，3代表用户级。
- tf_eflags: 用来保存标志寄存器eflags的值，即中断发生时的状态标志位。但在这里，我们用其来设置输出优先级
 - tf->tf_eflags |= FL_IOPL_MASK;这条语句的作用是将tf指针所指向的结构体中的tf_eflags成员的IOPL字段设置为3，让用户态也能直接访问IO端口
 - tf->tf_eflags &= ~FL_IOPL_MASK;这条语句的作用是将tf指针所指向的结构体中的tf_eflags成员的IOPL字段设置为0，即最低的I/O特权级，从而起到
- tf_esp: 用来保存栈指针寄存器esp的值，即中断发生时的栈顶地址。
- tf_ss: 用来保存栈段寄存器ss的值，即中断发生时的栈段选择子。

我们先完成 `case T_SWITCH_TOK`: (用户->内核):

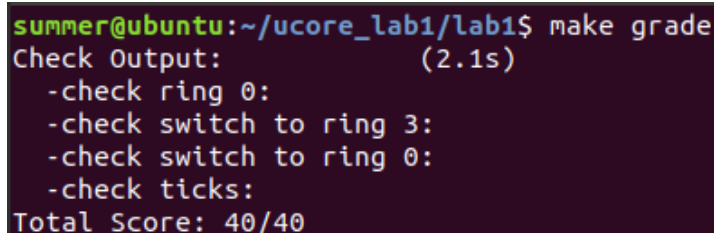
```
case T_SWITCH_TOK:
    if (tf -> tf_cs != KERNEL_CS){
        tf -> tf_cs = KERNEL_CS;
        tf -> tf_ss = tf -> tf_ds = tf -> tf_es = tf -> tf_gs = tf -> tf_fs = KERNEL_DS;
        tf->tf_eflags &= ~FL_IOPL_MASK;
    }
    break;
```

同理，再完成 `case T_SWITCH_TOU`: (内核->用户):

```
case T_SWITCH_TOU:
    if( tf -> tf_cs != USER_CS){
        tf -> tf_cs = USER_CS;
        tf -> tf_ss = tf -> tf_ds = tf -> tf_es = tf -> tf_gs = tf -> tf_fs = USER_DS;

        tf->tf_eflags |= FL_IOPL_MASK;
    }
    break;
```

实验结果截图:



```
summer@ubuntu:~/ucore_lab1/lab1$ make grade
Check Output: (2.1s)
-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Challenge 2

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。

我们在trap.c中的trap_dispatch(struct trapframe *tf)函数找到了 `case IRQ_OFFSET + IRQ_KBD`: 此处已经写好了一个接收 `c = cons_getc()`; 我们对其进行分类转换即可。

当输入值为0时:

```
case '0':
    t_switch_tok(tf);
    print_trapframe(tf);
    break;
```

这里的 `t_switch_tok(tf)`: 的功能其实跟case:T-SWITCH_TOK的功能一样，我们照搬过来即可

```
static void t_switch_tok(struct trapframe *tf) {
    if (tf -> tf_cs != KERNEL_CS){
        tf -> tf_cs = KERNEL_CS;
        tf -> tf_ss = tf -> tf_ds = tf -> tf_es = tf -> tf_gs = tf -> tf_fs = KERNEL_DS;
        tf->tf_eflags &= ~FL_IOPL_MASK;
    }
}
```

当输入值为3时，同理:

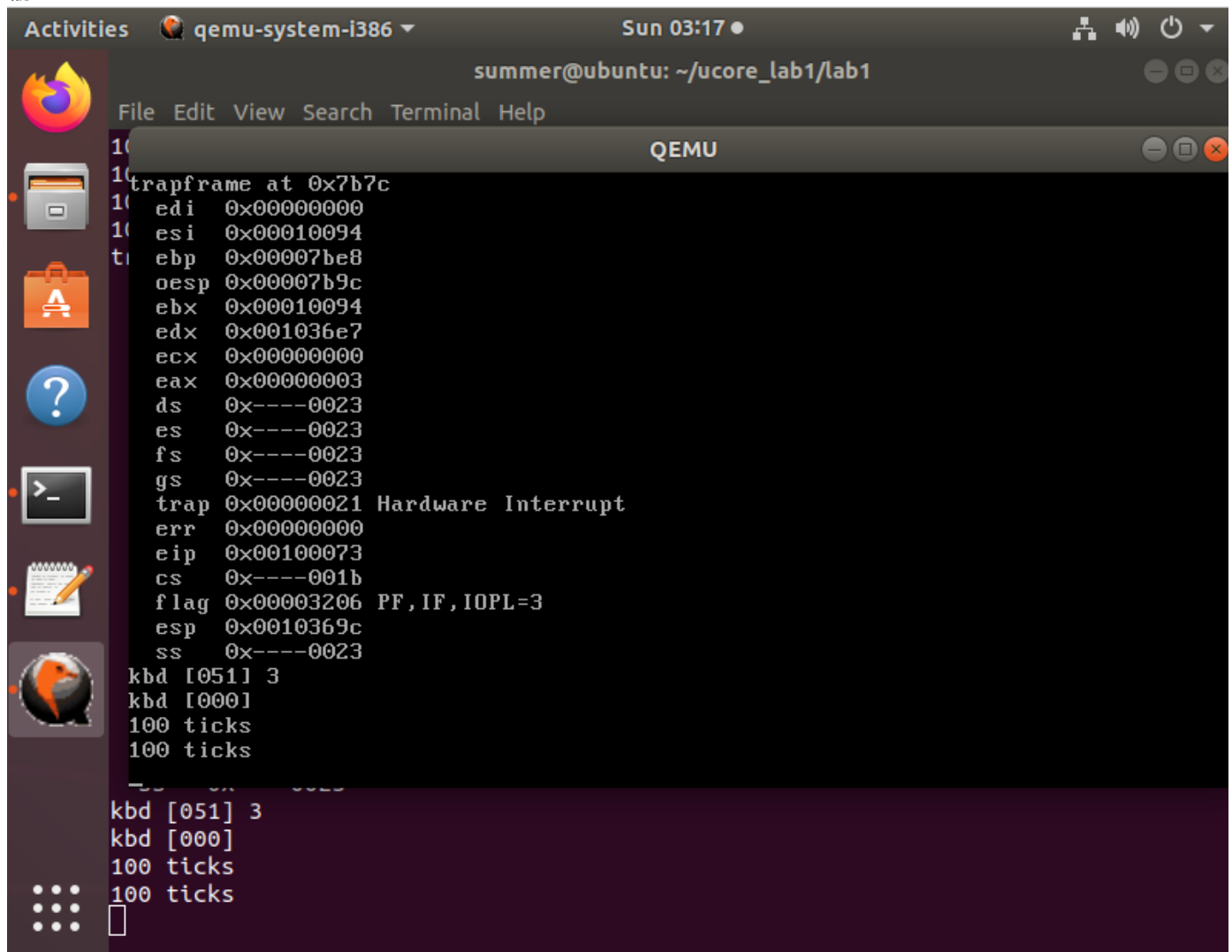
```
case '3':
    t_switch_tou(tf);
    print_trapframe(tf);
    break;
```

```
static void t_switch_tou(struct trapframe *tf) {
    if( tf -> tf_cs != USER_CS){
        tf -> tf_cs = USER_CS;
        tf -> tf_ss = tf -> tf_ds = tf -> tf_es = tf -> tf_gs = tf -> tf_fs = USER_DS;

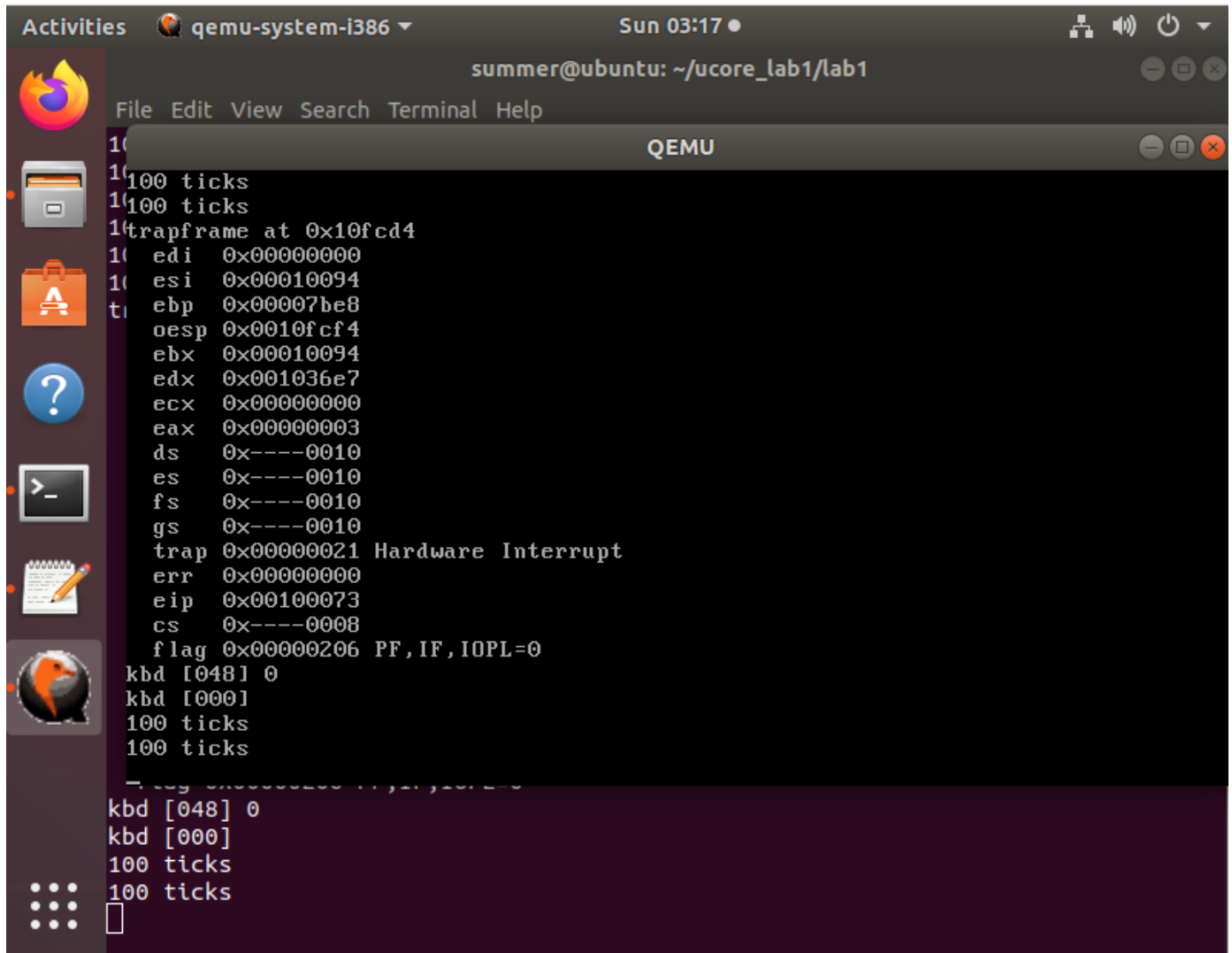
        tf->tf_eflags |= FL_IOPL_MASK;
    }
}
```

实验结果截图：

输入3:



输入0:



The screenshot shows a QEMU terminal window titled "QEMU" running on a system named "qemu-system-i386". The terminal output displays a hardware interrupt (trap) at address 0x10fcd4. It lists various registers and their values, including edi, esi, ebp, oesp, ebx, edx, ecx, eax, ds, es, fs, gs, trap, err, eip, cs, and flag. The flag register shows PF, IF, and IOPL bits. The output also includes keyboard input events (kbd [048] 0 and kbd [000]) and timing information (100 ticks).

```
100 ticks
100 ticks
trapframe at 0x10fcd4
edi 0x00000000
esi 0x00010094
ebp 0x00007be8
oesp 0x0010fcf4
ebx 0x00010094
edx 0x001036e7
ecx 0x00000000
eax 0x00000003
ds 0x----0010
es 0x----0010
fs 0x----0010
gs 0x----0010
trap 0x00000021 Hardware Interrupt
err 0x00000000
eip 0x00100073
cs 0x----0008
flag 0x00000206 PF, IF, IOPL=0
kbd [048] 0
kbd [000]
100 ticks
100 ticks
- flag 0x00000206 PF, IF, IOPL=0
kbd [048] 0
kbd [000]
100 ticks
100 ticks
█
```

再输入3:

```

100 ticks
trapframe at 0x10fcd4
edi 0x00000000
esi 0x00010094
ebp 0x00007be8
oesp 0x0010fcf4
ebx 0x00010094
edx 0x001036e7
ecx 0x00000000
eax 0x00000003
ds 0x----0023
es 0x----0023
fs 0x----0023
gs 0x----0023
trap 0x00000021 Hardware Interrupt
err 0x00000000
eip 0x00100073
cs 0x----001b
flag 0x00003206 PF, IF, IOPL=3
esp 0x0010369c
ss 0x----0023
kbd [051] 3
kbd [000]
100 ticks
ss 0x----0023
kbd [051] 3
kbd [000]
100 ticks

```

可以从flag的值看到模式的变化是符合预期的

过程中的问题附录

1. 实验内容的下载

用参考手册中给的链接克隆文件夹后没找到实验用的文件。

之后通过 <https://github.com/kiukotsu/ucore.git> 下载，结果文件可能有一些未知问题，在运行 `make qemu` 时会进入死循环，无法到达中断测试点。

最终，通过南开大学操作系统课程的仓库 https://github.com/suhipek/nku_os 下载了正确的文件，也就完成了实验。

2. 如何修改makefile文件？

按下ins, 进入修改模式，修改完后按下Esc退出，再输入：wq保存退出即可

3. 在qemu中输入字符

问题背景：

我们运行命令 `make qemu` 后要输入一个0/3来改变状态

经过测试，这个值不能使用九宫格内的数字。

经过查询：

数字键盘和数字行是两组不同的按键，它们有不同的扫描码和功能。在linux系统的qemu中，数字键盘的扫描码可能会被映射到其他功能上，比如方向键或者鼠标控制，而不是输入数字。

除官方文档外的参考资料：

lab_0 清华大学ucore实验环境配置详细步骤！（小白入）

<https://www.cnblogs.com/huilinmumu/p/16211900.html>

A20地址线： <https://blog.csdn.net/ruyanhai/article/details/7181842>

操作系统实验Ucore lab1： https://blog.csdn.net/qq_20549085/article/details/100128521

《Intel® 64 and IA-32 Architectures Software Developer' s Manual》第6.14.1节。

操作系统实验报告1： ucore Lab 1： <https://blog.csdn.net/StuGeek/article/details/118708567>

南开大学操作系统实验仓库 https://github.com/suhipek/nku_os

uCore lab1 操作系统实验 challenge https://blog.csdn.net/weixin_44677382/article/details/108646416

ucoreOS_lab1实验报告 <https://www.cnblogs.com/ECJTUACM-873284962/p/11178427.html>