目录

[作业一． 1](#_Toc68266610)

[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) 1](#_Toc68266611)

[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？ 4](#_Toc68266612)

[作业二 6](#_Toc68266613)

[1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行 6](#_Toc68266614)

[2. 在初始化位置0x7c00设置实地址断点,测试断点正常。 7](#_Toc68266615)

[通过杀掉进程的方法来关闭端口 8](#_Toc68266616)

[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。 8](#_Toc68266617)

[练习三：分析bootloader进入保护模式的过程。 11](#_Toc68266618)

[练习4——分析bootloader加载ELF格式的OS的过程 17](#_Toc68266619)

[如何加载ELF格式的OS： 21](#_Toc68266620)

[[练习5]：实现函数调用堆栈跟踪函数 21](#_Toc68266621)

[练习6]完善中断初始化和处理 23](#_Toc68266622)

[[6.1] 23](#_Toc68266623)

[[6.2] 25](#_Toc68266624)

[[6.3] 27](#_Toc68266625)

[附录 29](#_Toc68266626)

[中断处理过程 29](#_Toc68266627)

[GDT 30](#_Toc68266628)

[实现函数调用堆栈跟踪函数 32](#_Toc68266629)

[Kdedbg.c代码以及注释 33](#_Toc68266630)

作业一．

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)

##### makefile的规则

在讲述这个makefile之前，还是让我们先来粗略地看一看makefile的规则。

target ... : prerequisites ...

command

...

...

-ggdb : 生成可供gdb使用的调试信息

-m32 :生成适用于32位环境的代码

-gstabs ：生成stabs格式的调试信息

-nostdinc ：不在标准系统文件夹寻找头文件，只在-I等参数指定的文件夹中搜索头文件

-fno-stack-protector ：不生成用于检测缓冲区溢出的代码

-Os 为减小代码大小而进行优化

-Ttext是连接时将初始地址重定向为

————————————————

+ cc kern/init/init.c

gcc -Ikern/init/ -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/init/init.c -o obj/kern/init/init.o //编译init.c输出到init.o

kern/init/init.c:95:1: warning: ‘lab1\_switch\_test’ defined but not used [-Wunused-function]

lab1\_switch\_test(void) {

^

+ cc kern/libs/readline.c

gcc -Ikern/libs/ -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/libs/readline.c -o obj/kern/libs/readline.o //编译readline.c输出到readline.o

+ cc kern/libs/stdio.c

gcc -Ikern/libs/ -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/libs/stdio.c -o obj/kern/libs/stdio.o //编译stdio.c输出到stdio.o

+ cc kern/debug/kdebug.c

gcc -Ikern/debug/ -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/debug/kdebug.c -o obj/kern/debug/kdebug.o //编译kdebug.c输出到kdebug.o

kern/debug/kdebug.c:251:1: warning: ‘read\_eip’ defined but not used [-Wunused-function]

read\_eip(void) {

^

+ cc kern/debug/kmonitor.c

gcc -Ikern/debug/ -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/debug/kmonitor.c -o obj/kern/debug/kmonitor.o

+ cc kern/debug/panic.c

gcc -Ikern/debug/ -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/debug/panic.c -o obj/kern/debug/panic.o //编译panic.c输出到panic.o

kern/debug/panic.c: In function ‘\_\_panic’:

kern/debug/panic.c:27:5: warning: implicit declaration of function ‘print\_stackframe’ [-Wimplicit-function-declaration]

print\_stackframe();

^

+ cc kern/driver/clock.c

gcc -Ikern/driver/ -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/driver/clock.c -o obj/kern/driver/clock.o //编译clock.c输出到init.o

+ cc kern/driver/console.c

gcc -Ikern/driver/ -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/driver/console.c -o obj/kern/driver/console.o //编译console.c输出到console.o

+ cc kern/driver/intr.c

gcc -Ikern/driver/ -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/driver/intr.c -o obj/kern/driver/intr.o //编译intr.c输出到intr.o

+ cc kern/driver/picirq.c

gcc -Ikern/driver/ -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/driver/picirq.c -o obj/kern/driver/picirq.o

+ cc kern/trap/trap.c

gcc -Ikern/trap/ -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/trap/trap.c -o obj/kern/trap/trap.o

kern/trap/trap.c:14:13: warning: ‘print\_ticks’ defined but not used [-Wunused-function]

static void print\_ticks() {

^

kern/trap/trap.c:30:26: warning: ‘idt\_pd’ defined but not used [-Wunused-variable]

static struct pseudodesc idt\_pd = {

^

+ cc kern/trap/trapentry.S

gcc -Ikern/trap/ -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/trap/trapentry.S -o obj/kern/trap/trapentry.o

+ cc kern/trap/vectors.S

gcc -Ikern/trap/ -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/trap/vectors.S -o obj/kern/trap/vectors.o

+ 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/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

+ 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 //编译string.c输出到string.o

+ ld bin/kernel

ld -m elf\_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/readline.o obj/kern/libs/stdio.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/debug/panic.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/intr.o obj/kern/driver/picirq.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o obj/kern/trap/vectors.o obj/kern/mm/pmm.o obj/libs/printfmt.o obj/libs/string.o -m模拟指定的连接器 模拟i386链接成kernel

+ 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

+ 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//将bootblock起始地址重定向为0x7c00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o //链接生成bootblock.o

'obj/bootblock.out' size: 472 bytes

build 512 bytes boot sector: 'bin/bootblock' success!

if=文件名：输入文件名，默认为标准输入。即指定源文件

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

conv=<关键字>notrunc：不截短输出文件

dd if=/dev/zero of=bin/ucore.img count=10000 //输入ucore.img 10000个块，即创建10000个块大小的ucore.img文件

记录了10000+0 的读入

记录了10000+0 的写出

5120000字节(5.1 MB)已复制，0.0258643 秒，198 MB/秒

dd if=bin/bootblock of=bin/ucore.img conv=notrunc //把bootblock不截短的输入到ucore

记录了1+0 的读入

记录了1+0 的写出

512字节(512 B)已复制，0.000100014 秒，5.1 MB/秒

dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc //把kernel不截短的输入到ucore

记录了146+1 的读入

记录了146+1 的写出

74875字节(75 kB)已复制，0.000340129 秒，220 MB/秒

综上生成过程为

1. 先编译kern文件夹中的文件
2. 链接并生成kernel
3. 编译bootasm.S，bootmain.c，sign.c
4. 链接并生成bootblock.o
5. 将kernel和bootblock输入并生成ucore

├── bin // =======编译后生成===============================================

│ ├── bootblock // 是引导区

│ ├── kernel // 是操作系统内核

│ ├── sign // 用于生成一个符合规范的硬盘主引导扇区

│ └── ucore.img　// ucore.img 通过dd指令，将上面我们生成的　bootblock　和　kernel 的ELF文件拷贝到ucore.img

================================================

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？

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <sys/stat.h>

int

main(int argc, char \*argv[]) {

struct stat st;// \_stat函数用来获取指定路径的文件或者文件夹的信息。

if (argc != 3) {

fprintf(stderr, "Usage: <input filename> <output filename>\n");

return -1;

}

// 读取文件头

if (stat(argv[1], &st) != 0) {

fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));

return -1;

}

printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st\_size);

if (st.st\_size > 510) {

// 文件大小超过510字节报错返回，因为最后2个字节要用作结束标志位

fprintf(stderr, "%lld >> 510!!\n", (long long)st.st\_size);

return -1;

}

char buf[512];

memset(buf, 0, sizeof(buf));

FILE \*ifp = fopen(argv[1], "rb");

int size = fread(buf, 1, st.st\_size, ifp);

// 文件实际大小需和文件头描述一致

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;//最后两位

FILE \*ofp = fopen(argv[2], "wb+");

size = fwrite(buf, 1, 512, ofp);

if (size != 512) {//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;

}

1. 大小为512字节
2. 倒数第二位为0x55
3. 倒数第三位为0xAA

# 作业二

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行

1. 修改 lab1/tools/gdbinit,

set architecture i8086

target remote :1234

define hook-stop

x/i $pc

end

2. 在 lab1目录下，执行

make debug

然后I r cs得到cs为0xf000

这时gdb停在BIOS的第一条指令处：

0xffff0: ljmp $0xf000,$0xe05b

3. 在看到gdb的调试界面(gdb)后，执行如下命令，就可以看到BIOS在执行了

si

si

...

4 .此时的CS=0xf000, EIP=0xfff0，如果想看BIOS的代码

x/2i 0xffff0

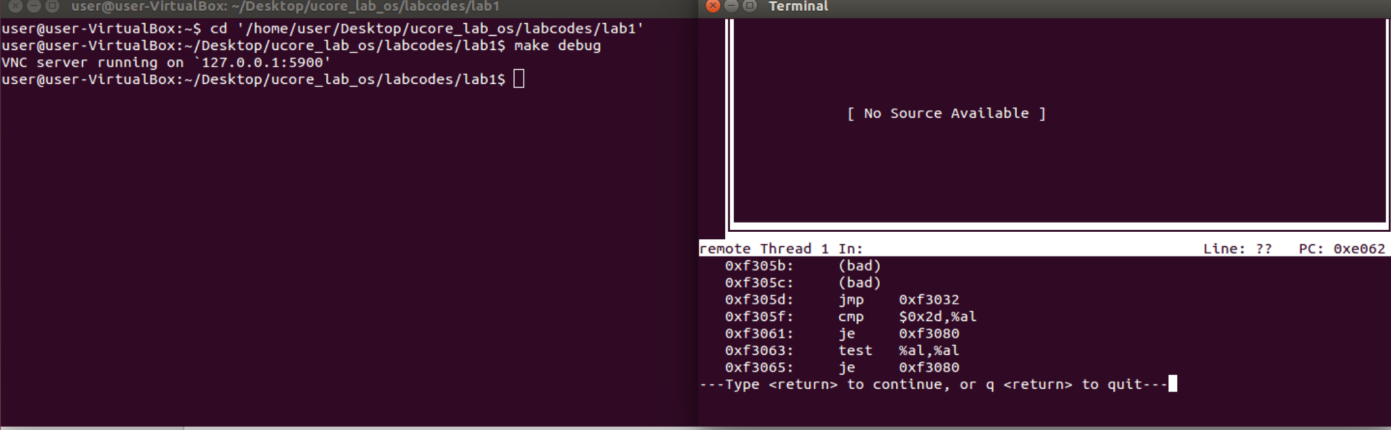
应该可以看到

0xffff0: ljmp $0xf000,$0xe05b

0xffff5: xor %dh,0x322f

进一步可以执行

x/10i 0xfe05b



用gdb查看内存

格式: x /nfu <addr>

说明  
x 是 examine 的缩写

n表示要显示的内存单元的个数

首先在CPU加电之后，CPU里面的ROM存储器会将其里面保存的初始值传给各个寄存器，

其中CS:IP = 0Xf000 : fff0（CS：代码段寄存器；IP：指令寄存器），

这个值决定了我们从内存中读数据的位置，PC = 16\*CS + IP。

1. 在初始化位置0x7c00设置实地址断点,测试断点正常。

修改gdbinit

set architecture i8086

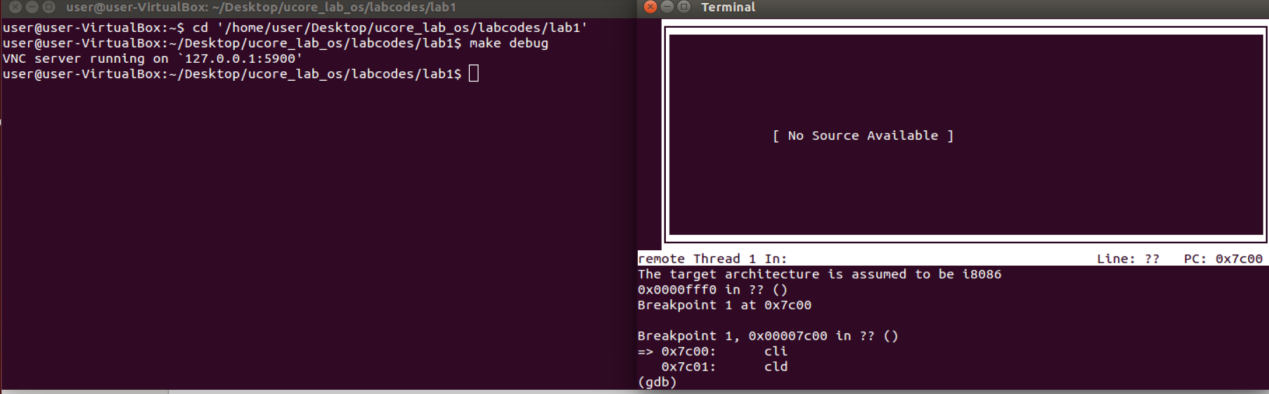
target remote :1234

b \*0x7c00

c

x/2i $pc

Gdb b+行号 直接在当前所在的文件添加断点



1.开第二遍debug前要先关闭端口

## 通过杀掉进程的方法来关闭端口

每个端口都有一个守护进程，kill掉这个守护进程就可以了

每个端口都是一个进程占用着，

第一步、用下面命令

netstat -anp |grep 端口

找出占用这个端口的进程，

第二步、用下面命令

kill -9 PID

杀掉就行了

2.地址前要加个\*

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。

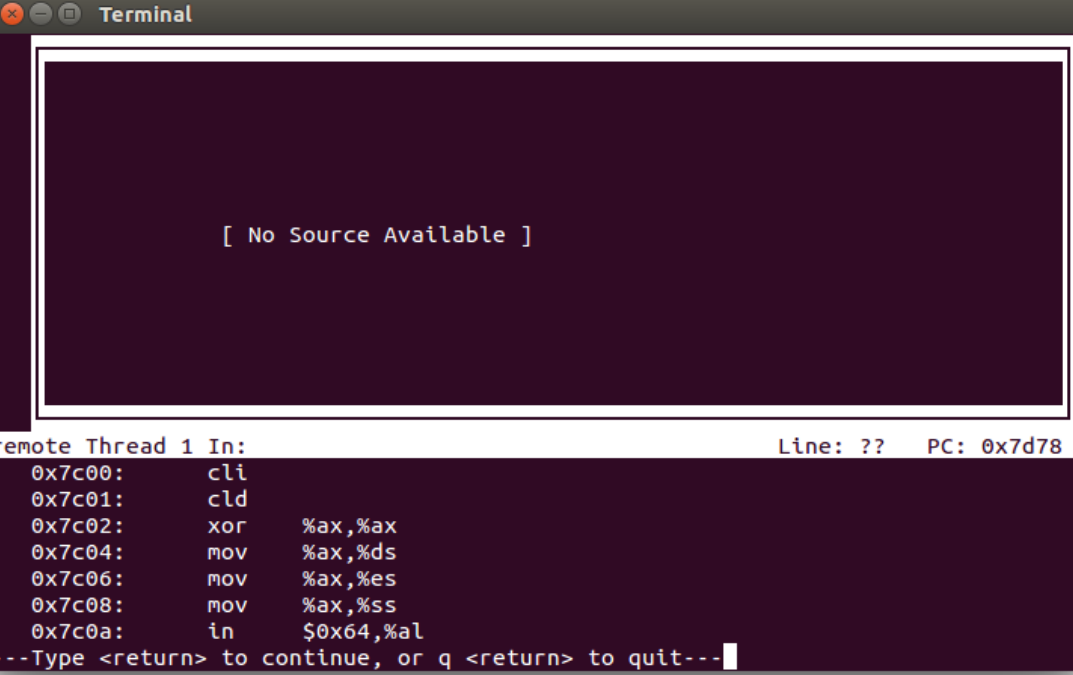
修改gdbinit

set architecture i8086

target remote :1234

b \*0x7c00

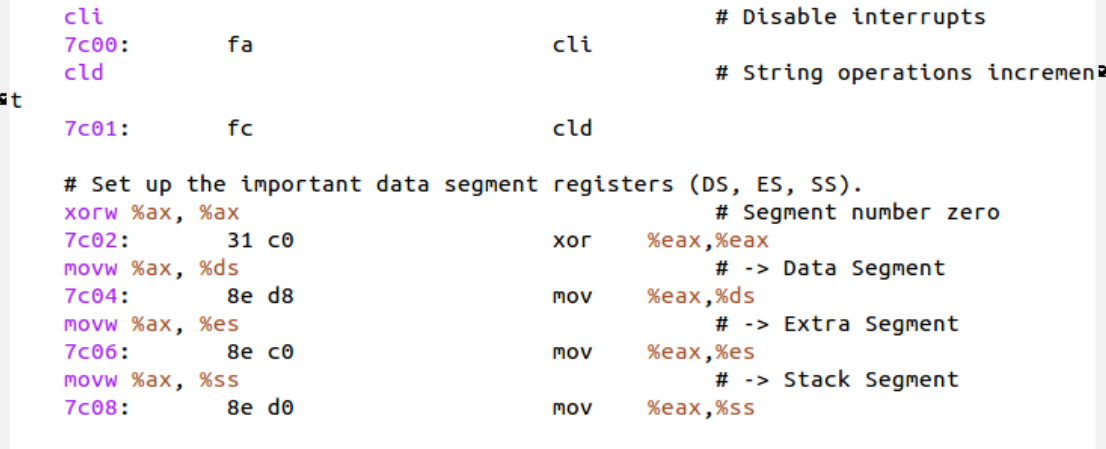
c



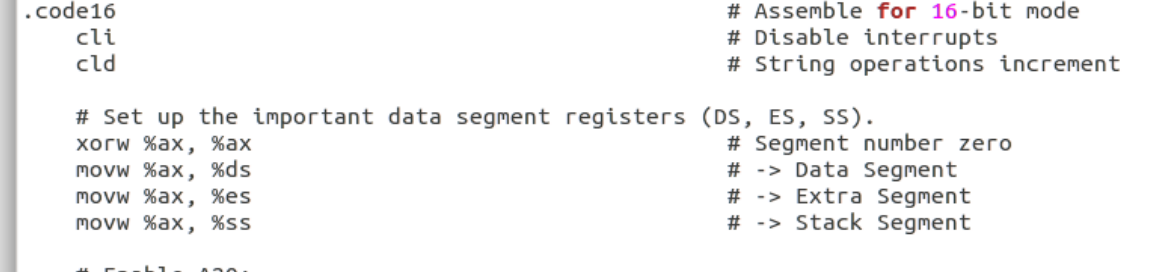
输入x/16i 0x7c00便可以从0x7C00开始跟踪到代码执行完毕。

对比

Bootasm.asm



Bootasm.s



CLI禁止中断发生

cld：清除方向标志，在字符串的比较，赋值，读取等一系列和rep连用的操作中，di或si是可以自动增减的而不需要人来加减它的值，cld即告诉程序si，di向前移动，std指令为设置方向，告诉程序si，di向后移动

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



# 练习三：分析bootloader进入保护模式的过程。

#include <inc/mmu.h>

# 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.

.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

.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

testb $0x2,%al

jnz seta20.1

movb $0xd1,%al # 0xd1 -> port 0x64

outb %al,$0x64

seta20.2:

inb $0x64,%al # Wait for not busy

testb $0x2,%al

jnz seta20.2

movb $0xdf,%al # 0xdf -> port 0x60

outb %al,$0x60

# Switch from real to protected mode, using a bootstrap GDT

# and segment translation that makes virtual addresses

# identical to their physical addresses, so that the

# effective memory map does not change during the switch.

lgdt gdtdesc

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

ljmp $PROT\_MODE\_CSEG, $protcseg

.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

# Set up the stack pointer and call into C.

movl $start, %esp

call bootmain

# If bootmain returns (it shouldn't), loop.

spin:

jmp spin

# Bootstrap GDT

.p2align 2 # force 4 byte alignment

gdt:

SEG\_NULL # null seg

SEG(STA\_X|STA\_R, 0x0, 0xffffffff) # code seg

SEG(STA\_W, 0x0, 0xffffffff) # data seg

gdtdesc:

.word 0x17 # sizeof(gdt) - 1

.long gdt # address gdt

1. **关闭中断，初始化寄存器。**

.code16 （CPU刚启动时位于实模式，CPU位宽和寄存器大小都为16位，该代码表示 进入16位实模式。）

cli （关闭中断，避免产生中断被BIOS中断处理程序处理）

cld （字符串操作时方向递增）

xorw %ax,%ax （xorw->异或操作）（通过异或操作将ax寄存器置0）

movw %ax,%ds （movw->将前一个寄存器内容赋给后一个）（这里将ds，es,ss寄存器

movw %ax,%es 的值全置位0 ）

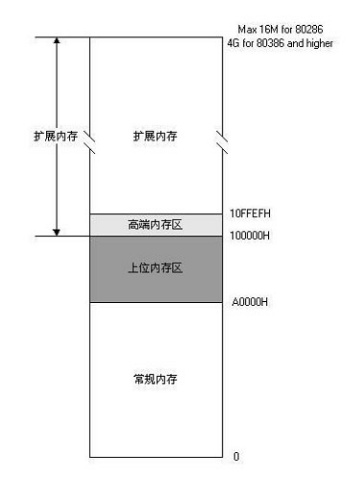
movw %ax,%ss

1. **打开A20**

**为什么打开A20?**

**历史问题的向下兼容：**

8088/8086只有20位地址线，按理它的寻址空间是2^20，应该是1024KB，但PC机的寻址结构是segment:offset，segment和offset都是16位的寄存器，最大值是0ffffh，换算成物理地址的计算方法是把segment左移4位，再加上offset，所以segment:offset所能表达的寻址空间最大应为0ffff0h + 0ffffh = 10ffefh（前面的0ffffh是segment=0ffffh并向左移动4位的结果，后面的0ffffh是可能的最大offset），大约是1088KB，就是说，segment:offset的地址表达能力，超过了20位地址线的物理寻址能力。在早先，由于所有的机器都没有那么大的内存，加上地址线只有20位，所以当你用segment:offset的方式企图寻址100000h这个地址时，由于没有实际的第21位地址线，你实际寻址的内存是00000h的位置，如果你企图寻址100001h这个地址时，你实际得到的内容是地址00001h上的内容，所以这个事对实际使用几乎没有任何影响，但是后来就不行了，出现了80286，地址线达到了24位，使segment:offset寻址100000h--10ffefh这将近64K的存储器成为可能，为了保持向下兼容，于是出现了A20 Gate



这张图很清楚地说明了问题，DOS下的“常规内存”只有640K，这640K就是从0--A0000H这段地址空间；所谓“上位内存区”，指的就是20位地址线所能寻址到的1M地址空间的上面384K空间，就是从A0001H--100000H这段地址空间，也就是我们说的用于ROM和系统设备的地址区域，这384K空间和常规内存的640K空间加起来就是20位地址线所能寻址的完整空间1024KB；由于80286和80386的出现使PC机的地址线从20位变成24位又变成32位，寻址能力极大地增加，1M以上的内存寻址空间，我们统称为“扩展内存”；这里面绝大部分内存区域只能在保护模式下才能寻址到。

出现80286以后，为了保持和8086的兼容，PC机在设计上在第21条地址线（也就是A20）上做了一个开关，当这个开关打开时，这条地址线和其它地址线一样可以使用，当这个开关关闭时，第21条地址线（A20）恒为0，这个开关就叫做A20 Gate，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......，所以在保护模式下，这个开关也必须打开。

由此我们可以得到实模式和保护模式的区别，实模式主要为兼容之前的cpu，如8086，只有20根地址线，cpu位宽为16位。因为寄存器为16位，无法完整访问整个存储空间，因此实模式采用分段的方式访问物理内存。而保护模式下80386可以使用32位地址线，访问4GB的物理内存空间。保护模式下程序访问内存使用分段存储管理机制，内存可被划分为若干段，每一段拥有基址、界限、特权级、权限等属性，程序在访问段时可进行安全检查，保证安全，因此叫做保护模式，突出对内存保护的安全特性。

**如何打开A20?**

回到BIOS源码。

seta20.1:

inb $0x64,%al （等待键盘控制器不繁忙，若繁忙，返回seta20.1）

testb $0x2,%al （检测是否繁忙）

jnz seta20.1

movb $0xd1,%al # 0xd1 -> port 0x64（发送写8042 outputPort命令）

outb %al,$0x64

seta20.2:

inb $0x64,%al 等待outputbuffer为空

testb $0x2,%al

jnz seta20.2

movb $0xdf,%al 我们将0xdf写入60h端口，至此来打开A20开关。

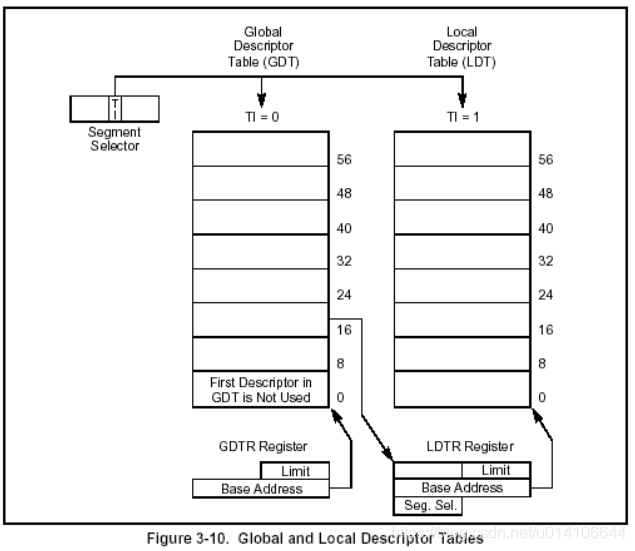
outb %al,$0x60

**如何初始化GDT表：**

lgdt gdtdesc 将gdtdesc指示的6字节加载到寄存器gdtr，一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

通过lgdt汇编指令可以把GDTR描述符表的大小和起始位置存入gdtr寄存器中，指令格式如下：

   lgdt      [描述段描述符表的地址]



GDTR寄存器保存了GDT的32位基地址和16位表界限，基地址指GDT的0字节的线性地址，表界限指表中的字节个数，LGDT和SGDT指令用来分别装载和保存GDTR寄存器。对于保护模式的操作，作为处理器初始化过程一部分，一个新基地址必须装入GDTR。

全局描述符表及其定义如下所示，一共定义了三个段，第一个为空段，第二个为代码段，第三个为数据段。

# 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

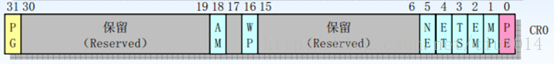
SEG\_ASM(STA\_W, 0x0, 0xffffffff) # data seg for bootloader and kernel

gdtdesc:

.word 0x17 # sizeof(gdt) - 1

.long gdt # address gdt

**如何使能和进入保护模式：**

****

**在这里由于我们需要进入保护模式，所以暂时可以先不用管其他的位，只需关注最低位的PE即可，**

**PE是启用保护位(protection enable)，当设置该位的时候即开启了保护模式，**

**系统上电复位的时候该位默认为0，于是便是实模式；**

**当PE置1的时候，进入保护模式，实质上是开启了段级保护，只是进行了分段，没有开启分页机制，**

通过将cr0寄存器PE位置1便开启了保护模式

movl %cr0, %eax cro->eax

orl $CR0\_PE\_ON, %eax 或逻辑运算置1

movl %eax, %cr0 将eax转回cro

通过长跳转更新cs的基地址（进入32位保护模式）

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32

设置段寄存器，并建立堆栈：

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

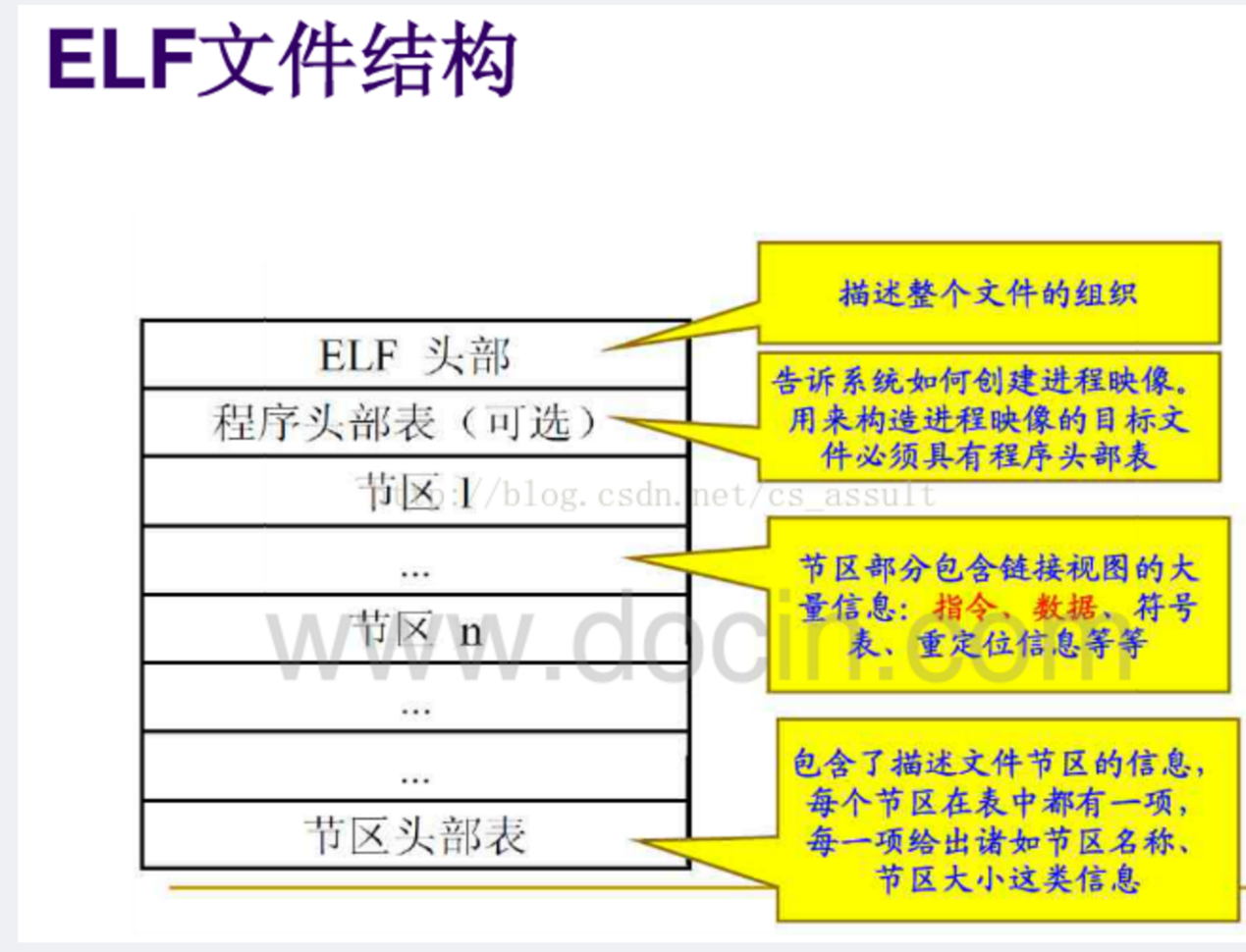
将kernel数据段基地址更新到数据段寄存器DS，ES，SS中。ebp指向栈顶，esp指向栈底。

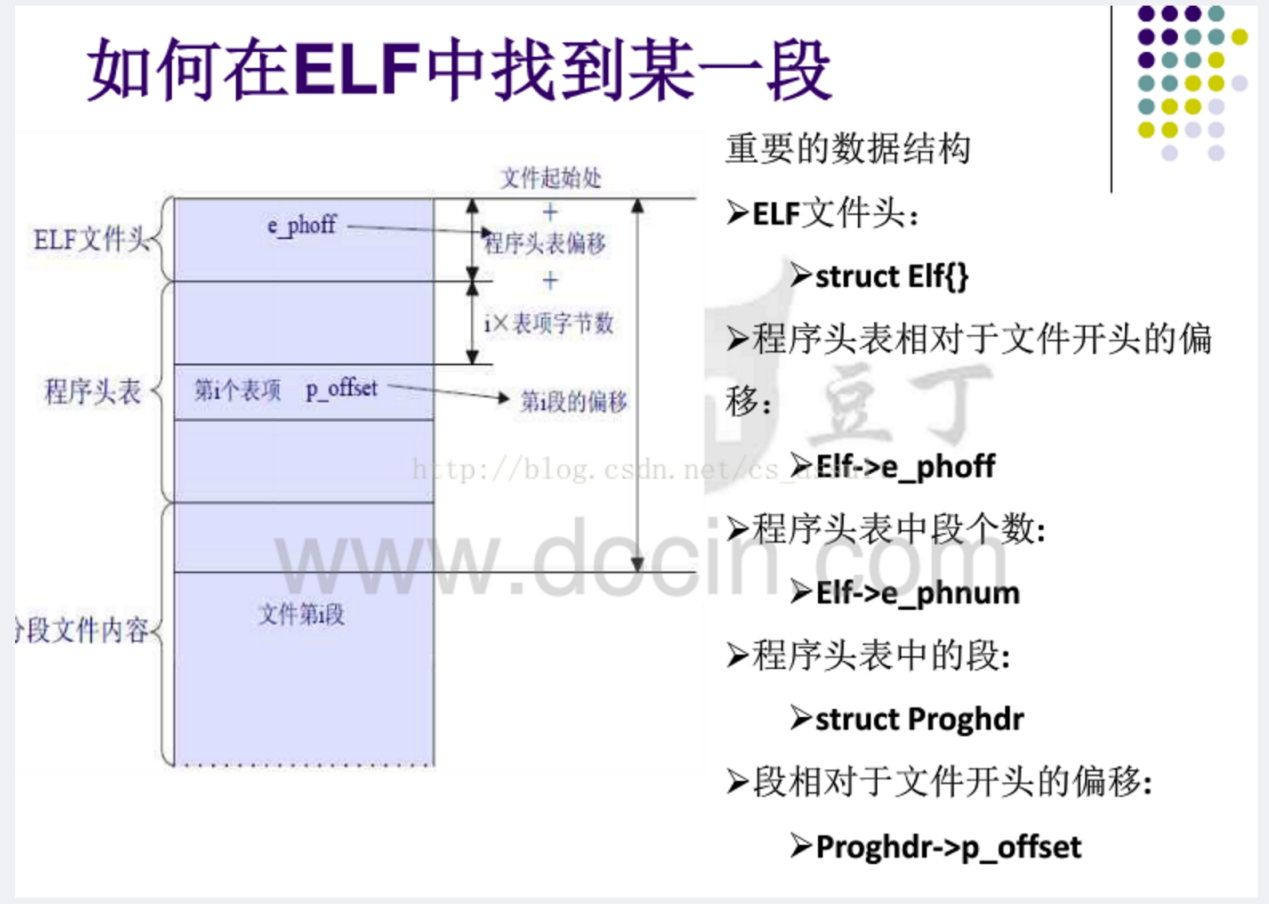
转到保护模式完成，进入boot主方法

call bootmain

bootmain方法则从硬盘开始加载操作系统文件。

# 练习4——分析bootloader加载ELF格式的OS的过程





在这里我们只需要关注其中的几个参数，e\_magic，是用来判断读出来的ELF格式的文件是否为正确的格式；e\_phoff，是program header表的位置偏移；e\_phnum，是program header表中的入口数目；e\_entry，是程序入口所对应的虚拟地址

#include <inc/x86.h>

#include <inc/elf.h>

#define SECTSIZE 512

#define ELFHDR ((struct Elf \*) 0x10000)

void readsect(void\*, uint32\_t);

void readseg(uint32\_t, uint32\_t, uint32\_t);

void

bootmain(void)

{

struct Proghdr \*ph, \*eph; 定义ph，eph

readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0); 首先读取ELF的头部

if (ELFHDR->e\_magic != ELF\_MAGIC) 检测是否是个ELF文件

goto bad;

ph = (struct Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR->e\_phoff);将描述表的头地址存在ph

eph = ph + ELFHDR->e\_phnum; 头地址加指令数量=尾地址

for (; ph < eph; ph++)

readseg(ph->p\_va, ph->p\_memsz, ph->p\_offset);按照描述表将ELF文件中数据载入内存

((void (\*)(void)) (ELFHDR->e\_entry))();根据ELF头部储存的入口信息，找到内核的入口

bad://异常处理

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1)

/\* do nothing \*/;

}

磁盘第一个扇区存储bootloader代码，从第二个扇区往后存储操作系统内核镜像，镜像使用ELF格式来存储

**readseg函数**

从内核镜像offset偏移位置处读取count字节到虚拟地址va位置处

readseg(uintptr\_t va, uint32\_t count, uint32\_t offset) {

uintptr\_t end\_va = va + count;//结束地址

// round down to sector boundary

va -= offset % SECTSIZE;函数会从内核的offset位置（因为内核起始位置为一个扇区的起始位置，所以offset % 512 也就是 offset这个位置相对于这个扇区的offset）读取count个字节，放置到va所对应的虚拟内存位置。函数传进来的offset和(offset % sectsize)意思不一样，前者是相对于1号扇区开始位置的偏移量，后者是相对于你要读的那个扇区的偏移量。

va -= offset % SECTSIZE; 这条语句表明我们读字节必须从扇区头部开始读；所以将va减去offset

// translate from bytes to sectors; kernel starts at sector 1

uint32\_t secno = (offset / SECTSIZE) + 1;// 存储我们需要读取的磁盘的位置

// 通过一个for循环一次从磁盘中读取一个整块

for (; va < end\_va; va += SECTSIZE, secno ++) {

// 继续对虚存va和secno进行自加操作，直到读完所需读的东西为止。

readsect((void \*)va, secno);// 磁盘中读取一个整块 存到相应的虚存va中

}

}

readseg简单包装了readsect，可以从设备读取任意长度的内容

**等待磁盘函数**

void

waitdisk(void)

{ while ((inb(0x1F7) & 0xC0) != 0x40)

}

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（这个状态应该意味着磁盘空闲）才返回。

**readsect函数**

从设备的第offset扇区读取数据到dst位置

void

readsect(void \*dst, uint32\_t offset)

{

等待硬盘空闲

waitdisk();

outb(0x1F2, 1); 读取扇区数目为1

outb(0x1F3, offset);

outb(0x1F4, offset >> 8);

outb(0x1F5, offset >> 16);

outb(0x1F6, (offset >> 24) | 0xE0);（上面四条指令联合制定了扇区号）

29-31位强制设为1

28位(=0)表示访问"Disk 0"

0-27位是28位的偏移量

outb(0x1F7, 0x20);

// cmd 0x20 - read sectors 0x20命令，读取扇区

waitdisk();

//读取到dst位置，以4位为单位

insl(0x1F0, dst, SECTSIZE/4);

}

/\* insl:从I/O端口port读取count个数据(单位双字)到以内存地址addr为开始的内存空间 \*/

void insl(unsigned port, void \*addr, unsigned long count);

如何读取硬盘扇区：

等待硬盘空闲。

硬盘空闲后，发出读取扇区的命令。对应的命令字为0x20，放在0x1F7寄存器中；读取的扇区数为1，放在0x1F2寄存器中；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。

等待硬盘空闲。

硬盘再次空闲后，开始从0x1F0寄存器中读数据。注意insl的作用是"That function will read cnt dwords from the input port specified by port into the supplied output array addr."，是以dword即4字节为单位的，因此这里SECTIZE需要除以4.

### 如何加载ELF格式的OS：

1.从硬盘读了8个扇区数据到内存0x10000处，并把这里强制转换成elfhdr使用；（使用readselect函数）

2.校验e\_magic字段；（if (ELFHDR->e\_magic != ELF\_MAGIC)）

3.根据偏移量分别把程序段的数据读取到内存中

# [练习5]：实现函数调用堆栈跟踪函数

对于函数堆栈来说可以分为以下三部分操作：

1、首先保存原相关寄存器的状态，即将相关参数以及寄存器的当前状态压入栈；

2、其次在栈中进行函数操作，即完成函数的相关功能；

3、最后释放栈空间，回复原寄存器状态。

首先，可以通过read\_ebp()和read\_eip()函数来获取当前ebp寄存器和eip 寄存器的信息。 然后通过ebp+12,ebp+16,ebp+20,ebp+24来输出4个参数的值，最后更新ebp：ebp=ebp[0],更新eip：eip=ebp[1]。直到ebp 对应地址的值为0（表示当前函数为bootmain）。

read\_eip() 函数定义在kdebug.c中：

static \_\_noinline uint32\_t

read\_eip(void) {

uint32\_t eip;

asm volatile("movl 4(%%ebp), %0" : "=r" (eip)); //内联汇编，读取（ebp-4）的值到变量eip

return eip; //返回eip的值

}

read\_ebp() 函数定义在x86.h中：

static inline uint32\_t

read\_ebp(void) {

uint32\_t ebp;

asm volatile ("movl %%ebp, %0" : "=r" (ebp)); //内联汇编,读取edp寄存器的值到变量ebp

return ebp; //返回ebp的值

}

​ **实现函数如下：**

void

print\_stackframe(void) {

uint32\_t ebp = read\_ebp(), eip = read\_eip(); //获取ebp和eip的值

int i, j;

//#define STACKFRAME\_DEPTH 20

for (i = 0; ebp != 0 && i < STACKFRAME\_DEPTH; i ++) {

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

uint32\_t \*args = (uint32\_t \*) ebp + 2; //参数的首地址

for (j = 0; j < 4; j ++) {

cprintf("0x%08x ", args[j]); //打印4个参数

}

cprintf("\n");

print\_debuginfo(eip - 1); //打印函数信息

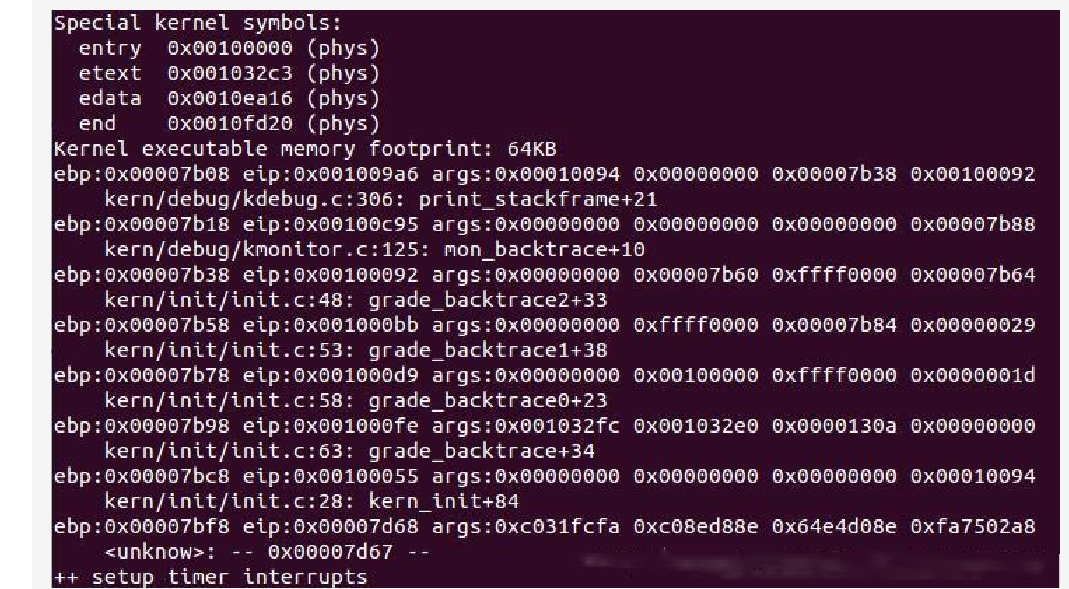
eip = ((uint32\_t \*)ebp)[1]; //更新eip

ebp = ((uint32\_t \*)ebp)[0]; //更新ebp

}

}

**执行 make qemu 得到输出**：



最后一行中给出了ebp，eip和args三个参数，其具体意义为

* **ebp=: 0x00007bf8** bootmain的地址
* **eip=: 0x00007d72** 是从bootasm.s跳转到bootmain前的地址，也就是bootmain的返回地址。
* **args=: 0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8** 通常状态下，args存放的四个dword是对应4个输入参数的值。但是在最底层处，即7c00往后增加的地址处，那里是bootloader的代码段，所以最后的args其实是bootloader指令的前十六个字节，下面这个例子就能很好的说明情况

# 练习6]完善中断初始化和处理

[6.1]

**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？**

我们知道CPU存在实模式和保护模式的操作模式。当计算机启动后，我们进入的是实模式，而实模式的内存寻址方式是通过：段值 × 16 + 偏移 = 物理地址。这样只能具有1MB的寻址能力。如果想要更为强大的寻址能力，就必须通过一定机制，进入具有更强大寻址能力的保护模式。

一个表项的结构如下:

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

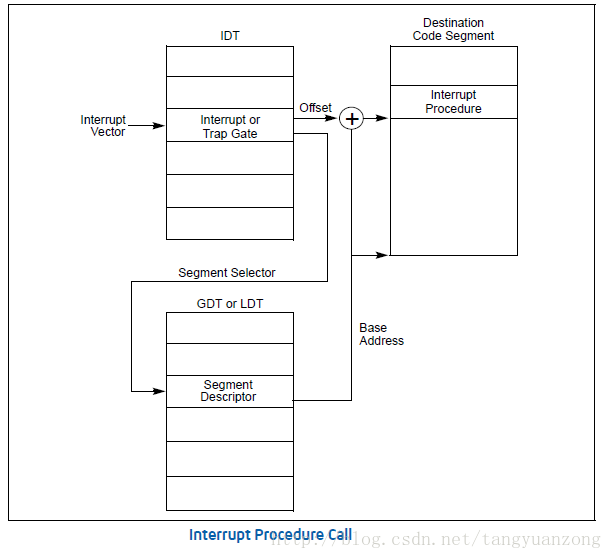
unsigned gd\_p : 1; // Present

unsigned gd\_off\_31\_16 : 16; // high bits of offset in segment

};

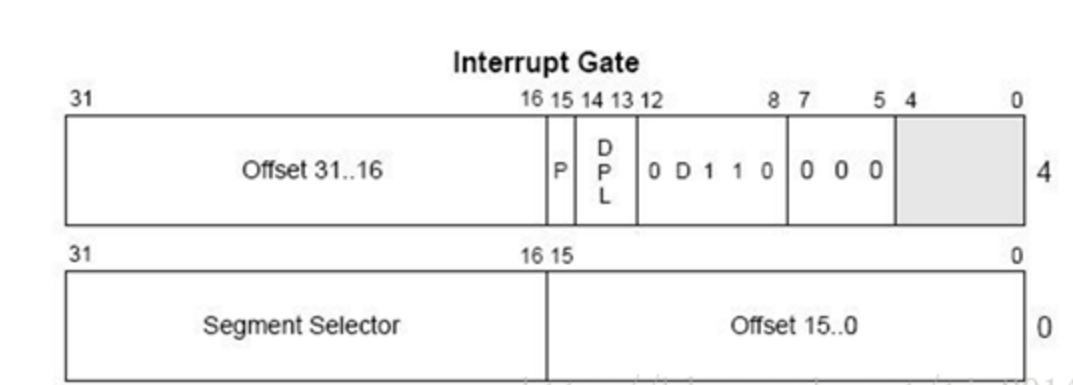
保护模式下，段寄存器含有段选择子；CPU收到中断信息后，需要先根据中断类型码找到对应的中断处理程序地址，这个过程通过查中断描述符表（获得特定中断处理程序的偏移量并以中断向量为索引查找中断处理程序的段选择子）以及全局描述符表（通过中断处理程序的段选择子获得段基址）完成；一个表项被称作一个门描述符，占**八个字节**。

**中断处理过程**:



**中断描述符表**一个表项占用8字节，其中2-3字节是段选择子，0-1字节和6-7字节拼成偏移量，为offset的低16位和高16位。

通过段选择子（16-31）去GDT中找到对应的基地址，然后基地址加上偏移量就是中断处理代码的入口。



[6.2]

**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。**

**SETGATE函数的实现**：

#define SETGATE(gate, istrap, sel, off, dpl) { \

(gate).gd\_off\_15\_0 = (uint32\_t)(off) & 0xffff; \

(gate).gd\_ss = (sel); \

(gate).gd\_args = 0; \

(gate).gd\_rsv1 = 0; \

(gate).gd\_type = (istrap) ? STS\_TG32 : STS\_IG32; \

(gate).gd\_s = 0; \

(gate).gd\_dpl = (dpl); \

(gate).gd\_p = 1; \

(gate).gd\_off\_31\_16 = (uint32\_t)(off) >> 16; \

}

**宏定义和数组说明：**

#define GD\_KTEXT ((SEG\_KTEXT) << 3) // kernel text

#define DPL\_KERNEL (0)

#define DPL\_USER (3)

#define T\_SWITCH\_TOK 121 // user/kernel switch

static struct gatedesc idt[256] = {{0}};

**idt\_init函数的实现：**

void

idt\_init(void) {

extern uintptr\_t \_\_vectors[]; //保存在vectors.S中的256个中断处理例程的入口地址数//组

int i;

//使用SETGATE宏，对中断描述符表中的每一个表项进行设置

for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) { //IDT表项的个数

//在中断门描述符表中通过建立中断门描述符，其中存储了中断处理例程的代码段GD\_KTEXT和偏移量\_\_vectors[i]，特权级为DPL\_KERNEL。//这样通过查询idt[i]就可定位到中断服务例程的起始地址。

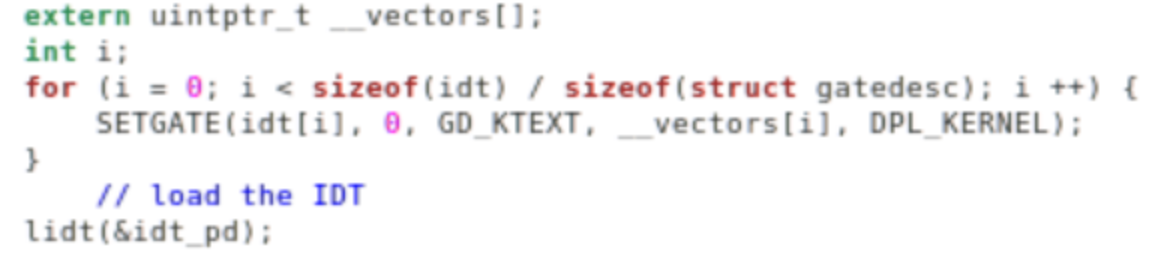
SETGATE(idt[i], 0, GD\_KTEXT, \_\_vectors[i], DPL\_KERNEL)；

}

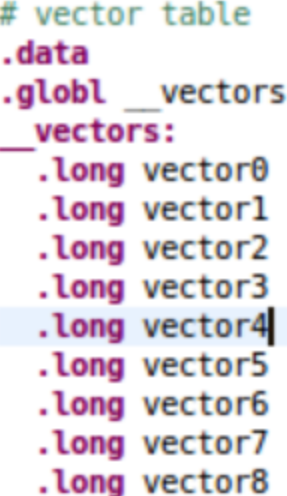
lidt(&idt\_pd);

}

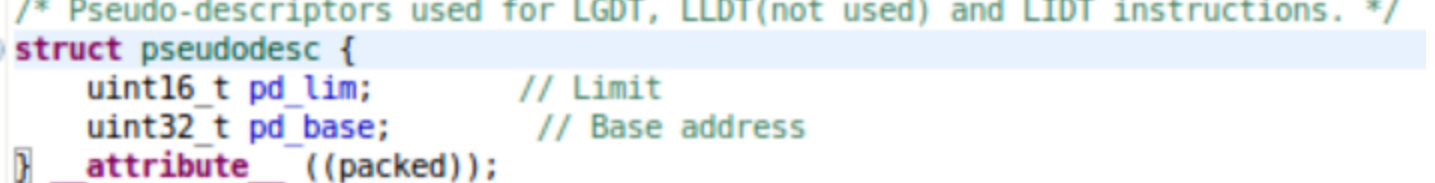
**IDT初始化整体流程是先初始化内核态中断，再初始化系统调用中断，最后在IDTR寄存器中存放IDT的地址，代码如下**：

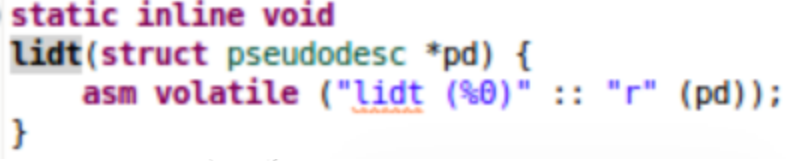


* 根据提示，中断服务例程（ISR）的入口地址都存放在uintptr\_t类型的\_\_vectors数组中，该数组由tools/vectors.c生成，存放在kern/trap/vector.S中 ；

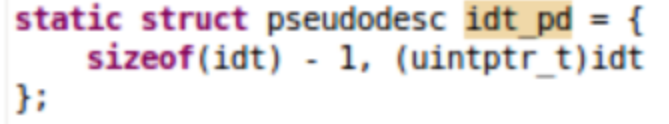


* vector即为中断向量，跳转到对应的中断向量之后，将调用kern/trap/trapentry.S中\_\_alltraps函数保存被打断的程序的现场；
* trap.c中为IDT准备了结构体数组定义：类型为gatedesc的idt[256]，每个IDT表项就是该数组中的一项，使用SETGATE宏进行初始化；
* 中断的特权级应设为ring 0，使用/kern/mm/memlayout.h中定义的系统、用户特权级变量（DPL\_KERNEL，值是0；DPL\_USER，值是3）；
* IDT的内容初始化完成后，还需要将IDT的起始地址加载到IDTR寄存器里，即需调用lidt指令——在C程序里写调用的方法是使用x86.h中定义的lidt函数（该函数参数类型pseudodesc也定义在x86.h中，可以看到存放了段的大小和基址），其功能是生成内联汇编来调用lidt指令（volatile关键字的意思是编译时拒绝优化）。





* 在trap.c中，可以方便地将该文件里定义的pseudodesc类型结构体idt\_pd实例化，作为lidt函数的参数：



[6.3]

**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”。**

case IRQ\_OFFSET + IRQ\_TIMER:

ticks++;

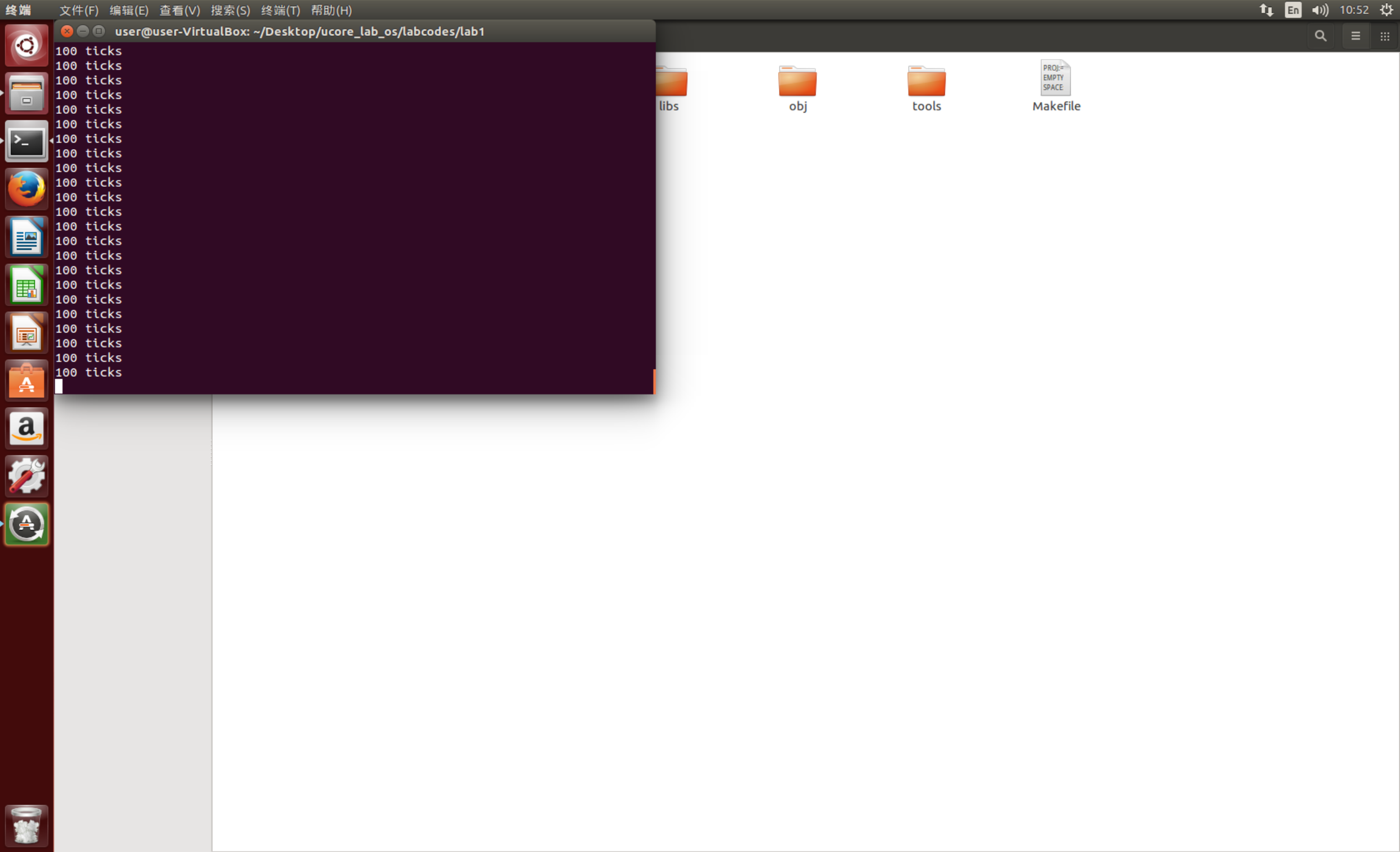
if(ticks == TICK\_NUM) {

print\_ticks();

ticks = 0;

}

break;**运行结果：**



# 附录

中断处理过程

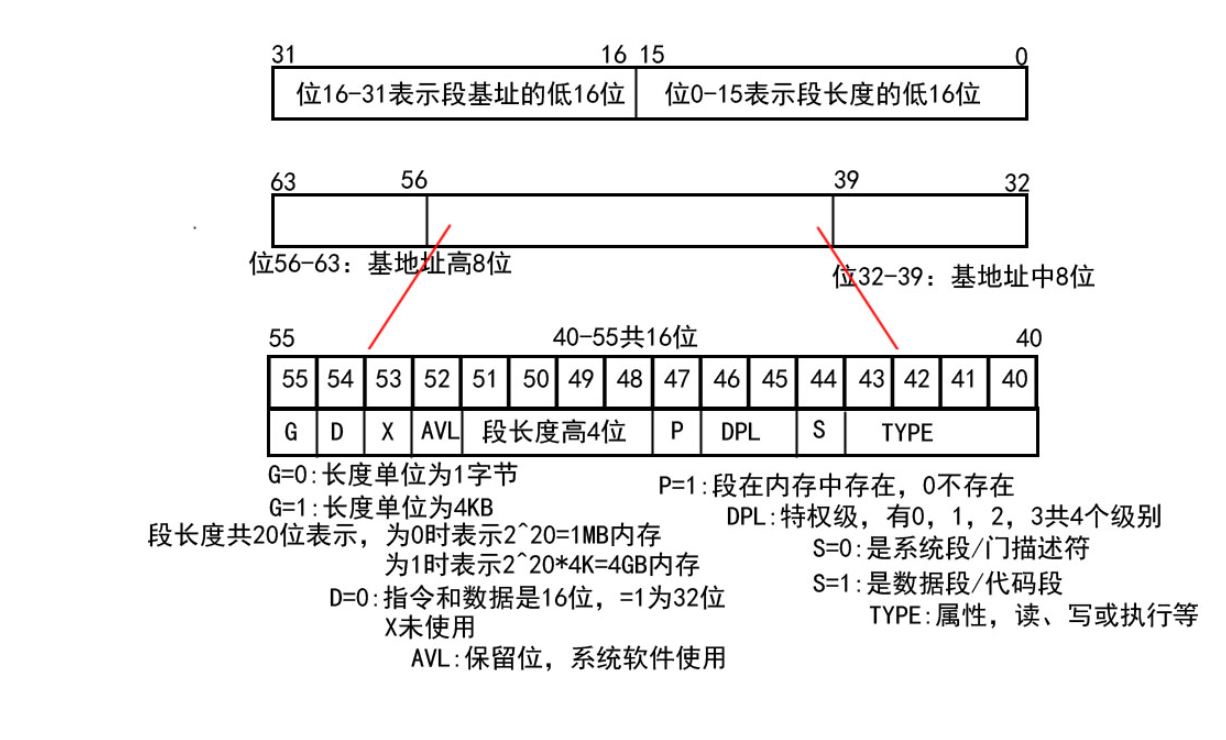
如果一个中断产生时任务正在用户代码中执行，那么该中断会引起CPU特权级从3到0的变化，此时CPU就会运行用户态堆栈到内核态堆栈的切换操作。CPU会从当前任务的任务状态段TSS（任务状态段）中取得新堆栈的段选择符和偏移量。因为中断服务程序在内核中，属于0级特权级代码，所以48位的内核态堆栈指针会从TSS的ss0和esp0字段中获得。在定位了新堆栈（内核态堆栈）之后，CPU就会首先把原用户态堆栈指针ss和esp压入内核态堆栈，随后把标志积存器eflags的内容和返回位置cs，eip压入内核态堆栈。   
  内核的系统调用是一个软件中断，因此任务调用系统调用时就会进入内核并执行内核中的中断服务代码。此时内核代码就会使用该任务的内核态堆栈进行操作。同样，当进入内核程序时，由于特权级别发生了改变，用户态堆栈的堆栈段和堆栈指针以及eflags会被保存在任务的内核态堆栈中。而在执行iret退出内核程序返回到用户程序时，将恢复用户态的堆栈和eflags。   
  如果一个任务正在内核态中运行，那么若CPU响应中断就不再需要进行堆栈切换操作。因为此时该任务运行的内核代码已经在使用内核态堆栈，并且不涉及到优先级别的变化，所以CPU仅把eflags和中断返回指针cs，eip压入当前内核态堆栈，然后执行中断服务过程。

GDT

我们知道CPU存在实模式和保护模式的操作模式。当计算机启动后，我们进入的是实模式，而实模式的内存寻址方式是通过：段值 × 16 + 偏移 = 物理地址。这样只能具有1MB的寻址能力。如果想要更为强大的寻址能力，就必须通过一定机制，进入具有更强大寻址能力的保护模式。

在保护模式下，不光提供了强大的寻址能力，还提供了内存保护，能够防止用户程序改写内核代码，并为操作系统提供了更好的硬件保障。在该模式下，有了32位的寄存器，一个寄存器就可以表示4G的地址空间，那段值加偏移这样的寻址方式是不是就不用了呢？当然不会。在这里，而是用了另外的方式来表示。

采用的方式就是引入了数据结构GDT，即全局描述符表，不知道有多少人听过？说简单点，GDT就是一个数组，每一个元素就是一个描述符，多个组合一起就构成了全局描述符表。而每一个描述符共64位，包含了以下的这些信息：段基址、段长度、属性。原来的段寄存器，比如CS，DS等存的值则不是段偏移了，而是GDT的索引，通过该索引就可以找到对应的描述符。现在是不是明白了呢？接下来我们再详细的解释一下一个描述符中各个位的意义吧。



上面说过，段寄存器中保存的是索引，那么段寄存器中的数据又是什么样呢？



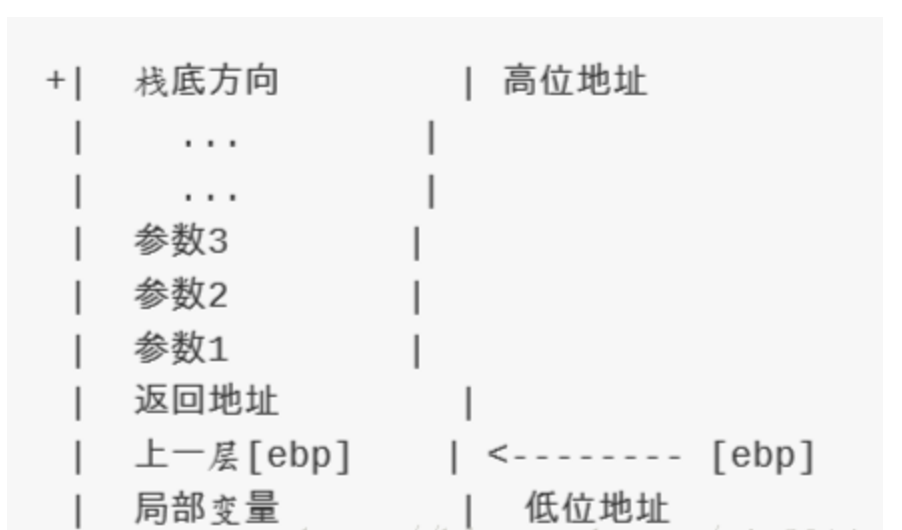
实现函数调用堆栈跟踪函数

对于函数堆栈来说可以分为以下三部分操作：

1、首先保存原相关寄存器的状态，即将相关参数以及寄存器的当前状态压入栈；

2、其次在栈中进行函数操作，即完成函数的相关功能；

3、最后释放栈空间，回复原寄存器状态。



pushl %ebp # 上一层的ebp的压入

movl %esp, %ebp # %ebp　指向%esp栈顶(存储的是上一层的ebp，也就是返回地址)

ss:[ebp+4]处为返回地址，

ss:[ebp-4]处为第一个局部变量，

ss:[ebp]处为上一层ebp值。