# 操作系统Ucore Lab1 实验报告

温雅康 MRYES666 032



$(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)

//这一段代码就是首先创建一个10000字节的空间，然后将bootblock复制一份到空间里，这里生成ucore.img还需要kernel和bootblock这两个文件。

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)

@echo "bbbbbbbbbbbbbbbbbbbbbb$(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)

//这里的代码是生成kernel所用的代码  
//我们在ucore中通过”make V=”这个指令得到生成kernel的具体过程具体的过程如下

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

//我们可以看到这个过程就是GCC编译器将kern目录下的c代码编译为一种后缀名为.o文件

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)

@echo "========================$(call toobj,$(bootfiles))"

@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)

//这里我们依旧使用”make V=”这个命令来看看生成bootblock的具体过程  
我们根据这一段代码来看生成bootblock首先就是需要bootasm.o bootmain.o和sign

bootfiles = $(call listf\_cc,boot)

$(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

//我们注意到这一段就是生成bootasm.o和bootmain.o的代码

gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o

gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o

//这两段是生成bootasm.o和bootmain.o的具体编译过程

最后我们罗列一下生成sign的具体过程与生成所用的代码

$(call add\_files\_host,tools/sign.c,sign,sign)

$(call create\_target\_host,sign,sign)

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

有了bootasm.o、bootmain.o、sign后我们就可以生成bootblock了

生成bootblock的具体过程如下

ld -m elf\_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

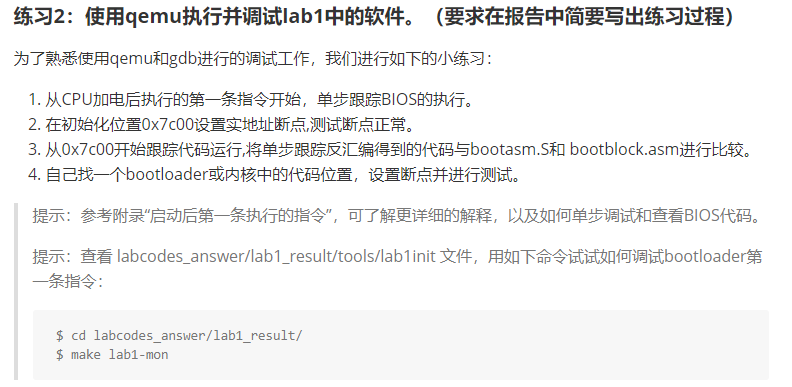
这个过程参数的具体解释如下

-m 模拟为i386上的连接器

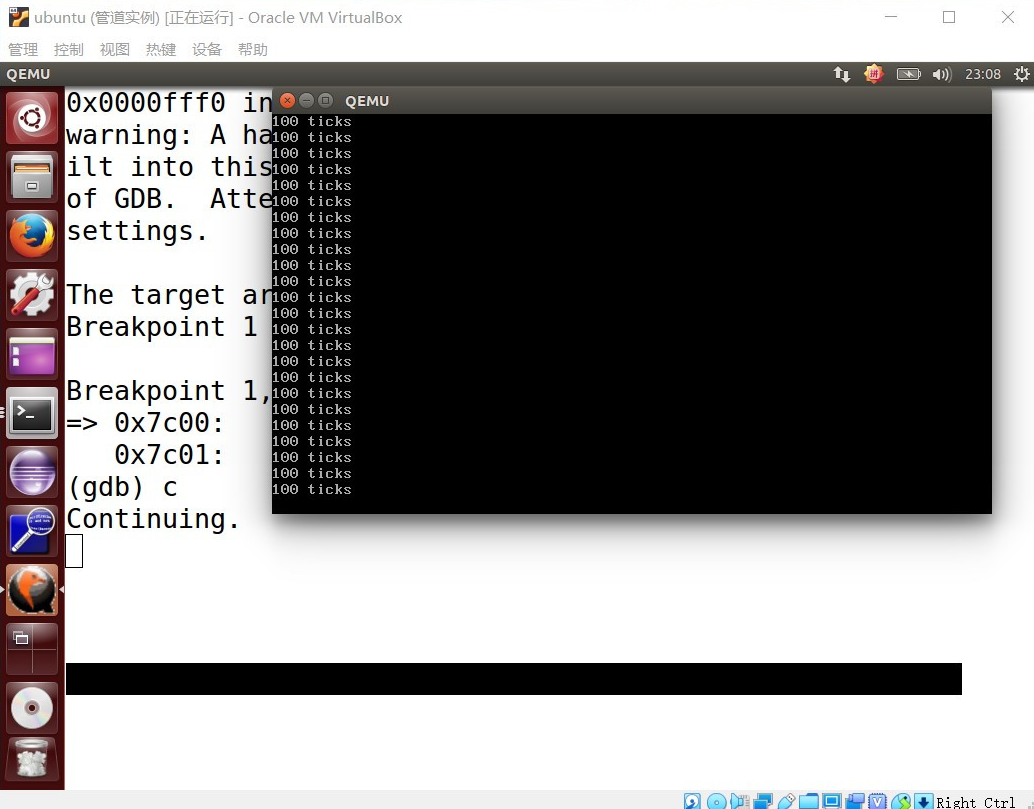
-N 设置代码段和数据段均可读写

-e 指定入口

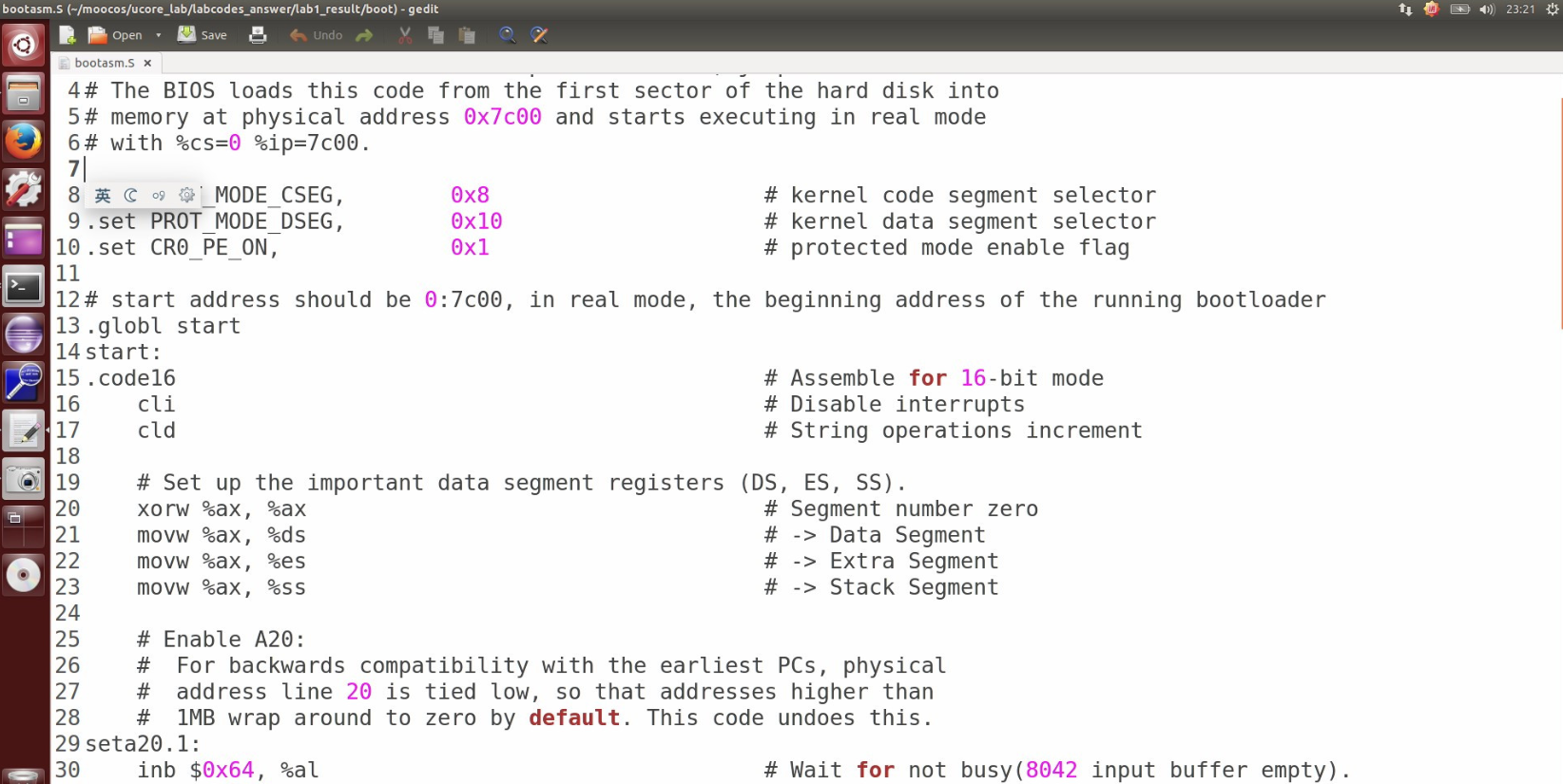
-Ttext 制定代码段开始位置

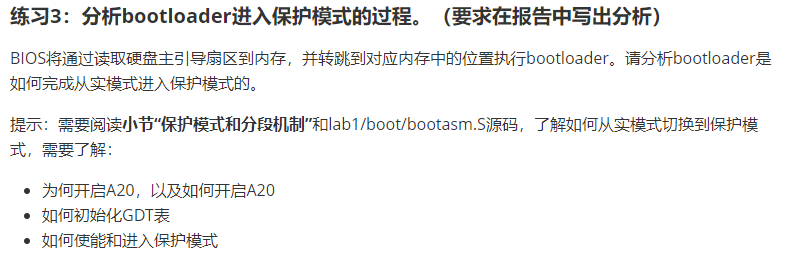


首先第一步进入qemu开始设置断点与0x7c00的位置  
b \*0x7c00  
第二步开始运行查看结果  
c



我们查看此处的反汇编代码对比此时的bootasm.s中的代码，发现是一样的





我们来查看bootasm.s的源码

cli # Disable interrupts

cld # String operations increment

xorw %ax, %ax # Segment number zero

movw %ax, %ds # -> Data Segment

movw %ax, %es # -> Extra Segment

movw %ax, %ss # -> Stack Segment

首先操作系统将各个寄存器置为0

seta20.1:  
inb $0x64, %al # 读取状态寄存器,等待8042键盘控制器闲置  
testb $0x2, %al # 判断输入缓存是否为空  
jnz seta20.1

movb $0xd1, %al # 0xd1表示写输出端口命令，参数随后通过0x60端口写入

outb %al, $0x64

seta20.2:

inb $0x64, %al

testb $0x2, %al

jnz seta20.2

movb $0xdf, %al # 通过0x60写入数据11011111 即将A20置1

outb %al, $0x60

下一段代码是开启A20  
首先我们介绍一下A20  
启用A20：为了与最早的PC向后兼容，物理地址线20在低电平，因此地址高于1MB默认回零。 此代码撤消了此操作。使得全部的32位地址线可用。

lgdt gdtdesc

这一串代码是加载GDT表

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

这一串代码是将CR0的第0位置1

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

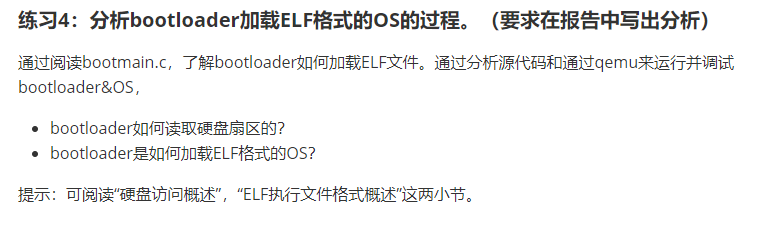
这里是将段寄存器重装

movl $0x0, %ebp

movl $start, %esp

call bootmain

最后进入bootmain继续下一步的执行



这里我们主要是分析bootmain函数

如下便是boot源码

bootmain(void) {

readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);

if (ELFHDR->e\_magic != ELF\_MAGIC) {

goto bad;

}

struct proghdr \*ph, \*eph;

ph = (struct proghdr )((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

for (; ph < eph; ph ++) {

readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

((void ()(void))(ELFHDR->e\_entry & 0xFFFFFF))();

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1);

}

分析源码我们可以得出这样的结论，bootloader中是通过调用readseg这个函数来读取硬盘的扇区的，而真正可以读取扇区的函数其实是readsect这个函数readsect每次可以读取一个扇区所以readseg会通过循环调用readsect来完成扇区的读取

readsect(void \*dst, uint32\_t secno) {

waitdisk(); // 等待硬盘就绪

// 写地址0x1f2~0x1f5,0x1f7,发出读取磁盘的命令

outb(0x1F2, 1);

outb(0x1F3, secno & 0xFF);

outb(0x1F4, (secno >> 8) & 0xFF);

outb(0x1F5, (secno >> 16) & 0xFF);

outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

outb(0x1F7, 0x20);

waitdisk();

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

}

我们来解释一下readsect的一些重要函数

outb() I/O上写入8位数据

inb() I/O上读入8位数据

在完成了磁盘的读取之后bootloader的下一个工作便是完成ELF格式文件的加载

bootmain(void) {  
…  
if (ELFHDR->e\_magic != ELF\_MAGIC) {  
goto bad;  
}  
struct proghdr \*ph, \*eph;

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

for (; ph < eph; ph ++) {

readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();

在这一段代码中我们首先要注意到的是判断条件的语句这一句的主要作用是用于判断文件的格式是否为ELF

然后是ph指针ph值存储的内容为ELF描述文件的加载位置

练习五

我们首先来看print\_stackframe函数的注释

void print\_stackframe(void) {

/\* LAB1 YOUR CODE : STEP 1 /

/ (1) call read\_ebp() to get the value of ebp. the type is (uint32\_t);

\* (2) call read\_eip() to get the value of eip. the type is (uint32\_t);

\* (3) from 0 … STACKFRAME\_DEPTH

\* (3.1) printf value of ebp, eip

\* (3.2) (uint32\_t)calling arguments [0…4] = the contents in address (unit32\_t)ebp +2 [0…4]

\* (3.3) cprintf("\n");

\* (3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.

\* (3.5) popup a calling stackframe

\* NOTICE: the calling funciton’s return addr eip = ss:[ebp+4]

\* the calling funciton’s ebp = ss:[ebp]

\*/

}

我们依据注释写出程序

void print\_stackframe(void) {

uint32\_t ebp=read\_ebp();//(1) call read\_ebp() to get the value of ebp. the type is (uint32\_t)

uint32\_t eip=read\_eip();//(2) call read\_eip() to get the value of eip. the type is (uint32\_t)

int i;

for(i=0;i<STACKFRAME\_DEPTH&&ebp!=0;i++){//(3) from 0 … STACKFRAME\_DEPTH

cprintf(“ebp:0x%08x eip:0x%08x “,ebp,eip);//(3.1)printf value of ebp, eip

uint32\_t tmp=(uint32\_t )ebp+2;

cprintf(“arg :0x%08x 0x%08x 0x%08x 0x%08x”,(tmp+0),(tmp+1),(tmp+2),(tmp+3));//(3.2)(uint32\_t)calling arguments [0…4] = the contents in address (unit32\_t)ebp +2 [0…4]

cprintf(”\n”);//(3.3) cprintf("\n");

print\_debuginfo(eip-1);//(3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.

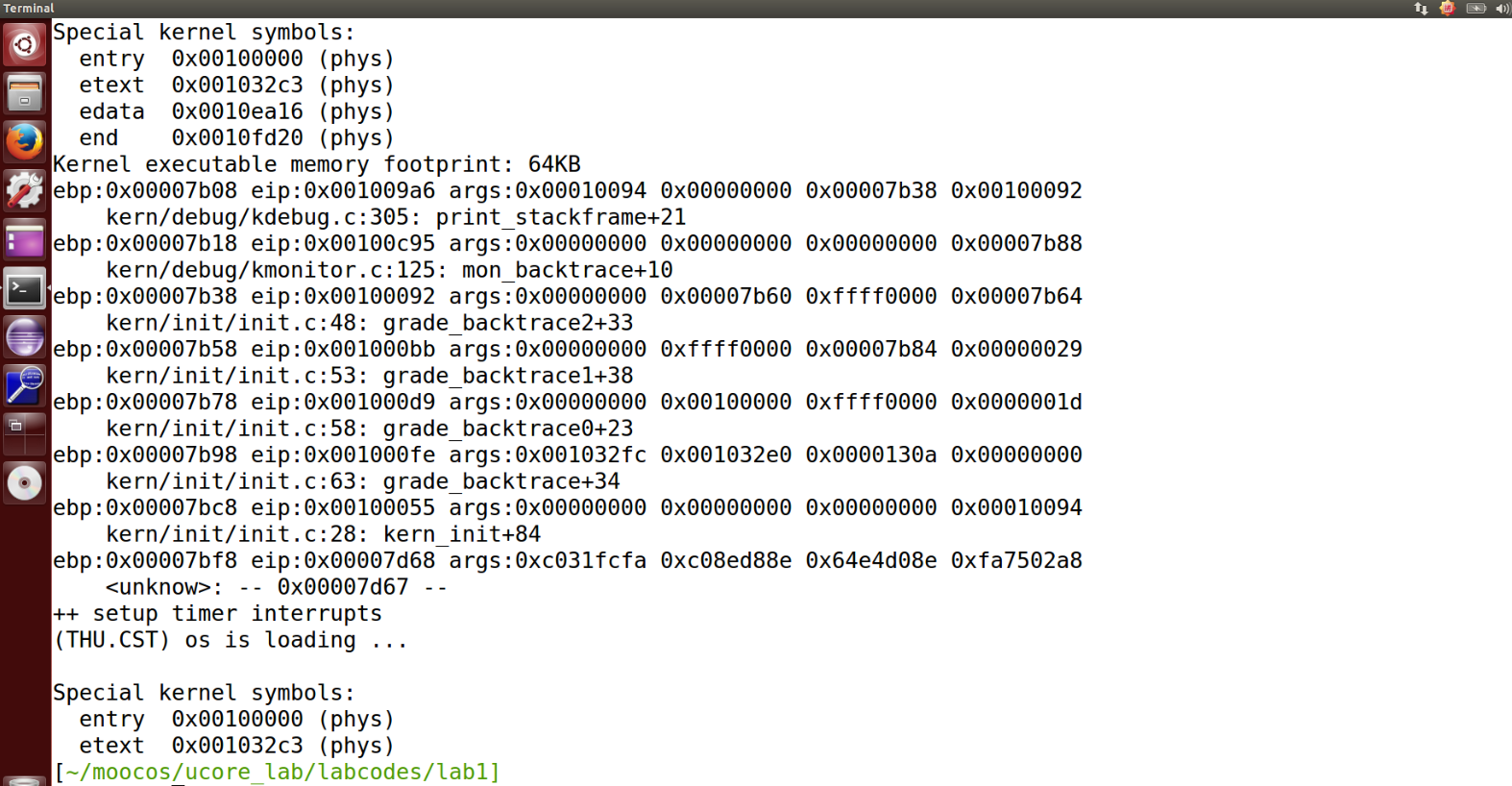
eip=((uint32\_t \*)ebp)[1];

ebp=((uint32\_t \*)ebp)[0];//(3.5) popup a calling stackframe

}

}

实验截图如下：





首先我们直接看到print\_stackframe函数的注释：

void print\_stackframe(void) {

/\* LAB1 YOUR CODE : STEP 1 \*/

/\* (1) call read\_ebp() to get the value of ebp. the type is (uint32\_t);

\* (2) call read\_eip() to get the value of eip. the type is (uint32\_t);

\* (3) from 0 .. STACKFRAME\_DEPTH

\* (3.1) printf value of ebp, eip

\* (3.2) (uint32\_t)calling arguments [0..4] = the contents in address (unit32\_t)ebp +2 [0..4]

\* (3.3) cprintf("\n");

\* (3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.

\* (3.5) popup a calling stackframe

\* NOTICE: the calling funciton's return addr eip = ss:[ebp+4]

\* the calling funciton's ebp = ss:[ebp]

\*/

}

这样我们直接根据注释以及之前的相关知识就能比较简单的编写成程序，如下所示：

void print\_stackframe(void) {

/\* LAB1 YOUR CODE : STEP 1 \*/

/\* (1) call read\_ebp() to get the value of ebp. the type is (uint32\_t);

\* (2) call read\_eip() to get the value of eip. the type is (uint32\_t);

\* (3) from 0 .. STACKFRAME\_DEPTH

\* (3.1) printf value of ebp, eip

\* (3.2) (uint32\_t)calling arguments [0..4] = the contents in address (unit32\_t)ebp +2 [0..4]

\* (3.3) cprintf("\n");

\* (3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.

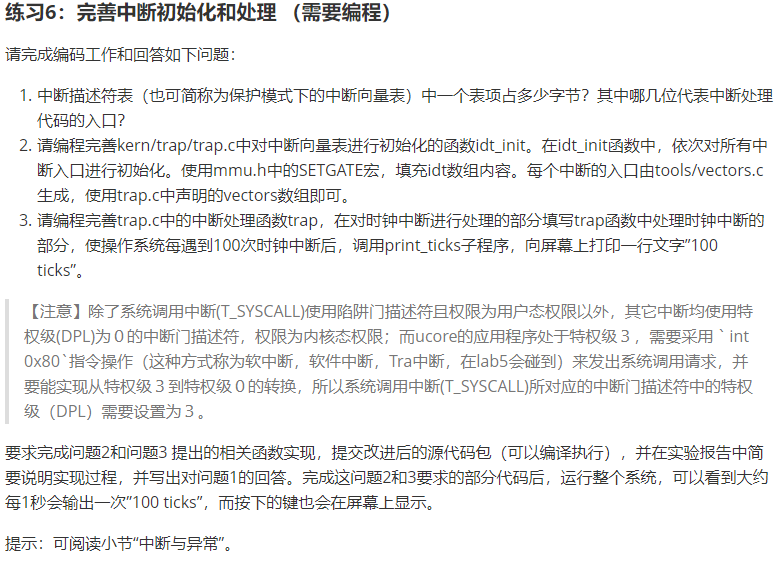
\* (3.5) popup a calling stackframe

\* NOTICE: the calling funciton's return addr eip = ss:[ebp+4]

\* the calling funciton's ebp = ss:[ebp]

\*/

}



任务目标：1.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？

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。注意除了系统调用中断(T\_SYSCALL)以外，其它中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常,权限为陷阱门描述符。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。

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。

任务1：中断描述符表一个表项占8字节。其中015位和4863位分别为offset的低16位和高16位。16~31位为段选择子。通过段选择子获得段基址，加上段内偏移量即可得到中断处理代码的入口。

任务2：这里需要实现对中断向量表的初始化

我们首先来看一下注释

void idt\_init(void) {

/\* LAB1 YOUR CODE : STEP 2 /

/ (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?

\* All ISR’s entry addrs are stored in \_\_vectors. where is uintptr\_t \_\_vectors[] ?

\* \_\_vectors[] is in kern/trap/vector.S which is produced by tools/vector.c

\* (try “make” command in lab1, then you will find vector.S in kern/trap DIR)

\* You can use “extern uintptr\_t \_\_vectors[];” to define this extern variable which will be used later.

\* (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).

\* Can you see idt[256] in this file? Yes, it’s IDT! you can use SETGATE macro to setup each item of IDT

\* (3) After setup the contents of IDT, you will let CPU know where is the IDT by using ‘lidt’ instruction.

\* You don’t know the meaning of this instruction? just google it! and check the libs/x86.h to know more.

\* Notice: the argument of lidt is idt\_pd. try to find it!

\*/

}

我们认真阅读注释得出

第一步，声明\_\_vertors[],其中存放着中断服务程序的入口地址。这个数组生成于vertor.S中。

第二步，填充中断描述符表IDT。

第三部，加载中断描述符表。

得到代码为

void idt\_init(void) {

extern uintptr\_t \_\_vectors[];//声明\_\_vertors[]

int i;

for(i=0;i<256;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);//使用lidt指令加载中断描述符表

}

任务3：

这里根据指导书查看函数trap\_dispatch，发现print\_ticks()子程序已经被实现了，所以我们直接进行判断输出即可

…

…

case IRQ\_OFFSET + IRQ\_TIMER:

ticks ++; //每一次时钟信号会使变量ticks加1

if (ticks==TICK\_NUM) {//TICK\_NUM已经被预定义成了100，每到100便调用print\_ticks()函数打印

ticks-=TICK\_NUM;

print\_ticks();

}

break;