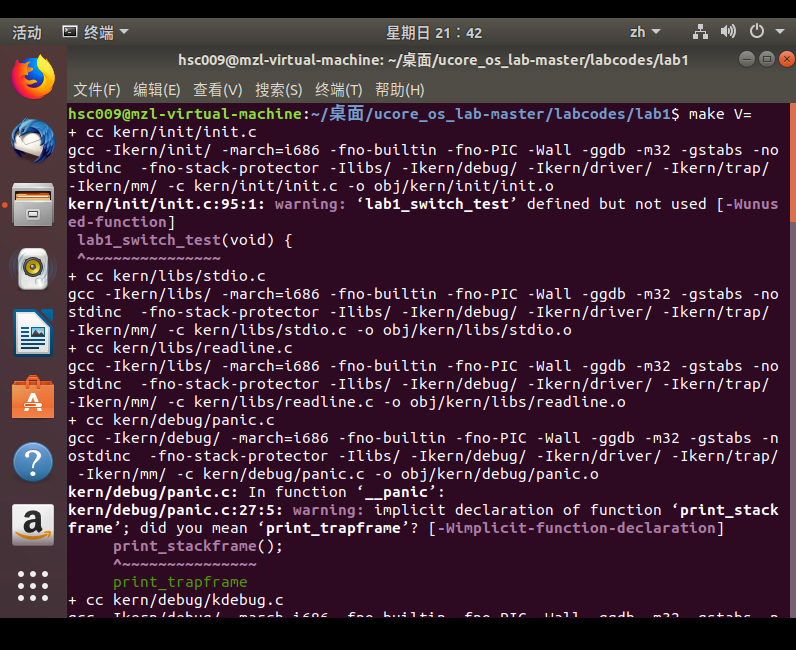
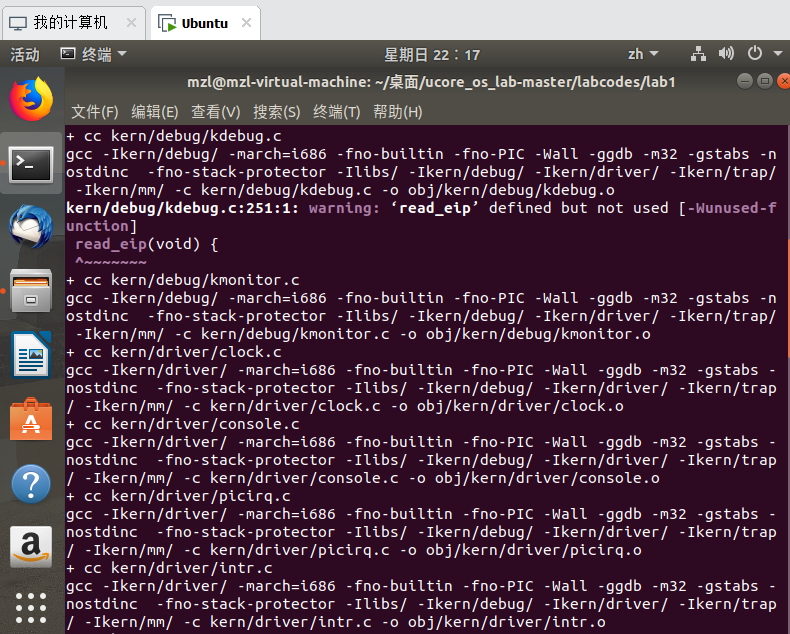
**ucoreOS实验报告**

**黄舒成 4042017009**

**练习1：理解通过make生成执行文件的过程**

问题1：操作系统镜像文件ucore.img是如何一步一步生成的？





由以上过程可知：

编译16个内核文件，构建出内核bin/kernel

生成 bin/bootblock 引导程序

编译bootasm.S，bootmain.c，链接生成obj/bootblock.o

编译sign.c生成sign.o工具

使用sign.o工具规范化bootblock.o生成bin/bootblock引导扇区

生成 ucore.img 虚拟磁盘

dd初始化ucore.img为5120000 bytes，内容为0的文件

dd拷贝bin/bootblock到ucore.img第一个扇区

dd拷贝bin/kernel到ucore.img第二个扇区往后的空间

问题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？

查看sign.c的内容：

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <sys/stat.h>

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

struct stat st;

// 输入状态判断

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;

}

// 问题1中输出的文件大小

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

// 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

if (st.st\_size > 510) {

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

return -1;

}

// 多余位用0填充

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;

// 写入结束位

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;

}

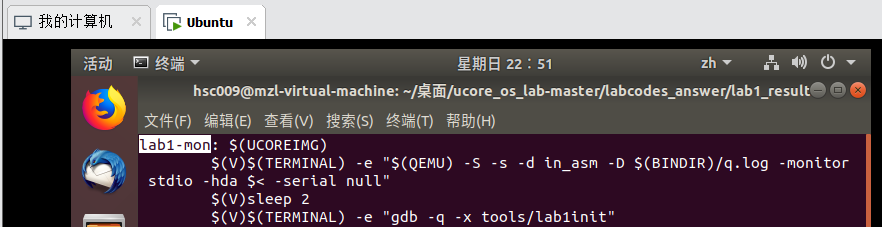
由以上代码可知，硬盘主引导扇区特征为：

1.大小为512字节，空余部分用0填充

2.文件内容不超过510 bytes

3.最后2 bytes为0x55 0xAA

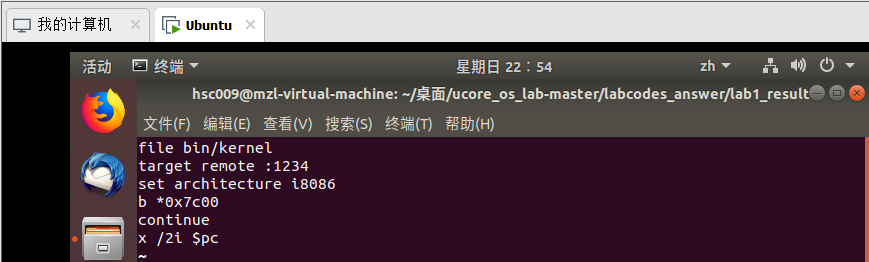
**练习2：使用qemu执行并调试lab1中的软件。**



可以看出这条命令干了两件事

第一个是让 qemu 把它执行的指令给记录下来，放到 q.log 这个地方

第二个是和 gdb 结合来调试正在执行的 Bootloader

使用命令：less tools/lab1init

第一条指令是加载 bin/kernel。（加载符号信息，事实上是ucore的信息）

第二条指令是与 qemu 进行连接，通过这个TRP进行连接

刚开始的时候，BIOS是进入8086的16位实模式方式，一直到0x7c00。在BIOS这个阶段，启动，最后把Bootloader加载进去，把控制权交给Bootloader，那么Bootloader第一条指令就是在0x7c00处，所以我们在这个地方设置一个断点，break 0x7c00

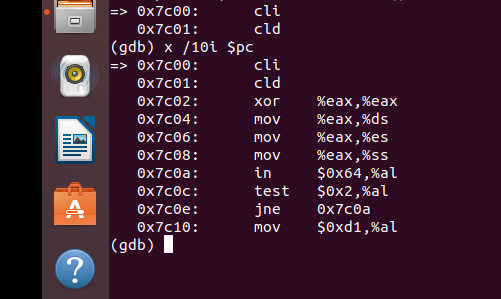
然后让这个系统继续运行，那么我们就会看到它会在这个断点处停下来，那我们可以把相应的这个指令给打印出来。

最后一条指令的意思是把PC(也就是EIP，即指令指针寄存器)，它存在当前正在执行这个指令的地址，  
那么x是显示的意思，／2i是显示两条，i是指令。

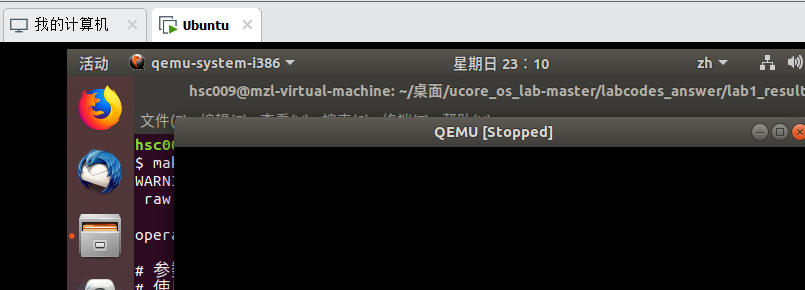
输入指令make lab1-mon

可以看到，qemu 已经断开

查看当前指令



执行make debug:



查看 boot/bootasm.S 文件

调试发现0x7C00为主引导程序的入口地址，代码与bootasm.S一致

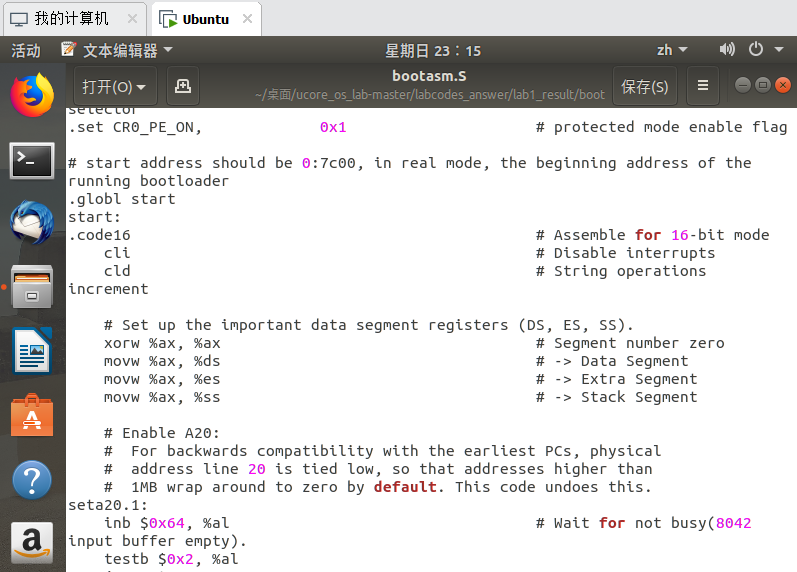
使用ni可进行单步调试

我们再对 tools/gdbinit 做如下修改：

1. file bin/kernel
2. set architecture i8086
3. target remote :1234
4. b kern\_init
5. continue

执行make debug

在内核入口处增加断点，可以看到代码停在kern\_init函数



**练习3：分析bootloader进入保护模式的过程**

Bootloader 完成了一些最基本的功能，比如 它能够把80386的保护模式给开启，使得现在的软件进入了一个32位的寻址空间，这就是我们的寻址方式发生了变化。为了做好这一步，它需要干如下一些事情：

开启A20

初始化GDT表(全局描述符表)

使能和进入保护模式

保护模式和分段机制

(1) 实模式

在bootloader接手BIOS的工作后，当前的PC系统处于实模式（16位模式）运行状态，在这种状态下软件可访问的物理内存空间不能超过1MB，且无法发挥Intel 80386以上级别的32位CPU的4GB内存管理能力。

实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样，用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。通过修改A20地址线可以完成从实模式到保护模式的转换。有关A20的进一步信息可参考附录“关于A20 Gate”。

(2) 保护模式

只有在保护模式下，80386的全部32根地址线有效，可寻址高达4G字节的线性地址空间和物理地址空间，可访问64TB（有2^14个段，每个段最大空间为2^32字节）的逻辑地址空间，可采用分段存储管理机制和分页存储管理机制。这不仅为存储共享和保护提供了硬件支持，而且为实现虚拟存储提供了硬件支持。通过提供4个特权级和完善的特权检查机制，既能实现资源共享又能保证代码数据的安全及任务的隔离。

(3) 分段存储管理机制

只有在保护模式下才能使用分段存储管理机制。分段机制将内存划分成以起始地址和长度限制这两个二维参数表示的内存块，这些内存块就称之为段（Segment）。编译器把源程序编译成执行程序时用到的代码段、数据段、堆和栈等概念在这里可以与段联系起来，二者在含义上是一致的。

分段机涉及4个关键内容：逻辑地址、段描述符（描述段的属性）、段描述符表（包含多个段描述符的“数组”）、段选择子（段寄存器，用于定位段描述符表中表项的索引）。转换逻辑地址（Logical Address,应用程序员看到的地址）到物理地址（Physical Address, 实际的物理内存地址）分以下两步：

[1] 分段地址转换：CPU把逻辑地址（由段选择子selector和段偏移offset组成）中的段选择子的内容作为段描述符表的索引，找到表中对应的段描述符，然后把段描述符中保存的段基址加上段偏移值，形成线性地址（Linear Address）。如果不启动分页存储管理机制，则线性地址等于物理地址。 [2] 分页地址转换，这一步中把线性地址转换为物理地址。（注意：这一步是可选的，由操作系统决定是否需要。在后续试验中会涉及。

上述转换过程对于应用程序员来说是不可见的。线性地址空间由一维的线性地址构成，线性地址空间和物理地址空间对等。线性地址32位长，线性地址空间容量为4G字节。分段地址转换的基本过程如下图所示。

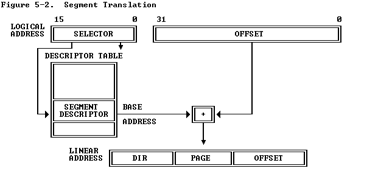
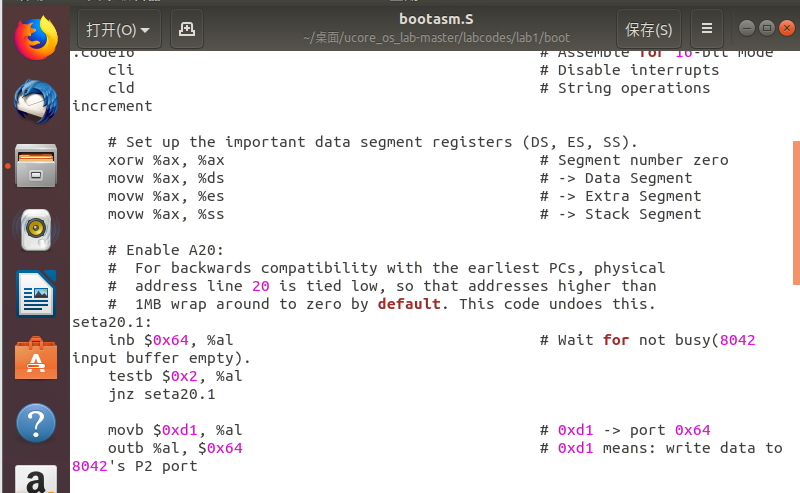


图1 分段地址转换基本过程

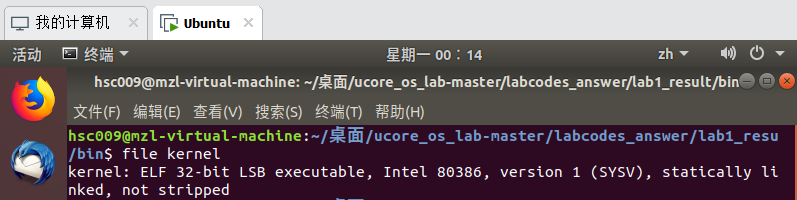
分段存储管理机制需要在启动保护模式的前提下建立。从上图可以看出，为了使得分段存储管理机制正常运行，需要建立好段描述符和段描述符表（参看bootasm.S，mmu.h，pmm.c）。

开启A20，初始化gdt后，将控制寄存器CR0的PE(bit0)置为1即可。

1. movl %cr0, %eax
2. orl 0x1, %eax
3. movl %eax, %cr0



**练习4：分析bootloader加载ELF格式的OS的过程**



执行完bootasm.S后，系统进入保护模式， 进行bootmain.c开始加载OS

* 定义ELF头指针，指向0x10000
* 读取8个扇区大小的ELF头到内存地址0x10000
* 校验ELF header中的模数，判断是否为0x464C457FU
* 读取ELF header中的程序段到内存中
* 跳转到操作系统入口

bootloader如何读取硬盘扇区的

bootloader进入保护模式并载入c程序bootmain

bootmain中readsect函数完成读取磁盘扇区的工作，函数传入一个指针和一个uint\_32类型secno，函数将secno对应的扇区内容拷贝至指针处

调用waitdisk函数等待地址0x1F7中低8、7位变为0,1，准备好磁盘

向0x1F2输出1，表示读1个扇区，0x1F3输出secno低8位，0x1F4输出secno的8~15位，0x1F5输出secno的16~23位，0x1F6输出0xe+secno的24~27位，第四位0表示主盘，第六位1表示LBA模式，0x1F7输出0x20

调用waitdisk函数等待磁盘准备好

调用insl函数把磁盘扇区数据读到指定内存

bootloader是如何加载ELF格式的OS

bootloader通过bootmain函数完成ELF格式OS的加载。

调用readseg函数从kernel头读取8个扇区得到elfher

判断elfher的成员变量magic是否等于ELF\_MAGIC，不等则进入bad死循环

相等表明是符合格式的ELF文件，循环调用readseg函数加载每一个程序段

调用elfher的入口指针进入OS

**练习5：实现函数调用堆栈跟踪函数**

要完成实验首先必须了解函数栈的构建过程

* ebp为基址指针寄存器
* esp为堆栈指针寄存器(指向栈顶)
* ebp寄存器处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原ebp入栈后的栈顶），从该地址为基准，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，而该地址处又存储着上一层函数调用时的ebp值  
  举一个实际的例子查看ebp与esp两个寄存器如何构建出完整的函数栈：  
  leave等同于movl %ebp, %esp，popl %ebp两条指令
* int g(int x) {
* return x + 10;
* }
* int f(int x) {
* return g(x);
* }
* int main(void) {
* return f(20) + 8;
* }

使用 read\_ebp()， read\_eip()函数获得ebp，eip的值

循环：

1. 输出ebp，eip的值

2. 输出4个参数的值，其中第一个参数的地址为ebp+8，依次加4得到下一个参数的地址

3. 更新ebp，eip，其中新的ebp的地址为ebp，新的eip的地址为ebp+4，即返回地址

4. ebp为0时表明程序返回到了最开始初始化的函数，ebp=0为循环的退出条件

void print\_stackframe(void){

uint32\_t ebp = read\_ebp(), eip = read\_eip();

int i, j;

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

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

// ebp向上移动4个字节为eip

uint32\_t \*args = (uint32\_t \*)ebp + 2;

// 再向上每4个字节都为输入的参数(这里只是假设4个参数，做实验)

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

cprintf("0x%08x ", args[j]);

}

cprintf("\n");

print\_debuginfo(eip - 1);

// ebp指针指向的位置向上一个地址为上一个函数的eip

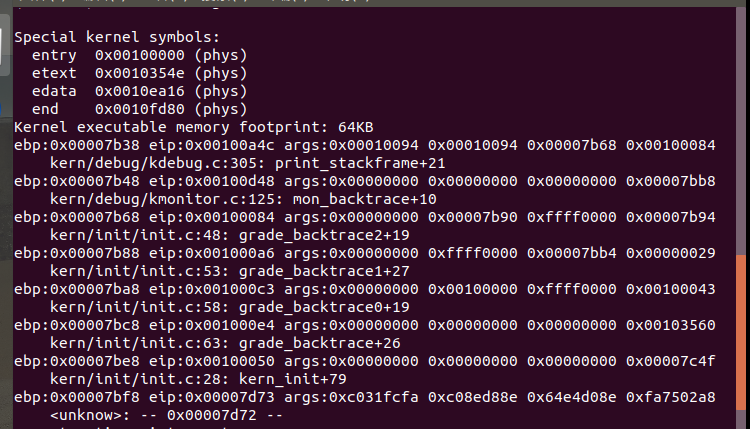
eip = ((uint32\_t \*)ebp)[1];

// ebp指针指向的位置存储的上一个ebp的地址

ebp = ((uint32\_t \*)ebp)[0];

}

}



**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？

当CPU收到中断时，会查找对应的中断描述符表（IDT），确定对应的中断服务例程。

IDT是一个8字节的描述符数组，IDT 可以位于内存的任意位置，CPU 通过IDT寄存器（IDTR）的内容来寻址IDT的起始地址。指令LIDT和SIDT用来操作IDTR。

DT的一个表项如下，4个字节分别存储offset的高位地址、段选择子和offset低位地址

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。

void idt\_init(void){

extern uintptr\_t \_\_vectors[];

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);

}

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”。

void trap(struct trapframe \*tf) {

*// dispatch based on what type of trap occurred*

trap\_dispatch(tf);

}

static void trap\_dispatch(struct trapframe \*tf) {

char c;

switch (tf->tf\_trapno) {

case IRQ\_OFFSET + IRQ\_TIMER:

ticks++;

if(ticks == TICK\_NUM) {

print\_ticks();

ticks = 0;

}

break;

case IRQ\_OFFSET + IRQ\_COM1:

c = cons\_getc();

cprintf("serial [%03d] %c\n", c, c);

break;

case IRQ\_OFFSET + IRQ\_KBD:

c = cons\_getc();

cprintf("kbd [%03d] %c\n", c, c);

break;

*//LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.*

case T\_SWITCH\_TOU:

case T\_SWITCH\_TOK:

panic("T\_SWITCH\_\*\* ??\n");

break;

case IRQ\_OFFSET + IRQ\_IDE1:

case IRQ\_OFFSET + IRQ\_IDE2:

*/\* do nothing \*/*

break;

default:

*// in kernel, it must be a mistake*

if ((tf->tf\_cs & 3) == 0) {

print\_trapframe(tf);

panic("unexpected trap in kernel.\n");

}

}

}