# 实验一

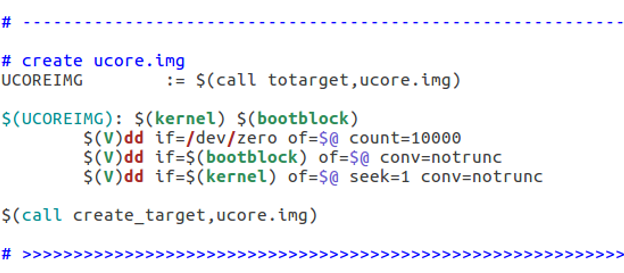
## 练习1：理解通过make生成执行文件的过程。

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：

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中与ucore.img的相关代码段：



### 相关代码段的解释：

$(UCORIMG): $(kernel) $(bootblock)

表示ucore.img依赖于kernel和bootblock这两个函数文件。

$(V)dd if=/dev/zero of=$@ count=10000:

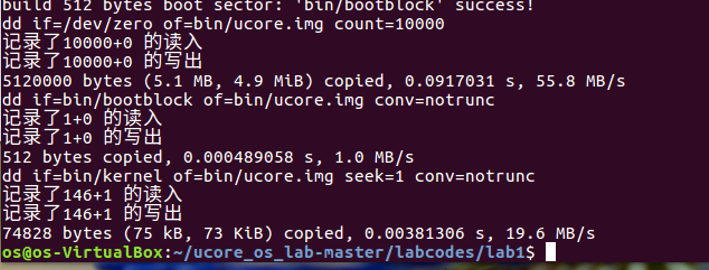
dd是一个连接文件，if=/dev/zero表示inputfile为空，of=$@表示outputfile为$@，  
$@表示的是上一个变量，即ucore.img，count=10000表示申请空白的大小为10000的空间，用来存放编译的ucore.img。

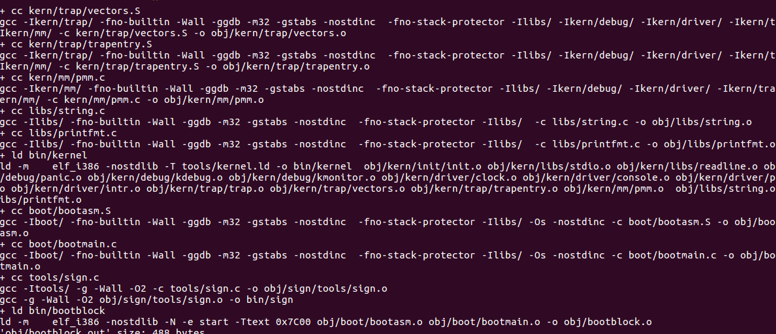
if = $(bootblock) of=$@

if = $(kernel) of=$@

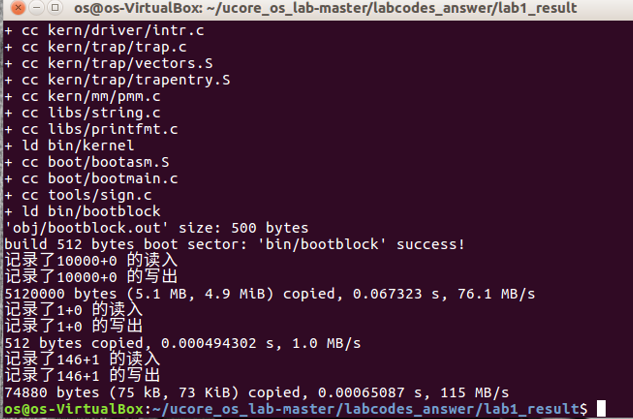
表示将kernel和bootblock放到ucore.img中

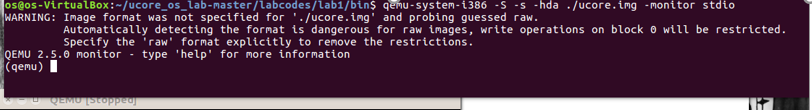
### ucore.img编译过程：

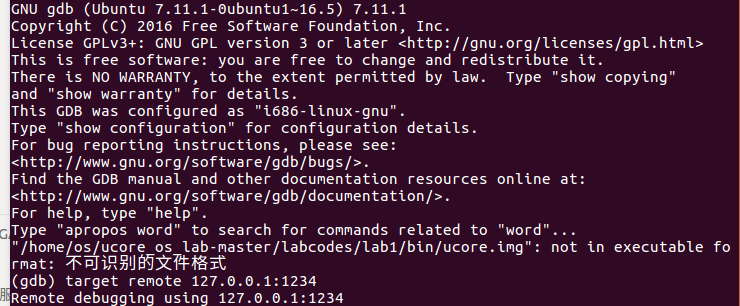




## 练习2：使用qemu执行并调试lab1中的软件。







## 练习3：分析bootloader进入保护模式的过程

### 为何要开启A20,以及如何开启A20

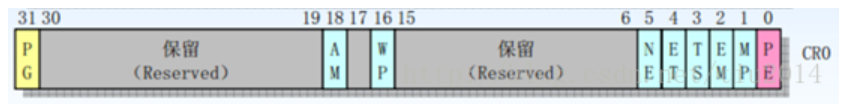
首先关于A20，我们通过查询资料以及说明文档可以知道早期的8086CPU所提供的地址线只有20位，所以可寻址空间为0~2^20(1MB)，但是8086的数据处理位宽16位，无法直接访问1M的地址空间，所以8086提供了段地址加偏移地址的转换机制。PC的寻址结构是segment:offset，segment和offset都是16位寄存器，最大值是0ffffh，所以换算成物理地址的计算方法是把segment左移4位，再加上offset，所以segment:offset所能表示的最大为10ffefh，而这个地址超过了1M，但是超过1M会发生“回卷”的现象不会报错，但是从下一代的80286开始，地址线成为了24位，所能访问的地址空间超过了1M，此时寻址超过1M时会报错，出现了向下不兼容，所以为了解决这个问题采用了A20机制。A20 Gate，将A20地址线控制器 和 键盘控制器的一个输出进行AND操作，这样来控制A20地址线的打开与关闭，所以在实模式下，需要确保A20开关处于关闭状态，这样可以防止访问大于1M的地址空间是在保护模式下，我们需要访问更大的内存空间，所以需要将A20的开关打开，如果在保护模式下，A20的开关未打开的话，此时我们只能访问奇数兆的内存，即只能访问0—1M，2—3M，4—5M……，所以如果我们要进入保护模式，首先就需要把A20开关给打开。

### 如何初始化GDT表

接下来我们需要了解下GDT表（全局描述符表），在整个操作系统中我们只有一张GDT表，GDT可以放在内存的任意位置，但是CPU必须知道GDT的入口，在Intel里面有一个专门的寄存器GDTR用来存放GDT的入口地址，程序员将GDT设定在内存的某个位置之后，可以通过LGDT指令将GDT的入口地址加载到该寄存器里面，以后CPU就可以通过GDTR来访问GDT了。

### 如何进入保护模式

寄存器CR0



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，如果要开启分页机制的话我们需要同时置位PE和PG。有了初步了解之后我们便知道的开启保护模式的相关操作，首先开启A20 Gate，其次加载全局描述符表GDT，最后只需要将CR0寄存器的最低位置为1即可。

我们通过观察代码来查看UCore具体是如何实现相应的操作的：

# 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 # 等待键盘控制器8042 0x64 端口 空闲，64h端口中的状态寄存器的值为0x2

jnz seta20.1 # 忙的话一直等待

movb $0xd1, %al # 等到64h空闲之后我们会写入0xd1　0xd1 -> port 0x64

outb %al, $0x64 # 表明我们要向60h里面写入数据， 0xd1 means: write data to 8042's P2 port

seta20.2:

inb $0x64, %al # 等待64h端口空闲　Wait for not busy(8042 input buffer empty).

testb $0x2, %al # 64h端口中的状态寄存器的值为0x2

jnz seta20.2 # 忙的话一直等待

movb $0xdf, %al # 0xdf -> port 0x60 等到空闲之后，我们将0xdf写入60h端口，至此来打开A20开关。

outb %al, $0x60 # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1 (第20位)

首先是开启A20，根据上文我们知道需要将第20位为1即可，但是我们需要知道在UCore里是如何将A20置为1的。根据说明书我们可以知道，A20地址线由键盘控制器8042进行控制，我们的A20所对应的是8042里面的P21引脚，所以问题就变成了我们需要将P21引脚置1。

对于8042芯片来说，有两个端口地址60h和64h。对于这两个端口来说，0x64用来发送一个键盘控制命令，0x60用来传递参数，所以将P21引脚置1的操作就变成了，我们首先利用0x64端口传递一个写入的指令，然后由0x60端口读进去相应的参数来将P21置1。由以下的资料我们可以知道，我们首先要先向 64h 发送 0xd1 的指令，写之前需要等待键盘控制器8042空闲，可以通过判断 64h端口中的状态寄存器的值0x2,来判断是否空闲。然后向 60h 发送 0xdf 的指令。

#include <asm.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

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader

.globl start

start:

.code16 # Assemble for 16-bit mode

cli # Disable interrupts

cld # String operations increment

# 初始化 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

# 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).

testb $0x2, %al # 等待键盘控制器8042 0x64 端口 空闲，64h端口中的状态寄存器的值为0x2

jnz seta20.1 # 忙的话一直等待

movb $0xd1, %al # 等到64h空闲之后我们会写入0xd1, 0xd1 -> port 0x64

outb %al, $0x64 # 表明我们要向60h里面写入数据，0xd1 means: write data to 8042's P2 port

seta20.2:

inb $0x64, %al # 等待64h端口空闲， Wait for not busy(8042 input buffer empty).

testb $0x2, %al # 64h端口中的状态寄存器的值为0x2

jnz seta20.2 # 忙的话一直等待

movb $0xdf, %al # 等到空闲之后，我们将0xdf写入60h端口，至此来打开A20开关，0xdf -> port 0x60

outb %al, $0x60 # 0xdf = 11011111, means set P2's A20 bit(the 1 bit(第20位)) to 1

# 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.

# 2 load gdt 加载GDT全局描述符 在后面可以看到

lgdt gdtdesc

# 使能和进入保护模式

movl %cr0, %eax # 首先将cr0寄存器里面的内容取出来

orl $CR0\_PE\_ON, %eax # 进行一个或操作, 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

movl %eax, %cr0 # 最后将得到的结果再写回 cr0中

# 只是进行了分段，没有开启分页机制，如果要开启分页机制的话我们需要同时置位PE和PG。

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

# Switches processor into 32-bit mode.

# 3 最后通过一个长跳转指令正式进入保护模式。

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32 # Assemble for 32-bit mode

protcseg:

# Set up the protected-mode data segment registers 初始化保护模式的数据段寄存器

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. The stack region is from 0--start(0x7c00)

movl $0x0, %ebp # 栈底指针

movl $start, %esp # 栈顶指针

call bootmain # 调用bootmain函数在bootmain.c中，进行　加载ELF格式的操作系统OS

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

spin:

jmp spin

# Bootstrap GDT

.p2align 2 # force 4 byte alignment

gdt: # GDT表的入口地址 GDT全局描述符表由三个全局描述符组成

SEG\_NULLASM # null seg 第一个均为空描述符

SEG\_ASM(STA\_X|STA\_R, 0x0, 0xffffffff) # code seg for bootloader and kernel 第二个为代码段描述符

SEG\_ASM(STA\_W, 0x0, 0xffffffff) # data seg for bootloader and kernel 第三个为数据段描述符

# gdt全局描述符表　 它里面有两个参数

gdtdesc:

.word 0x17 # 表示的是GDT表的大小 # sizeof(gdt) - 1

.long gdt # 表示的是GDT表的入口地址 # address gdt