1.1操作系统镜像文件ucore,img是如何一步一步生成的？（需要比较详细的解释Makfile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）

生成ucore.img的相关代码为

| $(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

|

| 为了生成ucore.img，首先需要生成bootblock、kernel

|

bin/bootblock

生成bootblock的相关代码为

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

| | @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)

为了生成bootblock，首先需要生成bootasm.o、bootmain.o、sign

| |

| |> obj/boot/bootasm.o, obj/boot/bootmain.o

| | | 生成bootasm.o,bootmain.o的相关makefile代码为

| | | bootfiles = $(call listf\_cc,boot)

| | | $(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),\

| | | $(CFLAGS) -Os -nostdinc))

| | | 实际代码由宏批量生成

| | |

| | | 生成bootasm.o需要bootasm.S

| | | 实际命令为

| | | gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs \

| | | -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc \

| | | -c boot/bootasm.S -o obj/boot/bootasm.o

| | | 其中关键的参数为

| | | -ggdb 生成可供gdb使用的调试信息。这样才能用qemu+gdb来调试bootloader or ucore。

| | | -m32 生成适用于32位环境的代码。我们用的模拟硬件是32bit的80386，所以ucore也要是32位的软件。

| | | -gstabs 生成stabs格式的调试信息。这样要ucore的monitor可以显示出便于开发者阅读的函数调用栈信息

| | | -nostdinc 不使用标准库。标准库是给应用程序用的，我们是编译ucore内核，OS内核是提供服务的，所以所有的服务要自给自足。

| | | -fno-stack-protector 不生成用于检测缓冲区溢出的代码。这是for 应用程序的，我们是编译内核，ucore内核好像还用不到此功能。

| | | -Os 为减小代码大小而进行优化。根据硬件spec，主引导扇区只有512字节，我们写的简单bootloader的最终大小不能大于510字节。

| | | -I<dir> 添加搜索头文件的路径

| | |

| | | 生成bootmain.o需要bootmain.c

| | | 实际命令为

| | | gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc \

| | | -fno-stack-protector -Ilibs/ -Os -nostdinc \

| | | -c boot/bootmain.c -o obj/boot/bootmain.o

| | | 新出现的关键参数有

| | | -fno-builtin 除非用\_\_builtin\_前缀，

| | | 否则不进行builtin函数的优化

| |

| |> bin/sign

| | | 生成sign工具的makefile代码为

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

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

| | |

| | | 实际命令为

| | | 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

| |

| | 首先生成bootblock.o

| | ld -m elf\_i386 -nostdlib -N -e start -Ttext 0x7C00 \

| | obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

| | 其中关键的参数为

| | -m <emulation> 模拟为i386上的连接器

| | -nostdlib 不使用标准库

| | -N 设置代码段和数据段均可读写

| | -e <entry> 指定入口

| | -Ttext 制定代码段开始位置

| |

| | 拷贝二进制代码bootblock.o到bootblock.out

| | objcopy -S -O binary obj/bootblock.o obj/bootblock.out

| | 其中关键的参数为

| | -S 移除所有符号和重定位信息

| | -O <bfdname> 指定输出格式

| |

| | 使用sign工具处理bootblock.out，生成bootblock

| | bin/sign obj/bootblock.out bin/bootblock

|

|> bin/kernel

| | 生成kernel的相关代码为

| | $(kernel): tools/kernel.ld

| | $(kernel): $(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)

| |

| | 为了生成kernel，首先需要 kernel.ld init.o readline.o stdio.o kdebug.o

| | kmonitor.o panic.o clock.o console.o intr.o picirq.o trap.o

| | trapentry.o vectors.o pmm.o printfmt.o string.o

| | kernel.ld已存在

| |

| |> obj/kern/\*/\*.o

| | | 生成这些.o文件的相关makefile代码为

| | | $(call add\_files\_cc,$(call listf\_cc,$(KSRCDIR)),kernel,\

| | | $(KCFLAGS))

| | | 这些.o生成方式和参数均类似，仅举init.o为例，其余不赘述

| |> obj/kern/init/init.o

| | | 编译需要init.c

| | | 实际命令为

| | | gcc -Ikern/init/ -fno-builtin -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

| |

| | 生成kernel时，makefile的几条指令中有@前缀的都不必需

| | 必需的命令只有

| | 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

| | 其中新出现的关键参数为

| | -T <scriptfile> 让连接器使用指定的脚本

|

| 生成一个有10000个块的文件，每个块默认512字节，用0填充

| dd if=/dev/zero of=bin/ucore.img count=10000

|

| 把bootblock中的内容写到第一个块

| dd if=bin/bootblock of=bin/ucore.img conv=notrunc

|

| 从第二个块开始写kernel中的内容

| dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc

```

练习2：使用qemu执行并调试lab1中的软件。（要求在报告中简 要写出练习过程）

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。
2. 修改lab1/tools/gdbinit

方法如下

set architecture i8086

target remote :1234

（2）在lab1目录下，执行 make debug

（3）看到dgb的调试界面后，在gdb调试界面执行以下输入 si

完成以上几步就可以单步跟踪BIOS了

1. 在初始化位置0x7c00设置实地址断点,测试断点正常。

在tools/gdbinit结尾加上

set architecture i8086 //设置当前调试的CPU是8086

b \*0x7c00 //在0x7c00处设置断点

c //continue简称，表示继续执行

x /2i $pc //显示当前eip处的汇编指令

set architecture i386 //设置当前调试的CPU是80386

运行“make debug”便可以清楚看到

1. 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。

在tools/gdbinit后面加上

‘’’b \*0x7c00

c

x /10i $pc’’’

便可以在q.log中读到"call bootmain"前执行的命令,且与bootasm.S和bootblock.asm中的代码相同。

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：

为何开启A20，以及如何开启A20 如何初始化GDT表 如何使能和进入保护模式

从`%cs=0 $pc=0x7c00`，进入后

（1）首先清理环境：包括将flag置0和将段寄存器置0

（2）开启A20：通过将键盘控制器上的A20线置于高电位，全部32条地址线可用，

可以访问4G的内存空间。

（3）初始化GDT表：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

进入保护模式：通过将cr0寄存器PE位置1便开启了保护模式

```

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

```

（4）通过长跳转更新cs的基地址

（5）设置段寄存器，并建立堆栈

```

movw $PROT\_MODE\_DSEG, %ax

movw %ax, %ds

movw %ax, %es

movw %ax, %fs

movw %ax, %gs

movw %ax, %ss

movl $0x0, %ebp

movl $start, %esp

```

转到保护模式完成，进入boot主方法

```

call bootmain

```