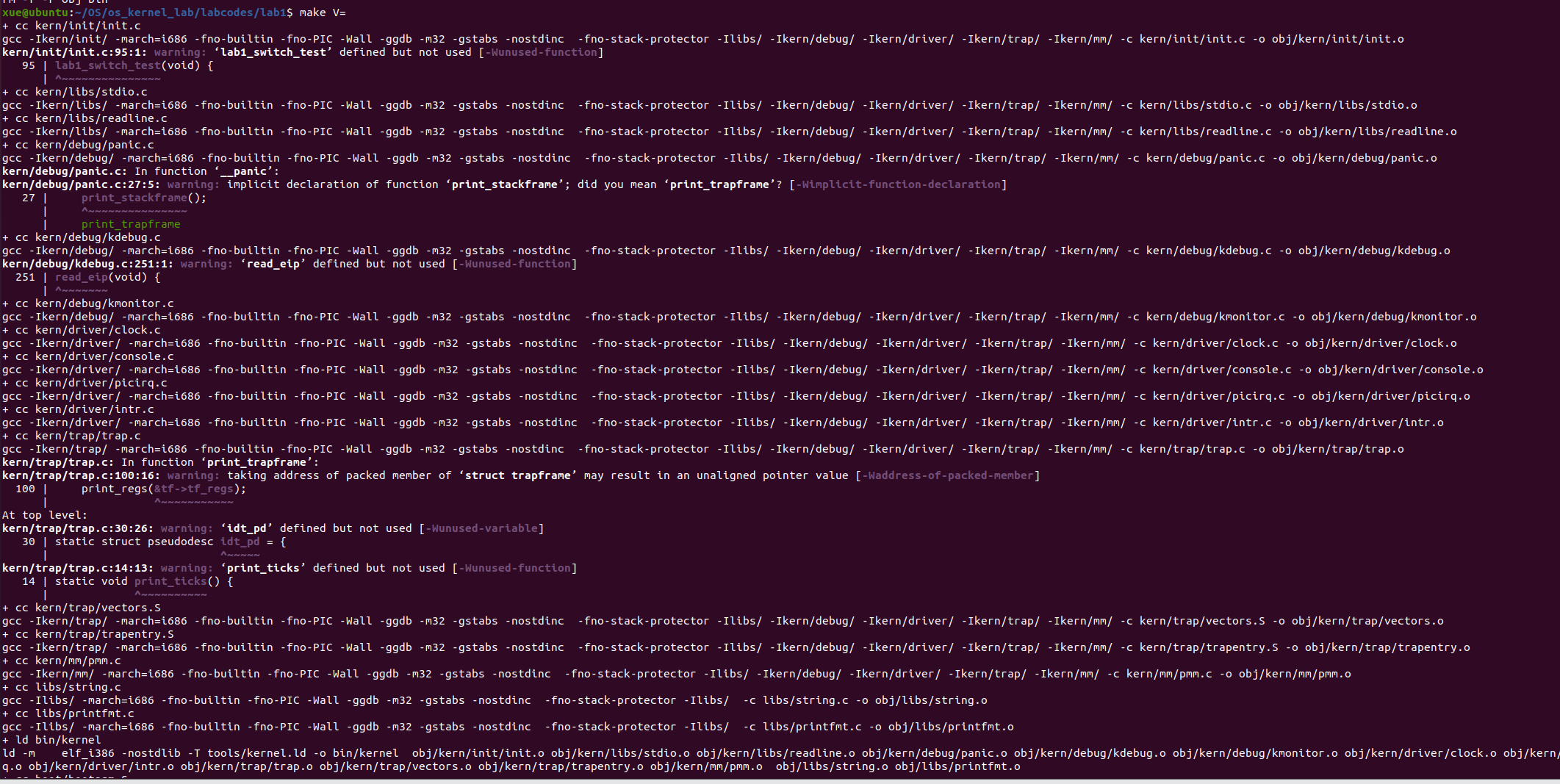
## 练习一：理解通过make生成执行文件的过程

**1、查看make具体指令**

**（1）进入lab1文件，首先使用make clean 命令清除，删除之前执行make命令产生的object文件【不然会出错！】**



1. **接着使用命令make V=，查看在生成kernel和bootblock执行的具体命令**



+ 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

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

95 | lab1\_switch\_test(void) {

| ^~~~~~~~~~~~~~~~

+ 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

+ 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

+ 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

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

kern/debug/panic.c:27:5: warning: implicit declaration of function ‘print\_stackframe’; did you mean ‘print\_trapframe’? [-Wimplicit-function-declaration]

27 | print\_stackframe();

| ^~~~~~~~~~~~~~~~

| print\_trapframe

+ 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

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

251 | 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/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

+ 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

+ 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/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

+ 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: In function ‘print\_trapframe’:

kern/trap/trap.c:100:16: warning: taking address of packed member of ‘struct trapframe’ may result in an unaligned pointer value [-Waddress-of-packed-member]

100 | print\_regs(&tf->tf\_regs);

| ^~~~~~~~~~~~

At top level:

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

30 | static struct pseudodesc idt\_pd = {

| ^~~~~~

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

14 | static void print\_ticks() {

| ^~~~~~~~~~~

+ 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/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/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/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

+ 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

+ ld bin/kernel

ld -m elf\_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/stdio.o obj/kern/libs/readline.o obj/kern/debug/panic.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/picirq.o obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmm.o obj/libs/string.o obj/libs/printfmt.o

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

'obj/bootblock.out' size: 496 bytes

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

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

10000+0 records in

10000+0 records out

5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0182781 s, 280 MB/s

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

1+0 records in

1+0 records out

512 bytes copied, 6.8477e-05 s, 7.5 MB/s

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

154+1 records in

154+1 records out

78916 bytes (79 kB, 77 KiB) copied, 0.000152803 s, 516 MB/s

**涉及到的命令及解释：**

ld：ld命令是二进制工具集GNU Binutils的一员，是GNU链接器，用于将目标文件与库链接为可执行程序或库文件

dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换。

-b <input-format>:指定目标代码输入文件的格式

-Bstatic:只使用静态库

-Bdynamic:只使用动态库

-Bsymbolic:把引用捆绑到共享库中的全局符号

-c <MRI-commandfile>,--mri-script=<MRI-commandfile>:为与MRI链接器兼容，ld接受由MRI命令语言编写的脚本文件

--cref:创建跨引用表

-d,-dc,-dp:即使指定了可重定位的输出文件（使用-r），也会为公共符号分配空间。脚本命令“FORCE\_COMMON\_ALLOCATION”具有相同的效果

-defsym:在输出文件中创建指定的全局符号

-demangle:在错误消息中还原符号名称

-e <entry>:使用指定的符号作为程序的初始执行点

-E,--export-dynamic:对于ELF格式文件，创建动态链接的可执行文件时，把所有符号添加到动态符号表

-f <name>,--auxiliary=<name>:对于ELF格式共享对象，设置 DT\_AUXILIARY 名称

-F <name>,--filter=<name>:对于ELF格式共享对象，设置 DT\_FILTER 名称。这告诉动态链接器，正在创建的共享对象的符号表应该用作共享对象名称的符号表的筛选器。

-g:被忽略。用于提供和其他工具的兼容性

-h:对于ELF格式共享对象，设置 DT\_SONAME 名称

-I<file>,--dynamic-linker=<file>:指定动态链接器。这仅在生成动态链接的ELF可执行文件时才有意义。默认的动态链接器通常是正确的，除非您知道正在做什么，否则不要使用该选项。

-l <namespec>,--library=<namespec>:把指定的库文件添加到要链接的文件清单

-L <searchdir>,--library-path=searchdir:把指定的路径添加添加到搜索库的目录清单

-M,--print-map:显示链接映射，用于诊断目的

-Map=<mapfile>: 将链接映射输出到指定的文件

-m <emulation>: 模拟指定的链接器

-N,--omagic: 指定读取/写入文本和数据段

-n,--nmagic: 关闭节的页面对齐，并禁用对共享库的链接。如果输出格式支持Unix样式的幻数，则将输出标记为"NMAGIC"

-noinhibit-exec:生成输出文件，即使出现非致命链接错误。通常，如果链接器在链接过程中遇到错误，它将不会生成输出文件。

-no-keep-memory:ld通常在内存中缓存输入文件的符号表来优化内存使用速度。此选项告诉ld不要缓存符号表。当链接大型可执行文件时，如果ld耗尽内存空间，则可能需要使用该选项

-O <level>:对于非零的优化等级，ld将优化输出。此操作会比较耗时，应该在生成最终的结果时使用。

-o <output>,--output=<output>:指定输出文件的名称

-oformat=<output-format>:指定输出文件的二进制格式

-R <filename>,--just-symbols=<filename>:从指定的文件读取符号名称和地址

-r,--relocatable:生成可重定位的输出（称为部分连接）

-rpath=<dir>:把指定的目录添加到运行时库搜索路径

-rpath-link=<dir>:指定搜索运行时共享库的目录

-S,--strip-debug:忽略来自输出文件的调试器符号信息

-s,--strip-all:忽略来自输出文件的所有符号信息

-shared,-Bshareable:创建共享库

-split-by-file[=size]:为每个目标文件在输出文件中创建额外的段大小达到size。size默认为1

-split-by-reloc[=count]:按照指定的长度在输出文件中创建额外的段

--section-start=<sectionname>=<org>:在输出文件中指定的地址定位指定的段

-T <scriptfile>,--script=<scriptfile>:使用scriptfile作为链接器脚本。此脚本将替换ld的默认链接器脚本（而不是添加到其中），因此脚本必须指定输出文件所需的所有内容。如果当前目录中不存在脚本文件，“ld”会在-L选项指定的目录中查找

-Ttext=<org>:使用指定的地址作为文本段的起始点

-Tdata=<org>:使用指定的地址作为数据段的起始点

-Tbss=<org>:使用指定的地址作为bss段的起始点

-t,--trace:在处理输入文件时显示它们的名称

-u <symbol>,--undefined=<symbol>:强制指定符号在输出文件中作为未定义符号

-v,-V,--version:显示ld版本号

-warn-common:当一个通用符号和另一个通用符号结合时发出警告

-warn-constructors:如果没有使用任何全局构造器，则发出警告

-warn-once:对于每个未定义的符号只发出一次警告

-warn-section-align:如果为了对齐而改动了输出段地址，则发出警告

--whole-archive:对于指定的存档文件，在存档中包含所有文件

-X,--discard-locals:删除所有本地临时符号

-x,--discard-al:删除所有本地符号Linux dd命令用于读取、转换并输出数据。

dd可从标准输入或文件中读取数据，根据指定的格式来转换数据，再输出到文件、设备或标准输出。

参数说明:

if=文件名：输入文件名，默认为标准输入。即指定源文件。

of=文件名：输出文件名，默认为标准输出。即指定目的文件。

ibs=bytes：一次读入bytes个字节，即指定一个块大小为bytes个字节。

obs=bytes：一次输出bytes个字节，即指定一个块大小为bytes个字节。

bs=bytes：同时设置读入/输出的块大小为bytes个字节。

cbs=bytes：一次转换bytes个字节，即指定转换缓冲区大小。

skip=blocks：从输入文件开头跳过blocks个块后再开始复制。

seek=blocks：从输出文件开头跳过blocks个块后再开始复制。

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

conv=<关键字>，关键字可以有以下11种：

conversion：用指定的参数转换文件。

ascii：转换ebcdic为ascii

ebcdic：转换ascii为ebcdic

ibm：转换ascii为alternate ebcdic

block：把每一行转换为长度为cbs，不足部分用空格填充

unblock：使每一行的长度都为cbs，不足部分用空格填充

lcase：把大写字符转换为小写字符

ucase：把小写字符转换为大写字符

swab：交换输入的每对字节

noerror：出错时不停止

notrunc：不截短输出文件

sync：将每个输入块填充到ibs个字节，不足部分用空（NUL）字符补齐。

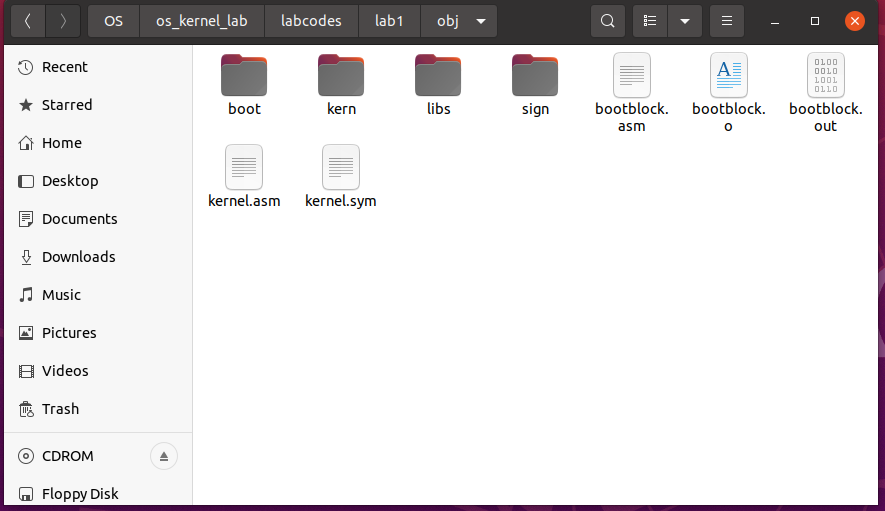
--help：显示帮助信息

--version：显示版本信息

**过程解析：**

①首先把C语言源代码编译为目标文件（.o）【标黄部分】

最后生成的目标文件全部存在obj文件夹中



②用命令ld把目标文件转换成可执行文件（.out）【标蓝部分】

把.o文件转换成可执行文件kernel与bootblock

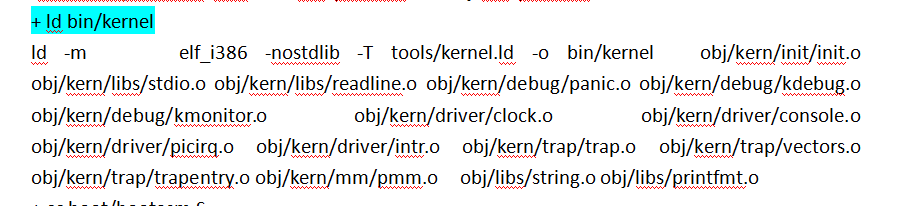
③dd命令把bootloader放到ucore.img.count的虚拟硬盘中【标绿部分】

此处生成一个虚拟硬盘叫ucore.img，这个虚拟硬盘有10000个扇区，每个扇区512字节，硬件模拟器会基于这个虚拟硬盘中的数据来执行相应的代码。其中bootblock大小为496字节，被存在虚拟磁盘的第一块

④生成了kernel与bootloader，其中kernel实际上是ucore的组成部分

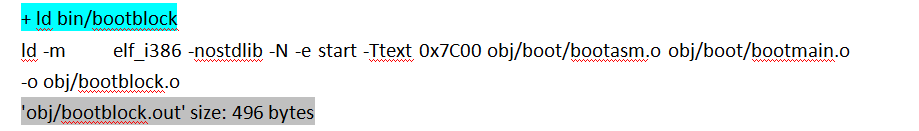
1. **具体分析ucore.img镜像文件生成过程**

A、首先是要生成kernel。ld将.o文件整合成可执行文件kernel，而这些.o文件是Makefile文件通过命令使用gcc把有关kernel的.c文件编译生成

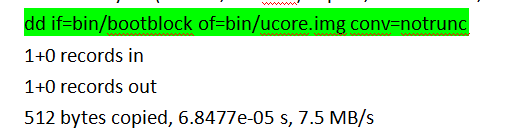


B、生成ucore.img还需要生成bootblock，而生成bootblock需要先生成bootmain.o和bootasm.o还有sign.o

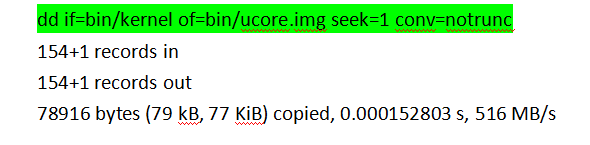
C、同理ld也将.o文件整合成可执行文件bootblock，大小为496字节,但还是放入512字节扇区中，但是，而这些.o文件也是Makefile文件通过命令使用gcc把有关bootloader的.c文件编译生成



D、创建10000块扇区，每个扇区512字节，制成ucore.img虚拟磁盘，将bootblock存到ucore.img虚拟磁盘的第一块



1. 将kernel存到ucore.img虚拟磁盘的第二块及之后几块，注意seek1，最终ucore.img虚拟磁盘制作完成



1. **分析MakeFile文件**

Makefile文件中，对应生成ucore.img的代码如下



其中$(kernel)的意思是生成kernel

Kernel生成代码如下



命令解析：

**·$(call add\_files\_cc,$(call listf\_cc,$(LIBDIR)),libs,)**

寻找libs目录下的所有具有.c, .s后缀的文件，并生成相应的.o文件，放置在obj/libs/文件夹下。

**·$(call add\_files\_cc,$(call listf\_cc,$(KSRCDIR)),kernel,$(KCFLAGS)**

用于生成kernel的所有子目录下包含的CTYPE文件（.s, .c文件）所对应的.o文件以及.d文件。【对应标黄的命令】

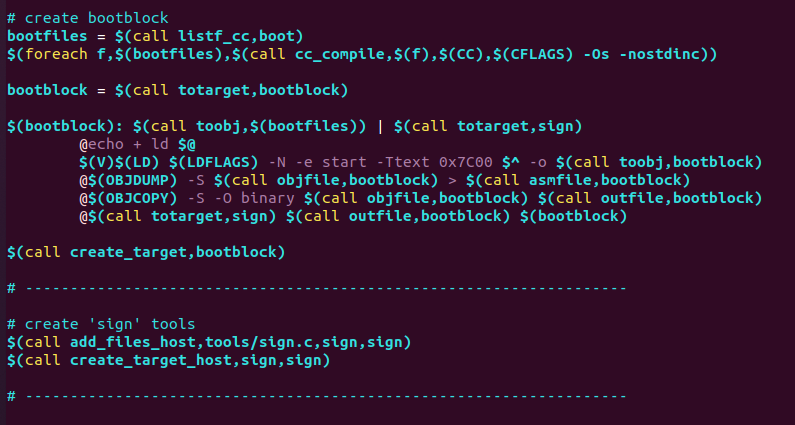
**·$(kernel): tools/kernel.ld**

/bin/kernel文件依赖于tools/kernel.ld文件，并且没有指定生成规则。

**·$(kernel): $(KOBJS)**

kernel文件的生成还依赖于上述生成的obj/libs, obj/kernels下的.o文件，并且生成规则为使用ld链接器将这些.o文件连接成kernel文件，其中ld的-T表示指定使用kernel.ld来替代默认的链接器脚本。【对应标蓝的命令】

$(bootblock)的意思是生成bootblock



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

将boot/文件夹下的bootasm.S, bootmain.c两个文件编译成相应的.o文件，并且生成依赖文件.d。其中：

-nostdinc: 不搜索默认路径头文件；

-0s: 针对生成代码的大小进行优化，这是因为bootloader的总大小被限制为不大于512-2=510字节。

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

链接.o文件，bootblock依赖于bootasm.o, bootmain.o文件与sign文件。

**·$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)**

使用ld链接器将依赖的.o文件链接成bootblock.o文件。其中：

-N：将代码段和数据段设置为可读可写；

-e：设置入口；

-Ttext：设置起始地址为0X7C00。

**·@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)**

使用objdump将编译结果反汇编出来，保存在bootclock.asm中，-S表示将源代码与汇编代码混合表示。

**·@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)**

使用objcopy将bootblock.o二进制拷贝到bootblock.out。

**·@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)**

使用sign程序, 利用bootblock.out生成bootblock。

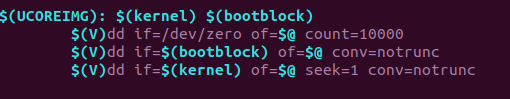
**·$(call add\_files\_host,tools/sign.c,sign,sign**

利用tools/sing.c生成sign.o。

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

利用sign.o生成sign，至此bootblock所依赖的文件均生成完毕。

其余命令



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

生成一个有10000个块的文件，每个块默认512字节，用0填充。

**·$(V)dd if=$(bootblock) of=$@ conv=notrunc**

把bootblock中的内容写到第一个块。

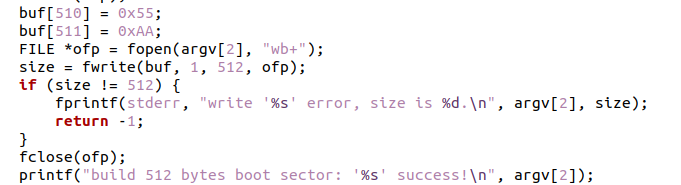
**·$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc**

从第二个块开始写kernel中的内容

1. **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文件

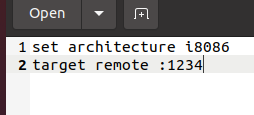
引导扇区的大小为512字节，最后两个字节为标志性结束字节0x55，0xAA，符合这两个要求才能认为是符合规范的磁盘主引导扇区



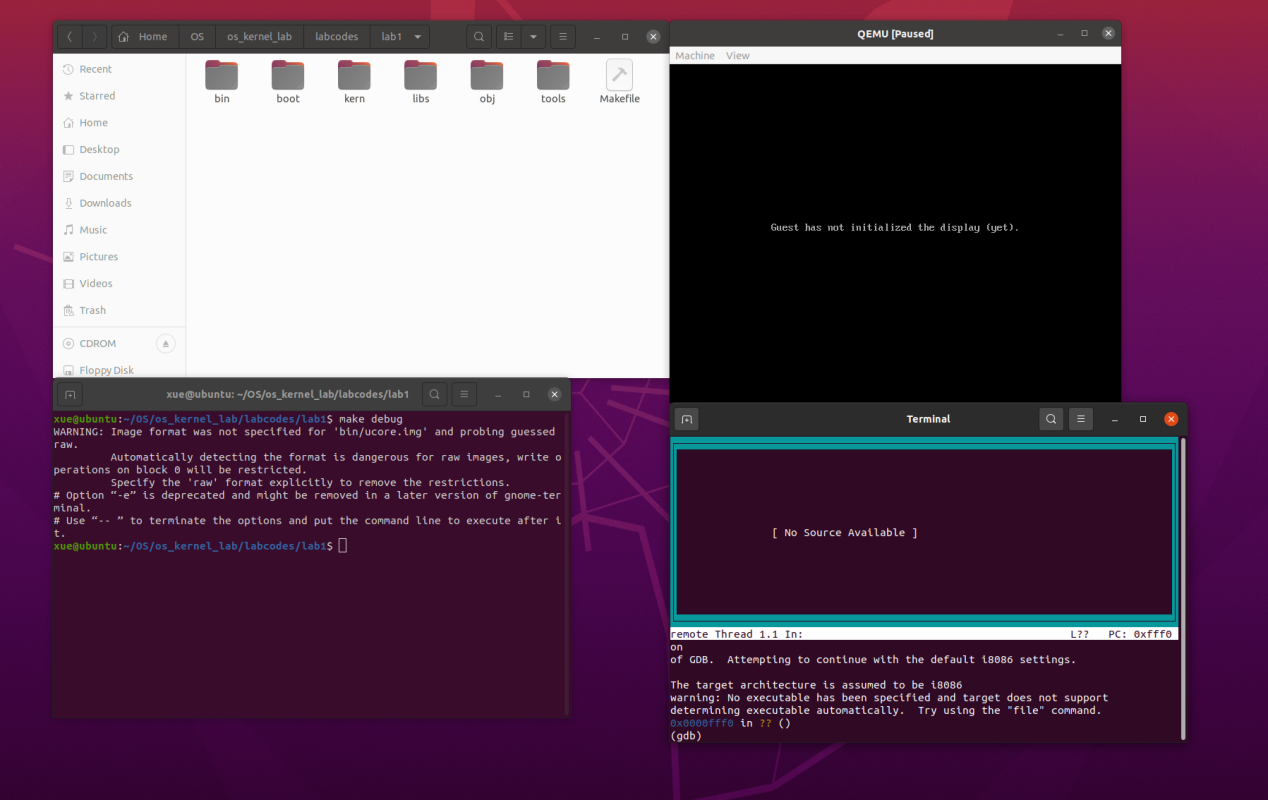
## 练习二：使用qemu执行并调试lab1中的软件

1. **从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。**

由于BIOS是在实模式下运行的，因此需要在lab1/tools/gdbinit里进行相应设置：



在lab1目录下执行make debug，就可以使用gdb单步追踪BIOS的指令执行



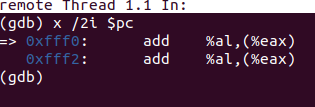
这时gdb停在BIOS的第一条指令处



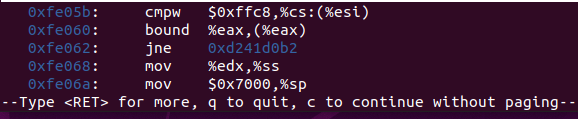
在gdb窗口中使用si命令即可单步追踪



在gdb界面下，可通过x /2i $pc命令来看BIOS的代码[显示当前eip所指处的汇编指令]

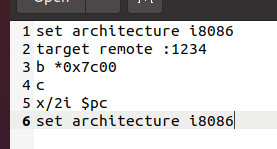


进一步执行x /10i 0xfe05b，可看到BIOS后续代码

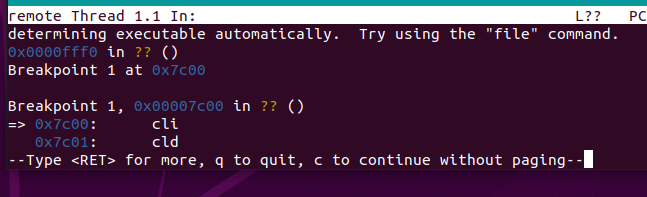


1. **在初始化位置0x7c00设置实地址断点,测试断点正常**

修改gdbinit文件，在结尾加上：

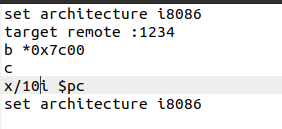


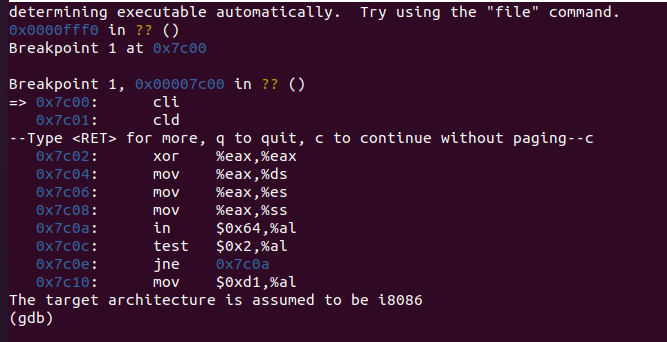
在lab1目录下执行make debug，得到如下结果，断点正常



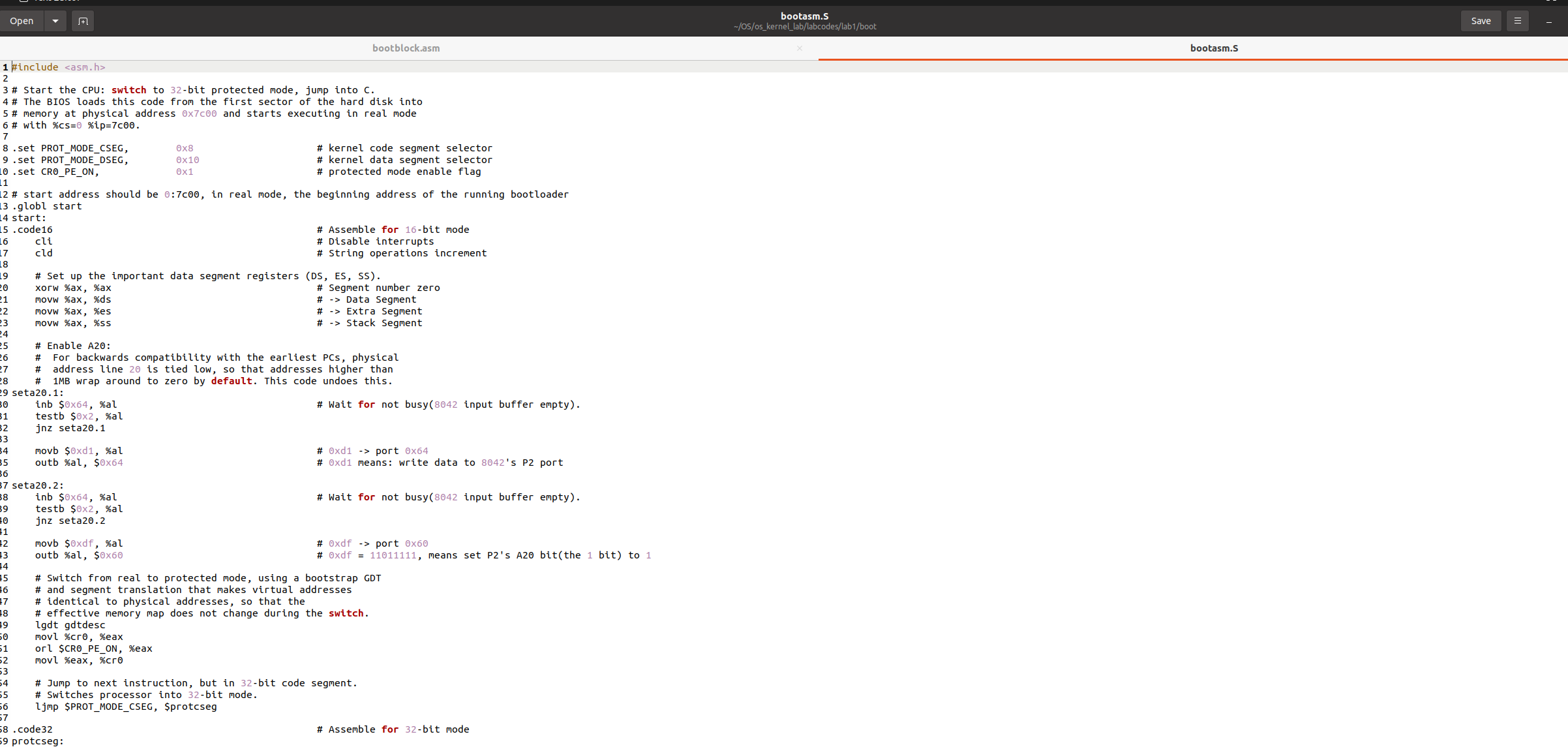
1. **从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较**

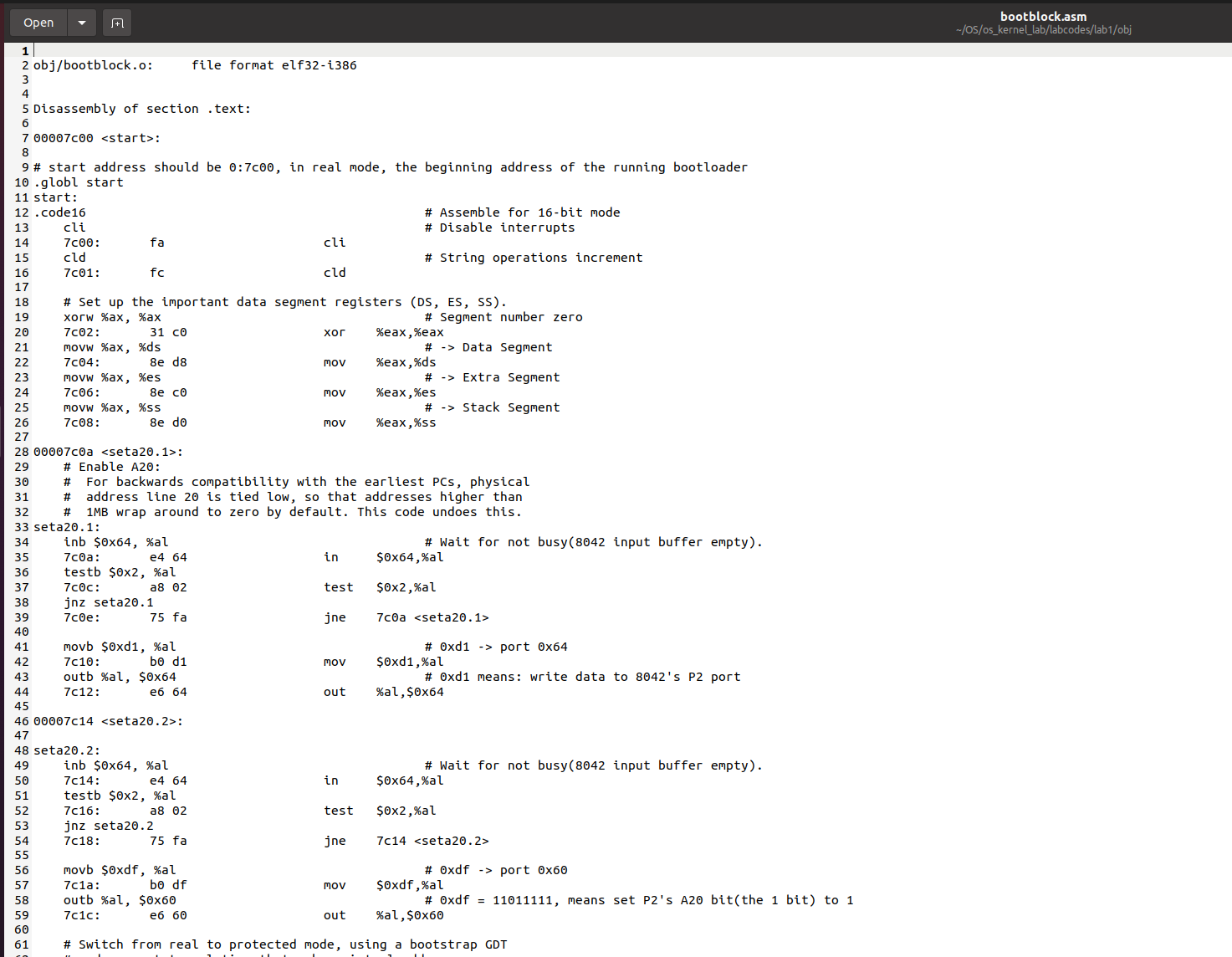
修改gbdinit



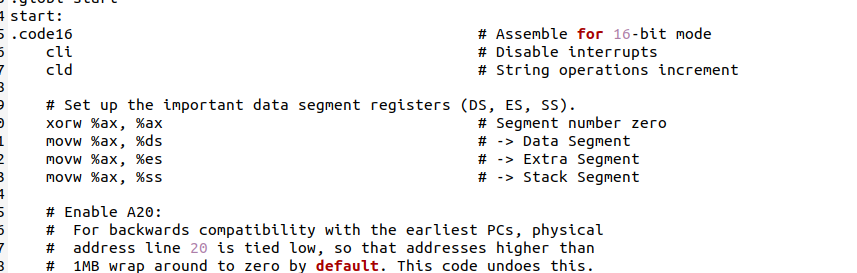


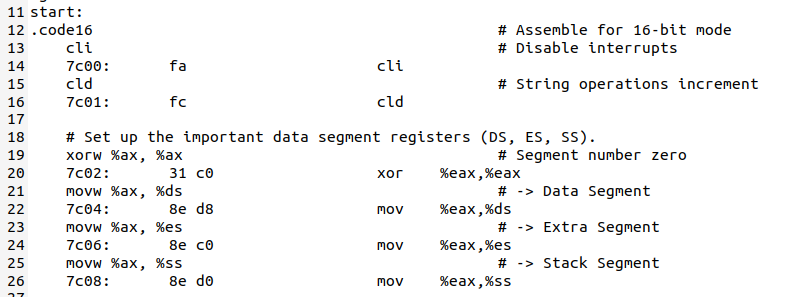
查看 bootasm.S、 bootblock.asm的完整代码：





两者的代码与反汇编结果基本一致





1. **自己找一个bootloader或内核中的代码位置，设置断点并进行测试**

b \*0x7c00的意思是在0x7c00处设置断点。此地址是bootloader入口点地址，可看boot/bootasm.S的start地址处

因此自己设置断点只要把0x7c00换成自己想要设置断点的位置即可

## 练习三：分析bootloader进入保护模式的过程

1. **为何开启A20，以及如何开启A20**

**为何要开启A20【历史遗留问题，为了实现兼容】**

1981年8月，IBM公司最初推出的个人计算机IBM PC使用的CPU是Inter 8088.在该微机中地址线只有20根。在当时内存RAM只有几百KB或不到1MB时，20根地址线已经足够用来寻址这些 内存。其所能寻址的最高地址是0xffff，

也就是0x10ffef。对于超出0x100000（1MB）的寻址地址将默认地环绕到0xffef。当IBM公司与1985年引入AT机时，使用的是Inter 80286 CPU，具有24根地址线，最高可寻址16MB，并且有一个与8088那样实现地址寻址的环绕。

但是当时已经有一些程序是利用这种环绕机制进行工作的。为了实现完全的兼容性，IBM公司发明了使用一个开关来开启或禁止0x100000地址比特位。由于当时的8042键盘控制器上恰好有空闲的端口引脚（输出端口P2，引脚P21），

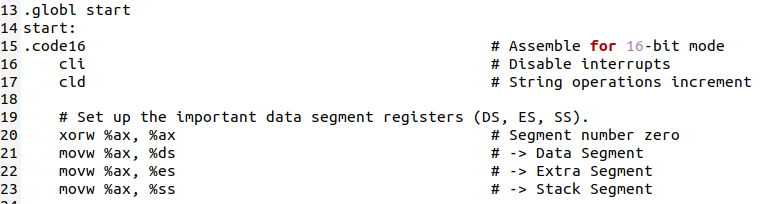
于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为A20。如果它为零，则比特20及以上地址都被清除。从而实现了兼容性。

当A20地址线控制禁止时，程序就像运行在8086上，1MB以上的地址是不可访问的，只能访问奇数MB的不连续的地址。为了使能所有地址位的寻址能力，必须向键盘控制器8082发送一个命令，键盘控制器8042会将A20线置于高电位，使全部32条地址线可用，实现访问4GB内存。

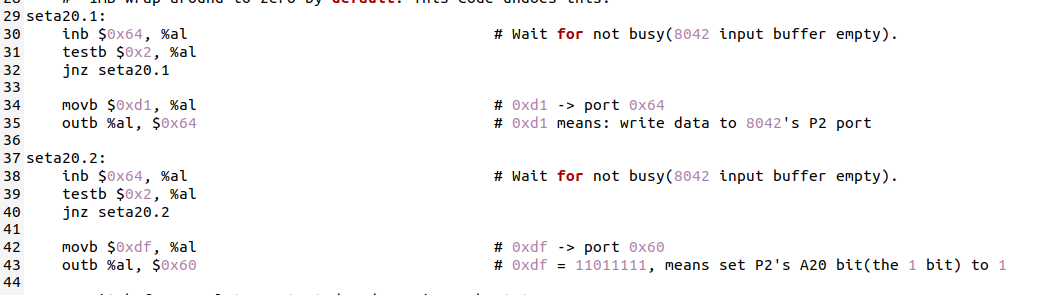
**如何开启A20**

bootloader中从实模式进到保护模式的代码保存在lab1/boot/bootasm.S文件下

bootloader入口为start, bootloader会被BIOS加载到内存的0x7c00处，此时cs=0, eip=0x7c00，在刚进入bootloader的时候，最先执行的操作分别为关闭中断、清除EFLAGS的DF位以及将ax, ds, es, ss寄存器初始化为0



接下来为了使得CPU进入保护模式之后能够充分使用32位的寻址能力，需要开启A20，关闭“回卷”机制。该过程主要分为等待8042控制器Inpute Buffer为空，发送P2命令到Input Buffer，等待Input Buffer为空，将P2得到的第二个位（A20选通）置为1，写回Input Buffer对应的汇编代码为：



第一步是从0x64内存地址读取一个字节到al中。接下来传送命令 0xd1，这个命令的意思是要向键盘控制器（8042）的 P2 port写入数据。

第二步是向8042 P2 端口写数据。写数据的方法是把数据通过键盘控制器的 0x60 端口写进去。写入的数据是 0xdf，表示将P2 port的第二个位（A20）选通置为1。随着 0xdf 的写入，A20 gate 就被打开了，从实模式进入保护模式，可以充分使用4G的寻址能力了。

1. **如何初始化GDT表**

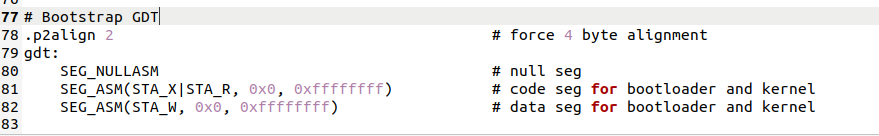
**什么是GDT表**

GDT全称是Global Descriptor Table，中文名称叫“全局描述符表”，想要在“保护模式”下对内存进行寻址就先要有 GDT。GDT 表里的每一项叫做“段描述符”，用来记录每个内存分段的一些属性信息，每个“段描述符”占 8 字节。

在保护模式下，我们通过设置GDT将内存空间被分割为了一个又一个的段(这些段是可以重叠的)，这样我们就能实现不同的程序访问不同的内存空间。这和实模式下的寻址方式是不同的, 在实模式下我们只能使用address = segment << 4 | offset的方式进行寻址(虽然也是segment + offset的，但在实模式下我们并不会真正的进行分段)。在这种情况下，任何程序都能访问整个1MB的空间。而在保护模式下，通过分段的方式，程序并不能访问整个内存空间。

**如何初始化GDT表**

bootasm.S中已经静态地描述了一个简单的GDT。GDT中将代码段和数据段的base均设置为了0，而limit设置为了2^32-1即4G，此时就使得逻辑地址等于线性地址，方便后续对于内存的操作：



因此在完成A20开启之后，只需要使用命令lgdt gdtdesc即可载入全局描述符表

【CPU单独为我们准备了一个寄存器叫做 GDTR 用来保存我们 GDT 在内存中的位置和我们 GDT 的长度。

GDTR 寄存器一共 48 位，其中高 32 位用来存储我们的 GDT 在内存中的位置，其余的低 16 位用来存我们的 GDT 有多少个段描述符。

16 位最大可以表示 65536 个数，这里我们把单位换成字节，而一个段描述符是 8 字节，所以 GDT 最多可以有 8192 个段描述符。

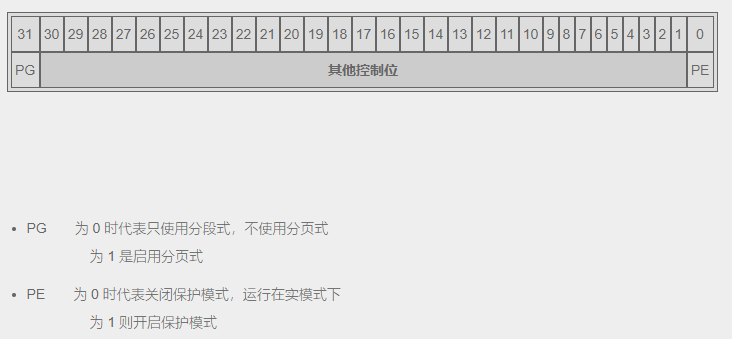
CPU 不仅用了一个单独的寄存器 GDTR 来存储我们的 GDT，而且还专门提供了一个指令用来让我们把 GDT 的地址和长度传给 GDTR 寄存器：lgdt gdtdesc】

因此把16位的GDT大小和32位的GDT地址传给GDTR寄存器，GDT就初始化好了。

1. **如何使能和进入保护模式**

x86 的控制寄存器一共有 4 个分别是 CR0、CR1、CR2、CR3（这四个寄存器都是 32 位的），而控制进入“保护模式”的开关在 CR0 上。

CR0中包含了6个预定义标志，0位是保护允许位PE(Protedted Enable)，用于启动保护模式，如果PE位置1，则保护模式启动，如果PE=0，则在实模式下运行

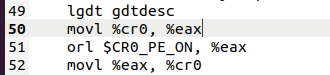


因此接下来只需要将cr0寄存器的PE位置1，即可从实模式切换到保护模式：【打开开关】

因为我们无法直接操作 cr0，所以我们首先要用一个通用寄存器来保存当前 cr0 寄存器的值，这里第一行就是用通用寄存器 eax 来保存 cr0 寄存器的值；

然后 CR0\_PE\_ON 这个宏的定义在 mmu.h 文件中，是个数值 0x00000001，将这个数值与 eax 中的 cr0 寄存器的值做“或”运算后，就保证将 cr0 的第 0 位设置成了 1 即 PE = 1 保证打开了保护模式的开关。

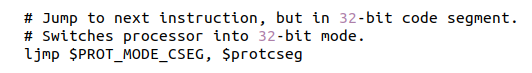
而 cr0 的第 31 位 PG = 0 表示我们只使用分段式，不使用分页，这时再将新的计算后的 eax 寄存器中的值写回到 cr0 寄存器中就完成了到保护模式的切换。

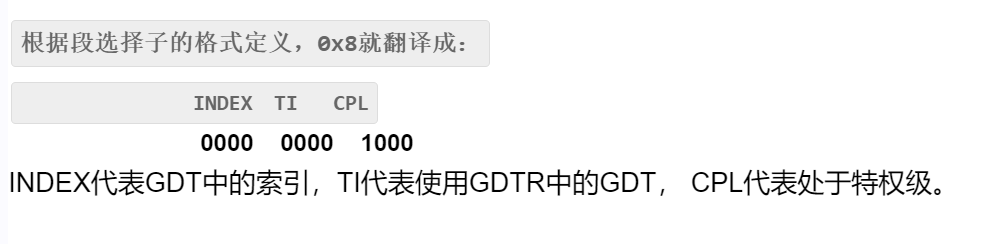


然后使用一个长跳转指令，更新cs寄存器基地址，将cs修改为32位段寄存器，以及跳转到protcseg这一32位代码入口处，此时CPU进入32位模式

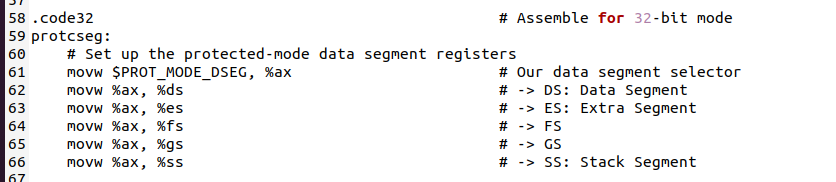
由于已经使能了保护模式，所以这里要使用逻辑地址，而不是实模式的地址

这里还要注意PROT\_MODE\_CSEG和PROT\_MODE\_DSEG，这两者分别定义为0x8和0x10，表示代码段和数据段的选择子。

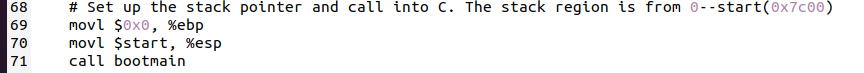




接下来执行的32位代码功能为设置段寄存器，并建立堆栈：设置ds、es, fs, gs, ss这几个段寄存器，然后初始化栈的frame pointer和stack pointer



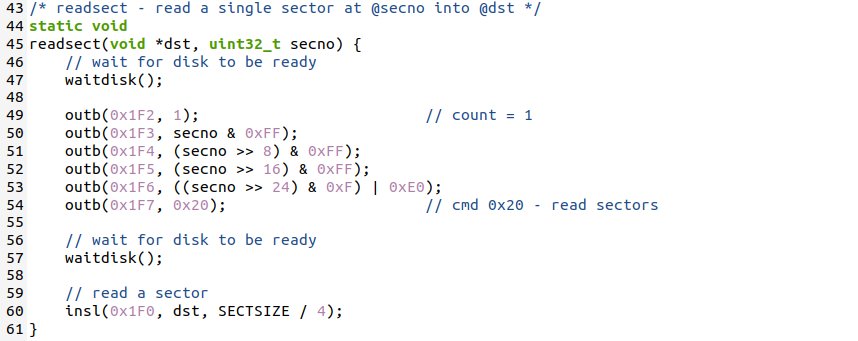
然后调用使用C语言编写的bootmain函数，进行操作系统内核的加载，至此，bootloader已经完成了从实模式进入到保护模式的任务。



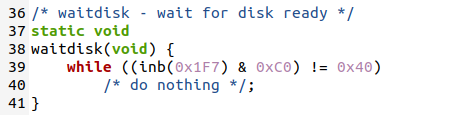
## 练习四：分析bootloader加载ELF格式的OS的过程

**1、bootloader如何读取硬盘扇区的？**

读取硬盘扇区的代码如下：



首先是waitdisk()函数，该函数的作用是连续不断地从0x1F7地址读取磁盘的状态，直到磁盘不忙为止



磁盘不忙后就可以开始读取磁盘扇区，具体操作为：

·往0x1F2到0X1F6中设置读取扇区需要的参数，包括读取扇区的数量以及LBA参数；

·往0x1F7端口发送读命令0X20；

outb()函数意味着这里是用LBA模式的PIO（Program IO）方式来访问硬盘的，即所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成，该函数每次只读取一个扇区

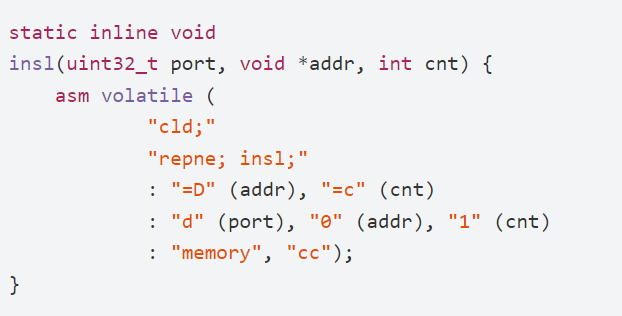


发出命令后再次等待磁盘不忙的时刻

硬盘再次空闲后，从数据端口0X1F0读取出数据到指定内存中

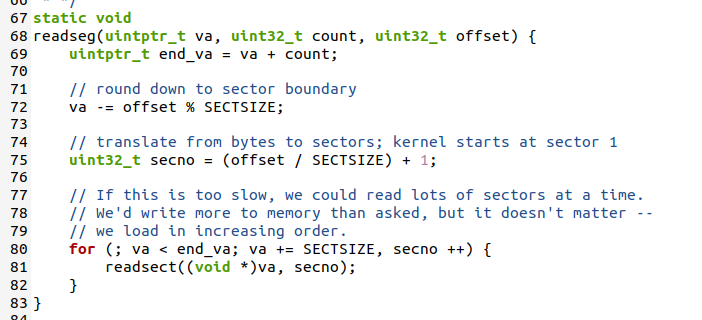
其中insl的实现如下

作用是从port IO端口读取cnt个双字(4Byte)大小的数据到基址为addr的内存中，由于读取单位是四字节，因此这里SECTIZE需要除以4。



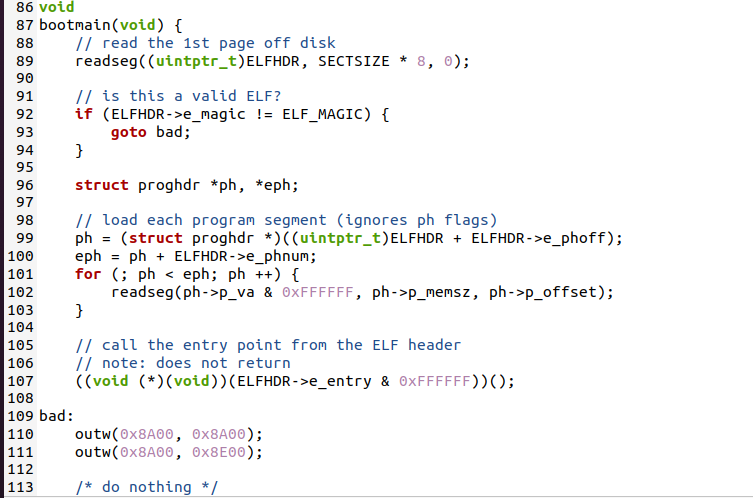
另外一个与读取磁盘相关的函数readseg，功能为将readsect进行进一步封装，提供能够从磁盘第二个扇区起（kernel起始位置）offset个位置处，读取count（任意）个字节到指定内存中。由于上述readsect函数只能就整个扇区进行读取，因此在readseg中，不得不连不完全包括了指定数据的首尾扇区内容也要一起读取进来。

此处还有一个小技巧就是将va减去了一个offset%512 Byte的偏移量，这使得就算是整个整个扇区读取，也可以使得要求的读取到的数据在内存中的起始位置恰好是指定的原始的va

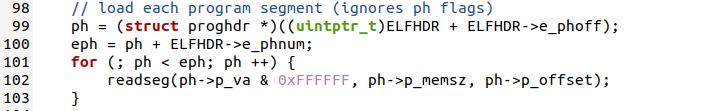


1. **bootloader是如何加载ELF格式的OS？**

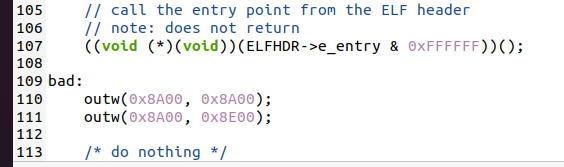
首先，从磁盘的第一个扇区（第零个扇区为bootloader）中读取OS kenerl最开始的4kB代码，然后判断其最开始四个字节是否等于指定的ELF\_MAGIC，用于判断该ELF header是否合法：



接下来从ELF头文件中获取program header表的位置，以及该表的入口数目，然后遍历该表的每一项，并且从每一个program header中获取到段应该被加载到内存中的位置（Load Address，虚拟地址），以及段的大小，然后调用readseg函数将每一个段加载到内存中，至此完成了将OS加载到内存中的操作：

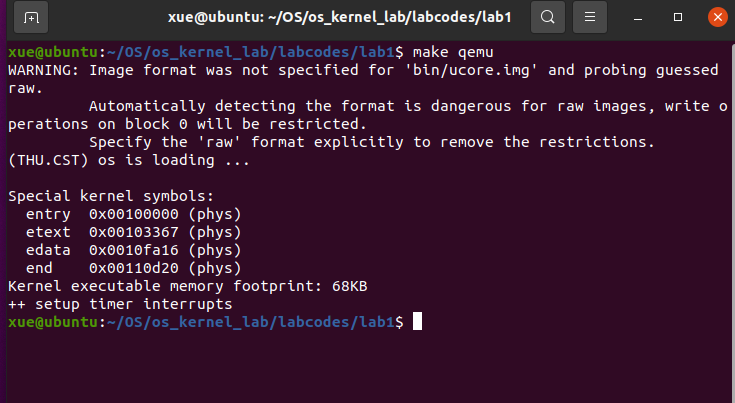


最后从ELF header中查询到OS kernel的入口地址，然后使用函数调用的方式跳转到该地址上去：

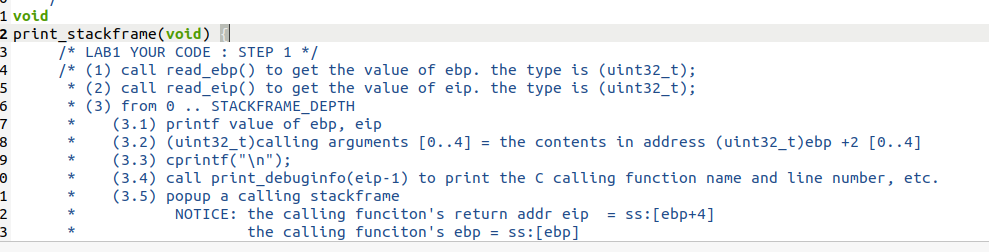


## 练习五：实现函数调用堆栈跟踪函数

首先使用make qemu 命令了解情况



根据提示写出代码



代码的具体步骤：

·使用read\_ebp和read\_eip函数获取当前stack frame的base pointer以及call read\_eip这条指令下一条指令的地址，存入ebp, eip两个临时变量中；

·使用cprint函数打印出ebp, eip的数值；

·打印出当前栈帧对应的函数可能的参数；

·使用print\_debuginfo打印出当前函数的函数名；

·根据动态链查找当前函数的调用者(caller)的栈帧；

·如果ebp非零并且没有达到规定的STACKFRAME DEPTH的上限，则跳转到（2），继续循环打印栈上栈帧和对应函数的信息。

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)

    for(int 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.

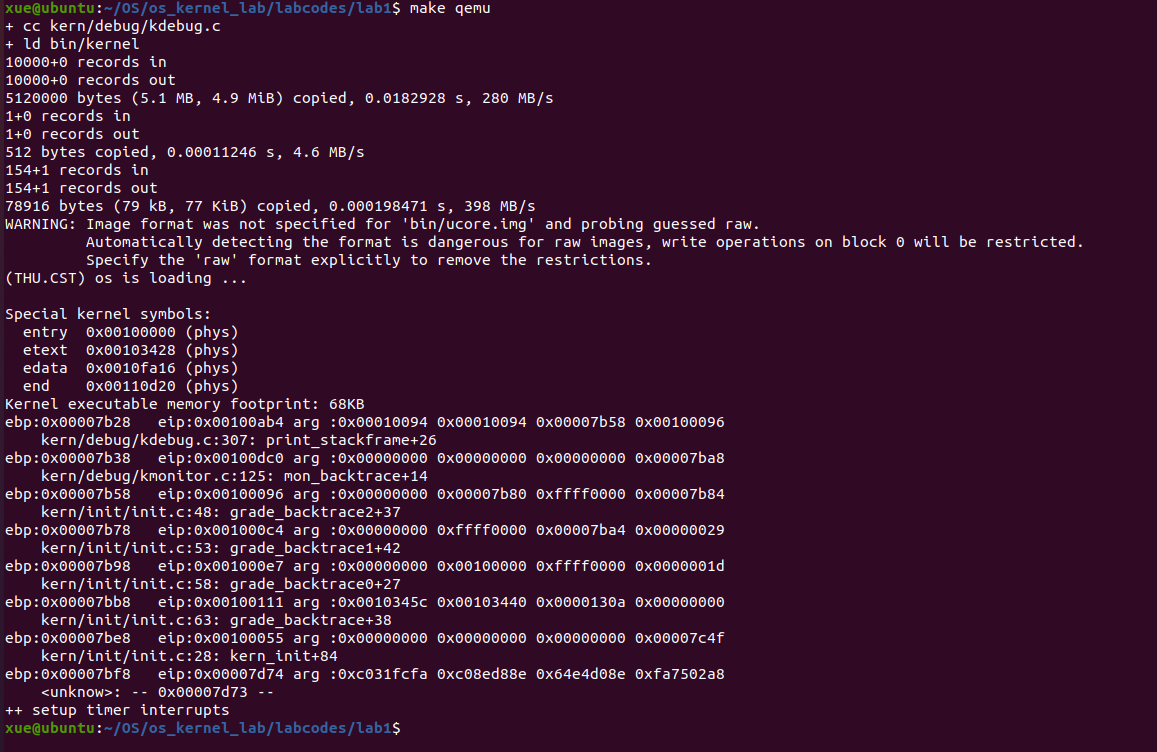
          eip=((uint32\_t \*)ebp)[1];

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

      }

}

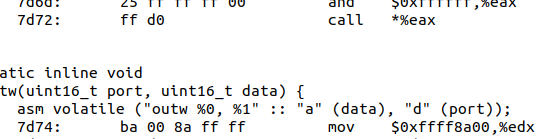
编码完成后，执行make qemu，打印结果如下所示，与实验指导书的结果大致一致



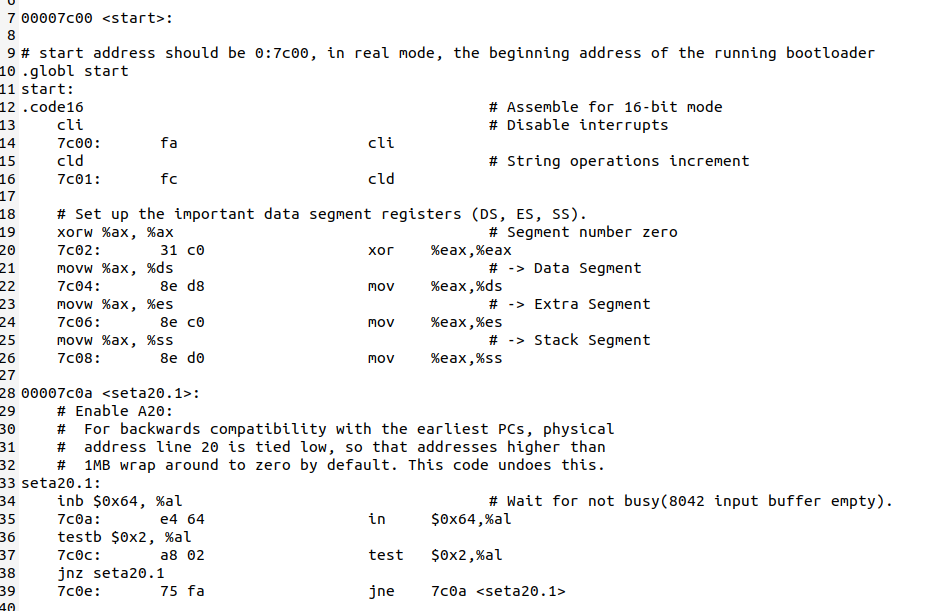
解释最后一行各个参数的含义：

ebp:0x0007bf8 此时ebp的值是kern\_init函数的栈顶地址，从obj/bootblock.asm文件中知道整个栈的栈顶地址为0x00007c00，ebp指向的栈位置存放调用者的ebp寄存器的值，ebp+4指向的栈位置存放返回地址的值，这意味着kern\_init函数的调用者（也就是bootmain函数）没有传递任何输入参数，因为单是存放旧的ebp、返回地址已经占用8字节了。

eip:0x00007d74 eip的值是kern\_init函数的返回地址，也就是bootmain函数调用kern\_init对应的指令的下一条指令的地址。这与obj/bootblock.asm是相符合的。



args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8 一般来说，args存放的4个dword是对应4个输入参数的值。但这里比较特殊，由于bootmain函数调用kern\_init并没传递任何输入参数，并且栈顶的位置恰好在boot loader第一条指令存放的地址的上面，而args恰好是kern\_int的ebp寄存器指向的栈顶往上第2~5个单元，因此args存放的就是boot loader指令的前16个字节。可以对比obj/bootblock.asm文件来验证（验证时要注意系统是小端字节序，可以看到，两者是相同的）。



## 练习六：完善中断初始化和处理

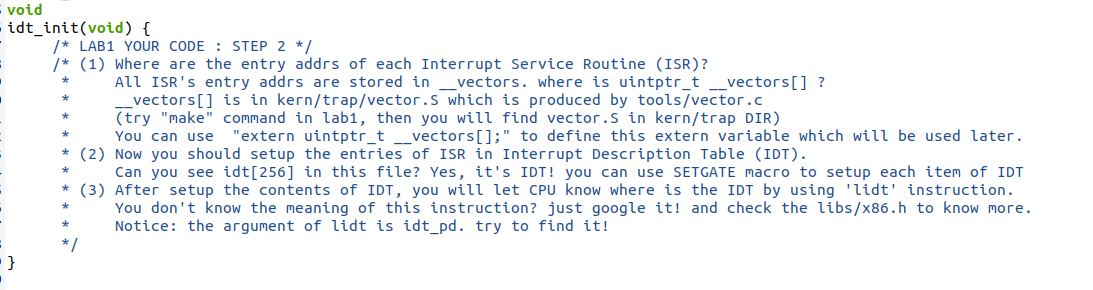
**1、中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？**

IDT中的每一个表项均占8个字节。

其中最开始2个字节和最末尾2个字节定义了offset，第16-31位定义了处理代码入口地址的段选择子，使用其在GDT中查找到相应段的base address，加上offset就是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。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。**

依旧是按照注释补全代码



代码的具体步骤：

·对整个idt数组进行初始化；

·把所有的中断都初始化为内核级的中断；

·使用lidt指令加载中断描述符表。

void idt\_init(void) {

    extern uintptr\_t \_\_vectors[];           //声明\_\_vertors[],其中存放着中断服务程序的入口地址

    int i;

    for(i=0;i<256;i++) {

        if(i!=T\_SYSCALL){

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

        }

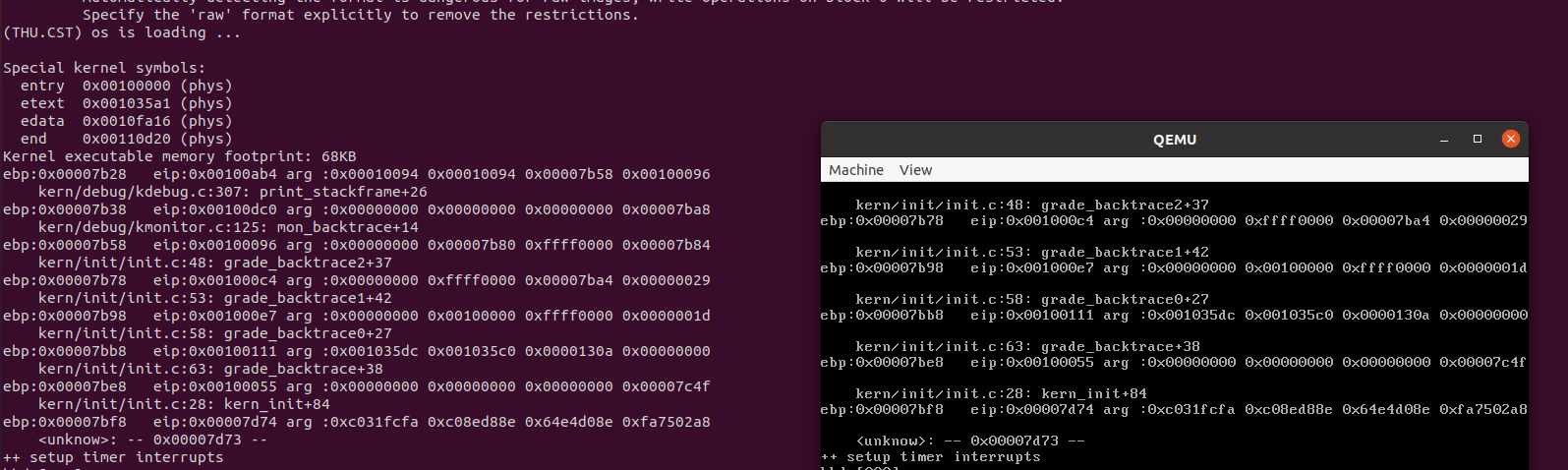
     }

    SETGATE(idt[T\_SWITCH\_TOK],0,GD\_KTEXT,\_\_vectors[T\_SWITCH\_TOK],DPL\_USER);     //填充中断描述符表IDT

    lidt(&idt\_pd);              //使用lidt指令加载中断描述符表

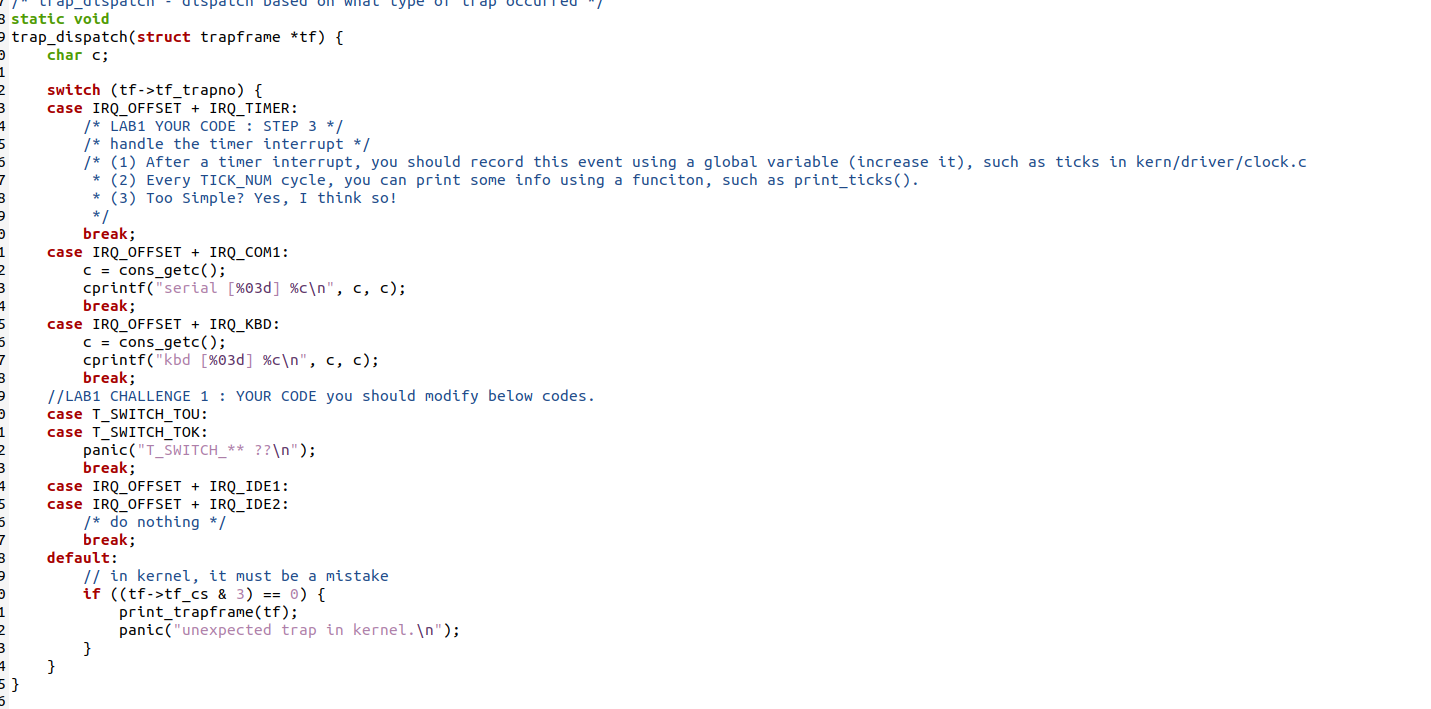
}

执行make qemu 结果：



1. **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”。**

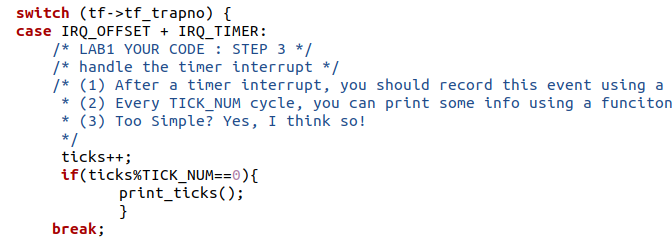
还是按照注释补全代码



代码的具体步骤：

·每次时钟中断之后ticks就会加一 当加到TICK\_NUM次数时 打印并重新开始；

·打印字符串；



执行“make qemu”结果

