

# Lab0 & Lab1

---

## 遇到的问题

---

### 配置环境

#### Win10装双系统时Windows启动器报错

Windows启动器显示“Windows未能启动，原因可能是最后更改了硬件或软件.....”，在网上也看到了类似的问题，但是没有合适的解决方法。向老师寻求帮助，然后收到了老师的“恭喜”。造成这种情况的原因可能是网上的文档有些旧了，适应不了win10。所以找了easyBCD的修复方法，然后选择装虚拟机。

#### 虚拟机能上网，物理机却上不了网

刚装好Ubuntu的那几天，碰巧校园网有点问题，于是出现了虚拟机能联网，物理机上不了网的情况，当时以为是虚拟机的网络设置有什么问题，查了一些资料也没有解决，然后某一个时刻开机之后，发现这个问题消失了，所以至今也不知道是什么原因。

#### qemu安装路径错误

执行“`sudo ln -s /usr/local/bin/qemu-system-i386 /usr/local/bin/qemu`”命令时失败了，报错：“`ln: target '/usr/local/bin/qemu' is not a directory`”，是因为没有把qemu装到正确的位置，默认解压路径是usr/bin，而不是usr/local/bin，在链接时会有问题。解决：在qemu官网上重新下载压缩包，解压，解压时使用tar xvjf命令，之后使用./configure，然后make。

#### 在Vmware中选择“启动虚拟机”之后，一直黑屏

qemu安装好之后就关机睡觉了，第二天发现点击开机一直黑屏。点击关闭时，提示系统繁忙，关不掉。解决方法：在任务管理器关掉进程，重启了一下电脑，然后可以正常打开虚拟机了。猜想原因可能是前一天晚上还没运行完就被我关机了。

## Lab1

#### recipe for target 'bin/bootblock' failed

执行make qemu时，在链接bin/bootblock时就不在向下进行了，报错：“`obj/bootblock.out`” size: 600 bytes 600>>510 !! Makefile:152: recipe for target 'bin/booblock' failed。错误是ld生成出来的bootblock..out的大小是600字节超过了510字节，网上给出的原因是新版本的GCC在生成执行代码时产生了更多的数据，导致bootloader的size超过了510字节，网上的解决方法有：（1）在Makfile中增加gcc的编译选项 -Os，做空间优化；（2）降低gcc的版本。

但是我执行了make clean之后，重新执行make qemu就没有问题了，上面的两种方法我都还没有尝试.....

#### 没有输出100 ticks

写完代码之后执行make qemu，并没有输出100 ticks，尝试了很多方法之后，发现可能是原来的Makefile有问题，替换掉了原来的Makefile，重新生成bin、obj文件，再次make qemu就可以正常输出了。用meld对比两个Makefile文件的区别，左侧是未替换前的Makefile，右侧是替换后的Makefile。两个是有区别的，好像有判断语句对编译器进行替换，但是我看不太懂这些区别会造成什么后果.....



## 没有键盘回显

键盘输入需要在qemu界面中才能显示出来，但是我执行make qemu之后，并没有弹出新的qemu界面。老师说可能是版本的问题。相同的代码在Mac里尝试时，就没有出现这个问题，不太清楚原因。解决：执行make qemu-nox命令，就可以在当前的控制台中实现键盘回显。make qemu-nox不同于make qemu的一点是它不使用图形化界面，直接在控制台运行。

## make debug时报错：Failed to execute child process "cgdb" (No such file or directory)

cgdb是gdb的一个图形化界面，最开始执行make debug还能正常进行调试，交作业之前突然就打不开调试窗口了，但是gdb指令还是可以使用。解决：执行sudo apt-get install cgdb，然后可以正常执行make debug。

## 中断门和陷阱门的区别

在SETGATE的参数设置里，0表示中断门，1表示陷阱门。对于中断门，在转移过程中把IF置为0，使得在处理程序执行期间屏蔽掉INTR中断(当然，在中断处理程序中可以人为设置IF标志打开中断，以使得在处理程序执行期间允许响应可屏蔽中断)；对于陷阱门，在转移过程中保持IF位不变，即如果IF位原来是1，那么通过陷阱门转移到处理程序之后仍允许INTR中断。因此，中断门最适宜于处理中断，而陷阱门适宜于处理异常。

## 函数调用的压栈出栈过程

在函数调用时，第一个进栈的是主函数中函数调用后的下一条指令(函数调用语句的下一条可执行语句)的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。静态变量是不入栈的。

## 参考文档

除了OSLab以外，在网上找到了其他我觉得比较好的文档：

[清华高级操作系统课程论文阅读笔记](#)

## 实验感想

没什么想吐槽的，就是坑太多了，而且对操作系统的代码比较陌生，出了问题都还不知道是哪里的的问题。尤其是交作业前两小时出问题，真的是非常难受。第一次实验的代码量还是比较小的，对之后的实验有一丝恐惧.....不过还是挺喜欢这种学习模式的，选择的空间很大，实验内容和教程都可以自己挑。

下面是实验过程中的笔记，没有什么独特的内容，只是为了帮助记忆。老师您可以不用往下看了。

## Lab1实验笔记

### 练习1：理解通过make生成执行文件的过程

#### Q1：操作系统镜像文件ucore.img是如何一步一步生成的？

.img文件是光盘映像文件。GNU make(简称make)是一种代码维护工具，在大中型项目中，它将根据程序各个模块的更新情况，自动的维护和生成目标代码。make命令执行时，需要一个 makefile （或Makefile）文件，makefile 就像一个shell脚本，告诉make命令需要怎么样的去编译和链接程序。我对Makefile的使用比较陌生，所以只能大致了解代码的意思，可能某几条代码的理解会有偏差。比较简单的、已经有英文注释的就不再赘述了。

#### Lab1/Makefile:

```
#定义了5个变量，其中@的意思是只输出结果，不输出后面的命令，使用make "v="可以查看make在执行整个流程的详细信息
```

```
PROJ      := challenge
EMPTY     :=
SPACE     := $(EMPTY) $(EMPTY)
SLASH     := /
```

```
V         := @
```

```
#若要使用LLVM (low level virtual machine, 一个编译器框架)，则去掉USELLVM前面的注释#
#need llvm/cang-3.5+
#USELLVM := 1
```

```
#定义变量GCCPREFIX，使用$var就是读取makefile的变量然后展开，将其值作为参数传给了一个shell命令。
grep主要是命令的简单过滤。\\是换行符的意思。|是管道符号。1>&2 是把结果输出到和标准错误一样。
```

```
#这部分做的工作大概是检查GCCPREFIX是否被设置，如果没有设置，那么判断运行环境，自定义GCCPREFIX
# try to infer the correct GCCPREFIX,
```

```
ifndef GCCPREFIX
```

```
GCCPREFIX := $(shell if i386-elf-objdump -i 2>&1 | grep '^elf32-i386$$' >/dev/null 2>&1; \
```

```
then echo 'i386-elf-'; \
```

```
elif objdump -i 2>&1 | grep 'elf32-i386' >/dev/null 2>&1; \
```

```
then echo ''; \
```

```
else echo "****" 1>&2; \
```

```
echo "**** Error: Couldn't find an i386-elf version of GCC/binutils." 1>&2; \
```

```
echo "**** Is the directory with i386-elf-gcc in your PATH?" 1>&2; \
```

```

echo "*** If your i386-elf toolchain is installed with a command" 1>&2; \
echo "*** prefix other than 'i386-elf-', set your GCCPREFIX" 1>&2; \
echo "*** environment variable to that prefix and run 'make' again." 1>&2; \
echo "*** To turn off this error, run 'gmake GCCPREFIX= ...'." 1>&2; \
echo "" 1>&2; exit 1; fi)
endif

```

#.....这一部分与上面GCCPREFIX类似，设置QEMU（一款模拟处理器），代码省略。

#设置默认的后缀规则，.c .S .h作为后缀列表

# eliminate default suffix rules

.SUFFIXES: .c .S .h

# delete target files if there is an error (or make is interrupted)

.DELETE\_ON\_ERROR:

# define compiler and flags

ifndef USELLVM

#HOSTCC这里指定给主机用的编译器是GCC，参数-g方便程序调试，-Wall生成警告信息，-O2优化选项，这个今天编译课刚学过嘿嘿嘿，有0,1,2,3四中优化选项，优化程度依次升高。

HOSTCC := gcc

HOSTCFLAGS := -g -Wall -O2

#cc是unix下面用的编译命令，gcc是linux下面用的编译命令，这句话或许是把cc连接到gcc命令上，运行cc就是运行gcc？

CC := \$(GCCPREFIX)gcc

#下面这几句是直接来自网上粘贴的对参数的解释：-fno-builtin 不接受非“\_\_”开头的内建函数，-ggdb让gcc 为gdb生成比较丰富的调试信息，-m32 编译32位程序，-gstabs 此选项以stabs格式声称调试信息，但是不包括gdb调试信息，-nostdinc 不在标准系统目录中搜索头文件，只在-I指定的目录中搜索，DEFS是未定义量。可用来对CFLAGS进行扩展。

CFLAGS := -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc \$(DEFS)

#对shell的命令不太熟，这里没有太懂。-E 仅作预处理，不进行编译、汇编和链接，-x c 指明使用的语言为c语言，/dev/null用来指定目标文件，>/dev/null 2>&1 将标准输出与错误输出重定向到/dev/null。如果&&前面的语句可以执行，那么CFLAGS += -fno-stack-protector

CFLAGS += \$(shell \$(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)

else

#clang是另外一种轻量级编译器

HOSTCC := clang

HOSTCFLAGS := -g -Wall -O2

CC := clang

CFLAGS := -fno-builtin -Wall -g -m32 -mno-sse -nostdinc \$(DEFS)

CFLAGS += \$(shell \$(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)

endif

#定义源文件类型为c和S

CTYPE := c S

#设置链接选项

LD := \$(GCCPREFIX)ld

#ld -v命令会输出连接器的版本与支持的模拟器。在其中搜索elf\_i386。若支持，则LDFLAGS := -m

elf\_i386，使用elf\_i386模拟器，-nostdlib 不连接系统标准启动文件和标准库文件，只把指定的文件传递给连接器

```
LDFLAGS := -m $(shell $(LD) -v | grep elf_i386 2>/dev/null)
LDFLAGS += -nostdlib
```

#.....省略一些命令定义语句

```
include tools/function.mk
```

#listf函数在function.mk中定义，列出某地址（变量1）下某些类型（变量2）文件。listf\_cc:列出某地址（变量1）下.c与.s文件

```
listf_cc = $(call listf,$(1),$(CTYPE))
```

# for cc

```
add_files_cc = $(call add_files,$(1),$(CC),$(CFLAGS) $(3),$(2),$(4))
```

```
create_target_cc = $(call create_target,$(1),$(2),$(3),$(CC),$(CFLAGS))
```

# for hostcc

```
add_files_host = $(call add_files,$(1),$(HOSTCC),$(HOSTCFLAGS),$(2),$(3))
```

```
create_target_host = $(call create_target,$(1),$(2),$(3),$(HOSTCC),$(HOSTCFLAGS))
```

#cgtype (filenames,type1, type2) 把文件名中后缀是type1的改为type2，这里可能是\*.c改为\*.o。

opatsubst是替换通配符。接下来列出所有的.o文件，将.o改为.asm，.o改为.out，.o改为.sym

```
cgtype = $(patsubst %.$(2),%.$(3),$(1))
```

```
objfile = $(call toobj,$(1))
```

```
asmfile = $(call cgtype,$(call toobj,$(1)),o,asm)
```

```
outfile = $(call cgtype,$(call toobj,$(1)),o,out)
```

```
symfile = $(call cgtype,$(call toobj,$(1)),o,sym)
```

# for match pattern

```
match = $(shell echo $(2) | $(AWK) '{for(i=1;i<=NF;i++){if(match("$(1)","^"$$i)$$")}{exit 1;}}'}; echo $$?)
```

#.....先生成bin/kernel，再生成bin/bootblock，之后生成bin/sign，最后生成bin/ucore.img。每一部分都有明确的注释，只不过我没有看懂其中一些参数的含义，这里省略代码。

#这一部分之前都是在定义各种变量/函数，设置参数，为这一部分做准备工作，之后的部分就是一些收尾工作。

# files for grade script

```
TARGETS: $(TARGETS)
```

```
.DEFAULT_GOAL := TARGETS
```

```
.PHONY: qemu qemu-nox debug debug-nox
```

#终端模式打开qemu，新窗口打开qemu，生成log文件，qemu调试以及删除的过程，代码省略。

#打包并输出语句

```
handin: packall
```

```
    @echo Please visit http://learn.tsinghua.edu.cn and upload $(HANDIN). Thanks!
```

```
packall: clean
```

```
    @$(RM) -f $(HANDIN)
```

```
    @tar -czf $(HANDIN) `find . -type f -o -type d | grep -v '^\.?$$' | grep -vF
```

```
 '$(HANDIN)'`
```

#输出所有tags

```
tags:
```

```
    @echo TAGS ALL
```

```
    $(V)rm -f cscope.files cscope.in.out cscope.out cscope.po.out tags
```

```
$(V)find . -type f -name "*.chs" >cscope.files
$(V)cscope -bq
$(V)ctags -L cscope.files
```

将最开始的“V:=@”改为“V=”，然后make；或者执行make “V=”，可以看到makefile的执行过程：

1. 先生成init.o, stdio.o, readline.o, panic.o, kdebug.o, kmonitor.o, clock.o, console.o, picirq.o, intr.o, trap.o, vectors.o, trapentry.o, pmm.o, string.o, printfmt.o（Makefile文件通过命令使用gcc把有关kernel的.c文件编译生成.o文件），使用ld链接命令，将.o文件整合成ELF格式的可执行文件bin/kernel；
2. 生成bootasm.o, bootmain.o（Makefile使用gcc把有关bootloader的.c文件编译生成的.o文件），生成bin/sign, sign是一个外部执行程序，用来生成虚拟的硬盘主引导扇区；使用ld链接命令生成bin/bootblock；
3. 使用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）命令，创建10000块扇区，每个扇区512字节，生成ucore.img，将bootblock存到ucore.img的第一块，使用seek=1设置下一个文件读写的位置，将kernel存到ucore.img虚拟磁盘的第二块。

## Q2：一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？

sign是用来生成虚拟的硬盘主引导扇区的，查看tool/sign.c中关于主引导扇区的主要部分：

```
//引用头文件
int
main(int argc, char *argv[]) {

    //错误情况的检查，代码省略

    //设置主引导扇区的大小为512个字节（特征1）
    char buf[512];
    //用0填充（特征2）
    memset(buf, 0, sizeof(buf));

    //读文件，不清楚读文件的目的，代码省略

    //主引导扇区最后两个字节为0x55和0xAA（特征3）
    buf[510] = 0x55;
    buf[511] = 0xAA;

    //写文件，不清楚这一部分目的，代码省略

    return 0;
}
```

## 练习2：使用qemu执行并调试lab1中的软件

### T1：从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行

BIOS, Basic Input Output System, 基本输入输出系统。CPU加电后执行的第一条指令的物理地址是 FFFFFFF0H, 在0xFFFFFFF0存放了一条跳转指令，跳到BIOS例行程序起始点。BIOS做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的主引导扇区到内存地址0x7c00处，然后CPU控制权会转移到那个地址继续执行。



这一部分主要使用了si和x/n/f/u/<addr>两条命令：si是执行一行汇编代码，如果有函数调用，则进入该函数。x/n/f/u/<addr>是以addr为起始地址，返回n个单元的值，每个单元对应u个字节（GDB默认是4个bytes），输出格式是f，例如x /2i 0xffff0。

刚开始误以为每次make debug都是从0x0000fff0开始，然后发现每次调试的初始地址都不一样，只有重启虚拟机之后第一次调试地址显示是0x0000fff0，0x0000fff0是CPU加电后执行的第一条指令。

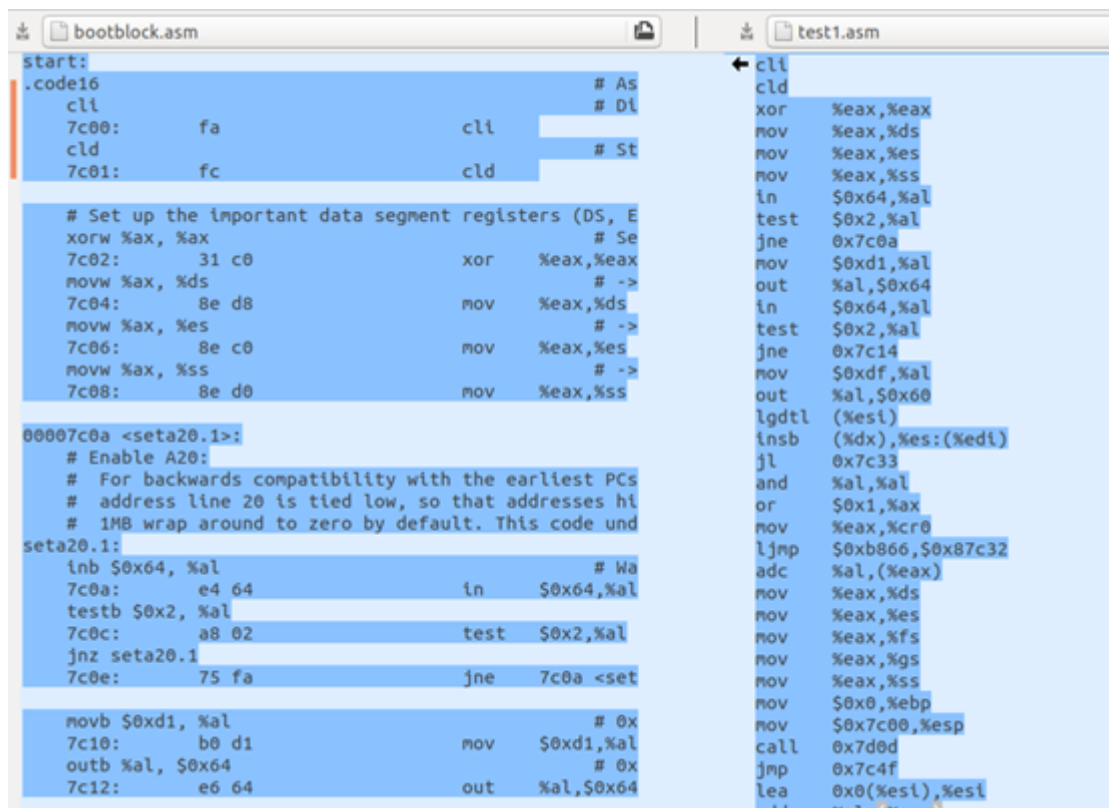
## T2：在初始化位置0x7c00设置实地址断点,测试断点正常

这一部分没有遇到什么问题，使用b \*0x7c00设置断点，使用c运行，遇到断点暂停。

## T3：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

要求中说“单步调试”，所以我刚开始不停地输入si，输入几次之后觉得这样太傻了，就直接使用x命令输出了之后的一部分代码，存储在test.asm中。

本来想用meld对test.asm、bootasm.S和 bootblock.asm进行比较，打开bootblock.asm之后，发现里面不仅有汇编指令，还有好像是地址和机器码的东西？所以最后还是自己一行行比较，对比之后，发现bootblock.asm中的汇编指令和test.asm没有区别。bootasm.S和test.asm相比，没发现代码含义有什么大的区别，只是test.asm中用类似%eax或者%ax表示对寄存器进行操作的位数，而bootasm.S中用l,w,b表示对寄存器进行操作的位数。



```
bootblock.asm
start:
.code16
cli
7c00: fa cli
cld
7c01: fc cld

# Set up the important data segment registers (DS, ES, SS)
xorw %ax, %ax
7c02: 31 c0 xor %eax, %eax
movw %ax, %ds
7c04: 8e d8 mov %eax, %ds
movw %ax, %es
7c06: 8e c0 mov %eax, %es
movw %ax, %ss
7c08: 8e d0 mov %eax, %ss

00007c0a <seta20.1>:
# Enable A20:
# For backwards compatibility with the earliest PCs
# address line 20 is tied low, so that addresses hi
# 1MB wrap around to zero by default. This code und
seta20.1:
inb $0x64, %al
7c0a: e4 64 in $0x64, %al
testb $0x2, %al
7c0c: a8 02 test $0x2, %al
jnz seta20.1
7c0e: 75 fa jne 7c0a <seta20.1>

movb $0xd1, %al
7c10: b0 d1 mov $0xd1, %al
outb %al, $0x64
7c12: e6 64 out %al, $0x64

test1.asm
cli
cld
xor %eax, %eax
mov %eax, %ds
mov %eax, %es
mov %eax, %ss
in $0x64, %al
test $0x2, %al
jne 0x7c0a
mov $0xd1, %al
out %al, $0x64
in $0x64, %al
test $0x2, %al
jne 0x7c14
mov $0xdf, %al
out %al, $0x60
lgdtl (%esi)
insb (%dx), %es: (%edi)
jl 0x7c33
and %al, %al
or $0x1, %ax
mov %eax, %cr0
ljmp $0xb866, $0x87c32
adc %al, (%eax)
mov %eax, %ds
mov %eax, %es
mov %eax, %fs
mov %eax, %gs
mov %eax, %ss
mov $0x0, %ebp
mov $0x7c00, %esp
call 0x7d0d
jmp 0x7c4f
lea 0x0(%esi), %esi
```

bootasm.S	test1.asm
cli	cli
cld	cld
# Set up the important data segment register	xor %eax,%eax
xorw %ax, %ax	mov %eax,%ds
movw %ax, %ds	mov %eax,%es
movw %ax, %es	mov %eax,%ss
movw %ax, %ss	in \$0x64,%al
# Enable A20:	test \$0x2,%al
# For backwards compatibility with the earl	jne 0x7c0a
# address line 20 is tied low, so that addr	mov \$0xd1,%al
# 1MB wrap around to zero by default. This	out %al,\$0x64
a20.1:	in \$0x64,%al
inb \$0x64, %al	test \$0x2,%al
testb \$0x2, %al	jne 0x7c14
jnz seta20.1	mov \$0xdf,%al
movb \$0xd1, %al	out %al,\$0x60
outb %al, \$0x64	lgdtl (%esi)
a20.2:	insb (%dx),%es:(%edi)
inb \$0x64, %al	j1 0x7c33
testb \$0x2, %al	and %al,%al
jnz seta20.2	or \$0x1,%ax
movb \$0xdf, %al	mov %eax,%cr0
outb %al, \$0x60	ljmp \$0xb866,\$0x87c32
# Switch from real to protected mode, using	adc %al,(%eax)
# and segment translation that makes virtual	mov %eax,%ds
# identical to physical addresses, so that t	mov %eax,%es
# effective memory map does not change durir	mov %eax,%fs
lgdt gdt desc	mov %eax,%gs
movl %cr0, %eax	mov %eax,%ss
orl \$CR0_PE_ON, %eax	mov \$0x0,%ebp
movl %eax, %cr0	mov \$0x7c00,%esp
	call 0x7d0d
	jmp 0x7c4f
	lea 0x0(%esi),%esi
	add %al,(%eax)
	add %al,(%eax)
	add %al,(%eax)

## T4: 找一个bootloader或内核中的代码位置，设置断点并进行测试

对kern\_init()代码进行调试

```
int
kern_init(void) {
    .....
    print_kerninfo();
    .....
    pmm_init();           // init physical memory management
    .....
}
```



```
(gdb) b print_kerninfo
```

```
No symbol table is loaded. Use the "file" command.
```

```
Make breakpoint pending on future shared library load? (y or [n]) y
```

```
Breakpoint 1 (print_kerninfo) pending.
```

```
(gdb) x/i $pc
```

```
=> 0xf195b:      rep movsb %ds:(%esi),%es:(%edi)
```

## 练习3：分析bootloader进入保护模式的过程

BIOS的初始化工作做完后，进一步的工作交给了ucore的bootloader，bootloader让CPU进入保护模式，启用分段机制。这个练习中的代码都来自于lab1/boot/bootasm.S源码，看了“保护模式和分段机制”这一小节，但是里面的词长得都太像了，我对这个空间的映射变换也不太清楚，这一部分没太看懂，幸好bootasm.S的注释很详细，大体上还是能理解的。

### Q1：为何开启A20，如何开启A20

为了满足系统升级的兼容性问题，设置了A20地址线，修改A20地址线可以完成从实模式到保护模式的转换，只有在保护模式下才能使用分段存储管理机制，访问1MB以上的内存。

```
# 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.
```

bootasm.S最开始先为段选择器赋值，cli关中断、cld清除方向标志、将段寄存器清零，之后通过设置8042芯片端口打开A20地址线的代码，但在写8024端口是，需要对可能存在的缓冲数据进行处理。

```
seta20.1:      #等待直到8042 Input buffer为空为止
    inb $0x64, %al      #从0x64端口中读入一个字节到al中
    testb $0x2, %al
    jnz seta20.1      #al的第2位为0，则跳出循环

    movb $0xd1, %al      # 0xd1 -> port 0x64
    outb %al, $0x64      #将0xd1写入到0x64端口中

seta20.2:
    inb $0x64, %al      #等待8042键盘控制器不忙
    testb $0x2, %al
    jnz seta20.2      #al的第2位为0，则跳出循环

    movb $0xdf, %al      # 0xdf -> port 0x60
    outb %al, $0x60      # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1
```

### Q2：如何初始化GDT表

段表：GDT（Global Descriptor Table，全局描述符表），第一个段描述符设定为空段描述符。

```
lgdt gdt_desc    #载入GDT表
movl %cr0, %eax  #加载cr0到eax
orl $CR0_PE_ON, %eax #将eax的第0位置为1
movl %eax, %cr0  #将cr0的第0位置为1，cr0的第0位为1表示处于保护模式

# 使用逻辑地址，跳到下一条指令
ljmp $PROT_MODE_CSEG, $protcseg

.code32          # Assemble for 32-bit mode
protcseg:
    # 设置保护模式下的段寄存器和堆栈
    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
    movl $start, %esp
    call bootmain    #调用bootmain函数
```

ljmp指令的功能：跳转到由PROT\_MODE\_CSEG段选择子和protcseg偏移量指定的代码入口处。而跳转到的代码所完成的工作是：通过.code32伪指令编码的对各段寄存器的初始化，随后利用start标号指向的地址作为esp调用main.c中的bootmain()函数。由于上面的代码已经打开了保护模式了，所以这里要使用逻辑地址，而不是之前实模式的地址了。PROT\_MODE\_CSEG的值是0x8。根据段选择子的格式定义，0x8就翻译成：

INDEX     TI CPL            0000 0000 0000 1 00 0 INDEX代表GDT中的索引，TI代表使用GDTR中的GDT，CPL代表处于特权级。

## Q3：如何使能和进入保护模式

cr0的第0位为1表示处于保护模式，切换到保护模式：

```
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
```

## 练习4：分析bootloader加载ELF格式的OS的过程

这一节中的代码都来自于lab1/boot/bootmain.c

### Q1：bootloader如何读取硬盘扇区的？

操作系统文件存在0号硬盘上，bootloader的所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成，硬盘的每个扇区大小为512字节。

readsect实现了读一个扇区的功能：

```

static void
readsect(void *dst, uint32_t secno) {
    //等待磁盘准备好
    waitdisk();
    //发出读取扇区的命令
    outb(0x1F2, 1); //读取一个扇区, 0x1F2是0号硬盘数据扇区计数端口
    outb(0x1F3, secno & 0xFF); //要读取的扇区编号, 0x1F3是0号硬盘扇区数
    outb(0x1F4, (secno >> 8) & 0xFF); //存放读写柱面的低8位字节, 0x1F4是0号硬盘柱面 (低字节)
    outb(0x1F5, (secno >> 16) & 0xFF); //存放读写柱面的高2位字节, 0x1F5是0号硬盘柱面 (高字节)
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0); // 0x1F6端口为0号硬盘驱动器, 存放要读/写的磁
    盘号及磁头号
    outb(0x1F7, 0x20); //0x1F7是0号硬盘状态寄存器和命令寄存器
    //等待磁盘准备好
    waitdisk();
    //把磁盘扇区数据读到指定内存
    insl(0x1F0, dst, SECTSIZE / 4); //0x1F0是0号硬盘数据寄存器
}

```

readseg包装了readsect, 可以从磁盘读取任意长度的内容。

## Q2: bootloader是如何加载ELF格式的OS?

ELF(Executable and linking format)文件格式在本实验中用于提供程序的进程映像, 加载的内存执行。

```

void
bootmain(void) {
    // 读取ELF的头部,从硬盘读8个扇区数据到内存0x10000处
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // e_magic用来判断读出来的ELF格式的文件是否为正确的格式
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags), ELF头部有描述ELF文件应加载到内存什么位置
    的描述表, 描述表的头地址存在ph. e_phoff, 是program header表的位置偏移; e_phnum, 是program header
    表中的入口数目
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    //将ELF文件中数据载入内存
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFF, ph->p_memsz, ph->p_offset);
    }

    // call the entry point from the ELF header
    // 内核的入口
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
}

```

```

    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}

```

## 练习5：实现函数调用堆栈跟踪函数

EBP其内存放一个指针，该指针指向系统栈最上面一个栈帧的底部。EIP存储着下一条指令的地址，每执行一条指令，该寄存器变化一次。

kern/debug/kdebug.c::print\_stackframe()

```

void
print_stackframe(void) {
    uint32_t v_ebp=read_ebp();
    uint32_t v_eip=read_eip();
    for(int i=0;i<STACKFRAME_DEPTH&&v_ebp!=0;++i)
    {
        cprintf("ebp: 0x%08x , eip: 0x%08x , ",v_ebp,v_eip);
        uint32_t *args=(uint32_t *)v_ebp +2;
        for(int j=0;j<4;++j)
        {
            cprintf("args[%d] = 0x%08x  ",j,args[j]);
        }
        cprintf("\n");
        print_debuginfo(v_eip-1);
        v_eip=((uint32_t *)v_ebp)[1];    //v_ebp[1]是返回地址
        v_ebp=((uint32_t *)v_ebp)[0];    //v_ebp[0]是上一层的ebp
    }
}

```

执行make qemu，输出结果：

```

ebp: 0x00007b38 , eip: 0x00100bcc , args[0] = 0x00010094   args[1] = 0x0010e950
args[2] = 0x00007b68   args[3] = 0x001000a2
    kern/debug/kdebug.c:306: print_stackframe+33
ebp: 0x00007b48 , eip: 0x00100f50 , args[0] = 0x00000000   args[1] = 0x00000000
args[2] = 0x00000000   args[3] = 0x0010008d
    kern/debug/kmonitor.c:125: mon_backtrace+23
ebp: 0x00007b68 , eip: 0x001000a2 , args[0] = 0x00000000   args[1] = 0x00007b90
args[2] = 0xffff0000   args[3] = 0x00007b94
    kern/init/init.c:48: grade_backtrace2+32
ebp: 0x00007b88 , eip: 0x001000d1 , args[0] = 0x00000000   args[1] = 0xffff0000
args[2] = 0x00007bb4   args[3] = 0x001000e5
    kern/init/init.c:53: grade_backtrace1+37
ebp: 0x00007ba8 , eip: 0x001000f8 , args[0] = 0x00000000   args[1] = 0x00100000
args[2] = 0xffff0000   args[3] = 0x00100109
    kern/init/init.c:58: grade_backtrace0+29
ebp: 0x00007bc8 , eip: 0x00100124 , args[0] = 0x00000000   args[1] = 0x00000000
args[2] = 0x00000000   args[3] = 0x001037a8
    kern/init/init.c:63: grade_backtrace+37

```

```

ebp: 0x00007be8 , eip: 0x00100066 , args[0] = 0x00000000   args[1] = 0x00000000
args[2] = 0x00000000   args[3] = 0x00007c4f
    kern/init/init.c:28: kern_init+101
ebp: 0x00007bf8 , eip: 0x00007d6e , args[0] = 0xc031fcfa   args[1] = 0xc08ed88e
args[2] = 0x64e4d08e   args[3] = 0xfa7502a8
<unknown>: -- 0x00007d6d --

```

当bootloader通过读取硬盘扇区把ucore在系统加载到内存后，就转跳到ucore操作系统在内存中的入口位置（kern/init.c中的kern\_init函数的起始地址），这样ucore就接管了整个控制权。输出结果的最后一行，bootloader设置的堆栈从0x7c00开始，使用“call bootmain”转入bootmain函数，call指令压栈，所以bootmain中ebp为0x7bf8。eip的值对应调用kern\_init后的指令地址，由于kern\_init()并没有参数传入，因此这里输出的是bootloader的二进制代码。

lab1/obj/bootblock.asm:

```

# Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
    movl $0x0, %ebp
7c40:   bd 00 00 00 00      mov     $0x0,%ebp
    movl $start, %esp
7c45:   bc 00 7c 00 00      mov     $0x7c00,%esp
    call bootmain
7c4a:   e8 be 00 00 00      call    7d0d <bootmain>

```

lab1/obj/kernel.asm:

```

/* *
 * mon_backtrace - call print_stackframe in kern/debug/kdebug.c to
 * print a backtrace of the stack.
 * */
int
mon_backtrace(int argc, char **argv, struct trapframe *tf) {
    100f32:   55                push    %ebp
    100f33:   89 e5             mov     %esp,%ebp
    100f35:   53                push    %ebx
    100f36:   83 ec 04          sub     $0x4,%esp
    100f39:   e8 3e f3 ff ff    call    10027c <__x86.get_pc_thunk.ax>
    100f3e:   05 12 da 00 00    add     $0xda12,%eax
    print_stackframe();
    100f43:   89 c3             mov     %eax,%ebx
    100f45:   e8 60 fc ff ff    call    100baa <print_stackframe>
    return 0;
    100f4a:   b8 00 00 00 00    mov     $0x0,%eax
}

```

关于cprintf(): %08x 以16进制输出后续对应参数的整型数字，08的含义为，输出占8位，不足部分左侧补0；

第一遍写的时候把下标j误写成i了，导致结果输出异常；

最开始照着注释写，外层循环里没有加v\_ebp!=0条件，导致输出结果后半部分反复出现下面两句，ebp=0时，应该返回调用的位置。

```

ebp: 0x00000000 , eip: 0x00000000 , args[0] = 0xf000e2c3   args[1] = 0xf000ff53
args[2] = 0xf000ff53   args[3] = 0xf000ff54
<unknow>: -- 0xffffffff --
ebp: 0xf000ff53 , eip: 0xf000ff53 , args[0] = 0x00000000   args[1] = 0x00000000
args[2] = 0x00000000   args[3] = 0x00000000
<unknow>: -- 0xf000ff52 --

```

(这一部分的代码照着注释还是比较好写的，不过那些地址的对应我还是没有完全弄清楚。)

## 练习6：完善中断初始化和处理

中断描述符表（IDT，保护模式下的中断向量表）中一个表项占8个字节，其中0~15位和48~63位分别为offset的低16位和高16位。16~31位为中断服务例程的段寄存器。通过段寄存器获得段基址，加上段内偏移量即可得到中断处理代码的入口。

问题二中需要用到的已有的代码段：

mmu.h中的SETGATE宏：

```

//gate即idt[]中的某一项。istrap:陷阱门设为1，中断门设为0。sel:段选择子
#define SETGATE(gate, istrap, sel, off, dpl) { \
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff; \
    (gate).gd_ss = (sel); \
    (gate).gd_args = 0; \
    (gate).gd_rsv1 = 0; \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32; \
    (gate).gd_s = 0; \
    (gate).gd_dpl = (dpl); \
    (gate).gd_p = 1; \
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16; \
}

```

kern/mm/memlayout.h

```

/* global segment number */
#define SEG_KTEXT    1
/* global descriptor numbers */
#define GD_KTEXT      ((SEG_KTEXT) << 3)           // kernel text

#define DPL_KERNEL    (0)
#define DPL_USER      (3)

```

trap.h:

```

#define T_SWITCH_TOU    120    // 从内核态跳转到用户态
#define T_SWITCH_TOK    121    // 从用户态跳转到内核态

```

## 编程

kern/trap/trap.c::idt\_init()



```

void
idt_init(void) {
    extern uintptr_t __vectors[];
    int num=sizeof(idt)/sizeof(struct gatedesc);//256
    for(int i=0;i<num;++i)
    {
        //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
        SETGATE(idt[i],0,GD_KTEXT,__vectors[i],DPL_KERNEL);
    }
    //除了系统调用中断(T_SYSCALL)使用陷阱门描述符且权限为用户态权限以外, 其它中断均使用特权级
    (DPL)为0的中断门描述符, 权限为内核态权限;
    SETGATE(idt[T_SWITCH_TOU],0,GD_KTEXT,__vectors[T_SWITCH_TOU],DPL_USER);
    //指令lidt把中断门描述符表的起始地址装入IDTR寄存器中
    lidt(&idt_pd);
}

```

print\_ticks:

```

ticks++;
if(ticks==TICK_NUM)
{
    ticks=0;
    print_ticks();
}

```

make qemu的输出结果:

```

ebp: 0x00007bf8 , eip: 0x00007d6e , args[0] = 0xc031fcfa   args[1] = 0xc08ed88e
args[2] = 0x64e4d08e   args[3] = 0xfa7502a8
<unknown>: -- 0x00007d6d --
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks

```

编程的过程中, idt\_init写的有些艰难, 因为很多已有的变量和函数都不太了解, 比如SEG\_KTEXT, 所以是对照着答案边理解边谷歌边写的。

ISR: 中断服务程序

CPL: 当前执行的程序或任务的特权级

## 拓展练习

需要增加一用户态函数，当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务。这一部分主要修改了五个地方：init.c中的switch\_to\_user和switch\_to\_kernel，trap.c中的idt\_init()、trap\_dispatch()的case T\_SWITCH\_TOU和case T\_SWITCH\_TOK。

为了使得能够在用户态下产生中断号为T\_SWITCH\_TOK的软中断，需要在IDT初始化的时候，将该终端号对应的表项的DPL设置为3。在idt\_init()里面增加下面的语句，将用户态调用SWITCH\_TOK中断的权限打开。

```
SETGATE(idt[T_SWITCH_TOK], 1, KERNEL_CS, __vectors[T_SWITCH_TOK], 3);
```

init.c中的switch\_to\_user和switch\_to\_kernel:

```
static void
lab1_switch_to_user(void) {
    //从中断返回时，会多pop两位，并用这两位值更新ss,sp，损坏堆栈。
    //所以要先将栈压两位，并在从中断返回后修复esp。
    asm volatile(
        "sub $0x8,%%esp \n"
        "int 0 \n" //调用了int指令之后，会最终跳转到T_SWITCH_TOU终端号对应的ISR入口，最终跳转到
        trap_dispatch函数处统一处理
        "movl %%ebp,%%esp" //恢复栈指针
        :
        : "i"(T_SWITCH_TOU)
    );
}

static void
lab1_switch_to_kernel(void) {
    //从中断返回时，esp仍在TSS指示的堆栈中。所以要在从中断返回后修复esp
    //把tf->tf_cs和tf->tf_ds都设置为内核代码段和内核数据段
    asm volatile(
        "int 0 \n" //调用T_SWITCH_TOK号中断
        "movl %%ebp, %%esp \n" //强行改为内核态，会让cpu认为没有发生特权级转换，需要在中断返回之后
        将栈上保存的原本应当被恢复的esp给pop回到esp上去
        :
        : "i"(T_SWITCH_TOK)
    );
}
```

trap\_dispatch()的case T\_SWITCH\_TOU和case T\_SWITCH\_TOK:

```

case T_SWITCH_TOU:
    //将调用io所需权限降低, 才能正常输出文本
    tf->tf_eflags |= FL_IOPL_MASK;
    tf->tf_cs = USER_CS;
    tf->tf_ds = tf->tf_es = tf->tf_gs = tf->tf_ss = tf->tf_fs = USER_DS;
    break;
case T_SWITCH_TOK:
    //将trapframe中保存的cs修改为指向DPL为0的段描述子的段选择子KERNEL_CS, 并且将ds, es, ss, gs, fs也
    相应地修改为KERNEL_DS, 然后进行正常的中断返回
    tf->tf_cs = KERNEL_CS;
    tf->tf_ds = tf->tf_es = tf->tf_gs = tf->tf_ss = tf->tf_fs = KERNEL_DS;
    break;

```

最后make grade, 结果如下图所示:

```

pcy@ubuntu:~/Desktop/labcodes/lab1$ make grade
Check Output: (2.2s)
-check ring 0: OK
-check switch to ring 3: OK
-check switch to ring 0: OK
-check ticks: OK
Total Score: 40/40

```