**练习一**

PROJ := challenge

EMPTY :=

SPACE := $(EMPTY) $(EMPTY)

SLASH := /

V := @

#定义了五个变量PROJ、EMPTY、SPACE、SLASH、V

#need llvm/cang-3.5+

#USELLVM := 1

# try to infer the correct GCCPREFX

ifndef GCCPREFIX

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

#如果没有定义GCCPREFIX，则运行shell脚本赋值

then echo 'i386-elf-'; \

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

#执行两条语句中的一条

1. 将标准错误重新定向到标准输出
2. 通过grep查找上调至领的输出中是否有elf32-i386开头并加上shell本身的PID的字符，同时将grep的输出重定向到/dev/null

then echo ''; \

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

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

#错误：找不到gcc/binutils的i386 elf版本

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

# try to infer the correct QEMU

#使用shell脚本查找qemu-system-i386文件

#如果是则打印qemu-system-i386然后退出

#如果不是则查找i386-elf-qemu文件

#如果有则打印i386-elf-qemu然后退出

#如果没有则查找qemu文件

#如果有则打印qemu然后退出

#如果没有则打印相关错误信息

#QEMU会被赋值为未被重定向的标准输出

ifndef QEMU

QEMU := $(shell if which qemu-system-i386 > /dev/null; \

then echo 'qemu-system-i386'; exit; \

elif which i386-elf-qemu > /dev/null; \

then echo 'i386-elf-qemu'; exit; \

elif which qemu > /dev/null; \

then echo 'qemu'; exit; \

else \

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

echo "\*\*\* Error: Couldn't find a working QEMU executable." 1>&2; \

echo "\*\*\* Is the directory containing the qemu binary in your PATH" 1>&2; \

echo "\*\*\*" 1>&2; exit 1; fi)

endif

# eliminate default suffix rules

.SUFFIXES: .c .S .h

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

.DELETE\_ON\_ERROR:

#设置编译器选项

# define compiler and flags

ifndef USELLVM

# hostcc是给主机用的编译器，按照主机格式。cc是i386,elf32格式的编译器。

HOSTCC := gcc

# -g 是为了gdb能够对程序进行调试

# -Wall 生成警告信息

# -O2 优化处理（0,1,2,3表示不同的优化程度，0为不优化）

HOSTCFLAGS := -g -Wall -O2

CC := $(GCCPREFIX)gcc

# -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)

#如果-fno-stack-protector选项存在，就添加它。过程蛮复杂的。

# -fstack-protector-all 启用堆栈保护,为所有函数插入保护代码

# -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

# 前一个/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

# /dev/null是一个垃圾桶一样的东西

# ‘&&’之前的半句表示，试着对一个垃圾跑一下这个命令，所有的输出都作为垃圾，为了快一点，开了-E。

# 如果不能运行，那么&&前面的条件不成立，后面的就被忽视。

# 如果可以运行，那么&&后面的句子得到执行，于是CFLAGS += -fno-stack-protector

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

else

HOSTCC := clang

HOSTCFLAGS := -g -Wall -O2

CC := $(GCCPREFIX)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

#源文件类型为c和S

CTYPE := c S

#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

LD := $(GCCPREFIX)ld

LDFLAGS := -m $(shell $(LD) -V | grep elf\_i386 2>/dev/null)

LDFLAGS += -nostdlib

OBJCOPY := $(GCCPREFIX)objcopy

OBJDUMP := $(GCCPREFIX)objdump

#定义一些命令

COPY := cp

MKDIR := mkdir -p

MV := mv

RM := rm -f

AWK := awk

SED := sed

SH := sh

TR := tr

TOUCH := touch -c

OBJDIR := obj

BINDIR := bin

ALLOBJS :=

ALLDEPS :=

TARGETS :=

include tools/function.mk

#列出$(1)中所有.c和.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 = $(patsubst %.$(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 $$?)

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

# include kernel/user

INCLUDE += libs/

CFLAGS += $(addprefix -I,$(INCLUDE))

LIBDIR += libs

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

# -------------------------------------------------------------------

# kernel

KINCLUDE += kern/debug/ \

kern/driver/ \

kern/trap/ \

kern/mm/

KSRCDIR += kern/init \

kern/libs \

kern/debug \

kern/driver \

kern/trap \

kern/mm

KCFLAGS += $(addprefix -I,$(KINCLUDE))

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

#将kernel、libs加上前缀\_\_objs\_ 并返回

KOBJS = $(call read\_packet,kernel libs)

# create kernel target

#为kernel加上 bin/ 的前缀

kernel = $(call totarget,kernel)

$(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)

$(call create\_target,kernel)

# -------------------------------------------------------------------

# create bootblock

#列出boot中所有.c和.S文件，bootasm.S和bootmain.c

bootfiles = $(call listf\_cc,boot)

$(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc)) # -I libs/

bootblock = $(call totarget,bootblock)

$(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)

$(call create\_target,bootblock)

# -------------------------------------------------------------------

# create 'sign' tools

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

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

# -------------------------------------------------------------------

# create ucore.img

#为ucore.img加上 bin/ 的前缀

UCOREIMG := $(call totarget,ucore.img)

#if=/dev/zero if=文件名：输入文件名，缺省为标准输入。

#/dev/zero ： 在类UNIX 操作系统中, /dev/zero 是一个特殊的文件，

#当你读它的时候，它会提供无限的空字符(NULL, ASCII NUL, 0x00)。

#Makefile有三个非常有用的变量。分别是$@，$^，$<代表的意义分别是：

#$@--目标文件，$^--所有的依赖文件，$<--第一个依赖文件。

#下面的of=$@中的$@指的是UCOREIMG

#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。

#首先分配一块10000\*512字节的空间给UCOREIMG

#将bootblock拷贝到UCOREIMG

#root@moocos-VirtualBox:/home/moocos/codeTest# dd if=/dev/zero of=zero.txt bs=1024 count=1

#1+0 records in

#1+0 records out

#1024 bytes (1.0 kB) copied, 0.000305041 s, 3.4 MB/s

#root@moocos-VirtualBox:/home/moocos/codeTest# ls

#hello helloworld.c zero.txt

#root@moocos-VirtualBox:/home/moocos/codeTest# ls -l zero.txt

#-rw-r--r-- 1 root root 1024 1月 21 19:05 zero.txt

#root@moocos-VirtualBox:/home/moocos/codeTest# dd if=/dev/zero of=zero.txt bs=512 count=1 conv=notrunc

#1+0 records in

#1+0 records out

#512 bytes (512 B) copied, 0.0002284 s, 2.2 MB/s

#root@moocos-VirtualBox:/home/moocos/codeTest# ls -l zero.txt

#-rw-r--r-- 1 root root 1024 1月 21 19:06 zero.txt

#root@moocos-VirtualBox:/home/moocos/codeTest# dd if=/dev/zero of=zero.txt bs=512 count=1

#1+0 records in

#1+0 records out

#512 bytes (512 B) copied, 0.000203208 s, 2.5 MB/s

#root@moocos-VirtualBox:/home/moocos/codeTest# ls -l zero.txt

#-rw-r--r-- 1 root root 512 1月 21 19:06 zero.txt

#root@moocos-VirtualBox:/home/moocos/codeTest# dd if=/dev/zero of=zero.txt count=100

#100+0 records in

#100+0 records out

#51200 bytes (51 kB) copied, 0.000504666 s, 101 MB/s

#root@moocos-VirtualBox:/home/moocos/codeTest# ls -l zero.txt

#-rw-r--r-- 1 root root 51200 1月 21 19:07 zero.txt

#上述实验表明，在不指定bs、ibs、obs的初值的情况下，其默认一块的大小为512字节

#conv=notrunc 不截短输出文件，在拷贝的大小小于输出文件大小情况下不截断输出文件大小

#那么下面的操作便是

#UCOREIMG 依赖于 kernel 和 bootblock两个文件

#首先分配一块10000 \* 512 字节大小给目标文件UCOREIMG

#将bootblock（大小为512个字节）拷贝到UCOREIMG

#将kernel拷贝到目标文件，从输出文件开头跳过seek个块后再开始复制。 （seek \* bs）个字节

$(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

$(call create\_target,ucore.img)

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

$(call finish\_all)

IGNORE\_ALLDEPS = clean \

dist-clean \

grade \

touch \

print-.+ \

handin

ifeq ($(call match,$(MAKECMDGOALS),$(IGNORE\_ALLDEPS)),0)

-include $(ALLDEPS)

endif

# files for grade script

TARGETS: $(TARGETS)

.DEFAULT\_GOAL := TARGETS

.PHONY: qemu qemu-nox debug debug-nox

qemu-mon: $(UCOREIMG)

$(V)$(QEMU) -monitor stdio -hda $< -serial null

qemu: $(UCOREIMG)

$(V)$(QEMU) -parallel stdio -hda $< -serial null

qemu-nox: $(UCOREIMG)

$(V)$(QEMU) -serial mon:stdio -hda $< -nographic

TERMINAL :=gnome-terminal

debug: $(UCOREIMG)

$(V)$(QEMU) -S -s -parallel stdio -hda $< -serial null &

$(V)sleep 2

$(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"

debug-nox: $(UCOREIMG)

$(V)$(QEMU) -S -s -serial mon:stdio -hda $< -nographic &

$(V)sleep 2

$(V)$(TERMINAL) -e "gdb -q -x tools/gdbinit"

.PHONY: grade touch

GRADE\_GDB\_IN := .gdb.in

GRADE\_QEMU\_OUT := .qemu.out

HANDIN := proj$(PROJ)-handin.tar.gz

TOUCH\_FILES := kern/trap/trap.c

MAKEOPTS := --quiet --no-print-directory

grade:

$(V)$(MAKE) $(MAKEOPTS) clean

$(V)$(SH) tools/grade.sh

touch:

$(V)$(foreach f,$(TOUCH\_FILES),$(TOUCH) $(f))

print-%:

@echo $($(shell echo $(patsubst print-%,%,$@) | $(TR) [a-z] [A-Z]))

.PHONY: clean dist-clean handin packall tags

clean:

$(V)$(RM) $(GRADE\_GDB\_IN) $(GRADE\_QEMU\_OUT) cscope\* tags

-$(RM) -r $(OBJDIR) $(BINDIR)

dist-clean: clean

-$(RM) $(HANDIN)

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)'`

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

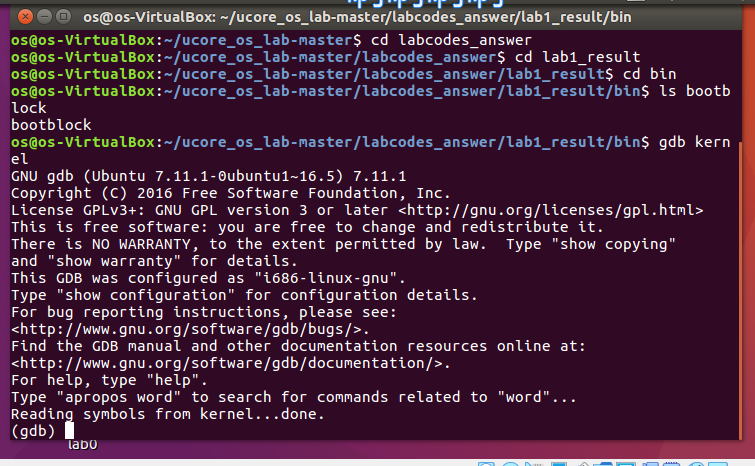
$(V)cscope -bq

$(V)ctags -L cscope.files

**\*\*\*多有借鉴\*\*\***

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，做完这样的检查才能认为是符合规范的磁盘主引导扇区。

**练习二**



**练习三**

1.为何开启A20，以及如何开启A20

  a）何为A20

  intel早起8086CPU提供了20根地址线，寻址空间为0~2^20（00000H~FFFFF）的1MB内存空间，但是8086数据处理位宽为16位，

  无法直接寻址1MB内存空间；8086提供了段地址加偏移地址的地址转换机制，这样寻址空间最大为0ffff0h+0ffffh = 10ffefh（大约1088KB），

  也就是说8086在使用这套寻址机制时寻址能力大于1MB，寻址超过1MB内存时会发生回卷。在基于intel 80286CPU 计算机系统提供24根地址线，

  此时寻址超过1MB时系统不会再发生回卷，这样便造成了向下不兼容，为了保持完全的向下兼容性，IMB在计算机系统上加了个硬件逻辑，

  来模仿以上的回绕特征，于是出现了A20 Gate，其实是第21根地址线，在实模式下一直被拉低，寻址超过1MB时，也会回卷。这样就能够向下兼容

  b）为何要开启A20

  在80286、80386以后，地址线来到了24根和32根，同时也提供了保护模式，如果A20一直置0的话，便无法寻到所有的地址空间，

  所以在保护模式中或者实模式时需要访问高端内存区，均需要将A20地址线开启

  c）如何开启A20

  在Bootblock.asm中有这样一段代码，是开启A20的：

00007c0a <seta20.1>:

# 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).

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 # 0xd1 -> port 0x64

7c10: b0 d1 mov $0xd1,%al

outb %al, $0x64 # 0xd1 means: write data to 8042's P2 port

7c12: e6 64 out %al,$0x64

00007c14 <seta20.2>:

seta20.2:

inb $0x64, %al # Wait for not busy(8042 input buffer empty).

7c14: e4 64 in $0x64,%al

testb $0x2, %al

7c16: a8 02 test $0x2,%al

jnz seta20.2

7c18: 75 fa jne 7c14 <seta20.2>

movb $0xdf, %al # 0xdf -> port 0x60

7c1a: b0 df mov $0xdf,%al

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

7c1c: e6 60 out %al,$0x60

初始化GDT表：lgdt gdtdesc

如何使能和进入保护模式

CR0的位0是启用保护（Protection Enable）标志。当设置该位时即开启了保护模式

当复位时即进入实地址模式。

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0