一．

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)

-ggdb : 生成可供gdb使用的调试信息

-m32 :生成适用于32位环境的代码

-gstabs ：生成stabs格式的调试信息

-nostdinc ：不在标准系统文件夹寻找头文件，只在-I等参数指定的文件夹中搜索头文件

-fno-stack-protector ：不生成用于检测缓冲区溢出的代码

-Os 为减小代码大小而进行优化

————————————————

+ 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 //编译init.c输出到init.o

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

lab1\_switch\_test(void) {

^

+ 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 //编译readline.c输出到readline.o

+ 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 //编译stdio.c输出到stdio.o

+ 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 //编译kdebug.c输出到kdebug.o

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

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/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 //编译panic.c输出到panic.o

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

kern/debug/panic.c:27:5: warning: implicit declaration of function ‘print\_stackframe’ [-Wimplicit-function-declaration]

print\_stackframe();

^

+ 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 //编译clock.c输出到init.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 //编译console.c输出到console.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 //编译intr.c输出到intr.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/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:14:13: warning: ‘print\_ticks’ defined but not used [-Wunused-function]

static void print\_ticks() {

^

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

static struct pseudodesc idt\_pd = {

^

+ 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/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/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/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

+ 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 //编译string.c输出到string.o

+ ld bin/kernel

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 -m模拟指定的连接器 模拟i386链接成kernel

+ 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 //链接生成bootblock.o

'obj/bootblock.out' size: 472 bytes

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

if=文件名：输入文件名，默认为标准输入。即指定源文件

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

conv=<关键字>notrunc：不截短输出文件

dd if=/dev/zero of=bin/ucore.img count=10000 //输入ucore.img 10000个块，即创建10000个块大小的ucore.img文件

记录了10000+0 的读入

记录了10000+0 的写出

5120000字节(5.1 MB)已复制，0.0258643 秒，198 MB/秒

dd if=bin/bootblock of=bin/ucore.img conv=notrunc //把bootblock不截短的输入到ucore

记录了1+0 的读入

记录了1+0 的写出

512字节(512 B)已复制，0.000100014 秒，5.1 MB/秒

dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc //把kernel不截短的输入到ucore

记录了146+1 的读入

记录了146+1 的写出

74875字节(75 kB)已复制，0.000340129 秒，220 MB/秒

综上生成过程为

1. 先编译kern文件夹中的文件
2. 链接并生成kernel
3. 编译bootasm.S，bootmain.c，sign.c
4. 链接并生成bootblock.o
5. 将kernel和bootblock输入并生成ucore

2.一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？

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <sys/stat.h>

int

main(int argc, char \*argv[]) {

struct stat st;

if (argc != 3) {

fprintf(stderr, "Usage: <input filename> <output filename>\n");

return -1;

}

if (stat(argv[1], &st) != 0) {

fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));

return -1;

}

printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st\_size);

if (st.st\_size > 510) {

fprintf(stderr, "%lld >> 510!!\n", (long long)st.st\_size);

return -1;

}

char buf[512];

memset(buf, 0, sizeof(buf));

FILE \*ifp = fopen(argv[1], "rb");

int size = fread(buf, 1, st.st\_size, ifp);

if (size != st.st\_size) {

fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);

return -1;

}

fclose(ifp);

buf[510] = 0x55;

buf[511] = 0xAA;//最后两位

FILE \*ofp = fopen(argv[2], "wb+");

size = fwrite(buf, 1, 512, ofp);

if (size != 512) {//size=512

fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);

return -1;

}

fclose(ofp);

printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);

return 0;

}

1. 大小为512字节
2. 倒数第二位为0x55
3. 倒数第三位为0xAA

# 作业二

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行



用gdb查看内存

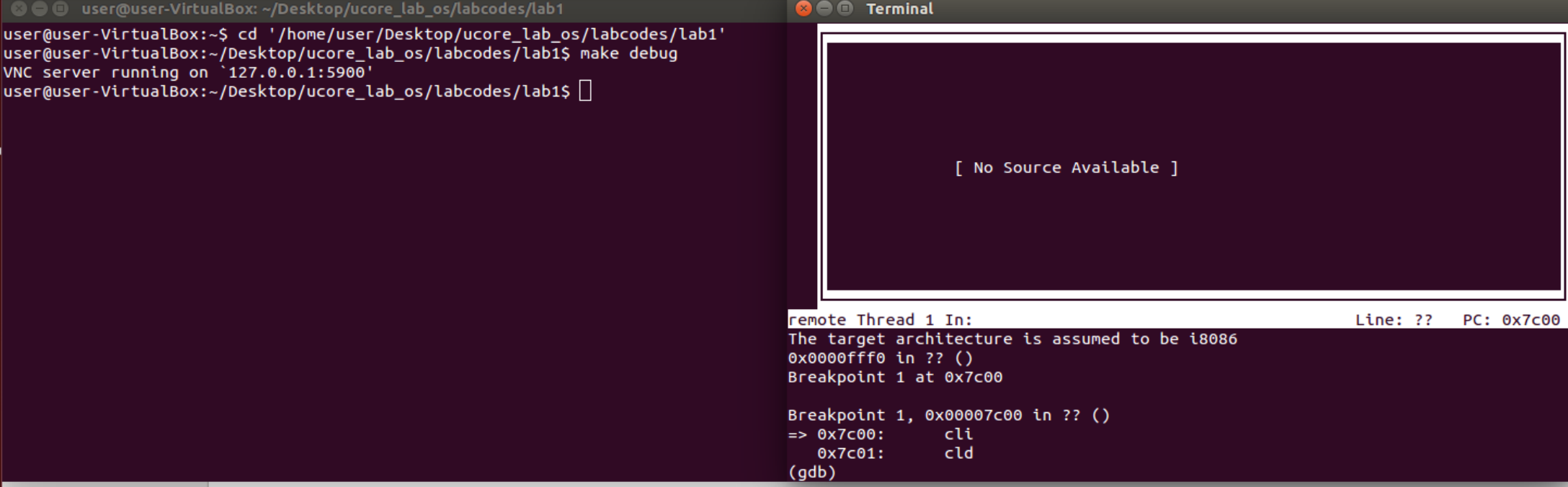
格式: x /nfu <addr>

说明  
x 是 examine 的缩写

n表示要显示的内存单元的个数

1. 在初始化位置0x7c00设置实地址断点,测试断点正常。

Gdb b+行号 直接在当前所在的文件添加断点



问题：

1.开第二遍debug前要先关闭端口

## 通过杀掉进程的方法来关闭端口

每个端口都有一个守护进程，kill掉这个守护进程就可以了

每个端口都是一个进程占用着，

第一步、用下面命令

netstat -anp |grep 端口

找出占用这个端口的进程，

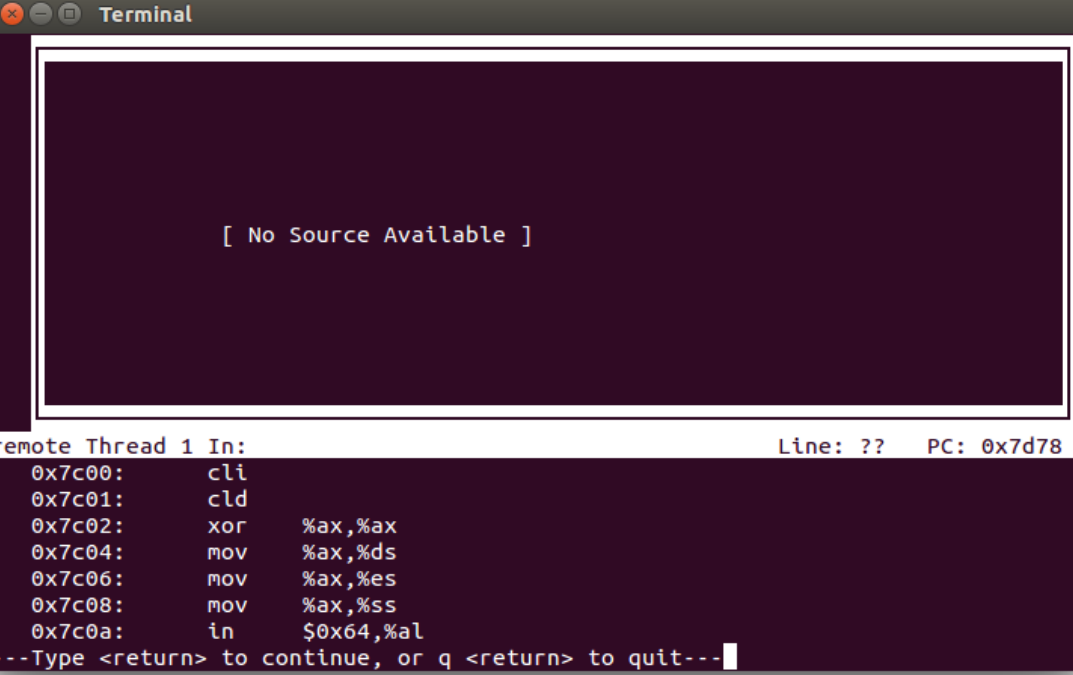
第二步、用下面命令

kill -9 PID

杀掉就行了

2.地址前要加个\*

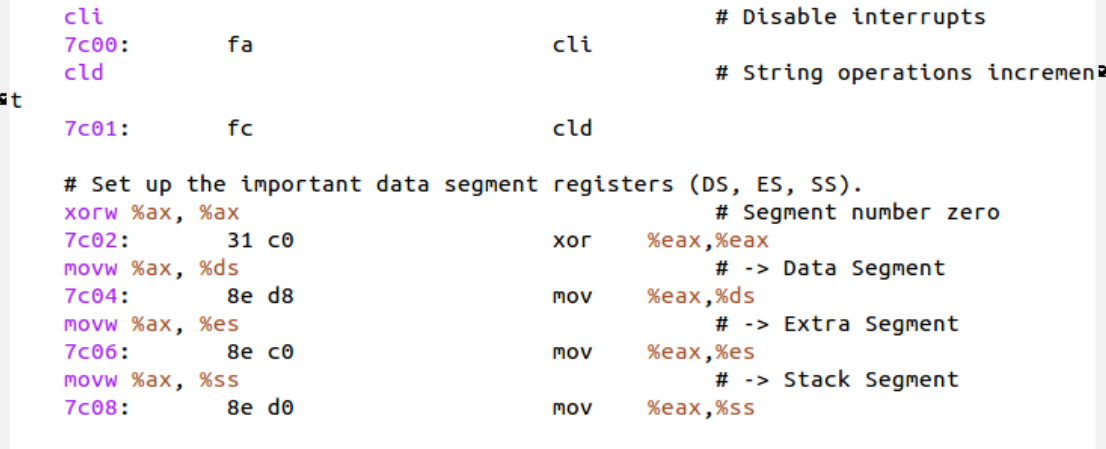
从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。



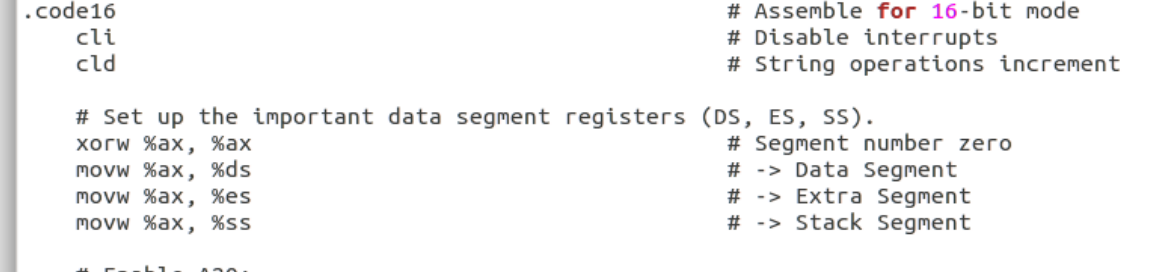
输入x/16i 0x7c00便可以从0x7C00开始跟踪。

对比

Bootasm.asm



Bootasm.s



4．自己找一个bootloader或内核中的代码位置，设置断点并进行测试

