

实验课题：Linux 启动初始化过程探析

计科 1903 班 19281171 王雨潇

1. 实验目的

探索、分析和理解操作系统引导及自启动初始化的基本流程、设计机理。

2. 实验环境

编译环境：GNU 编译器 GCC-1.4.0，汇编器 as86，链接器 ld86

运行环境：Bochs-2.2.1 虚拟机，Intel x86 计算机模拟器

测试环境：Bochs-2.2.1 虚拟机和附带的 bochsdbg 调试器

宿主机操作系统为 Windows 10 家庭版，CPU 为 Intel Core i7-10710 (x86-64 架构)

该实验报告使用 Plantuml 绘制全部流程图，源码截图使用 VS Code

3. 实验内容

3.1. Linux-0.11 内核源码下载和研读

下载并解压一份 devel 开发者版本（包含 include）的 Linux-0.11 源码



3.1.1. Makefile 文件解读

Makefile 文件就是整个工程的编译、连接规则，对于 Linux 源码根目录下的 Makefile 解释如下：

开头主要是编译器参数，这里是选择 8006 汇编的编译器和连接器，使用 GNU 的编译器和连接器，后面主要是各类编译参数，比如 CFLAGS =-Wall 就是指打印所有警告信息，此处不多赘述；

```
#
# if you want the ram-disk device, define this to be the
# size in blocks.
#
RAMDISK = #-DRAMDISK=512

AS86      =as86 -0 -a
LD86      =ld86 -0

AS  =gas
LD  =gld
LDLFLAGS =-s -x -M
CC  =gcc $(RAMDISK)
CFLAGS =-Wall -O -fstrength-reduce -fomit-frame-pointer \
-fcombine-regs -mstring-insns
CPP  =cpp -nostdinc -Iinclude
```

推测这一部分用意是给这些经常被“打包”一起编译或一起引用的文件起了别名。

- (1) ROOT_DEV 是创建内核映像文件时所使用的默认根文件系统所在的设备；
- (2) ARCHIVES 是 kernel, mm, fs 三个目录生成的*.o 目标文件路径；
- (3) DRIVERS 块和字符设备库文件，

查阅它包括的第一个目录 kernel\blk_drv\Makefile, 得知*.a 是一类用 gar 生成的归档文件；

- (4) MATH 是数学运算库归档文件，LIBS 是库文件归档文件；

```
#
# ROOT_DEV specifies the default root-device when making the image.
# This can be either FLOPPY, /dev/xxxx or empty, in which case the
# default of /dev/hd6 is used by 'build'.
#
ROOT_DEV=/dev/hd6
```

```
ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o
DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a
MATH     =kernel/math/math.a
LIBS     =lib/lib.a
```

三组后缀规则，顺序是[源文件格式后缀][模板文件格式后缀]，功能应该是指对不同文件编译和连接过程的 gcc 和 as 参数；

```
.c.s:
    $(CC) $(CFLAGS) \
    -nostdinc -Iinclude -S -o $*.s $<
.s.o:
    $(AS) -c -o $*.o $<
.c.o:
    $(CC) $(CFLAGS) \
    -nostdinc -Iinclude -c -o $*.o $<
```

下列就是生成规则，根据 Makefile 的编写规则，其中每一条的格式都是

[目标：生成该目标依赖的文件 要执行的指令]

```
all:    Image

Image: boot/bootsect boot/setup tools/system tools/build
    tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) > Image
    sync

disk: Image
    dd bs=8192 if=Image of=/dev/PS0

tools/build: tools/build.c
    $(CC) $(CFLAGS) \
    -o tools/build tools/build.c

boot/head.o: boot/head.s

tools/system:  boot/head.o init/main.o \
    $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)
    $(LD) $(LDFLAGS) boot/head.o init/main.o \
    $(ARCHIVES) \
    $(DRIVERS) \
    $(MATH) \
    $(LIBS) \
    -o tools/system > System.map
```

```
kernel/math/math.a:
    (cd kernel/math; make)

kernel/blk_drv/blk_drv.a:
    (cd kernel/blk_drv; make)

kernel/chr_drv/chr_drv.a:
    (cd kernel/chr_drv; make)

kernel/kernel.o:
    (cd kernel; make)

mm/mm.o:
    (cd mm; make)

fs/fs.o:
    (cd fs; make)

lib/lib.a:
    (cd lib; make)

boot/setup: boot/setup.s
    $(AS86) -o boot/setup.o boot/setup.s
    $(LD86) -s -o boot/setup boot/setup.o

boot/bootsect: boot/bootsect.s
    $(AS86) -o boot/bootsect.o boot/bootsect.s
    $(LD86) -s -o boot/bootsect boot/bootsect.o

tmp.s: boot/bootsect.s tools/system
    (echo -n "SYSSIZE = (" ; ls -l tools/system | grep system \
    | cut -c25-31 | tr '\012' ' '; echo "+ 15 ) / 16") > tmp.s
    cat boot/bootsect.s >> tmp.s

clean:
    rm -f Image System.map tmp_make core boot/bootsect boot/setup
    rm -f init/*.o tools/system tools/build boot/*.o
    (cd mm; make clean)
    (cd fs; make clean)
    (cd kernel; make clean)
    (cd lib; make clean)

backup: clean
    (cd .. ; tar cf - linux | compress - > backup.Z)
```

```
sync
```

```
dep:
```

```
sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
(for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
cp tmp_make Makefile
(cd fs; make dep)
(cd kernel; make dep)
(cd mm; make dep)
```

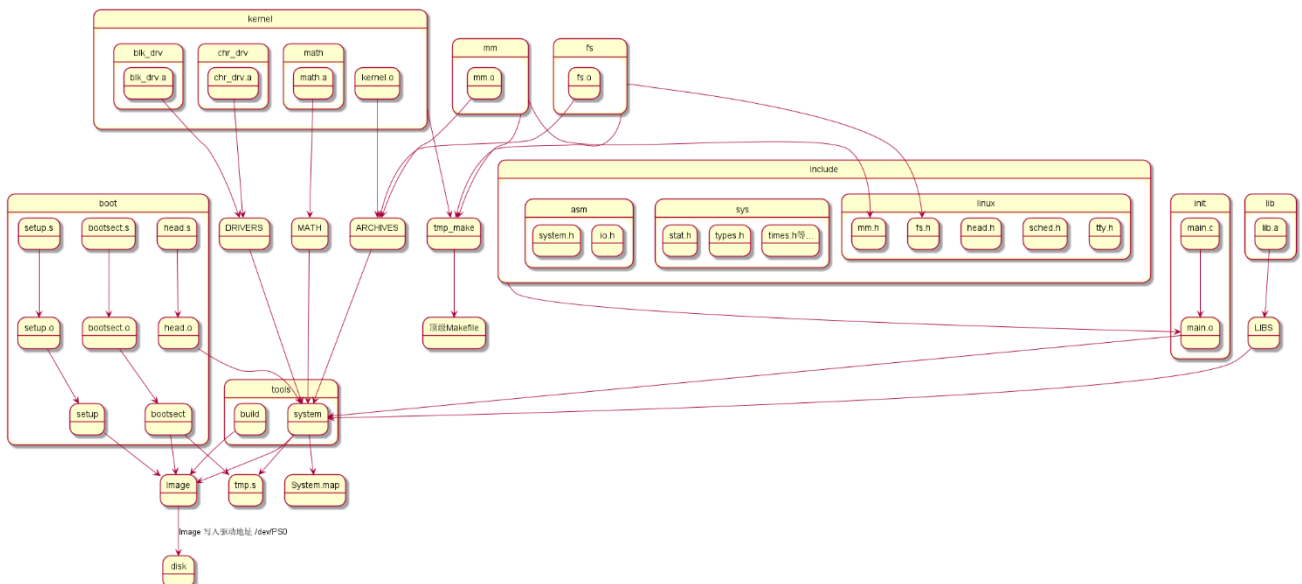
这里就是调用上述规则的命令

```
### Dependencies:
```

```
init/main.o : init/main.c include/unistd.h include/sys/stat.h \
include/sys/types.h include/sys/times.h include/sys/utsname.h \
include/utime.h include/time.h include/linux/tty.h include/termios.h \
include/linux/sched.h include/linux/head.h include/linux/fs.h \
include/linux/mm.h include/signal.h include/asm/system.h include/asm/io.h \
include/stddef.h include/stdarg.h include/fcntl.h
```

3.1.2. Makefile 的生成规则总结

如下图所示，依据顶级 Makefile 文件，可以画出编译过程文件/文件夹之间的依赖关系。

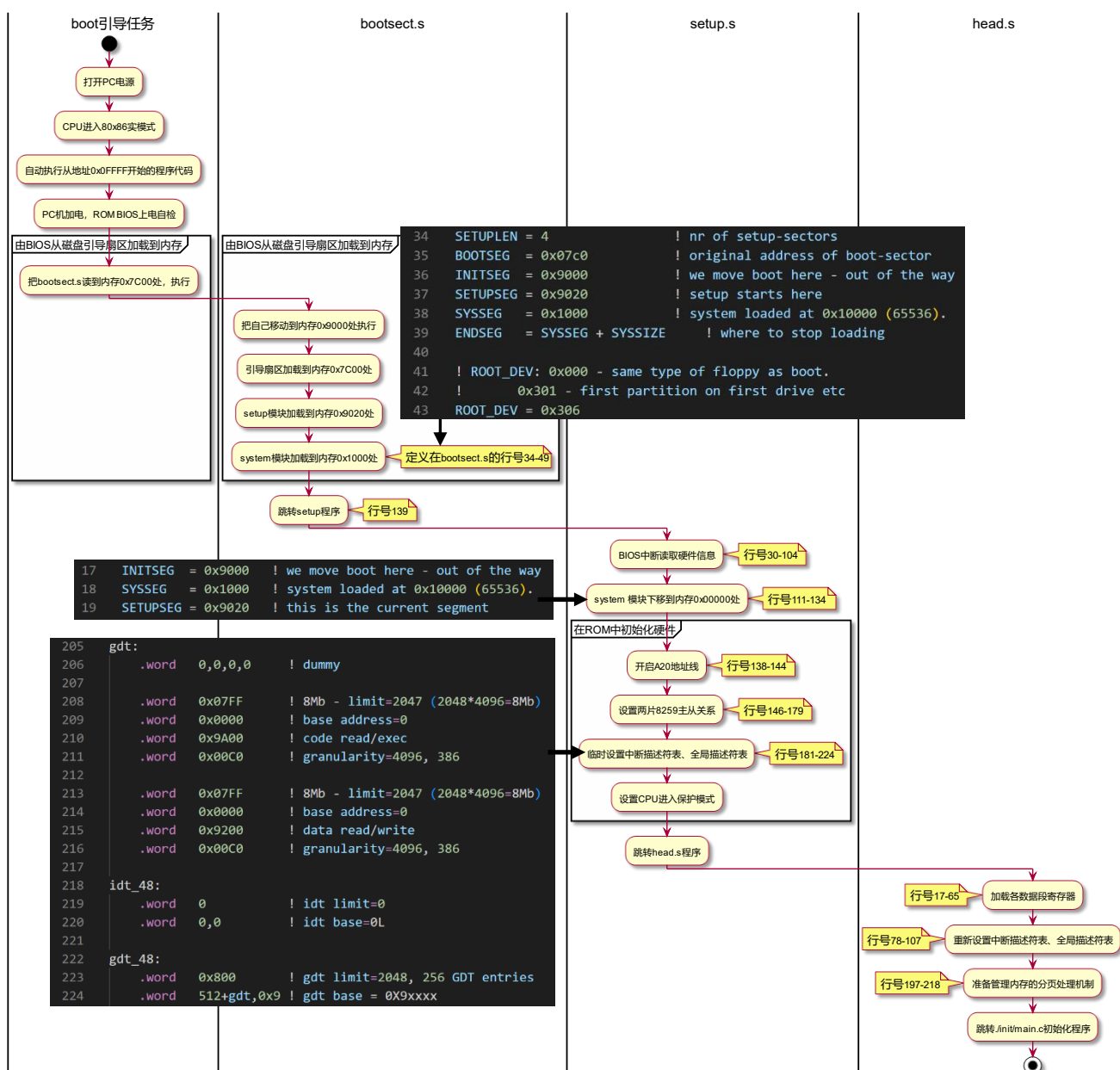


3.2. 操作系统引导和自启动初始化的流程分析

系统引导指的是将操作系统内核装入内存并启动系统的过程，系统的初始化过程是配置系统的服务和准备软件执行的环境的过程。以下分析采用泳道图表示，涉及的重要数据结构定义由箭头标识。

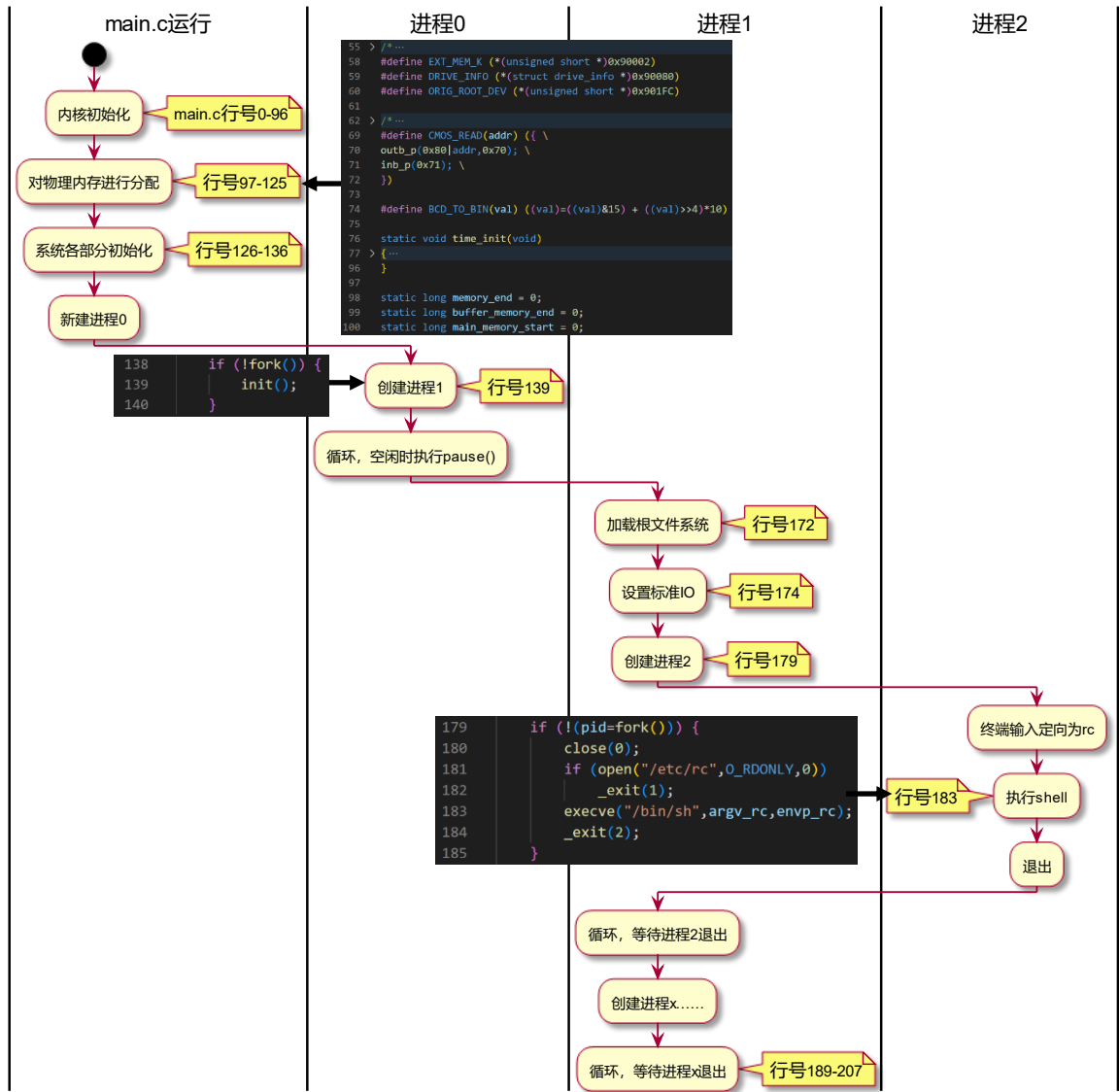
3.2.1. boot 引导程序

引导启动程序即.\boot\文件夹下的三个*.s 文件，它们在引导过程的功能分工如以下图所示；



3.2.2. init 初始化程序

初始化程序即.\init\main.c, 在引导过程的功能如以下图所示



3.3. 虚拟机平台测试验证

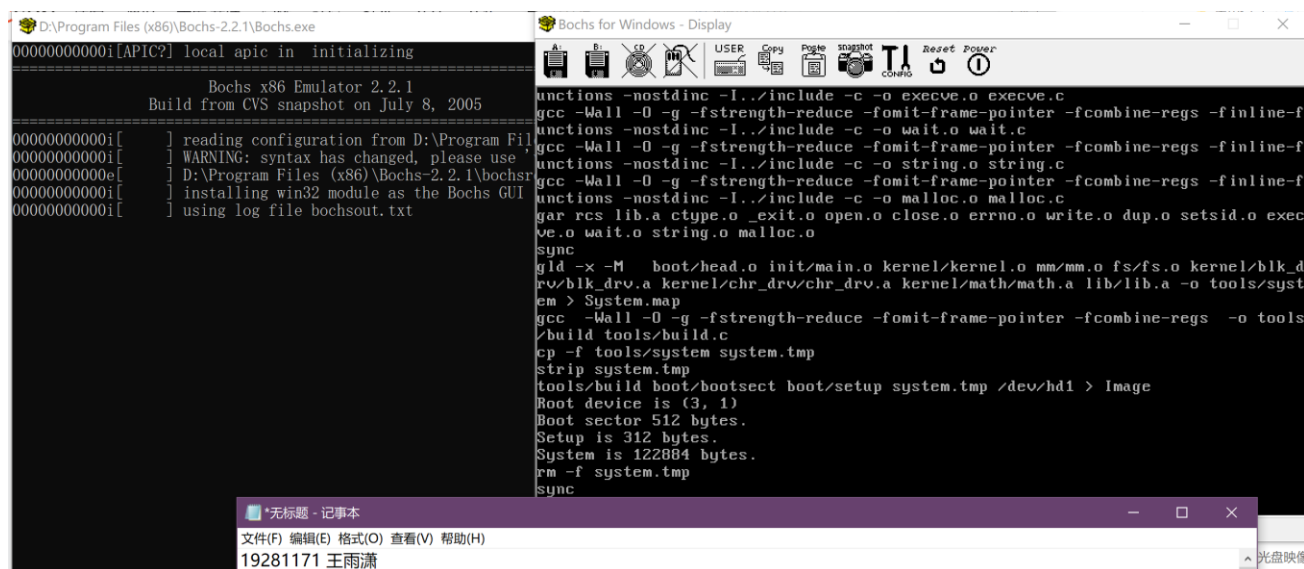
在 oldlinux.org/Linux.old/bochs/ 下载安装 linux-0.11-devel-060625.zip, 其中包括了 bochs 模拟器的安装文件和 Linux-0.11 的内核映像;

使用了 Bochs-2.2.1 的配置文件 bochsrc-hd.bxrc, 查阅资料得知*.bxrc 是 bochs 的配置文件格式, 而该配置表明从硬盘启动系统, 是从 Bochs-2.2.1 目录下内核映像文件 bootimage-0.11-hd 中读取加载

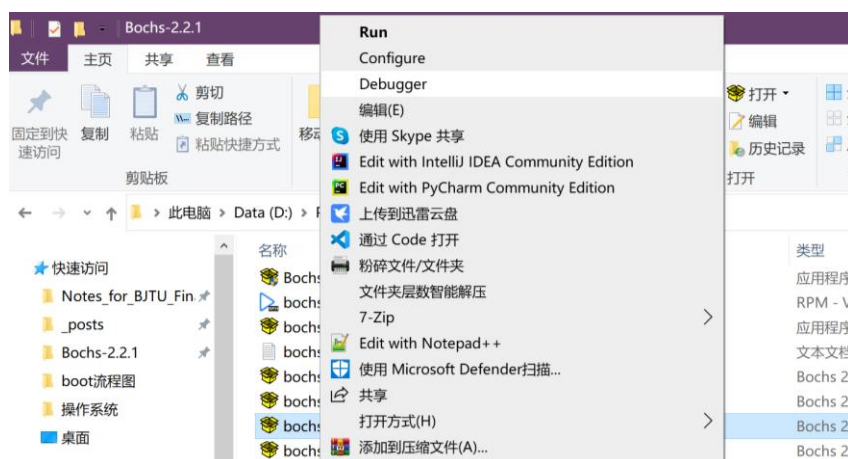
信息，使得内核初始化运行时会自动从虚拟 C 盘的第 1 个分区中加载根文件系统。

3.3.1. 编译 Linux-0.11 内核源码

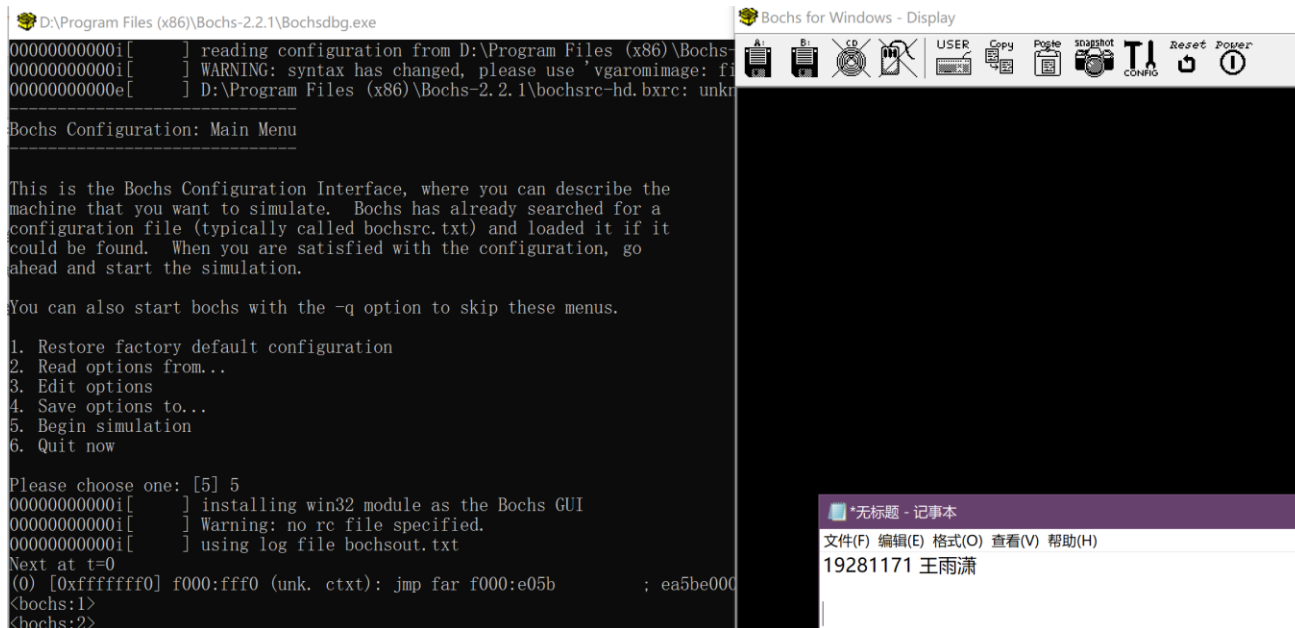
linux-0.11-devel-060625.zip 这个版本的 bochs 比课程组提供的教程新一些，节省了自己改 Makefile 文件的时间。因此，直接 cd 到路径下 make，即可成功编译内核部分。



3.3.2. 开始调试 bochsrc-hd.bxrc

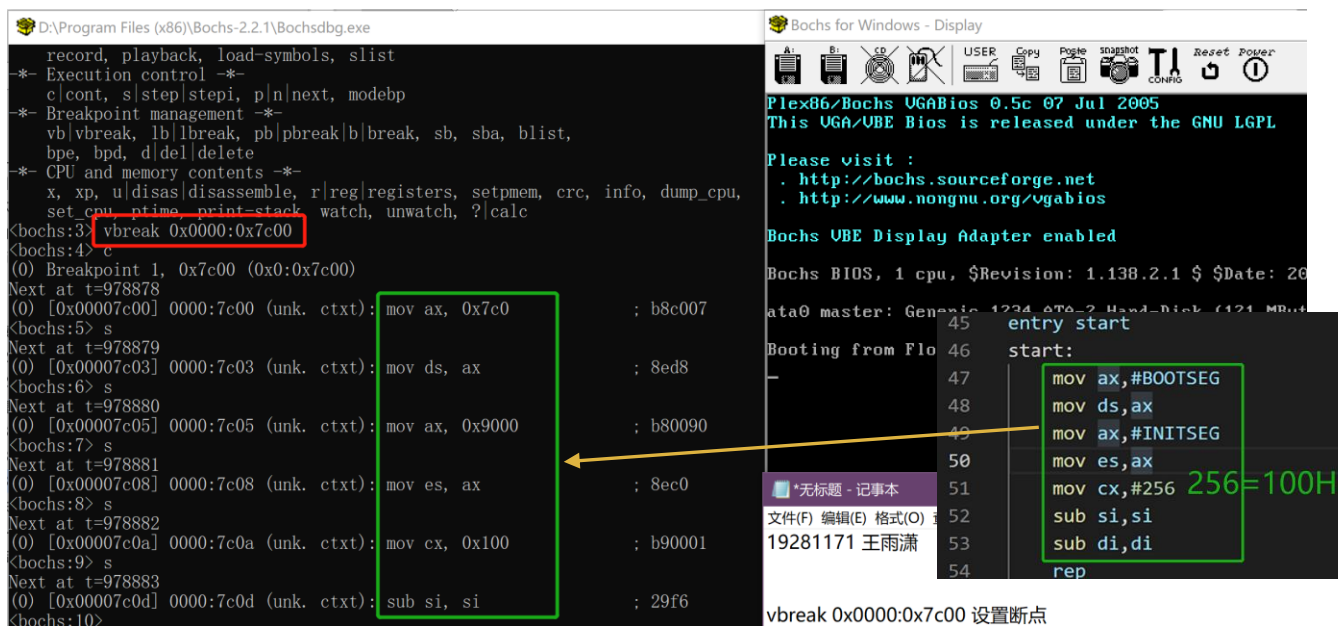


最开始我试图在 Windows Terminal 调试，遇到了# In bx_win32_gui_c::exit(void)!报错，但换成直接右键 bochsrc-hd.bxrc 就可以正常运行，结果如下图；

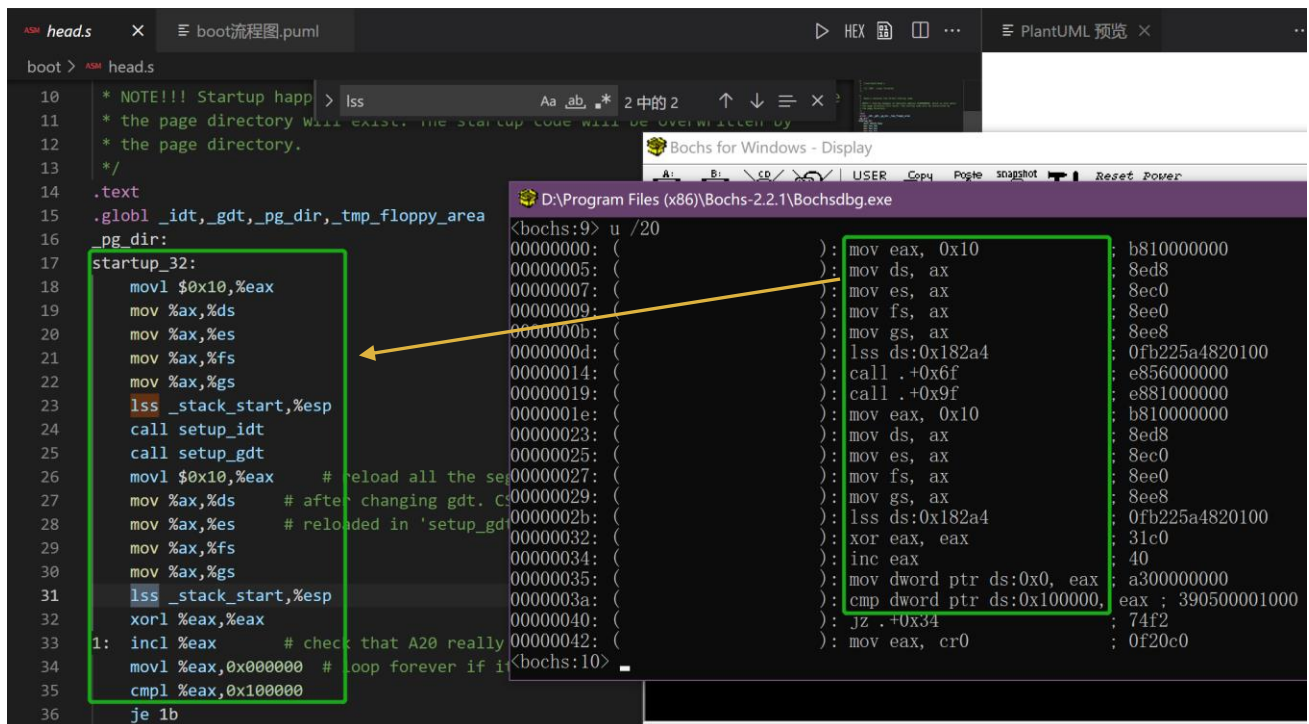


3.3.3. 断点调试

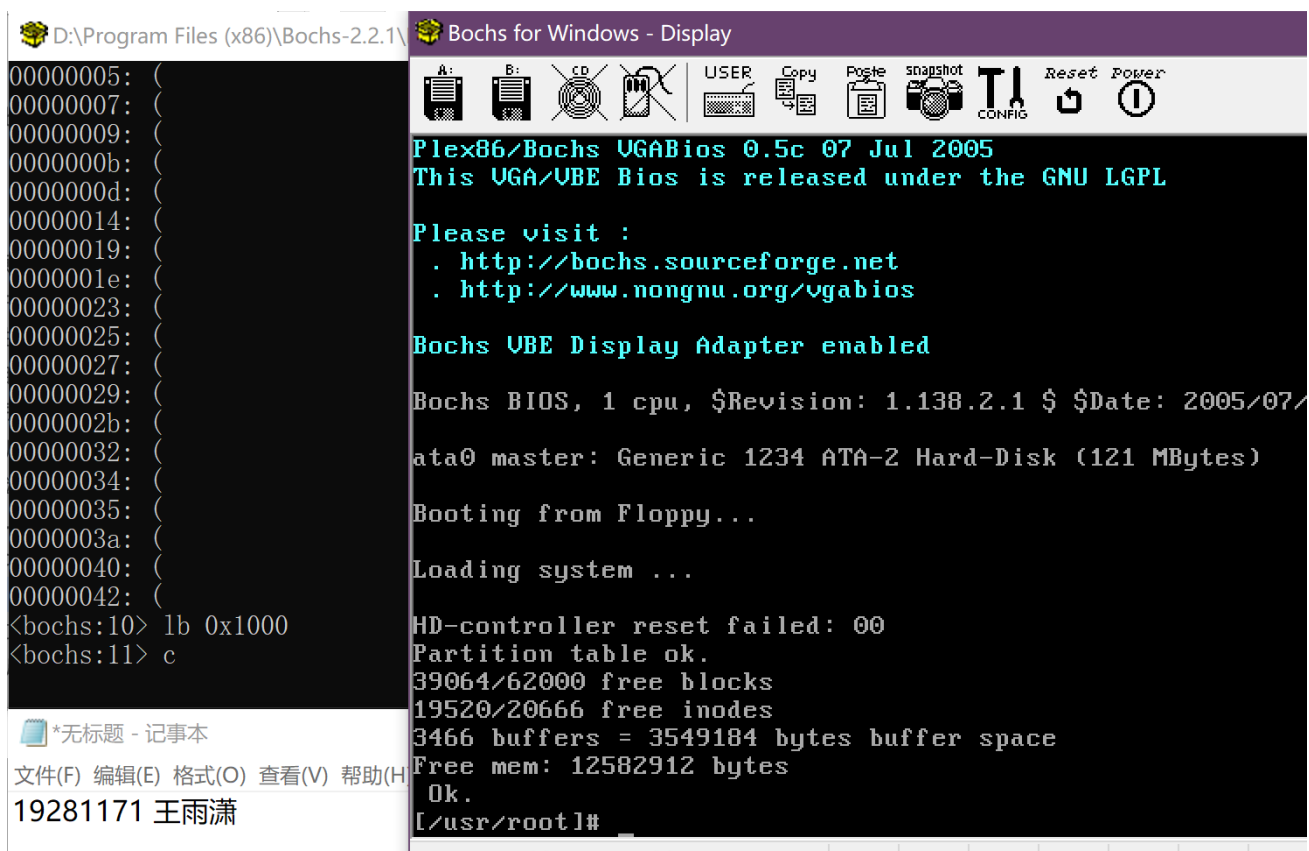
首先输入 `vbreak 0x0000:0x7c00` 指令，即在 0x7c00 处设置断点，然后输入 `c` 让程序连续执行，在断点位置后连续几次输入 `s`，观测到此时的汇编代码正好是 `\boot\bootsect.s` 的 `entry start` 部分读取 `bootsect.s` 到内存的汇编代码；



然后输入 `lb 0x0000` 指令，意为设置线性地址断点，观察此处反汇编该地址之后的 20 行代码，发现对应了 `\boot\head.s` 中设置中断描述符表的部分；



再次连续执行，从终端屏幕发现已经完成内核的编译运行。

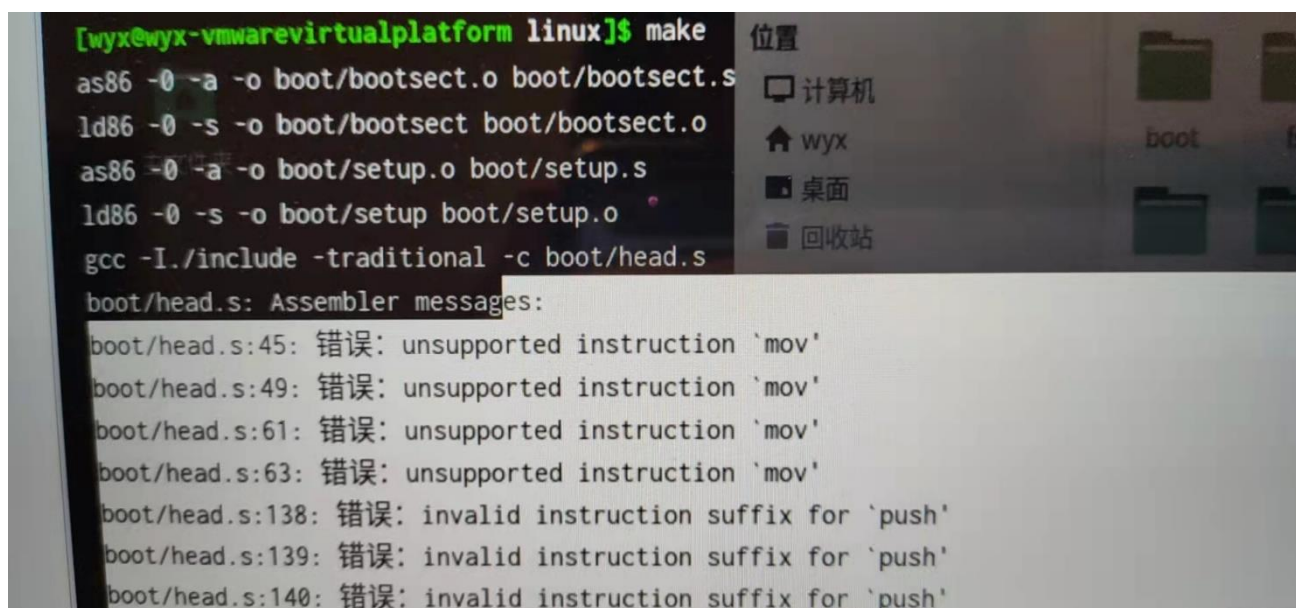


以这些“捕捉”为标志，证明断点调试观测到的运行过程与此前的流程分析一致。

3.4. 失败记录和经验

3.4.1. manjaro 虚拟机为什么不适合这个实验

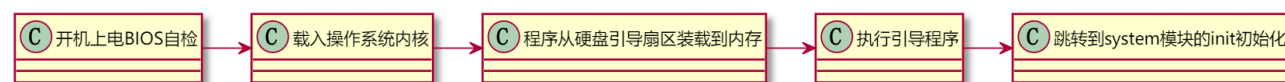
最开始我尝试过用 manjaro (Arch-Linux 的发行版之一) 虚拟机搭配旧版本 gcc 和 gdb 编译调试内核, 但 Linux-0.11 中的 80x86 汇编代码根本不能在 x86-64 位 CPU 上运行。我经过了两天的查资料和查错, 尝试包括装 32 位的早期 Linux, 设置虚拟化 CPU 为 32 位处理器, 但都没能解决这个问题。



```
[wyx@wyx-vmwarevirtualplatform linux]$ make
as86 -O -a -o boot/bootsect.o boot/bootsect.s
ld86 -O -s -o boot/bootsect boot/bootsect.o
as86 -O -a -o boot/setup.o boot/setup.s
ld86 -O -s -o boot/setup boot/setup.o
gcc -I./include -traditional -c boot/head.s
boot/head.s: Assembler messages:
boot/head.s:45: 错误: unsupported instruction `mov'
boot/head.s:49: 错误: unsupported instruction `mov'
boot/head.s:61: 错误: unsupported instruction `mov'
boot/head.s:63: 错误: unsupported instruction `mov'
boot/head.s:138: 错误: invalid instruction suffix for `push'
boot/head.s:139: 错误: invalid instruction suffix for `push'
boot/head.s:140: 错误: invalid instruction suffix for `push'
```

而 bochs 模拟器直接虚拟化全部硬件, 正如它官方手册的一段话: “也许你有一个重要的应用程序运行在一个旧的操作系统上, 它只在旧硬件上运行良好”, 或许更新版本的 Linux 源码才适合用虚拟机执行。

4. 实验结论和心得体会



通过本次实验, 我了解了 Linux-0.11 内核引导和自启动的过程, 验证了自己阅读 Makefile 文件和 boot 引导, .\init\main.c 文件得出的初始化过程分析的正确性。由此我们可以更深入的理解操作系统, 为在以后的实验中能够自己在操作系统内核上编写用户态的程序打下基础。