

Lab3 操作系统的引导

一、实验目的

1. 熟悉实验环境
2. 建立对操作系统引导过程的深入认识
3. 掌握操作系统的基本开发过程
4. 能对操作系统代码进行简单的控制，揭开操作系统的神秘面纱

二、实验内容

1. 阅读《Linux内核完全注释》的第 6 章，对计算机和 Linux 0.11 的引导过程进行初步的了解
2. 按照下面的要求改写 Linux 0.11 的引导程序 bootsect.s
3. 有兴趣同学可以做做进入保护模式前的设置程序 setup.s

2.1. 改写 bootsect.s 主要完成如下功能

bootsect.s 能在屏幕上打印一段提示信息

```
xxx is booting...
```

其中 xxx 是你给自己的操作系统起的名字，也可以显示一个特色 logo，以表示自己操作系统的与众不同。

2.2. 改写 setup.s 主要完成如下功能

1. bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。而 setup.s 向屏幕输出一行

```
Now we are in SETUP
```

2. setup.s 能获取至少一个基本的硬件参数（如内存参数、显卡参数、硬盘参数等），将其存放在内存的特定地址，并输出到屏幕上。
3. setup.s 不再加载Linux内核，保持上述信息显示在屏幕上即可。

三、实验过程

本实验所用环境为操作系统课题组提供的 vmware -- Linux 0.11 实验环境测试版。

3.0. Linux-0.11 的仿真运行

1. 考虑每次实验需要重置 Linux-0.11 目录，因此需要运行重置功能。

```
cd ~/oslab
./init
```

2. 进入仿真目录，将当前目录切换到 oslab 下，用 pwd，命令确认，用 ls -l 列目录内容。

```
cd ~/oslab
ls -l
```

3. 编译内核

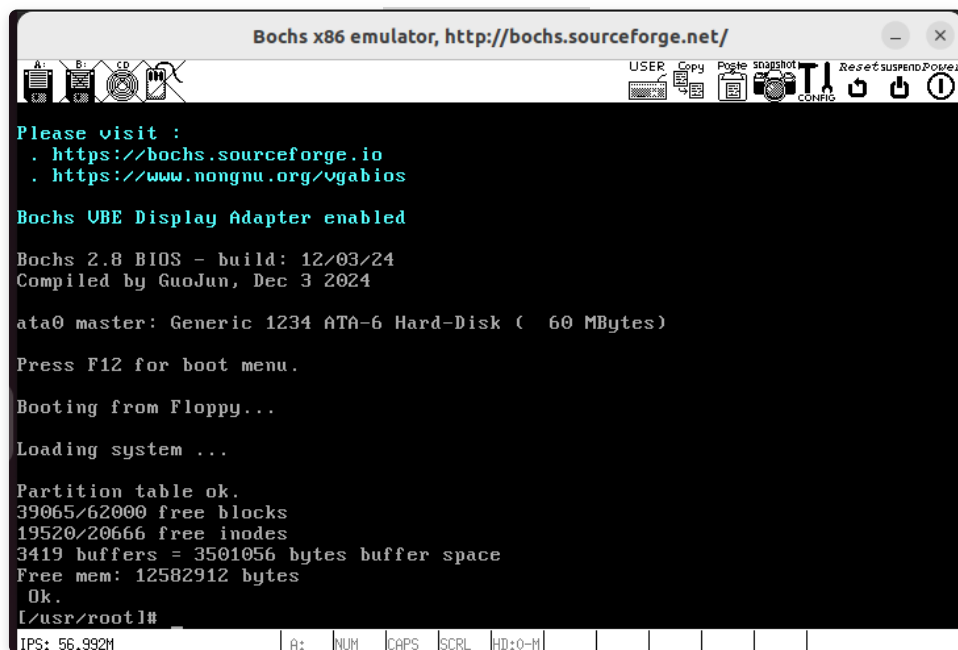
```
cd linux-0.11
# 因为 all 是最常用的参数，所以可以省略，只用 make ，效果一样
make all
# 在多处处理器的系统上，可以用 -j 参数进行并行编译，加快速度。例如双 CPU 的系统可以：
# make -j 2
# 删除上一次编译生成的所有中间文件和目标文件，确保是在全新的状态下编译整个工程
# make clean
```

4. 程序运行，在 oslab 目录下运行一下命令，如果出现 Bochs 的窗口，继续输入命令 `c`，里面显示 Linux 的引导过程，最后停止在 `[/usr/root/#]`，表示运行成功。

```
./dbg-bochs
```

```
guojun@guojunos:~/oslab$ ./dbg-bochs
=====
Bochs x86 Emulator 2.8
Built from GitHub snapshot on March 10, 2024
Timestamp: Sun Mar 10 08:00:00 CET 2024
Compiled by GuoJun, Dec 3 2024
=====
00000000000i[      ] BXSHARE is set to '/usr/local/bochs28/share/bo
chs'
00000000000i[      ] reading configuration from ./bochs/linux-0.11.
bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Bochs internal debugger, type 'help' for help or 'c' to continue
Switching to CPU0
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b
; ea5be000f0
<bochs:1> c
```

图3-0-1 运行程序



```
Bochs x86 emulator, http://bochs.sourceforge.net/
Please visit :
. https://bochs.sourceforge.io
. https://www.nongnu.org/vgabios
Bochs VBE Display Adapter enabled
Bochs 2.8 BIOS - build: 12/03/24
Compiled by GuoJun, Dec 3 2024
ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)
Press F12 for boot menu.
Booting from Floppy...
Loading system ...
Partition table ok.
39065/62000 free blocks
19520/20666 free inodes
3419 buffers = 3501056 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]#
IPS: 56,992M | A: | NUM | CAPS | SCRL | HD:0-M | | | | |
```

图3-0-2 bochs显示引导日志

3.1. bootsect.s 的改写

要做到在屏幕上打印一段提示信息

```
2022110829 YangMingda's os is booting...
```

3.1.1. bootsect.s 核心代码分析

查看 bootsect.s 代码，将其完成屏幕显示的关键代码提炼出来，并进行每行代码的详细分析。

```
# 首先读入光标位置
mov $0x03, %ah      # read cursor pos
xor %bh, %bh        # 首先读光标位置，返回光标位置值在dx中
int $0x10           # dh - 行 (0--24): dl - 列 (0--79)，供显示串用
# 显示字符串"Loading system ..."
mov $24, %cx        # 要显示的字符串长度 (共显示24个字符)
mov $0x0007, %bx     # page 0, attribute 7 (normal)
mov $msg1, %bp       # es:bp指向要显示的字符串
mov $0x1301, %ax     # write string, move cursor
int $0x10           # 写字符串并移动光标到串结尾处

ljmp    $SETUPSEG, $0 # 到此本程序就结束了

msg1:                # 调用BIOS中断显示信息
.byte 13,10         # 换行、回车的ASCII码
.ascii "Loading system ..." # 打印的字符串
.byte 13,10,13,10   # 两对换行、回车
# 表示下面的语句从地址508 (0x1FC) 开始
.org 508
# 下面0xAA55是启动盘具有有效引导扇区的标志。供BIOS中的程序加载引导扇区时识别使用。
# 他比具有位于引导扇区的最后两个字节中
boot_flag:
.word 0xAA55        # 必须有它，才能引导
```

3.1.2. bootsect.s 代码改写部分分析

分析上述代码。其中

```
mov $24, %cx

.byte 13,10
.ascii "Loading system ..."
.byte 13,10,13,10
```

表示显示字符串的长度，和要打印的字符串。所以，如果要实现打印属于自己独特的提示信息，需要改写这两行代码为

```
mov $46, %cx

.byte 13,10
.ascii "2022110829 YangMingda's os is booting..."
.byte 13,10,13,10
```

其中字符长度为46，包括提示信息的长度为40，和三对换行+回车长度为6。

3.1.3. bootsect.s 改动后核心代码展示

bootsect.s 改动后的核心代码如下所示。

```
# 首先读入光标位置
mov $0x03, %ah      # read cursor pos
xor %bh, %bh        # 首先读光标位置，返回光标位置值在dx中
int $0x10           # dh - 行 (0--24): dl - 列 (0--79)，供显示串用
# 显示字符串“Loading system ...”
mov $46, %cx        # 要显示的字符串长度（共显示46个字符）
mov $0x0007, %bx     # page 0, attribute 7 (normal)
mov $msg1, %bp       # es:bp指向要显示的字符串
mov $0x1301, %ax     # write string, move cursor
int $0x10           # 写字符串并移动光标到串结尾处

ljmp    $SETUPSEG, $0 # 到此本程序就结束了

msg1:                # 调用BIOS中断显示信息
.byte 13,10          # 换行、回车的ASCII码
.ascii "2022110829 YangMingda's os is booting..." # 打印的字符串
.byte 13,10,13,10    # 两对换行、回车
# 表示下面的语句从地址508（0x1FC）开始
.org 508
# 下面0xAA55是启动盘具有有效引导扇区的标志。供BIOS中的程序加载引导扇区时识别使用。
# 他比具有位于引导扇区的最后两个字节中
boot_flag:
.word 0xAA55         # 必须有它，才能引导
```

3.1.4. 程序仿真运行

1. 在~/oslab/linux-0.11/boot 目录下对 bootsect.s 进行改写

```
100      mov     $46, %cx
101      mov     $0x0007, %bx      # page 0, attribute 7 (normal)
102      #lea     msg1, %bp
103      mov     $msg1, %bp
104      mov     $0x1301, %ax      # write string, move cursor
105      int     $0x10
```

图3-1-1 字符串字节

```
247 msg1:
248      .byte 13,10
249      .ascii "2022110829 YangMingda's os is booting..."
250      .byte 13,10,13,10
```

图3-1-2 字符串提示信息

2. 进入Linux-0.11 目录，进行编译

```
cd ~/oslab
cd linux-0.11
make Image
```

```

guojun@guojunos:~/oslab/linux-0.11$ make Image
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/boot'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/boot'
1+0 records in
1+0 records out
512 bytes copied, 0.000720335 s, 711 kB/s
0+1 records in
0+1 records out
311 bytes copied, 0.000430545 s, 722 kB/s
309+1 records in
309+1 records out
158465 bytes (158 kB, 155 KiB) copied, 0.0010858 s, 146 MB/s
2+0 records in
2+0 records out
2 bytes copied, 0.000760516 s, 2.6 kB/s

```

图3-1-3 编译

3. 程序运行，可以看出在 Bochs 界面中，能成功打印出目标的字符串。

```
../dbg-bochs
```

```

guojun@guojunos:~/oslab/linux-0.11$ ../dbg-bochs
=====
Bochs x86 Emulator 2.8
Built from GitHub snapshot on March 10, 2024
Timestamp: Sun Mar 10 08:00:00 CET 2024
Compiled by GuoJun, Dec 3 2024
=====
=====
00000000000i[      ] BXSHARE is set to '/usr/local/bochs28/share/bo
chs'
00000000000i[      ] reading configuration from ../bochs/linux-0.11
.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ../bochsout.txt
Bochs internal debugger, type 'help' for help or 'c' to continue
Switching to CPU0
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b
; ea5be000f0
<bochs:1> c

```

图3-1-4 运行程序

```

Bochs x86 emulator, http://bochs.sourceforge.net/
Please visit :
. https://bochs.sourceforge.io
. https://www.nongnu.org/vgabios
Bochs UBE Display Adapter enabled
Bochs 2.8 BIOS - build: 12/03/24
Compiled by GuoJun, Dec 3 2024
ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)
Press F12 for boot menu.
Booting from Floppy...
2022110829 YangMingda's os is booting...
Partition table ok.
39065/62000 free blocks
19520/20666 free inodes
3419 buffers = 3501056 bytes buffer space
Free mem: 12582912 bytes
Ok.
[usr/root]#
IPS: 8.238M  A:  NUM  CAPS  SCRL  HD:0-M

```

图3-1-5 系统启动情况

3.2. bootsect.s 载入 setup.s

bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。而 setup.s 向屏幕输出一行

```
Now we are in SETUP
```

3.2.1. 总体分析

bootsect.s 代码是磁盘引导块程序，驻留在磁盘的第一个扇区（引导扇区，0磁道（柱面），0磁头，第1个扇区）。在磁盘上，引导块、setup 模块和 system 模块的扇区位置和大小示意图如下图所示。

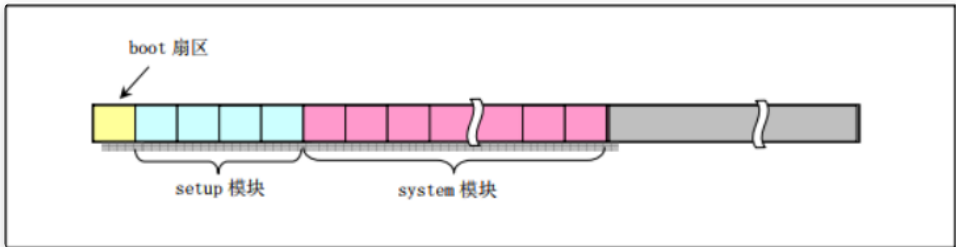


图3-2-1 Linux 0.11内核在1.44MB磁盘上的分布情况

1.44MB磁盘共有2880个扇区，其中引导程序代码占用第1个扇区，setup 模块占用随后的4个扇区。bootsect 程序的主要作用是首先从磁盘第2个扇区开始的4个扇区的 setup 模块（由 setup.s 编译而成）加载到内存紧接着 bootsect 后面位置处（0x90200），然后利用 BIOS 中断 0x13 取磁盘参数表中当前启动引导盘参数，接着在屏幕上显示“Loading system...”字符串。再把磁盘上 setup 模块后面的 system 模块加载到内存 0x10000 开始的地方。随后确定根文件系统的设备号，最后长跳转到 setup 程序的开始处（0x90200）执行 setup 程序。

对机器开始加电，开始顺序执行。

- 1. 系统上电后，会将 bootsec.s 加载入 0x7C00 位置
- 2. bootsec.s 将自己移动到 0x90000
- 3. 加载 setup.s 入内存，加载 system 入 0x10000 处，然后跳转至 setup.s 中执行
- 4. setup.s 加载一些硬件参数
- 5. 将系统移动至 0x0000 处，跳转至 system 中的 head.s 中执行

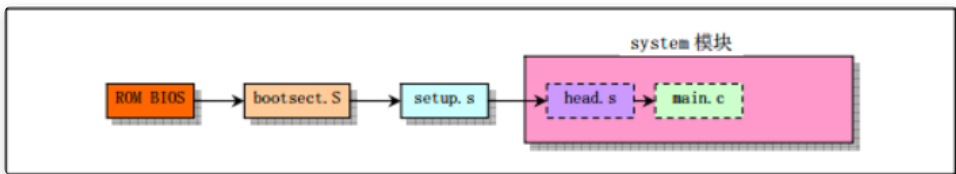


图3-2-2 从系统加电起所执行程序的顺序

Linux 系统时各个程序或模块在内存的动态位置如下图所示。其中每一竖条框代表某一时刻内存中各程序的映像位置图。

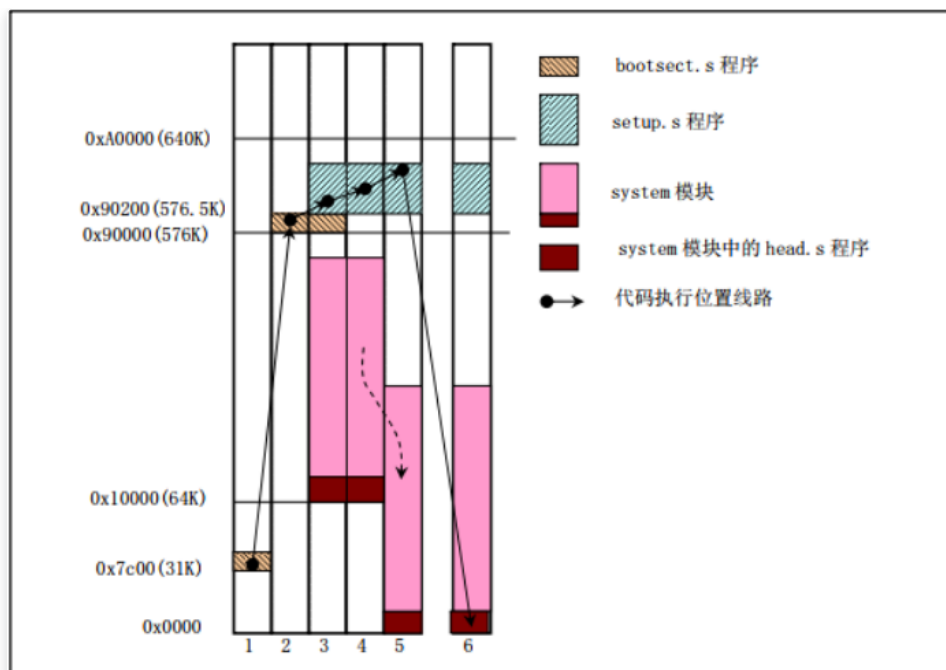


图3-2-3 启动引导时内核在内存中的位置和移动后的位置情况

3.2.2. bootsect.s 核心代码分析

bootsect.s 中载入 setup.s 的关键代码及每行代码具体分析如下所示。

```
load_setup:
    mov $0x0000, %dx      # 设置驱动器和磁头(drive 0, head 0): 软盘0磁头
    mov $0x0002, %cx      # 设置扇区号和磁道(sector 2, track 0):0磁头、0磁道、2
                           # 扇区
    mov $0x0200, %bx      # 设置读入的内存地址: BOOTSEG+address = 512, 偏移512字
                           # 节
    .equ    AX, 0x0200+SETUPLEN # 设置读入的扇区个数(service 2, nr of sectors),
    mov     $AX, %ax       # SETUPLEN是读入的扇区个数, Linux 0.11设置的是4,
                           # 我们不需要那么多, 我们设置为2
    int $0x13              # 应用0x13号BIOS中断读入2个setup.s扇区
    jnc ok_load_setup      # 读入成功, 跳转到ok_load_setup: ok - continue
    mov $0x0000, %dx      # 软驱、软盘有问题才会执行到这里。我们的镜像文件比它们可靠
                           # 多了
    mov $0x0000, %ax      # 否则复位软驱 reset the diskette
    int $0x13
    jmp load_setup         # 重新循环, 再次尝试读取

ok_load_setup:            # 第二章的打印字符串任务代码(此处省略)
```

3.2.3. setup.s 核心代码分析

在 bootsect.s 中实现跳转到 setup.s 中, 后开始实现 setup.s 打印提示词的核心代码。核心代码及每行代码的具体分析如下所示。

```
ljmp $SETUPSEG, $_start # 将程序跳转到SETUPSEG段的_start标签处
_start:                  # 程序的入口点, 执行从这里开始的代码
init_ds_es:
    mov %cs, %ax         # 将当前代码段寄存器cs的值移动到寄存器ax中
    mov %ax, %ds         # 将ax寄存器的值复制到数据段寄存器ds中
    mov %ax, %es         # 将ax寄存器的值复制到额外段寄存器es中
```

```

print_cur:
    mov     $0x03, %ah          # read cursor pos
    xor     %bh, %bh           # 首先读光标位置，返回光标位置值在dx中
    int     $0x10              # dh - 行 (0--24): dl - 列 (0--79)，供显示串用
    # 显示字符串"Now we are in SETUP..."
    mov     $28, %cx           # 要显示的字符串长度 (共显示28个字符)
    mov     $0x0007, %bx       # page 0, attribute 7 (normal)
    mov     $msg1, %bp         # es:bp指向要显示的字符串
    mov     $0x1301, %ax       # write string, move cursor
    int     $0x10              # 写字符串并移动光标到串结尾处

msg1:
    # 调用BIOS中断显示信息
    .ascii "Now we are in SETUP..." # 打印的字符串
    .byte 13,10,13,10             # 两对换行、回车

```

3.2.4. build.sh 修改

在 make 编译的过程中，因为 make 根据 Makefile 的指引执行了 tools/build.s，它是为生成整个内核的镜像文件而设计的，没考虑我们只需要 bootsect.s 和 setup.s 的情况。它在向我们要“系统”的核心代码。为完成实验，接下来给它打个小补丁。

改造 build.sh 的思路就是当 system 是 none 的时候，只写 bootsect 和 setup，忽略所有与 system 有关的工作，或者在该写 system 的位置都写上 0。具体代码如下所示。

```

#!/bin/bash
# build.sh -- a shell version of build.c for the new bootsect.s & setup.s
# author: falcon <wuzhangjin@gmail.com>
# update: 2008-10-10
bootsect=$1
setup=$2
system=$3
IMAGE=$4
root_dev=$5
# Set the biggest sys_size
# Changes from 0x20000 to 0x30000 by tiger.cn to avoid oversized code.
SYS_SIZE=$((0x3000*16))
# set the default "device" file for root image file
if [ -z "$root_dev" ]; then
    DEFAULT_MAJOR_ROOT=3
    DEFAULT_MINOR_ROOT=1
else
    DEFAULT_MAJOR_ROOT=${root_dev:0:2}
    DEFAULT_MINOR_ROOT=${root_dev:2:3}
fi
# Write bootsect (512 bytes, one sector) to stdout
[ ! -f "$bootsect" ] && echo "there is no bootsect binary file there" && exit -1
dd if=$bootsect bs=512 count=1 of=$IMAGE 2>&1 >/dev/null
# Write setup(4 * 512bytes, four sectors) to stdout
[ ! -f "$setup" ] && echo "there is no setup binary file there" && exit -1
dd if=$setup seek=1 bs=512 count=4 of=$IMAGE 2>&1 >/dev/null
# If system is 'none', skip system binary writing
if [ "$system" != "none" ]; then
# Write system(< SYS_SIZE) to stdout
[ ! -f "$system" ] && echo "there is no system binary file there" && exit -1

```



```

system_size=`wc -c $system |cut -d" " -f1`
[ $system_size -gt $SYS_SIZE ] && echo "the system binary is too big" && exit
-1
dd if=$system seek=5 bs=512 count=$((2888-1-4)) of=$IMAGE 2>&1 >/dev/null
else
dd if=/dev/zero seek=5 bs=512 count=$((2888-1-4)) of=$IMAGE 2>&1 >/dev/null
fi
# Set "device" for the root image file
echo -ne "\x$DEFAULT_MINOR_ROOT\x$DEFAULT_MAJOR_ROOT" | dd ibs=1 obs=1 count=2
seek=508 of=$IMAGE conv=notrunc 2>&1 >/dev/null

```

3.2.5. bootsect.s 完整代码

在本章任务下，bootsect.s 的完整代码如下所示。

```

.code16

.equ SYSSIZE, 0x3000

.global _start, begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

.equ SETUPLEN, 4
.equ BOOTSEG, 0x07c0
.equ INITSEG, 0x9000
.equ SETUPSEG, 0x9020
.equ SYSSEG, 0x1000
.equ ENDSEG, SYSSEG + SYSSIZE

.equ ROOT_DEV, 0x301
ljmp     $BOOTSEG, $_start
_start:
    mov $BOOTSEG, %ax
    mov %ax, %ds
    mov $INITSEG, %ax
    mov %ax, %es
    mov $256, %cx
    sub %si, %si
    sub %di, %di
    rep
    movsw
    ljmp     $INITSEG, $go
go: mov %cs, %ax
    mov %ax, %ds
    mov %ax, %es

load_setup:
    mov $0x0000, %dx
    mov $0x0002, %cx

```

```

mov $0x0200, %bx
.equ    AX, 0x0200+SETUPLEN
mov     $AX, %ax
int $0x13
jnc ok_load_setup
mov $0x0000, %dx
mov $0x0000, %ax
int $0x13
jmp load_setup

ok_load_setup:
mov $0x03, %ah
xor %bh, %bh
int $0x10
mov $46, %cx
mov $0x0007, %bx
mov $msg1, %bp
mov $0x1301, %ax
int $0x10

ljmp    $SETUPSEG, $0

sectors:
.word 0

msg1:
.byte 13,10
.ascii "2022110829 YangMingda's os is booting..."
.byte 13,10,13,10

.org 508
root_dev:
.word ROOT_DEV
boot_flag:
.word 0xAA55

.text
endtext:
.data
enddata:
.bss
endbss:

```

3.2.6. setup.s 完整代码

在本章任务下，`setup.s` 的完整代码如下所示。

```

.code16

.equ INITSEG, 0x9000
.equ SYSSEG, 0x1000
.equ SETUPSEG, 0x9020

.global _start, begtext, begdata, begbss, endtext, enddata, endbss
.text

```

```

begtext:
.data
begdata:
.bss
begbss:
.text

    ljmp $SETUPSEG, $_start
_start:

init_ds_es:
    mov    %cs, %ax
    mov    %ax, %ds
    mov    %ax, %es

print_cur:
    mov    $0x03, %ah
    xor    %bh, %bh
    int    $0x10
    mov    $26, %cx
    mov    $0x0007, %bx
    mov    $msg1, %bp
    mov    $0x1301, %ax
    int    $0x10

msg1:
    .ascii "Now we are in SETUP..."
    .byte 13,10,13,10

.text
endtext:
.data
enddata:
.bss
endbss:

```

3.2.7. 程序仿真运行

1. 在 `~/oslab/linux-0.11/boot` 目录下对 `bootsect.s` 和 `setup.s` 进行改写。
2. 在 `~/oslab/linux-0.11/tools` 目录下对 `build.sh` 进行改写。
3. 进入 `Linux-0.11` 目录，进行编译。

```

cd ~/oslab
cd linux-0.11
make Image

```

```

guojun@guojunos:~/oslab/linux-0.11$ make Image
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/boot'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/boot'
1+0 records in
1+0 records out
512 bytes copied, 0.000495921 s, 1.0 MB/s
0+1 records in
0+1 records out
57 bytes copied, 0.000365677 s, 156 kB/s
309+1 records in
309+1 records out
158465 bytes (158 kB, 155 KiB) copied, 0.00145037 s, 109 MB/s
2+0 records in
2+0 records out
2 bytes copied, 0.0003974 s, 5.0 kB/s

```

图3-2-4 编译

4. 程序运行，可以看出在 Bochs 界面中，能成功打印出目标的字符串。

./dbg-bochs

```

guojun@guojunos:~/oslab/linux-0.11$ ../dbg-bochs
=====
Bochs x86 Emulator 2.8
Built from GitHub snapshot on March 10, 2024
Timestamp: Sun Mar 10 08:00:00 CET 2024
Compiled by GuoJun, Dec 3 2024
=====
000000000000i[      ] BXSHARE is set to '/usr/local/bochs28/share/bo
chs'
000000000000i[      ] reading configuration from ../bochs/linux-0.11
.bxrc
000000000000i[      ] installing x module as the Bochs GUI
000000000000i[      ] using log file ../bochsout.txt
Bochs internal debugger, type 'help' for help or 'c' to continue
Switching to CPU0
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b
; ea5be000f0
<bochs:1> c

```

图3-2-5 运行程序

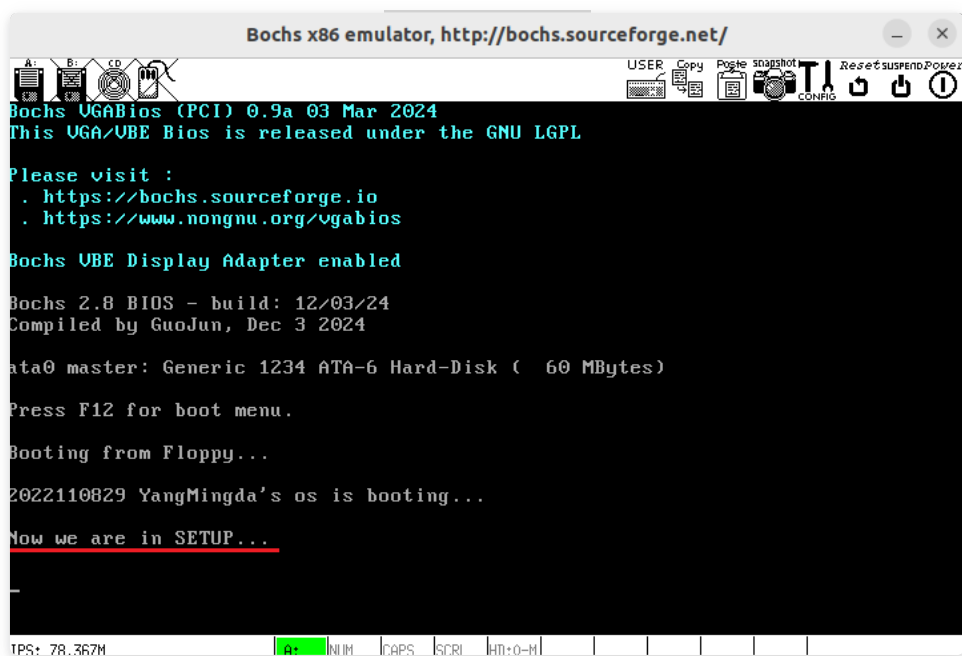


图3-2-6 系统启动情况

3.3. setup.s 获取基本的硬件参数

setup.s 将获得硬件参数放在内存的 0x90000 处。原版 setup.s 中已经完成了光标位置、内存大小、显存大小、显卡参数、第一和第二硬盘参数的保存。

用 ah=#0x03 调用 0x10 中断可以读出光标的位置，用 ah=#0x88 调用 0x15 中断可以读出内存的大小。有些硬件参数的获取要稍微复杂一些，如磁盘参数表。在 PC 机中 BIOS 设定的中断向量表中 int 0x41 的中断向量位置(4*0x41 = 0x0000:0x0104)存放的并不是中断程序的地址，而是第一个硬盘的基本参数表。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量位置处。每个硬盘参数表有 16 个字节大小。下表给出了硬盘基本参数表的内容：

位移	大小	说明
0x00	字	柱面数
0x02	字节	磁头数
0x03	字	开始减小写电流的柱面
0x05	字	开始写前预补偿柱面号
0x07	字节	最大ECC猝发长度
0x08	字节	控制字节
0x09	字节	标准超时值
0x0A	字节	格式化超时值
0x0B	字节	检测驱动器超时值
0x0C	字	磁头着陆（停止）柱面号
0x0E	字节	每磁道扇区数
0x0F	字节	保留

所以获得磁盘参数的方法就是复制数据。

下面是将硬件参数取出来放在内存 0x90000 的关键代码。

```
mov    $INITSEG, %ax
mov    %ax, %ds      # 设置ds=0x9000
mov    $0x03, %ah    # 读入光标位置
xor    %bh, %bh
int    $0x10         # 调用0x10中断
mov    %dx, [0]      # 将光标位置写入0x90000.
# 读入内存大小位置
mov    $0x88, %ah
int    $0x15
mov    %ax, [2]
# 从0x41处拷贝16个字节（磁盘参数表）
mov    $0x0000, %ax
mov    %ax, %ds
lds    %ds:4*0x41, %si
mov    $INITSEG, %ax
```

```

mov    %ax, %es
mov    $0x0004, %di
mov    $0x10, %cx
rep                                # 重复16次
movsb

```

现在已经将硬件参数取出来放在了 0x90000 处，接下来的工作是将这些参数显示在屏幕上。这些参数都是一些无符号整数，所以需要做的的主要工作是用汇编程序在屏幕上将这些整数显示出来。

以十六进制方式显示比较简单。这是因为十六进制与二进制有很好的对应关系（每 4 位二进制数和 1 位十六进制数存在一一对应关系），显示时只需将原二进制数每 4 位划成一组，按组求对应的 ASCII 码送显示器即可。ASCII 码与十六进制数字的对应关系为：0x30~0x39 对应数字 0~9，0x41~0x46 对应数字 a~f。从数字 9 到 a，其 ASCII 码间隔了 7h，这一点在转换时要特别注意。为使一个十六进制数能按高位到低位依次显示，实际编程中，需对 bx 中的数每次循环左移一组（4 位二进制），然后屏蔽掉当前高 12 位，对当前余下的 4 位（即 1 位十六进制数）求其 ASCII 码，要判断它是 0~9 还是 a~f，是前者则加 0x30 得对应的 ASCII 码，后者则要加 0x37 才行，最后送显示器输出。以上步骤重复 4 次，就可以完成 bx 中数以 4 位十六进制数的形式显示出来。

下面是完成显示 16 进制数的汇编语言程序的关键代码，其中用到的 BIOS 中断为 INT 0x10，功能号 0x0E（显示一个字符），即 AH=0x0E，AL=要显示字符的 ASCII 码。

```

# 以16进制方式打印栈顶的16位数
print_hex:
    mov    $4, %cx                # 4个十六进制数字
    mov    %ax, %dx               # 将ax所指的放入dx中
print_digit:
    rol    $4, %dx                # 循环以使低4比特用上！！取dx的高4比特移到低4比特处
    mov    $0xe0f, %ax            # ah = 请求的功能值, al = 半字节(4个比特)掩码
    and    %dl, %al               # 取dl的低4比特值
    add    $0x30, %al             # 给al数字加上十六进制0x30
    cmp    $0x3a, %al             # 比较al是否小于0x3a（即‘9’）
    jnl    outp                   # 如果是一个不大于10的数字，跳转到输出
    add    $0x07, %al             # 如果是a~f，要多加7
outp:
    int    $0x10                  # 调用BIOS中断输出字符
    loop   print_digit            # 循环，直到打印完4个数字
    ret

```

这里用到了一个 loop 指令，每次执行 loop 指令，cx 减 1，然后判断 cx 是否等于 0。如果不为 0 则转移到 loop 指令后的标号处，实现循环；如果为 0 顺序执行。另外还有一个非常相似的指令：rep 指令，每次执行 rep 指令，cx 减 1，然后判断 cx 是否等于 0，如果不为 0 则继续执行 rep 指令后的串操作指令，直到 cx 为 0，实现重复。

```

# 打印回车换行
print_nl:
    mov    $0xe0d, %ax            # CR
    int    $0x10                  # 调用BIOS中断输出CR
    mov    $0xa, %al              # LF
    int    $0x10                  # 调用BIOS中断输出LF
    ret

```

只要在适当的位置调用 print_hex 和 print_nl（注意，一定要设置好栈，才能进行函数调用）就能将获得硬件参数打印到屏幕上，完成此次实验的任务。

3.3.1. 显示光标位置

在显示光标位置的任务中，获取光标位置并存入结果的代码如下所示。

```
mov    $INITSEG, %ax
mov    %ax, %ds      # set ds=0x9000
mov    $0x03, %ah     # get cursor pos
xor    %bh, %bh
int    $0x10         # interrupt
mov    %dx, [0]      # write cursor pos to 0x90000.
```

打印光标位置的代码如下所示。

```
print_cursor_pos:
    mov    $0x03, %ah      # read cursor pos
    xor    %bh, %bh
    int    $0x10

    mov    $12, %cx
    mov    $0x0007, %bx     # page 0, attribute 7 (normal)
    mov    $cursor, %bp
    mov    $0x1301, %ax     # write string, move cursor
    int    $0x10

    mov    [0], %ax
    call   print_hex
    call   print_nl
```

3.3.2. 显示内存大小

在显示内存大小的任务中，获取内存大小并存入结果的代码如下所示。

```
mov    $0x88, %ah
int    $0x15
mov    %ax, [2]
```

打印内存大小的代码如下所示。

```
print_memory_size:
    mov    $0x03, %ah      # read cursor pos
    xor    %bh, %bh
    int    $0x10

    mov    $13, %cx
    mov    $0x0007, %bx     # page 0, attribute 7 (normal)
    mov    $memory_size, %bp
    mov    $0x1301, %ax     # write string, move cursor
    int    $0x10

    mov    [2], %ax
    call   print_hex
```

3.3.3. 显示硬盘参数

在显示硬盘参数的任务中，获取硬盘参数并存入结果的代码如下所示。

```
mov    $0x0000, %ax
mov    %ax, %ds
lds    %ds:4*0x41, %si
mov    $INITSEG, %ax
mov    %ax, %es
mov    $0x0004, %di
mov    $0x10, %cx
rep
movsb
```

打印硬盘参数的代码如下所示。

```
print_hd_info:
    # cylinders
    mov    $0x03, %ah          # read cursor pos
    xor    %bh, %bh
    int    $0x10

    mov    $15, %cx
    mov    $0x0007, %bx        # page 0, attribute 7 (normal)
    mov    $hdinfo, %bp
    mov    $0x1301, %ax        # write string, move cursor
    int    $0x10

    mov    [4], %ax
    call   print_hex
    call   print_n1

    # head
    mov    $0x03, %ah          # read cursor pos
    xor    %bh, %bh
    int    $0x10

    mov    $9, %cx
    mov    $0x0007, %bx        # page 0, attribute 7 (normal)
    mov    $head, %bp
    mov    $0x1301, %ax        # write string, move cursor
    int    $0x10

    xor    %ax, %ax
    mov    [4+2], %al
    call   print_hex
    call   print_n1

    # sect
    mov    $0x03, %ah          # read cursor pos
    xor    %bh, %bh
    int    $0x10

    mov    $9, %cx
    mov    $0x0007, %bx        # page 0, attribute 7 (normal)
```



```

    mov    $sect, %bp
    mov    $0x1301, %ax    # write string, move cursor
    int    $0x10

    xor    %ax, %ax
    mov    [0x4+0x0E], %a1
    call   print_hex
    call   print_n1

```

3.3.4. bootsect.s 完整代码

在本章任务下，bootsect.s 的完整代码如下所示。

```

.code16

.equ SYSSIZE, 0x3000

.global _start, begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

.equ SETUPLEN, 4
.equ BOOTSEG, 0x07c0
.equ INITSEG, 0x9000
.equ SETUPSEG, 0x9020
.equ SYSSEG, 0x1000
.equ ENDSEG, SYSSEG + SYSSIZE

.equ ROOT_DEV, 0x301
ljmp     $BOOTSEG, $_start
_start:
    mov $BOOTSEG, %ax
    mov %ax, %ds
    mov $INITSEG, %ax
    mov %ax, %es
    mov $256, %cx
    sub %si, %si
    sub %di, %di
    rep
    movsw
    ljmp     $INITSEG, $go
go: mov %cs, %ax
    mov %ax, %ds
    mov %ax, %es

load_setup:
    mov $0x0000, %dx
    mov $0x0002, %cx
    mov $0x0200, %bx
    .equ     AX, 0x0200+SETUPLEN

```

```

    mov     $AX, %ax
    int $0x13
    jnc ok_load_setup
    mov $0x0000, %dx
    mov $0x0000, %ax
    int $0x13
    jmp load_setup

ok_load_setup:
    mov $0x03, %ah
    xor %bh, %bh
    int $0x10

    mov $46, %cx
    mov $0x0007, %bx
    mov     $msg1, %bp
    mov $0x1301, %ax
    int $0x10

    ljmp    $SETUPSEG, $0

sectors:
    .word 0

msg1:
    .byte 13,10
    .ascii "2022110829 YangMingda's os is booting..."
    .byte 13,10,13,10

    .org 508
root_dev:
    .word ROOT_DEV
boot_flag:
    .word 0xAA55

    .text
endtext:
    .data
enddata:
    .bss
endbss:

```

3.3.5. setup.s 完整代码

在本章任务下，`setup.s` 的完整代码如下所示。

```

.code16

.global _start, begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:

```

```

.text

.equ SETUPLEN, 4
.equ BOOTSEG, 0x07c0
.equ INITSEG, 0x9000
.equ SETUPSEG, 0x9020
.equ SYSSEG, 0x1000
.equ ENDSEG, SYSSEG + SYSSIZE

.equ ROOT_DEV, 0x306
ljmp $SETUPSEG, $_start
_start:

init_ds_es:
    mov    %cs, %ax
    mov    %ax, %ds
    mov    %ax, %es

print_cur:
    mov    $0x03, %ah
    xor    %bh, %bh
    int    $0x10
    mov    $26, %cx
    mov    $0x0007, %bx
    mov    $msg1, %bp
    mov    $0x1301, %ax
    int    $0x10
    # get cursor pos
    mov    $INITSEG, %ax
    mov    %ax, %ds
    mov    $0x03, %ah
    xor    %bh, %bh
    int    $0x10
    mov    %dx, [0]
    # get memory size
    mov    $0x88, %ah
    int    $0x15
    mov    %ax, [2]
    # copy disk parameters from 0x41
    mov    $0x0000, %ax
    mov    %ax, %ds
    lds    %ds:4*0x41, %si
    mov    $INITSEG, %ax
    mov    %ax, %es
    mov    $0x0004, %di
    mov    $0x10, %cx
    rep
    movsb

reset_ds_es:
    mov    $INITSEG, %ax
    mov    %ax, %ds
    mov    $SETUPSEG, %ax
    mov    %ax, %es

print_cursor_pos:

```

```
mov    $0x03, %ah
xor     %bh, %bh
int     $0x10
mov     $12, %cx
mov     $0x0007, %bx
mov     $cursor, %bp
mov     $0x1301, %ax
int     $0x10
mov     [0], %ax
call    print_hex
call    print_nl
```

print_memory_size:

```
mov     $0x03, %ah
xor     %bh, %bh
int     $0x10
mov     $13, %cx
mov     $0x0007, %bx
mov     $memory_size, %bp
mov     $0x1301, %ax
int     $0x10
mov     [2], %ax
call    print_hex
```

print_hd_info:

```
# Cylinders
mov     $0x03, %ah
xor     %bh, %bh
int     $0x10
mov     $26, %cx
mov     $0x0007, %bx
mov     $hdinfo, %bp
mov     $0x1301, %ax
int     $0x10
mov     [4], %ax
call    print_hex
call    print_nl

# head
mov     $0x03, %ah
xor     %bh, %bh
int     $0x10
mov     $9, %cx
mov     $0x0007, %bx
mov     $head, %bp
mov     $0x1301, %ax
int     $0x10
xor     %ax, %ax
mov     [4+2], %ax
call    print_hex
call    print_nl

# sect
mov     $0x03, %ah
xor     %bh, %bh
int     $0x10
mov     $9, %cx
mov     $0x0007, %bx
```

```

        mov     $sect, %bp
        mov     $0x1301, %ax
        int     $0x10
        xor     %ax, %ax
        mov     [0x4+0x0E], %al
        call    print_hex
        call    print_n1

# print hex
print_hex:
        mov     $4, %cx
        mov     %ax, %dx

print_digit:
        rol     $4, %dx
        mov     $0xe0f, %ax
        and     %dl, %al
        add     $0x30, %al
        cmp     $0x3a, %al
        jl      outp

        add     $0x07, %al
outp:
        int     $0x10
        loop    print_digit
        ret

# 打印换行
print_n1:
        mov     $0xe0d, %ax
        int     $0x10
        mov     $0xa, %al
        int     $0x10
        ret

msg1:
        .ascii "Now we are in SETUP..."
        .byte 13,10,13,10
cursor:
        .ascii "Cursor POS: "
memory_size:
        .ascii "Memory SIZE: "
hdinfo:
        .ascii "KB"
        .byte 13,10,13,10
        .ascii "HD Info"
        .byte 13,10
        .ascii "Cylinders: "
head:
        .ascii "Headers: "
sect:
        .ascii "Sectors: "

        .org 508
root_dev:
        .word ROOT_DEV
boot_flag:

```

```
.word 0xAA55
.text
endtext:
.data
enddata:
.bss
endbss:
```

3.3.6. 程序仿真运行

1. 在 `~/oslab/linux-0.11/boot` 目录下对 `setup.s` 进行改写。
2. 进入 `Linux-0.11` 目录，进行编译。

```
cd ~/oslab
cd linux-0.11
make Image
```

```
guojun@guojunos:~/oslab/linux-0.11$ make Image
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/boot'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/boot'
1+0 records in
1+0 records out
512 bytes copied, 0.00061572 s, 832 kB/s
1+0 records in
1+0 records out
512 bytes copied, 0.000179932 s, 2.8 MB/s
309+1 records in
309+1 records out
158465 bytes (158 kB, 155 KiB) copied, 0.00208243 s, 76.1 MB/s
2+0 records in
2+0 records out
2 bytes copied, 0.000231951 s, 8.6 kB/s
```

图3-3-1 编译

3. 程序运行，可以看出在 `Bochs` 界面中，能成功打印出目标的字符串。

```
../dbg-bochs
```

```
guojun@guojunos:~/oslab/linux-0.11$ ../dbg-bochs
=====
=====
Bochs x86 Emulator 2.8
Built from GitHub snapshot on March 10, 2024
Timestamp: Sun Mar 10 08:00:00 CET 2024
Compiled by GuoJun, Dec 3 2024
=====
=====
00000000000i[      ] BXSHARE is set to '/usr/local/bochs28/share/bo
chs'
00000000000i[      ] reading configuration from ../bochs/linux-0.11
.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ../bochsout.txt
Bochs internal debugger, type 'help' for help or 'c' to continue
Switching to CPU0
Next at t=0
(0) [0x0000ffffffff] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b
; ea5be000f0
<bochs:1> c
```

图3-3-2 运行程序

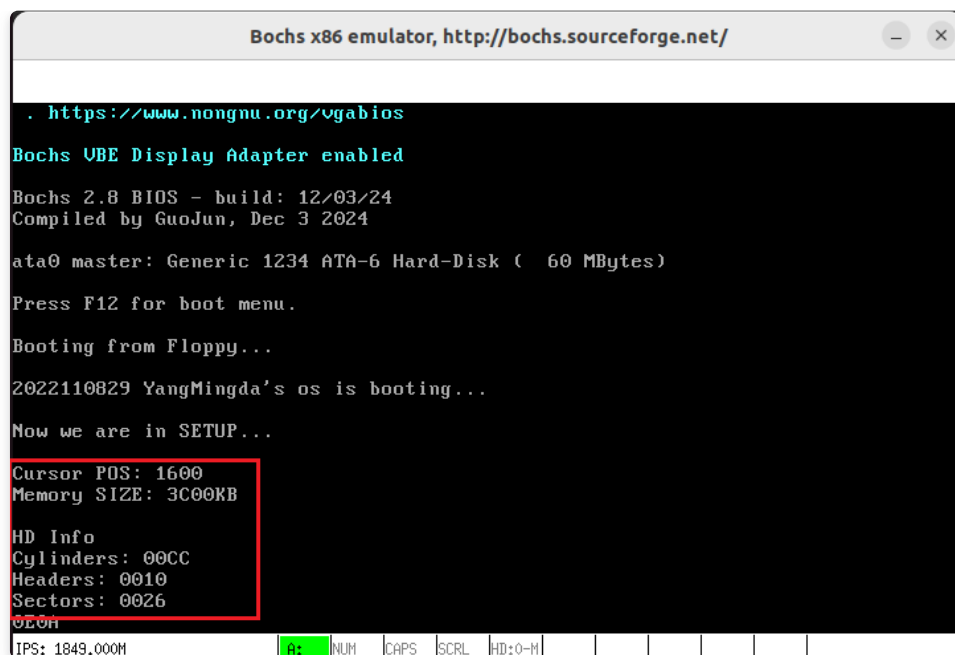


图3-3-3 系统启动情况

3.4. setup.s 不再加载Linux内核

3.4.1. setup.s 代码修改

要实现 setup.s 不再加载Linux内核，保持上述信息显示在屏幕上，需要在上一节基础上添加下面的两行代码，让程序进入无限的死循环。

```
dead_loop:
    jmp    dead_loop
```

3.4.2. bootsect.s 完整代码

在本章任务下，bootsect.s 的完整代码如下所示。

```
.code16

.equ SYSSIZE, 0x3000

.global _start, begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

.equ SETUPLEN, 4
.equ BOOTSEG, 0x07c0
.equ INITSEG, 0x9000
.equ SETUPSEG, 0x9020
.equ SYSSEG, 0x1000
.equ ENDSEG, SYSSEG + SYSSIZE

.equ ROOT_DEV, 0x301
```

```

    ljmp    $BOOTSEG, $_start
_start:
    mov $BOOTSEG, %ax
    mov %ax, %ds
    mov $INITSEG, %ax
    mov %ax, %es
    mov $256, %cx
    sub %si, %si
    sub %di, %di
    rep
    movsw
    ljmp    $INITSEG, $go
go: mov %cs, %ax
    mov %ax, %ds
    mov %ax, %es

load_setup:
    mov $0x0000, %dx
    mov $0x0002, %cx
    mov $0x0200, %bx
    .equ    AX, 0x0200+SETUPLEN
    mov     $AX, %ax
    int $0x13
    jnc ok_load_setup
    mov $0x0000, %dx
    mov $0x0000, %ax
    int $0x13
    jmp load_setup

ok_load_setup:
    mov $0x03, %ah
    xor %bh, %bh
    int $0x10

    mov $46, %cx
    mov $0x0007, %bx
    mov     $msg1, %bp
    mov $0x1301, %ax
    int $0x10

    ljmp    $SETUPSEG, $0

sectors:
    .word 0

msg1:
    .byte 13,10
    .ascii "2022110829 YangMingda's os is booting..."
    .byte 13,10,13,10

    .org 508
root_dev:
    .word ROOT_DEV
boot_flag:
    .word 0xAA55

```



```

.text
endtext:
.data
enddata:
.bss
endbss:

```

3.4.3. setup.s 完整代码

在本章任务下，`setup.s` 的完整代码如下所示。

```

.code16

.global _start, begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

.equ SETUPLEN, 4
.equ BOOTSEG, 0x07c0
.equ INITSEG, 0x9000
.equ SETUPSEG, 0x9020
.equ SYSSEG, 0x1000
.equ ENDSEG, SYSSEG + SYSSIZE

.equ ROOT_DEV, 0x306
ljmp $SETUPSEG, $_start
_start:

init_ds_es:
    mov    %cs, %ax
    mov    %ax, %ds
    mov    %ax, %es

print_cur:
    mov    $0x03, %ah
    xor    %bh, %bh
    int    $0x10
    mov    $26, %cx
    mov    $0x0007, %bx
    mov    $msg1, %bp
    mov    $0x1301, %ax
    int    $0x10
    # get cursor pos
    mov    $INITSEG, %ax
    mov    %ax, %ds
    mov    $0x03, %ah
    xor    %bh, %bh
    int    $0x10
    mov    %dx, [0]
    # get memory size

```

```

mov    $0x88, %ah
int    $0x15
mov    %ax, [2]
# copy disk parameters from 0x41
mov    $0x0000, %ax
mov    %ax, %ds
lds    %ds:4*0x41, %si
mov    $INITSEG, %ax
mov    %ax, %es
mov    $0x0004, %di
mov    $0x10, %cx
rep
movsb

```

reset_ds_es:

```

mov    $INITSEG, %ax
mov    %ax, %ds
mov    $SETUPSEG, %ax
mov    %ax, %es

```

print_cursor_pos:

```

mov    $0x03, %ah
xor    %bh, %bh
int    $0x10
mov    $12, %cx
mov    $0x0007, %bx
mov    $cursor, %bp
mov    $0x1301, %ax
int    $0x10
mov    [0], %ax
call   print_hex
call   print_nl

```

print_memory_size:

```

mov    $0x03, %ah
xor    %bh, %bh
int    $0x10
mov    $13, %cx
mov    $0x0007, %bx
mov    $memory_size, %bp
mov    $0x1301, %ax
int    $0x10
mov    [2], %ax
call   print_hex

```

print_hd_info:

```

# Cylinders
mov    $0x03, %ah
xor    %bh, %bh
int    $0x10
mov    $26, %cx
mov    $0x0007, %bx
mov    $hdinfo, %bp
mov    $0x1301, %ax
int    $0x10
mov    [4], %ax

```

```

    call    print_hex
    call    print_n1
# head
    mov     $0x03, %ah
    xor     %bh, %bh
    int     $0x10
    mov     $9, %cx
    mov     $0x0007, %bx
    mov     $head, %bp
    mov     $0x1301, %ax
    int     $0x10
    xor     %ax, %ax
    mov     [4+2], %al
    call    print_hex
    call    print_n1
# sect
    mov     $0x03, %ah
    xor     %bh, %bh
    int     $0x10
    mov     $9, %cx
    mov     $0x0007, %bx
    mov     $sect, %bp
    mov     $0x1301, %ax
    int     $0x10
    xor     %ax, %ax
    mov     [0x4+0x0E], %al
    call    print_hex
    call    print_n1

dead_loop:
    jmp     dead_loop
# print hex
print_hex:
    mov     $4, %cx
    mov     %ax, %dx

print_digit:
    rol     $4, %dx
    mov     $0xe0f, %ax
    and     %dl, %al
    add     $0x30, %al
    cmp     $0x3a, %al
    jl      outp

    add     $0x07, %al
outp:
    int     $0x10
    loop    print_digit
    ret

# 打印换行
print_n1:
    mov     $0xe0d, %ax
    int     $0x10
    mov     $0xa, %al
    int     $0x10
    ret

```

```

msg1:
    .ascii "Now we are in SETUP..."
    .byte 13,10,13,10
cursor:
    .ascii "Cursor POS: "
memory_size:
    .ascii "Memory SIZE: "
hdinfo:
    .ascii "KB"
    .byte 13,10,13,10
    .ascii "HD Info"
    .byte 13,10
    .ascii "Cylinders: "
head:
    .ascii "Headers: "
sect:
    .ascii "Sectors: "

    .org 508
root_dev:
    .word ROOT_DEV
boot_flag:
    .word 0xAA55
.text
endtext:
.data
enddata:
.bss
endbss:

```

3.4.4. 程序仿真运行

1. 在 `~/oslab/linux-0.11/boot` 目录下对 `setup.s` 进行改写。
2. 进入 `Linux-0.11` 目录，进行编译。

```

cd ~/oslab
cd linux-0.11
make Image

```

```

guojun@guojunos:~/oslab/linux-0.11$ make Image
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/boot'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/boot'
1+0 records in
1+0 records out
512 bytes copied, 0.00067128 s, 763 kB/s
1+0 records in
1+0 records out
512 bytes copied, 0.000197005 s, 2.6 MB/s
309+1 records in
309+1 records out
158465 bytes (158 kB, 155 KiB) copied, 0.000930966 s, 170 MB/s
2+0 records in
2+0 records out
2 bytes copied, 0.000288306 s, 6.9 kB/s

```

图3-4-1 编译

3. 程序运行，可以看出在 `Bochs` 界面中，能成功打印出目标的字符串。

```
../dbg-bochs
```

```
guojun@guojunos:~/oslab/linux-0.11$ ../dbg-bochs
=====
Bochs x86 Emulator 2.8
Built from GitHub snapshot on March 10, 2024
Timestamp: Sun Mar 10 08:00:00 CET 2024
Compiled by GuoJun, Dec 3 2024
=====
=====
00000000000i[      ] BXSHARE is set to '/usr/local/bochs28/share/bo
chs'
00000000000i[      ] reading configuration from ../bochs/linux-0.11
.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ../bochsout.txt
Bochs internal debugger, type 'help' for help or 'c' to continue
Switching to CPU0
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b
; ea5be000f0
<bochs:1> c
```

图3-4-2 运行程序

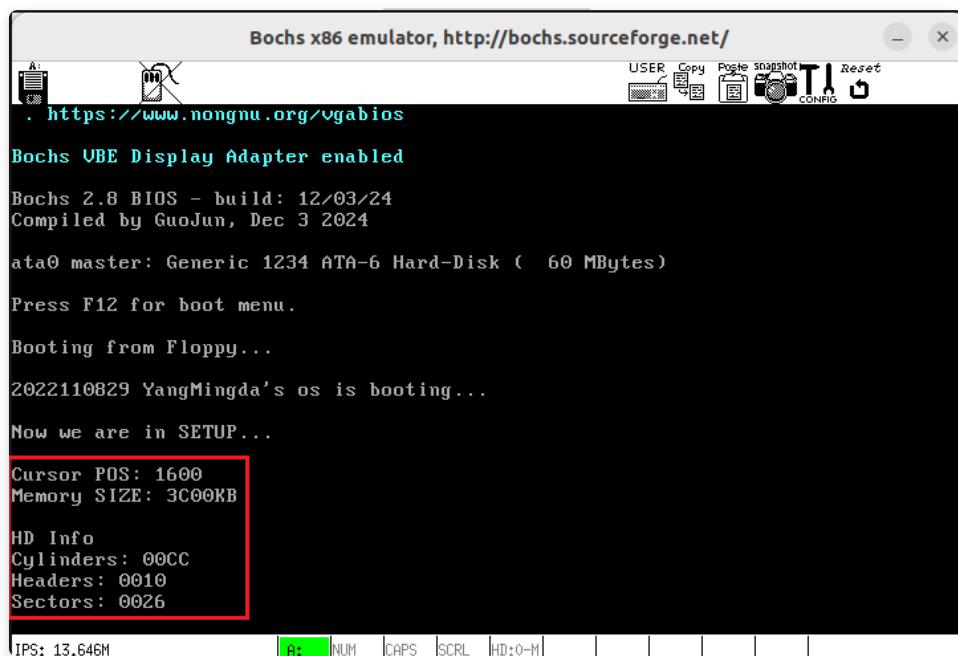


图3-4-3 系统启动情况

四、实验报告

有时，继承传统意味着别手蹩脚。x86 计算机为了向下兼容，导致启动过程比较复杂。请找出 x86 计算机启动过程中，被硬件强制，软件必须遵守的两个“多此一举”的步骤（多找几个也无妨），说说它们为什么多此一举，并设计更简洁的替代方案。

1. 多此一举：SETUPSEG 本应通过计算就能够算出来的，在 0x07e0，但在代码中是 0x9000。开始的时候是引导至 0x07e0，但是在执行结束之后为了后续内容的方便执行，又会将自身移动到比较靠后的位置，这个位置在 Linux-0.11 中就是 0x9000。

替代方案：可以直接引导至 0x9000 的位置进行执行，在执行后再进行向前跳转就可以了。

2. 多此一举：当 PC 的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xffff0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后将启动设备的第一个扇区 512 字节读入内存绝对地址 0x7c00 处。因为当时 system 模块的长度不会超过 0x80000 字节大小 512KB，所以 bootsect 程

序把 system 模块读入物理地址 0x10000 开始位置处时并不会覆盖在 0x90000 处开始的 bootsect 和 setup 模块，多此一举的是 system 模块移到内存中相对靠后的位置，以便加载系统主模块。

替代方案：在保证操作系统启动引导成功的前提下尽量扩大 ROM-BIOS 的内存寻址范围。

3. 多此一举：在 Linux-0.11 中，会将 system 先加载到不与中断向量表冲突的地方，然后再将主模块移动到内存的初始位置，再将这个中断向量表覆盖掉。在 BIOS 初始化时，会在内存的初始位置放置 1kb 的中断向量表，为 BIOS 中断使用。如果在主模块中需要使用一些由 BIOS 中断得到的硬件参数的情况下，就不能在主模块的加载开始过程中直接覆盖掉这 1kb 的中断向量表而让主模块直接从内存的初始位置加载。

替代方案：可以将 1kb 的内容直接放到可以寻址的其他位置，避免先加载再覆盖这一多余的操作。

五、总结

通过本次实验，在 Linux-0.11 下通过修改 bootsect.s 和 setup.s 代码完成操作系统的引导，我熟悉实验环境，建立对操作系统引导过程的深入认识，掌握操作系统的基本开发过程，能对操作系统代码进行简单的控制，揭开操作系统的神秘面纱。在实验过程中，我还不断学习 linux0.11 中 bootsect.s 和 setup.s 汇编代码，了解其中的作用和含义。通过本次实验还锻炼了我解决问题的能力。