



北京交通大学

BEIJING JIAOTONG UNIVERSITY

# 北京交通大学

课程名称：操作系统

实验题目：Linux 启动初始化过程探析

学号：22281188

姓名：江家玮

班级：计科2204班

指导老师：吕凯老师

报告日期：2024-09-20

## 目录

### 一、实验目的

### 二、实验内容

### 三、代码研读

3.1 Linux0.11的基本架构

3.2 操作系统引导及自启动初始化的基本流程

3.2.1 系统引导

3.2.2 进入内核初始化

3.2.3 系统任务初始化

3.3 原理及代码解释

3.3.1 BIOS启动原理

3.3.2 bootsect.s

3.3.3 setup.s

3.3.4 head.s

### 四、实验过程

4.1 实验环境搭建

4.2 测试和验证

### 五、实验心得

### 六、参考资料

# 一、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。

下载和研读Linux内核源码（可以是任意版本），探索、分析和理解操作系统引导和自启动初始化的基本流程，完成相应Linux内核源码的编译和启用，并在虚拟机平台上加以测试验证。

# 二、实验内容

实验主要完成的内容如下：

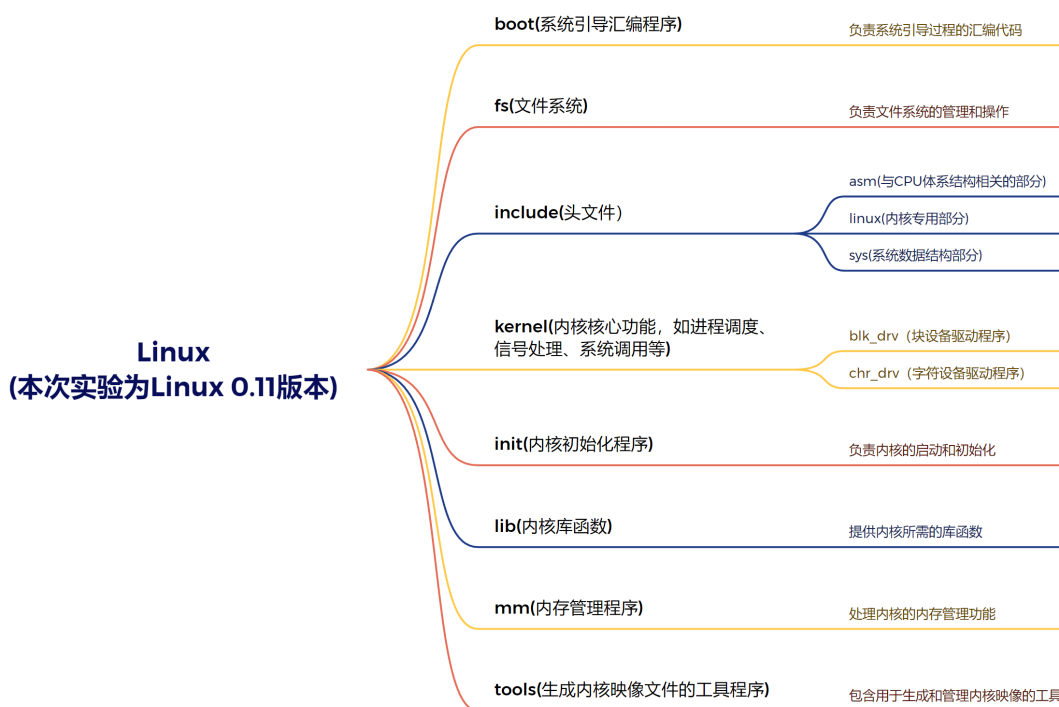
- 下载和研读Linux内核源码（本实验研读的是 linux 0.11 版本）
- 探索、分析和理解操作系统引导和自启动初始化的基本流程
- 完成相应Linux内核源码的编译和启用
- 并在虚拟机平台上加以测试验证

Linux启动初始化过程探析实验基本要求如下：

- (1) 下载和研读Linux内核源码（可以是任意版本）；
- (2) 围绕操作系统引导和自启动初始化过程，研读Linux内核对应源码（包括汇编代码、C程序、Makefile及相关配置文件），整理操作系统引导及自启动初始化的基本流程和设计机理（包括处理器平台相关部分和无关部分，前者可针对x86体系结构或MIPS体系结构或arm64体系结构）
- (3) 完成相应Linux内核源码的编译和启用；
- (4) 在虚拟机平台上启用相应Linux内核并测试验证，对照启动时所显示的系统信息和自己关于内核源码分析结果的一致性。

# 三、代码研读

## 3.1 Linux0.11的基本架构



- **boot**（系统引导汇编程序）：负责系统引导过程的汇编代码。
  - 其中 **boot** 目录下的文件负责**引导和启动**，目录中含有3个汇编语言文件：
    - **bootsect.s** 程序是磁盘引导块程序,编译后会驻留在磁盘的第一个扇区中(引导扇区, 0磁道(柱面),0磁头, 第1个扇区)。在PC机加电ROM BIOS自检后, 将被BIOS加载到内存 **0x7C00** 处进行执行。
    - **setup.s** 程序主要用于读取机器的硬件配置参数, 并把内核模块system移动到适当的内存位置处
    - **head.s** 程序会被编译连接在system模块的最前部分, 主要进行硬件设备的探测设置和内存管理页面
- **fs**（文件系统）：负责文件系统的管理和操作。
- **include**（头文件）：包含多个子目录, 包括 **asm**（与CPU体系结构相关的部分）、**linux**（内核专用部分）和 **sys**（系统数据结构部分）。
- **init**（内核初始化程序）：负责内核的启动和初始化。
- **kernel**（内核核心功能, 如进程调度、信号处理、系统调用等）：包含两个子目录 **blk\_drv**（块设备驱动程序）和 **chr\_drv**（字符设备驱动程序）。
- **lib**（内核库函数）：提供内核所需的库函数。
- **mm**（内存管理程序）：处理内核的内存管理功能。
- **tools**（生成内核映像文件的工具程序）：包含用于生成和管理内核映像的工具。

## 3.2 操作系统引导及自启动初始化的基本流程

在操作系统的引导及自启动初始化过程中, 涉及多个步骤, 以下是具体描述 (以Linux0.11为例) :

### 3.2.1 系统引导

操作系统引导是启动过程中最初的一部分, 通常由引导加载程序 (如BIOS或UEFI) 负责, 将内核从存储设备加载到内存中并执行启动代码。

**关键步骤:**

**BIOS/UEFI加载:** 计算机电源启动后, BIOS/UEFI会执行自检 (POST) 并选择引导设备 (如硬盘、CD-ROM、USB驱动器等) 。

**MBR加载 (主引导记录):** BIOS查找存储设备的主引导记录 (MBR), 并加载其中的引导加载程序到内存。MBR通常位于存储设备的第一个扇区。

**引导加载程序:** 引导加载程序 (如GRUB、LILO等) 负责将内核从磁盘读取并加载到内存中。它会设置加载内核所需的环境, 并将控制权移交给内核。

在Linux0.11中, **bootsect.s**和**setup.s**分别负责加载内核和执行基本的硬件初始化。

### 3.2.2 进入内核初始化

一旦引导加载程序将内核加载到内存中, 控制权转移给内核, 系统进入内核模式, 开始执行内核的初始化代码。

**关键步骤:**

**切换到保护模式：**内核首先会从实模式切换到保护模式，开启多任务和内存保护。这一步通过 `head.s` 实现，负责设置内存管理结构（GDT/IDT）和段寄存器。

**硬件初始化：**内核开始初始化各种硬件子系统，如中断控制器、定时器、串口、键盘等设备驱动。

**内存初始化：**分页机制和物理内存管理结构在此阶段初始化，设置内存页表，启用虚拟内存（由 `mm/memory.c` 负责）。

**设备初始化：**块设备（如硬盘）和字符设备（如终端、串口）的驱动程序在内核启动期间被加载和初始化（如 `drivers/block/hd.c` 和 `drivers/char/tty.c`）。

### 3.2.3 系统任务初始化

内核初始化完成后，进入任务管理的初始化阶段，内核会启动调度器和进程管理系统。

**关键步骤：**

**创建空闲进程 (IdleTask)：**操作系统首先创建一个空闲进程，用于当系统没有其他任务时保持系统的运行。空闲任务通常是PID为0的特殊进程，由 `kernel/sched.c` 管理。

**启动第一个用户态进程 (init进程)：**这是系统启动的第一个用户态进程，PID为1。它负责启动所有其他系统服务和进程。init进程会读取系统的启动配置（如 `/etc/inittab` 并启动相关服务。

## 3.3 原理及代码解释

在研究 `Linux 0.11` 的启动和初始化过程中，关键文件包括 `bootsect.s`、`setup.s`、`head.s`（负责引导与硬件初始化）。

### 3.3.1 BIOS启动原理

BIOS的启动原理是在计算机开机后，首先执行加电自检（POST），检测和初始化硬件设备。接着，BIOS按预设的启动顺序寻找引导设备（如硬盘、光驱或USB），加载主引导记录（MBR）中的引导加载程序。引导加载程序负责将操作系统的内核加载到内存中，并将控制权交给操作系统，从而完成启动。BIOS提供了硬件初始化和引导功能，是操作系统启动前的关键步骤。

### 3.3.2 bootsect.s

`bootsect.s` 是 `Linux 0.11` 内核的引导扇区代码，负责在系统启动时将内核加载到内存中。它是最早执行的代码之一，位于磁盘的第一个扇区（主引导记录，MBR），大小为512字节。`bootsect.s` 的主要任务是读取操作系统的剩余部分，并将其加载到内存的特定位置，准备将控制权交给后续的引导程序或内核。

**主要功能：**

**加载内核：**从启动设备（通常是软盘或硬盘）读取更多的数据块，并将其加载到内存中指定的地址，通常是低地址区域。

**切换模式准备：**虽然 `bootsect.s` 运行在实模式（16位模式），但它准备将CPU切换到保护模式，以便操作系统能够利用现代的内存管理和多任务处理功能。

**跳转到内核：**在内核成功加载到内存后，`bootsect.s` 会将控制权交给下一阶段的启动代码（如 `setup.s`），并退出。

以下是 `bootsect.s` 代码注释：

```
1 | !SYSSIZE = 0x3000
2 | ! system 模块的最大大小（单位为16字节一节）。这个值提供了一个默认的最大值。
```

```

3
4 ! bootsect.s 是被BIOS加载到内存地址0x7c00处的引导程序，并将自己移到0x90000处，
5 ! 随后加载setup和system程序。
6 !
7 ! bootsect 程序将 BIOS 使用的启动扇区加载到内存，并负责将 setup 和 system
8 ! 程序加载到内存中的合适位置。
9
10 .globl begtext, begdata, begbss, endtext, enddata, endbss ! 定义全局符号
11 .text ! 开始文本段
12 begtext:
13 .data ! 开始数据段
14 begdata:
15 .bss ! 开始未初始化数据段（BSS 段）
16 begbss:
17 .text ! 再次回到文本段
18
19 SETUPLEN = 4 ! setup 程序占用的扇区数
20 BOOTSEG = 0x07c0 ! 引导扇区在内存中的原始段地址（BIOS 加载到此地址）
21 INITSEG = 0x9000 ! 将引导程序移动到此段地址（0x90000处）
22 SETUPSEG = 0x9020 ! setup 程序加载的内存段地址（0x90200处）
23 SYSSEG = 0x1000 ! system 模块加载到此处（0x10000处）
24 ENDSEG = SYSSEG + SYSSIZE ! 停止加载的段地址
25
26 ROOT_DEV = 0x306 ! 指定根文件系统设备号为第二个硬盘的第一个分区
27
28 entry start ! 程序从 start 开始执行
29 start:
30     mov ax, #BOOTSEG ! 设置 ds 段寄存器为引导扇区段地址 0x07C0
31     mov ds, ax
32     mov ax, #INITSEG ! 设置 es 段寄存器为 0x9000，即引导程序的新地址
33     mov es, ax
34     mov cx, #256 ! 设置复制的字数为 256 字（512 字节）
35     sub si, si ! 源地址 0x07C0:0x0000
36     sub di, di ! 目的地址 0x9000:0x0000
37     rep
38     movw ! 复制一个字，重复直到 cx = 0
39     jmp i go, INITSEG ! 跳转到新位置，开始执行移动后的引导代码
40
41 go:
42     mov ax, cs ! 将代码段寄存器 CS 的值加载到 ax
43     mov ds, ax ! 设置 ds 段寄存器为代码段
44     mov es, ax ! 设置 es 段寄存器为代码段
45     mov ss, ax ! 设置 ss 段寄存器为代码段，指向堆栈
46     mov sp, #0xFF00 ! 将栈顶指针设置为0xFF00
47
48 ! 加载 setup 程序到 0x90200 地址处（setup 程序位于软盘第2扇区开始）
49 load_setup:
50     mov dx, #0x0000 ! 设置驱动器号为0（软盘）
51     mov cx, #0x0002 ! 设置读取的扇区为第二扇区
52     mov bx, #0x0200 ! 设置缓冲区地址为0x0200
53     mov ax, #0x0200+SETUPLEN ! 读取 setup 程序的扇区数
54     int 0x13 ! 调用 BIOS 读盘中断
55     jnc ok_load_setup ! 如果成功读取则继续
56     mov dx, #0x0000 ! 如果读取失败，重置驱动器
57     mov ax, #0x0000

```

```

58     int 0x13
59     jmp load_setup    ! 重新尝试加载
60
61 ok_load_setup:
62     ! 读取驱动器参数, 如每磁道的扇区数量
63     mov dl,#0x00
64     mov ax,#0x0800    ! 使用 BIOS 中断获取驱动器参数
65     int 0x13
66     mov ch,#0x00
67     mov sectors,cx    ! 保存每磁道的扇区数量
68
69     ! 显示 "Loading system..." 提示
70     mov ah,#0x03      ! 获取光标位置
71     xor bh,bh
72     int 0x10
73     mov cx,#24        ! 显示24个字符
74     mov bx,#0x0007    ! 设置显示属性为普通
75     mov bp,msg1       ! 设置显示字符串的起始地址
76     mov ax,#0x1301    ! 显示字符串并移动光标
77     int 0x10
78
79     ! 将 system 模块从软盘加载到内存
80     mov ax,#SYSSEG    ! 设置 es 段寄存器为 system 模块的加载地址
81     mov es,ax
82     call read_it      ! 调用读取系统模块的子程序
83     call kill_motor   ! 关闭驱动器马达
84
85     ! 设置根设备号
86     seg cs
87     mov ax,root_dev
88     cmp ax,#0
89     jne root_defined
90     mov ax,#0x0208    ! 设置为1.2MB软驱设备号
91     cmp bx,#15
92     je root_defined
93     mov ax,#0x021c    ! 设置为1.44MB软驱设备号
94     cmp bx,#18
95     je root_defined
96     jmp undef_root    ! 如果无法识别根设备则进入死循环
97
98 root_defined:
99     seg cs
100    mov root_dev,ax    ! 保存根设备号
101
102    ! 跳转到 setup 程序, 继续执行
103    jmp 0,SETUPSEG
104
105    ! 读取 system 模块到内存的子程序
106 read_it:
107    ! 检查 es 段地址是否位于 64KB 边界
108    mov ax,es
109    test ax,#0xffff
110    jne die            ! 如果不在 64KB 边界, 则进入死循环
111
112    xor bx,bx          ! 清除段内偏移值

```

```

113      ! 开始读取 system 模块
114      rp_read:
115          mov ax,es
116          cmp ax,#ENDSEG  ! 检查是否已经加载完所有数据
117          jb ok1_read      ! 如果还未加载完，则继续
118          ret              ! 如果加载完，则返回
119      ok1_read:
120          call read_track
121          mov sread,ax     ! 保存当前磁道已读扇区数
122          add ax,sread
123          cmp ax,sectors
124          jne ok3_read
125          mov ax,#1
126          sub ax,head
127          jne ok4_read
128          inc track
129      ok4_read:
130          mov head,ax
131          xor ax,ax
132      ok3_read:
133          mov sread,ax
134          shl cx,#9
135          add bx,cx
136          jnc rp_read
137          add ax,#0x1000
138          xor bx,bx
139          jmp rp_read
140
141      kill_motor:
142          mov dx,#0x3f2
143          mov al,#0
144          outb
145          ret
146
147      msg1:
148          .byte 13,10
149          .ascii "Loading system ..."
150          .byte 13,10,13,10
151
152      root_dev:
153          .word ROOT_DEV
154
155      boot_flag:
156          .word 0xAA55

```

### 3.3.3 setup.s

前对系统进行必要的硬件初始化，确保内存布局正确并设置好中断和段寄存器。`setup.s` 在 `bootsect.s` 将其加载到内存后执行。

**主要功能：**

**初始化中断描述符表 (IDT) 和全局描述符表 (GDT)：** 设置中断向量表和全局段描述符表，使系统能够正确处理中断和内存段管理。

**切换到保护模式：**将CPU从实模式切换到保护模式，使操作系统能够访问超过1MB的内存并启用分页等现代功能。

**硬件初始化：**初始化包括定时器、键盘等基本硬件设备，确保系统能够正常运行。

**内存布局设置：**配置内存段寄存器和分页机制，为操作系统管理内存做好准备。

**跳转到内核：**所有硬件和内存配置完成后，`setup.s` 最终跳转到内核的入口地址，开始执行内核代码。

以下是 `setup.s` 代码注释：

```
1  ! setup.s (C) 1991 Linus Torvalds
2  !
3  ! setup.s 的主要功能是从 BIOS 获取系统参数（如内存、硬盘信息等），并将这些数据存放到安全的内存位置。
4  ! 这些数据将供保护模式系统使用，确保系统能够正确运行。
5  ! setup.s 和系统内核已经由引导块（bootsect.s）加载到了内存中。
6  !
7
8  INITSEG = 0x9000      ! 引导扇区移动后的地址，用来存放 BIOS 参数
9  SYSSEG  = 0x1000      ! 系统内核加载的地址（0x10000）
10 SETUPSEG = 0x9020     ! setup 程序当前加载的段地址
11
12 .globl begtext, begdata, begbss, endtext, enddata, endbss ! 声明全局变量
13 .text
14 begtext: ! 文本段开始
15 .data
16 begdata: ! 数据段开始
17 .bss
18 begbss: ! BSS 段（未初始化数据段）开始
19 .text
20
21 .entry start ! 程序入口，从 start 开始执行
22 start:
23
24 ! 设置堆栈：虽然在 bootsect.s 中已经完成，但这里再次设置堆栈
25 mov ax, #INITSEG      ! 初始化段寄存器 ss 和堆栈指针 sp
26 mov ss, ax
27 mov sp, #0xFF00       ! 设置栈顶指针为 0xFF00
28
29 ! 输出消息，表示进入 setup 阶段
30 mov ax, #SETUPSEG     ! 设置 es 寄存器为当前段地址
31 mov es, ax
32 call read_cursor      ! 获取当前光标位置
33 mov cx, #14           ! 字符串长度（即要显示的字符数量）
34 mov bx, #0x0007       ! 显示属性：亮绿色
35 mov bp, #msg          ! 消息字符串的地址
36 mov ax, #0x1301       ! BIOS 中断 0x10: 显示字符串
37 int 0x10              ! 调用 BIOS 中断显示消息
38
39 ! 获取系统光标位置并保存
40 mov ax, #INITSEG      ! 设置段寄存器 ds
41 mov ds, ax
42 mov ah, #0x03         ! BIOS 中断 0x10: 获取光标位置
43 xor bh, bh
```



```

44 int 0x10                ! 将光标位置存储在 0x90000 地址
45
46 ! 获取系统内存大小（扩展内存，以 KB 计）
47 mov ah, #0x88          ! BIOS 中断 0x15: 获取内存大小
48 int 0x15
49 mov [2], ax             ! 将内存大小保存到 0x90000 + 2 地址
50
51 ! 获取显卡数据
52 mov ah, #0x0f          ! BIOS 中断 0x10: 获取显卡信息
53 int 0x10
54 mov [4], bx             ! 保存显示页面信息
55 mov [6], ax             ! 保存视频模式和窗口宽度信息
56
57 ! 检测 EGA/VGA 显示卡
58 mov ah, #0x12
59 mov bl, #0x10          ! 传递参数，调用 BIOS 中断 0x10 获取 VGA 信息
60 int 0x10
61 mov [8], ax             ! 保存 VGA 配置数据
62 mov [10], bx
63 mov [12], cx
64
65 ! 获取第一个硬盘的数据
66 mov ax, #0x0000        ! 选择 BIOS 数据区
67 mov ds, ax
68 lds si, [4*0x41]        ! 从 BIOS 数据区加载硬盘信息指针
69 mov ax, #INITSEG
70 mov es, ax
71 mov di, #0x0080        ! 将硬盘数据保存到 0x90000 + 0x80
72 mov cx, #0x10          ! 复制 16 字节的数据
73 rep
74 movsb
75
76 ! 获取第二个硬盘的数据
77 mov ax, #0x0000
78 mov ds, ax
79 lds si, [4*0x46]        ! 从 BIOS 数据区加载第二个硬盘信息指针
80 mov ax, #INITSEG
81 mov es, ax
82 mov di, #0x0090        ! 将数据保存到 0x90000 + 0x90
83 mov cx, #0x10          ! 复制 16 字节
84 rep
85 movsb
86
87 ! 检测是否存在第二块硬盘
88 mov ax, #0x01500        ! 使用 BIOS 中断 0x13 检测硬盘
89 mov dl, #0x81          ! 第二块硬盘编号
90 int 0x13
91 jc no_disk1            ! 如果出错，则跳转到 no_disk1 标签
92 cmp ah, #3              ! 检查 BIOS 返回码，若没有错误，跳转到 is_disk1
93 je is_disk1
94 no_disk1:
95 mov ax, #INITSEG
96 mov es, ax
97 mov di, #0x0090        ! 清空第二块硬盘的相关数据
98 mov cx, #0x10

```

```

99  mov ax, #0x00
100 rep
101 stosb
102 is_disk1:
103
104 ! 重新设置中断描述符表 (IDT) 和全局描述符表 (GDT)
105 mov ax, #SETUPSEG
106 mov ds, ax
107 lidt idt_48      ! 加载 IDT 表
108 lgdt gdt_48      ! 加载 GDT 表
109
110 ! 打开 A20 地址线, 允许访问 1MB 以上的内存
111 call empty_8042   ! 等待键盘控制器为空
112 mov al, #0xD1     ! 发送命令到键盘控制器
113 out #0x64, al
114 call empty_8042
115 mov al, #0xDF     ! 启用 A20 地址线
116 out #0x60, al
117 call empty_8042
118
119 ! 重新编程 8259 中断控制器, 避免中断冲突
120 mov al, #0x11     ! 初始化 8259 中断控制器
121 out #0x20, al     ! 发送到主 8259
122 out #0xA0, al     ! 发送到从 8259
123 mov al, #0x20     ! 设置主 8259 的中断偏移地址为 0x20
124 out #0x21, al
125 mov al, #0x28     ! 设置从 8259 的中断偏移地址为 0x28
126 out #0xA1, al
127
128 ! 切换到保护模式
129 mov ax, #0x0001   ! 设置 CR0 的保护模式位
130 lmsw ax           ! 启用保护模式
131 jmp 0, 8          ! 跳转到段选择子 8 的代码段, 进入 32 位保护模式

```

### 3.3.4 head.s

`head.s` 是 Linux 0.11 中的一个关键汇编文件, 它承担了内核启动的第一步, 负责把系统从实模式 (16位) 切换到保护模式 (32位)。

#### 主要功能:

**设置全局描述符表 (GDT)** `head.s` 初始化 GDT, 全局描述符表是保护模式下管理内存段的方式。每个内存段 (代码段、数据段等) 都会在 GDT 中有一个条目。这个就像给操作系统的各个部分发通行证, 让它们可以访问相应的内存区域。

**切换到保护模式:** 它会打开 CPU 的保护模式开关, 从 16 位转移到 32 位。这让系统能处理更多内存, 开启多任务处理等强大的功能。

**分页 (不完全启动):** `head.s` 还为分页机制做了准备, 不过 Linux 0.11 的分页还比较基础。分页可以让每个程序以为自己独占了内存。

**跳转到内核:** 最后, 当所有的基础设施搭建好后, `head.s` 把控制权交给了 Linux 内核的 C 语言代码, 接下来的事情就由内核处理了。

```

1  ! head.s 包含了 32 位的启动代码, 启动地址为 0x00000000, 并会被页目录覆盖。
2

```



```

57  xorl %edi, %edi
58  cld; rep; stosl          ! 清空页目录
59  movl $pg0+7, pg_dir     ! 设置页目录项
60  movl $pg1+7, pg_dir+4
61  movl $pg2+7, pg_dir+8
62  movl $pg3+7, pg_dir+12
63  movl $pg3+4092, %edi     ! 页表项
64  movl $0xffff007, %eax    ! 初始化页表
65  std
66  1: stosl                ! 向后填充页表
67  subl $0x1000, %eax
68  jge 1b
69  movl %eax, %cr3          ! 加载页目录基地址到 CR3
70  movl %cr0, %eax
71  orl $0x80000000, %eax    ! 设置分页位 (PG)
72  movl %eax, %cr0
73  ret

```

## 四、实验过程

### 4.1 实验环境搭建

首先在Vmware Workstation上搭建虚拟机Ubuntu18.04

```

coconut@Coconut:~$ hostnamectl
Static hostname: Coconut
Icon name: computer-vm
Chassis: vm
Machine ID: a6a94eea8d0f47238c4f98aeaa361a2a
Boot ID: 27b708d13d574c77b0d2c605b2692258
Virtualization: vmware
Operating System: Ubuntu 18.04.6 LTS
Kernel: Linux 5.4.0-150-generic
Architecture: x86-64

```

而后下载Linux 0.11

```

1  sudo apt-get install git
2  git clone https://github.com/Wangzhike/HIT-Linux-0.11.git

```

进入文件夹后运行脚本

```

1  cd JJW-Linux-0.11/0-prepEnv/hit-oslab-qiuyu/setup.sh

```

```

coconut@Coconut: ~/JJW-Linux-0.11/0-prepEnv/hit-oslab-qiuyu
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
coconut@Coconut:~$ cd JJW-Linux-0.11/0-prepEnv/hit-oslab-qiuyu/
coconut@Coconut:~/JJW-Linux-0.11/0-prepEnv/hit-oslab-qiuyu$ ./setup.sh
* Update apt sources.....
[sudo] coconut 的密码:
命中:4 http://security.ubuntu.com/ubuntu bionic-security InRelease
命中:1 http://mirrors.tuna.tsinghua.edu.cn/ubuntu bionic InRelease
命中:2 http://mirrors.tuna.tsinghua.edu.cn/ubuntu bionic-updates InRelease
命中:3 http://mirrors.tuna.tsinghua.edu.cn/ubuntu bionic-backports InRelease
正在读取软件包列表... 完成
* Create oslab main directory..... Done
* Extract linux 0.11 and bochs and bde image..... Done

```

本脚本会将实验环境安装在当前登录用户的家目录下，文件名为 `oslab`，即我们的实验目录是

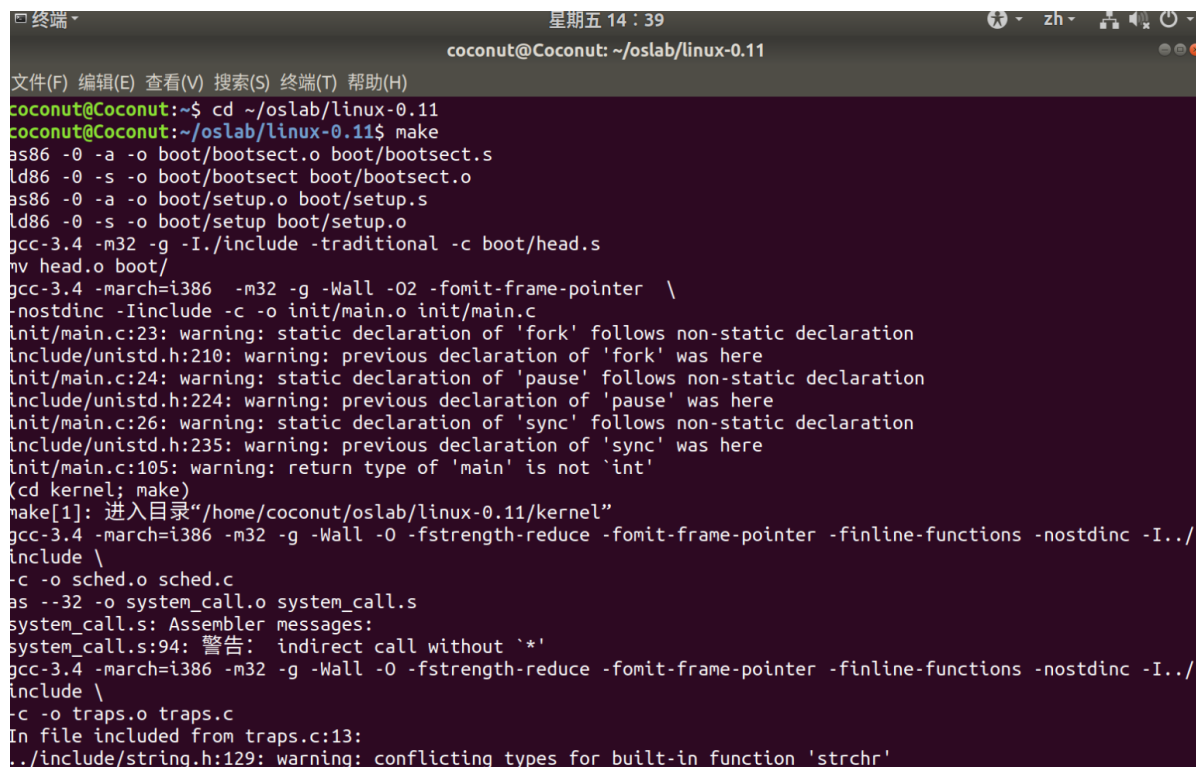
`~/oslab`

这个脚本会下载并安装许多软件包如：

```
1 gcc-3.4
2 bin86
3 libc6-dev-i386
4 build-essential
5 libsm6:i386
6 libx11-6:i386
7 libxpm4:i386
```

编译Linux 0.11、运行

```
1 cd ~/oslab/linux-0.11
2 make
```



```
coconut@Coconut:~/oslab/linux-0.11$ cd ~/oslab/linux-0.11
coconut@Coconut:~/oslab/linux-0.11$ make
as86 -0 -a -o boot/bootsect.o boot/bootsect.s
ld86 -0 -s -o boot/bootsect boot/bootsect.o
as86 -0 -a -o boot/setup.o boot/setup.s
ld86 -0 -s -o boot/setup boot/setup.o
gcc-3.4 -m32 -g -I./include -traditional -c boot/head.s
mv head.o boot/
gcc-3.4 -march=i386 -m32 -g -Wall -O2 -fomit-frame-pointer \
-nostdinc -Iinclude -c -o init/main.o init/main.c
init/main.c:23: warning: static declaration of 'fork' follows non-static declaration
include/unistd.h:210: warning: previous declaration of 'fork' was here
init/main.c:24: warning: static declaration of 'pause' follows non-static declaration
include/unistd.h:224: warning: previous declaration of 'pause' was here
init/main.c:26: warning: static declaration of 'sync' follows non-static declaration
include/unistd.h:235: warning: previous declaration of 'sync' was here
init/main.c:105: warning: return type of 'main' is not 'int'
(cd kernel; make)
make[1]: 进入目录“/home/coconut/oslab/linux-0.11/kernel”
gcc-3.4 -march=i386 -m32 -g -Wall -O -fstrength-reduce -fomit-frame-pointer -finline-functions -nostdinc -I../
include \
-c -o sched.o sched.c
as --32 -o system_call.o system_call.s
system_call.s: Assembler messages:
system_call.s:94: 警告: indirect call without '*'
gcc-3.4 -march=i386 -m32 -g -Wall -O -fstrength-reduce -fomit-frame-pointer -finline-functions -nostdinc -I../
include \
-c -o traps.o traps.c
In file included from traps.c:13:
./include/string.h:129: warning: conflicting types for built-in function 'strchr'
```

此时会生成镜像文件Image

```
1 cd ~/oslab
2 ./run
```

同时弹出 Bochs 的界面，其中 Loading system 是 bootsect 在加载 system 的时候输出的提示信息。接下来显示的是一些系统的信息如空闲的内存容量是 12582912 bytes

```
Bochs x86 emulator, http://bochs.sourceforge.net/
his VGA/VE BIOS is released under the GNU LGPL

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/13/08
Revision: 1.194 $ $Date: 2007/12/23 19:46:27 $
Options: apmbios pcibios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)

Booting from Floppy...

Loading system ...

Partition table ok.
39065/62000 free blocks
19520/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Ok.
[usr/root]# ls
README          hello          hello.o        linux0.tgz     shoe
gcclib140       hello.c        linux-0.00     mtools.howto  shoelace.tar.Z
[usr/root]#

Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008

00000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
```

而后在 Bochs 界面输入 `ls`，出现下图说明编译成功：

```
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/13/08
$Revision: 1.194 $ $Date: 2007/12/23 19:46:27 $
Options: apmbios pcibios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)

Booting from Floppy...

Loading system ...

Partition table ok.
39065/62000 free blocks
19520/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Ok.
[usr/root]# ls
README          hello          hello.o        linux0.tgz     shoe
gcclib140       hello.c        linux-0.00     mtools.howto  shoelace.tar.Z
[usr/root]#
```

然而在运行此语句时：

```
1 | ./rungdb
```

出现找不到 `libncurses.so.5` 找不到 `libexpat.so.1` 两个报错

```
1 | /rungdb
2 | ./gdb: error while loading shared libraries: libncurses.so.5: cannot open
  | shared object file: No such file or directory
```

解决办法是：

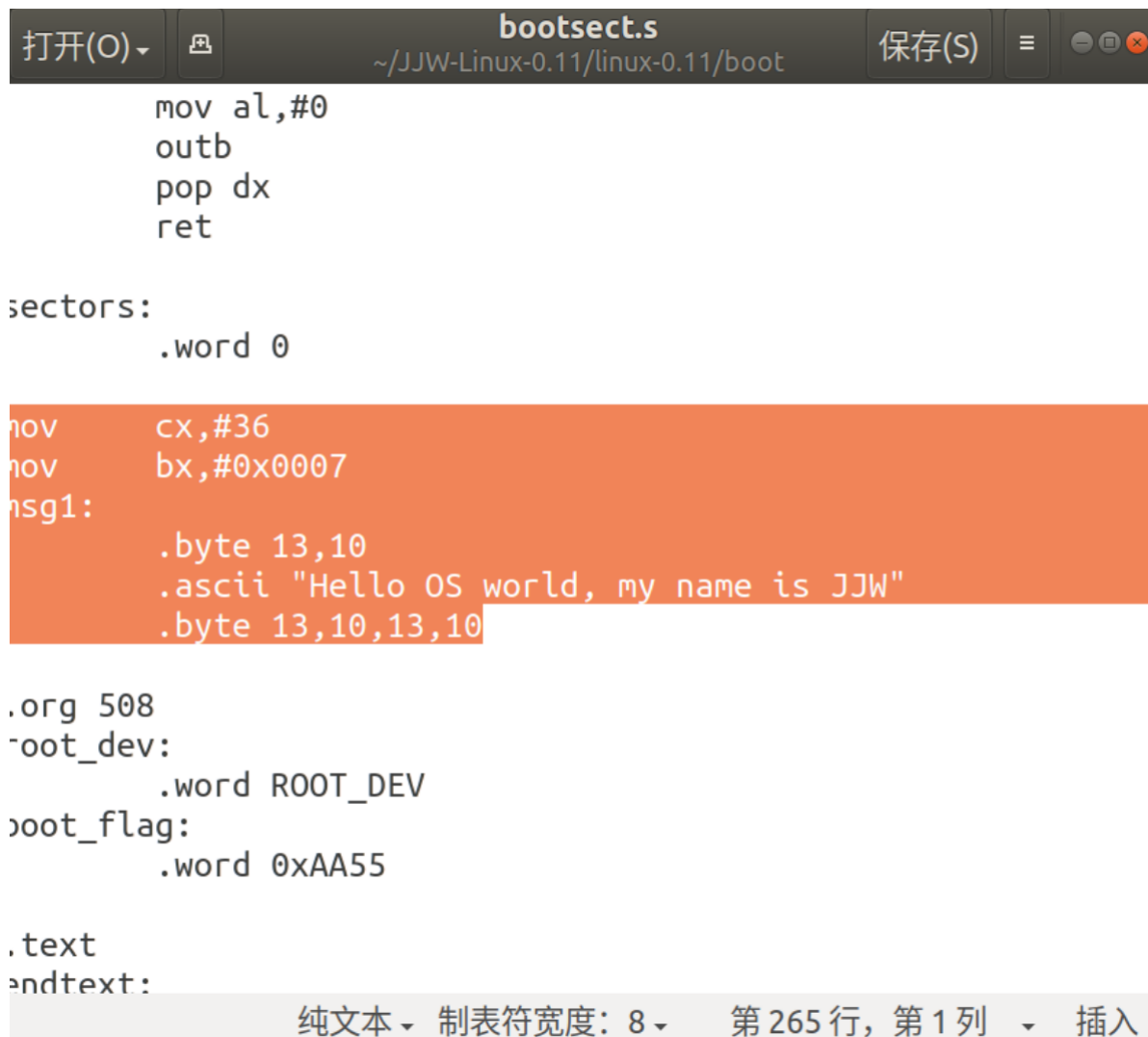
```
1 | sudo apt-get install libncurses5:i386
2 | sudo apt-get install libexpat1-dev:i386
```

## 4.2 测试和验证

通过修改 `bootsect.s` 代码，在 bochs 模拟器上打印出一句话验证一致性，说明了系统确实到达并正确执行了修改的代码路径。且如果打印的消息如预期显示，说明启动流程中的 `bootsect.s` 部分被正确加载和执行。

完成目标：改写 `bootsect.s` 代码，在屏幕上输出一句话（本实验以输出 `Hello OS world, my name is JJW` 为例）

首先修改 `bootsect.s`：



```
bootsect.s
~/JJW-Linux-0.11/linux-0.11/boot

    mov al, #0
    outb
    pop dx
    ret

sectors:
    .word 0

mov     cx, #36
mov     bx, #0x0007
msg1:
    .byte 13, 10
    .ascii "Hello OS world, my name is JJW"
    .byte 13, 10, 13, 10

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

.text
endtext:
纯文本 制表符宽度: 8 第 265 行, 第 1 列 插入
```

进入 `~/linux-0.11/boot/` 目录下运行以下两条指令实现编译链接：

```
1 | as86 -O -a -o bootsect.o bootsect.s
2 | ld86 -O -s -o nootsect bootsect.o
```

其中运行上述两条命令，若无其他输出，说明编译链接均已通过，继续在终端输入 `ls -l`，则如下图：



```

coconut@Coconut: ~/JJW-Linux-0.11/linux-0.11/boot
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
coconut@Coconut:~/JJW-Linux-0.11/linux-0.11/boot$ as86 -0 -a -o bootsect.o bootsect.s
coconut@Coconut:~/JJW-Linux-0.11/linux-0.11/boot$ ld86 -0 -s -o bootsect bootsect.o
coconut@Coconut:~/JJW-Linux-0.11/linux-0.11/boot$ ls -l
总用量 72
-rw-rw-r-- 1 coconut coconut 0 9月 20 14:32 a.out
-rwxrwxr-x 1 coconut coconut 544 9月 20 15:37 bootsect
-rw-rw-r-- 1 coconut coconut 1002 9月 20 15:37 bootsect.o
-rw-rw-r-- 1 coconut coconut 5486 9月 20 15:37 bootsect.s

```

`bootsect` 文件的大小为 544 字节，而引导程序的大小必须正好为一个磁盘扇区，即 512 字节。多出的 32 字节的原因在于 `ld86` 生成了 Minix 可执行文件格式。除了包含文本段和数据段之外，这种格式的可执行文件还带有一个 Minix 文件头。因此，`bootsect` 文件的前几个字节应该是 01 03 10 04。要验证这一点，可以在 Ubuntu 系统下使用 `hexdump -C bootsect` 命令进行查看：

```

coconut@Coconut: ~/JJW-Linux-0.11/linux-0.11/boot
coconut@Coconut:~/JJW-Linux-0.11/linux-0.11/boot$ hexdump -C bootsect
00000000 01 03 10 04 20 00 00 00 00 02 00 00 00 00 00 00 |....|
00000010 00 00 00 00 00 00 00 00 00 82 00 00 00 00 00 00 |.....|
00000020 b8 c0 07 8e d8 b8 00 90 8e c0 b9 00 01 29 f6 29 |.....).)|
00000030 ff f3 a5 ea 18 00 00 90 8c c8 8e d8 8e c0 8e d0 |.....|
00000040 bc 00 ff ba 00 00 b9 02 00 bb 00 02 b8 04 02 cd |.....|
00000050 13 73 0a ba 00 00 b8 00 00 cd 13 eb e6 b2 00 b8 |.s.....|
00000060 00 08 cd 13 b5 00 2e 89 0e 67 01 b8 00 90 8e c0 |.....g.....|

```

然后使用命令 `dd bs=1 if=bootsect of=Image skip=32` 要去掉这 32 个字节的文件头部。

```

coconut@Coconut:~/oslab/linux-0.11/boot$ as86 -0 -a -o bootsect.o bootsect.s
coconut@Coconut:~/oslab/linux-0.11/boot$ ld86 -0 -s -o bootsect bootsect.o
coconut@Coconut:~/oslab/linux-0.11/boot$ dd bs=1 if=bootsect of=Image skip=32
记录了512+0 的读入
记录了512+0 的写出
512 bytes copied, 0.000828527 s, 618 kB/s

```

去掉这 32 个字节后，将生成的 `Image` 文件拷贝到 `linux-0.11` 目录下，而后执行：

```

1 | cd ~/oslab
2 | ./run

```

可以看到 `bootsect.s` 修改的部分可以在 `Bochs` 中运行并打印出来：

```

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/13/08
$Revision: 1.194 $ $Date: 2007/12/23 19:46:27 $
Options: apmbios pcibios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)

Booting from Floppy...
Hello OS world, my name is JJW

```

## 五、实验心得



通过这次实验，我对 Linux 内核的启动过程有了更深入的理解。在阅读 Linux 内核源码的过程中，我也加深了对计算机底层架构和原理的认识。例如，之前在课堂上提到的实模式和保护模式，当时对这两者的概念不太清晰。然而，通过此次探究 Linux 启动过程中从实模式切换到保护模式的过程，我更加明确了这两种模式的差异，以及它们在中断处理方式上的不同。在分析 bootsect 源码时，我深感操作系统设计者的巧妙之处，特别是在内存规划方面，设计者经过了精密计算，考虑了全面的需求。在研究 setup.s 程序时，我发现 system 模块复制到 0x00000 地址的设计非常精妙，它废除了 BIOS 的 16 位中断机制，并为 32 位操作系统的中断机制铺平了道路。此外，setup 程序中对中断描述符表寄存器（IDTR）和全局描述符表寄存器（GDTR）的初始化设置，也加深了我对课堂上描述符表内容的理解。

Linux 0.11 版本的代码量相对较小，比较适合入门学习。然而，在学习过程中，我也遇到了一些难以理解的部分。通过参考赵炯博士的解析，我逐步理解了这些难点。尽管这一过程枯燥且充满挑战，但我也从中获得了很大的收获。最终，我通过修改一些脚本和代码，并成功编译和启动系统，这不仅提升了我的实际操作能力，也进一步加深了对源码的理解。

## 六、参考资料

---

【1】<https://www.lanqiao.cn/courses/115>

【2】CSDN博客, "Linux 内核启动过程解析," [https://blog.csdn.net/qq\\_39557240/article/details/85336730](https://blog.csdn.net/qq_39557240/article/details/85336730).

【3】《Linux内核完全注释》