

Linux启动初始化过程探析实验报告

|  |  |
| --- | --- |
| 学 号： |  |
| 姓 名： |  |
| 学 院： |  |
| 提交日期： | 2022.9.15 |

**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。

**二、实验内容**

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。

**三、实验要求**

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。

**四、整体研读Linux内核源码**

在Linux操作系统中，内核是其最基础也是最重要的模块，它有着了提供硬件抽象层、磁盘及文件系统控制、多任务等功能，以及一些其他计算机必须用到的组件。严格意义来说，Linux这个词本身也是专指Linux内核，但是目前大部分人都将Linux理解为整个Linux系统。本实验中我完成的内容是专指Linux内核部分源码的研读。

### 进程管理（Process Scheduler)

Linux内核对进程和线程没有特别区分，在实现上没有特别的调度算法或定义数据结构来标识线程，进程和线程都是用相同的进程PCB数据结构。而创建和操作进程需要通过fork()、vfork()、clone()的系统调用，最后进入内核空间。

内核里使用clone()方法来创建线程，其工作方式与fork()类似。但是会区分哪些资源与进程共享，哪些资源是线程独有。

  Linux系统通过fork()系统调用复制父进程来创建子进程，接着调用exec()函数来创建新的地址空间，并把新的程序载入其中。最终，程序通过exit()系统调用退出执行，进入僵死状态，等待父进程调用wait()或waitpid()函数回收其资源。

进程的所有信息被task\_struct数据结构所描述，通常称它为进程PCB。每一个进程对应一个PCB，系统中的所有PCB存放在一个双向循环链表中，称这个链表为任务[队列](https://so.csdn.net/so/search?q=%E9%98%9F%E5%88%97&spm=1001.2101.3001.7020" \t "_blank)。内核中current全局变量利用内核栈的特性可以获取当前进程的PCB。首先，从SP寄存器中得到当前内核栈的地址，然后获取到 thread\_info数据结构的指针。进而 thread\_info->task的到PCB结构。

### Memory Manager内存管理

Linux内存管理引入了虚拟地址的概念，但本质上来说，内存管理的工作是对物理内存进行组织。这两者结合起来，就出现了Linux系统关于内存管理的一个重要的功能：将虚拟地址与实际地址对应。

首先，页（page）是内核空间管理的基本单位，内核用struct page结构体表示每个物理页，struct page占用40个字节。而内存管理单元（MMU,把虚拟地址转换为物理地址的硬件设备）通常是以页为单位处理。

而区（zone）是内核将页继续进行了划分，一般Linux系统共有三个区，包括：

ZONE\_DMA(DMA使用的页，物理内存<16MB)  
ZONE\_NORMAL(可以正常寻址的页，物理内存16—896MB)  
ZONE\_HIGHMEM（动态映射的页，物理内存>896MB）

并且页也需要分配和释放，其中用到的函数有alloc\_page、alloc\_pages、\_get\_free\_pages、\_\_get\_free\_pages、\_\_get\_zeroed\_page。

Linux系统内存管理最重要的部分就是slab分配器，它有着缓存频繁分配和释放数据结构的功能。一般情况下，slab由一个或多个物理地址上连续的页组成。每当进程调用fork时，一定会创建一个新的进程描符，这是在dup\_task\_struct()进程中完成，而该函数会被do\_fork()调用.执行完毕后，如果没有子进程在等待的话，它的进程描述符就会被释放。返回给task\_struct\_cachep slab高速缓存。

（3）VFS（Virtual File System），虚拟文件系统。Linux内核将不同功能的外部设备，例如Disk设备（硬盘、磁盘、NAND Flash、Nor Flash等）、输入输出设备、显示设备等等，抽象为可以通过统一的文件操作接口（open、close、read、write等）来访问。

（4）Network，网络子系统。负责管理系统的网络设备，并实现多种多样的网络标准。

（5）IPC（Inter-Process Communication），进程间通信。IPC不管理任何的硬件，它主要负责Linux系统中进程之间的通信。

（3）（4）（5）三部分由于在课程部分还未提及，我只简略了解了这几部分的功能以及主要的函数，故在此不详细列出。

总结：Linux内核中有许多代码，上述列出的是Linux内核中最重要的几个部分，其余部分非核心代码或者编译脚本，配置文件等没有列出。

**五、操作系统引导和自启动初始化过程相关的 Linux 内核源码的研读及对应设计机理的归纳提取**

1.计算机的启动要首先从BIOS说起。在计算机通电后产生一个脉冲给CPU开始工作当cpu收到该复位脉冲后，其硬件逻辑就会置CS寄存器为FFFFH，IP寄存器为0000H，也就是说，会自动地到FFFF:0000H去取第一条指令。

并且BIOS程序在内存最开始的位置（0x00000）用1KB的内存空间（0x00000～0x003FF）构建中断向量表，在紧挨着它的位置用256字节的内存空间构建BIOS数据区（0x00400～0x004FF），并在大约57 KB以后的位置（0x0E05B）加载了8 KB左右的与中断向量表相应的若干中断服务程序。这些是第一步内容，在Linux中后续将在bootsect指挥下开始启动和加载。

2.bootsect.s研读归纳

### （1）首先BIOS在启动自减后，将0号磁头对应的盘面0磁道的1扇区加载到0x07c00。然后bootsect接管cpu，复制bootsect到 0x90000位置，并跳转到这个位置开始执行。

entry \_start  ; 告知链接程序，程序从 start 标号开始执行

\_start:

    mov ax,#BOOTSEG

    mov ds,ax      ! ds = 0x07c0

    mov ax,#INITSEG

    mov es,ax      ! es = 0x9000

    mov cx,#256    ! 搬运 256 次

    sub si,si      ! si = 0

    sub di,di      ! di = 0

                   ! ds:si = 0x07c0:0x0

                   ! es:di = 0x9000:0x0

    rep

    movw           ! 每次搬运 2 字节

    jmpi go,INITSEG  ! 跳转到 0x9000:go

### 其中：jmpi go,INITSEG 是段间跳转，INITSEG 是段地址，go 是偏移地址。

### （2）加载setup到0x90200

load\_setup:

    mov dx,#0x0000      ! 驱动器号(DL)0，磁头号(DH)0

    mov cx,#0x0002      ! 起始扇区号2, 磁道号0

    mov bx,#0x0200      ! 偏移地址0x200

    mov ax,#0x0200+SETUPLEN ! 功能号AH=0x02,AL=要读的扇区数目=SETUPLEN=4

    int 0x13            ! read it

    jnc ok\_load\_setup   ! CF = 0 则跳转

    mov dx,#0x0000      !需要复位的驱动器号=DL=0

    mov ax,#0x0000      !功能号AH=0

    int 0x13            ! 复位磁盘

j   load\_setup

1. 获得磁盘驱动器参数

    mov dl,#0x00    !驱动器号为0，说明是软盘

    mov ax,#0x0800  ! AH=8 is get drive parameters

    int 0x13

    mov ch,#0x00    !这里用不上软盘的最大磁道号，可以使CH=0

    ; seg 用于将段超越前缀中指明的段寄存器去取代指令中默认的段寄存器，它只影响其下一条语句

    ; 实际上，由于本程序代码段和数据段重叠，因此本程序中此处可以不使用该指令

    seg cs

    mov sectors,cx

    ; 以上两句也可以写为  mov cs:[sectors], cx

    ; 保存每磁道最大扇区数。对于软盘，最大磁道号不会超过256，所以CH足以表示，CL[7:6]为0

    mov ax,#INITSEG

    mov es,ax       !因为上面ES的值被修改，所以令 ES=0x9000

1. 加载 system 到 0x10000

    ! we want to load the system (at 0x10000)

    mov ax,#SYSSEG  ! SYSSEG=0x1000

    mov es,ax       ! segment of 0x010000

    call    read\_it

    call    kill\_motor

1. 确认根文件系统设备号

    seg cs

    mov ax,root\_dev     ! ax = ROOT\_DEV

    cmp ax,#0

    jne root\_defined    ! 如果 ROOT\_DEV 不等于 0 则跳转到 root\_defined

    seg cs

    mov bx,sectors       ! 取每磁道扇区数

    mov ax,#0x0208       ! /dev/ps0 - 1.2Mb

    cmp bx,#15           ! 判断每磁道扇区数是否等于15

    je  root\_defined     ! 如果等于15，说明是1.2MB的软盘

    mov ax,#0x021c       ! /dev/PS0 - 1.44Mb

    cmp bx,#18           ! 判断每磁道扇区数是否等于18

    je  root\_defined     ! 如果等于18，说明是1.44MB的软盘

undef\_root:

    jmp undef\_root       ! 死循环

root\_defined:

    seg cs

    mov root\_dev,ax      ! 将检查过的设备号保存到 root\_dev 中

1. 跳转到 setup 去执行

    jmpi    0,SETUPSEG   !到此本程序就结束了。

总结：这六条是bootsec.s这部分代码干的基本工作。这其实是一个引导扇区，主要作用就是帮助后续的加载和启动。这部分代码编译后仅有512字节，可谓是干练简洁。我在此列出的是基本功能解释，而具体的每一个函数，每一个地址的作用我并未一一列出，因为对于整体逻辑来说这部分不是重点。

#### 3. setup.s研读归纳

（1）获取参数保存在0x90000

这些参数包括保存光标位置，获取从1M处开始的扩展内存大小，获取显示模式，检查显示方式并获取参数，赋值硬盘参数表。

保存光标的位置

    mov ax,#INITSEG  ! INITSEG = 0x9000

    mov ds,ax        ! ds = 0x9000

    mov ah,#0x03     ! 功能号=3，获取光标的位置

    xor bh,bh        ! bh = 页号 = 0（输入）

    int 0x10         ! 输出： DH=行号，DL=列号

    mov [0],dx       ! 保存光标的行号和列号到 0x90000，共占2字节.

获取从 1M 处开始的扩展内存大小

    ! 利用 BIOS 中断 0x15 功能号 ah = 0x88 取系统所含扩展内存大小，并保存在内存 0x90002 处

    ! 返回：ax=从0xl00000(lM)处开始的扩展内存大小(KB).若出错则CF置位,ax=出错码

    mov ah,#0x88

    int 0x15

    mov [2],ax ! ax = 从1M处开始的扩展内存大小

获取显示模式

    ! 获取显示卡当前的显示模式

    ! 调用 BIOS 中断 0x10，功能号 ah = 0x0f

    ! 返回： ah=字符列数； al=显示模式；bh=当前显示页。

    ! 0x90004(l个字)存放当前页；0x90006(1字节)存放显示模式；0x90007(1字节)存放字符列数。

    mov ah,#0x0f

    int 0x10

    mov [4],bx      ! bh = 当前显示页

    mov [6],ax      ! al = 显示模式, ah = 字符列数（窗口宽度）

复制硬盘参数表

复制 HD0 的硬盘参数表

! 复制 hd0 的硬盘参数表，参数表地址是中断向量0x41的值，表长度16B

! 中断向量在中断向量表中的位置 = 中断类型号N × 4

! (N\*4)的字单元存放偏移地址；

! (N\*4+2)的字单元存放段基址。

    mov ax,#0x0000

    mov ds,ax      ! ds=0

! 将内存[4\*0x41]处的低2字节(偏移地址)传给si,高2字节(段地址)传给ds

    lds si,[4\*0x41]

    mov ax,#INITSEG

    mov es,ax          !es = 0x9000

    mov di,#0x0080

    mov cx,#0x10       !重复16次

! ds:si --> es:di(0x9000:0x0080),共传送16B

    rep

    movsb

复制 HD1 的硬盘参数表

! 复制 hd1 的硬盘参数表，参数表地址是中断向量0x46的值，表长度16B

! 道理同上一小节,此处不赘述

    mov ax,#0x0000

    mov ds,ax

    lds si,[4\*0x46]

    mov ax,#INITSEG ! INITSEG = 0x9000

    mov es,ax

    mov di,#0x0090

    mov cx,#0x10

! ds:si --> es:di(0x9000:0x0090),共传送16B

    rep

    movsb

检查系统是否有第2个硬盘

! 检查系统是否有第2个硬盘，如果没有就把第2个参数表清零

! 利用 BIOS 中断调用 0x13 的取盘类型功能，功能号 ah = 0xl5;

! 输入： dl=驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）

! 输出： ah=类型码；00-没有这个盘，CF 置位；

!                  01-是软驱，没有 change-line 支持；

!                  02 -是软驱(或其他可移动设备），有 change-line 支持；

!                  03 -是硬盘。

!

    mov ax,#0x01500 ! 功能号 ah=0x15，读取盘类型

    mov dl,#0x81    ! dl=驱动器号，0x81代表第2个硬盘

    int 0x13

    jc  no\_disk1    ! CF置位,表示没有这个盘

    cmp ah,#3

    je  is\_disk1    ! ah=3表示存在第2个硬盘,跳转到is\_disk1

no\_disk1:

! 清空第2个表

    mov ax,#INITSEG

    mov es,ax

    mov di,#0x0090  ! es:di = 0x9000:0x0090

    mov cx,#0x10

    mov ax,#0x00    ! AL=0

    rep

    stosb           ! Store AL at address es:di

is\_disk1:

（2）关中断

关中断

! 为进入保护模式做准备

  cli         ! no interrupts allowed !

（3）移动 SYSTEM 模块到 0X00000

bootsect.s 引导程序将 system 模块读入到 0xl0000 开始的位置。由于当时假设 system 模块最大长度不会超过 0x80000 (512KB)，即其末端不会超过内存地址 0x90000，所以 bootsect.s 会把自己移动到0x90000 开始的地方，并把 setup 加载到它的后面。下面这段程序的用途是再把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff 的内存数据块(共512KB)整块地向内存低端移动了0x10000(64KB)。

! 从代码实现来看，是一小块(0x10000B=64KB)一小块移动的，共移动8小块。

    mov ax,#0x0000

    cld             ! 'direction'=0, movs moves forward

do\_move:

    mov es,ax       ! es是目的段地址

    add ax,#0x1000

    cmp ax,#0x9000   ! 当 ax==0x9000 时结束移动

    jz  end\_move

    mov ds,ax       ! ds是源段地址，ds比es大0x1000

    sub di,di        ! di = 0

    sub si,si       ！ si = 0

    mov cx,#0x8000  ! 重复 0x8000次

    rep             ! ds:si --> es:di

    movsw           ! 每次移动2B.

    jmp do\_move    ！ 本轮一共移动 0x8000\*2B = 0x10000B=64KB. 准备下一轮移动

end\_move:

（4）设置8259

; ICW1

    mov al,#0x11        ! initialization sequence

    out #0x20,al        ! send ICW1 to Master

    .word   0x00eb,0x00eb       ! jmp $+2, jmp $+2

    out #0xA0,al        ! send ICW1 to Slave

    .word   0x00eb,0x00eb

    ;------------------------------------------------------

    ; ICW2

    mov al,#0x20        ! 送主芯片ICW2命令字，设置起始中断号，要送奇端口

    out #0x21,al

    .word   0x00eb,0x00eb

    mov al,#0x28        ! 送从芯片ICW2命令字，设置起始中断号，要送奇端口

    out #0xA1,al

    .word   0x00eb,0x00eb

    ;-------------------------------------------------------

    ; ICW3

    mov al,#0x04        ! 8259-1 is master

    out #0x21,al

    .word   0x00eb,0x00eb

    mov al,#0x02        ! 8259-2 is slave

    out #0xA1,al

    .word   0x00eb,0x00eb

    ;------------------------------------------------------

    ; ICW4

    mov al,#0x01

    out #0x21,al

    .word   0x00eb,0x00eb

    out #0xA1,al

    .word   0x00eb,0x00eb

    ;------------------------------------------------------

    ; OCW1

    mov al,#0xFF        ! mask off all interrupts for now

    out #0x21,al

    .word   0x00eb,0x00eb

    out #0xA1,al

（5）进入保护模式

首先加载机器状态字（lmsw，Load Machine Status Word），也称控制寄存器 CR0，其比特位 0 置 1 将使 CPU 切换到保护模式，并且运行在特权级0，即当前特权级 CPL = 0。此时各个段寄存器仍然指向与实地址模式中相同的线性地址处（在实地址模式下线性地址与物理地址相同）。在设置该比特位后，随后一条指令必须是一条段间跳转指令，用于刷新 CPU 当前指令队列。因为 CPU 是在执行一条指令之前就已从内存读取该指令并对其进行译码。然而在进入保护模式以后那些属于实模式的预先取得的指令信息就变得不再有效。而一条段间跳转指令就会刷新 CPU 的当前指令队列，即丢弃这些无效信息。

    mov ax,#0x0001  ! protected mode (PE) bit

    lmsw    ax      ! This is it! LMSW:Load Machine State Word 加载机器状态字，将PE位置1，开启保护模式

    jmpi    0,8     ! jmp offset 0 of segment 8 (cs)， 这里是段选择符

总结：setup.s是启动的初始阶段，包括了基础的初始化，加载基本模块和从实模式进入保护模式，是Linux启动中不可获取的一步。

#### 4.head.s研读归纳

head的加载方式是：先将head.s汇编成目标代码，将C语言编写的内核程序编译成目标代码，然后链接成system模块。所以system模块既有内核程序，也有head程序。

setup将system模块复制到0x00000位置，因为head在system模块的前面部分，所以head程序就在0x00000这个位置。head程序占有25KB+184B的空间。head程序后面就是main函数。

head的工作：用程序自身的代码在程序自身所在的内存创建内核分页机制。即在0x00000创建了页目录表、页表、缓冲区、GDT、IDT。并覆盖已执行的代码。这意味着head将自己废弃，即将执行main函数。

(1)将DS,ES,FS和GS从实模式转变为保护模式

.text

.globl \_idt,\_gdt,\_pg\_dir,\_tmp\_floppy\_area

\_pg\_dir:  #标识内核分页机制完成后的内核起始地址(物理内存的起始地址0x00000)，head在此处建立页目录表

startup\_32:

  #ds,es,fs,gs值都为0x10

  #因为处理器已经工作在保护模式下，所以这些段寄存器都表示段选择子。0x10 写成16位二进制形式为0b0000 0000 0001 0000，所以值为该数的段选择子：请求特权级为 0（RPL=00）、所指向的描述符存放在GDT（TI=0）、所指向的描述符索引为2（DI=0000 000000010）

    movl $0x10,%eax

    mov %ax,%ds

    mov %ax,%es

    mov %ax,%fs

    mov %ax,%gs

    #lss:将操作数的值传送给指定寄存器ss:esp

    #stack\_start定义在kenel/sched.c文件中

    /\*long user\_stack [ PAGE\_SIZE>>2 ] ;

        struct {

        long \* a;

        short b;

        } stack\_start = { & user\_stack [PAGE\_SIZE>>2] , 0x10 };\*/

    #将结构体stack\_start的值传送到ss:esp，令 ss=0x10（段选择子）和 esp=&user\_stack [PAGE\_SIZE>>2]

    lss \_stack\_start,%esp

    call setup\_idt #设置IDT

    call setup\_gdt

    movl $0x10,%eax     # reload all the segment registers

    mov %ax,%ds     # after changing gdt. CS was already

    mov %ax,%es     # reloaded in 'setup\_gdt'

    mov %ax,%fs

    mov %ax,%gs

    lss \_stack\_start,%esp

    xorl %eax,%eax

1:  incl %eax       # check that A20 really IS enabled

    movl %eax,0x000000  # loop forever if it isn't

    cmpl %eax,0x100000

    je 1b

(2)设置IDT

setup\_idt:

    lea ignore\_int,%edx #先让所有的中断描述符默认指向ignore\_int

    movl $0x00080000,%eax  #这里的8看成1000，这个值会在初始化IDT的时候用到

    movw %dx,%ax        /\* selector = 0x0008 = cs \*/

    movw $0x8E00,%dx    /\* interrupt gate - dpl=0, present \*/

    lea \_idt,%edi

    mov $256,%ecx

rp\_sidt:

    movl %eax,(%edi)

    movl %edx,4(%edi)

    addl $8,%edi

    dec %ecx

    jne rp\_sidt

    lidt idt\_descr

    ret

中断描述符为64位，包含0-15+48-63组合成32位的中断服务程序的段内偏移地址。16-31位为段选择符，定位中断服务程序所在段，47:段存在标志，45-46:特权级标志，40-43:段描述符类型标志

(3)设置GDT

setup\_gdt:

    lgdt gdt\_descr

    ret

.align 2

.word 0

idt\_descr:

    .word 256\*8-1       # idt contains 256 entries

    .long \_idt

.align 2

.word 0

gdt\_descr:

    .word 256\*8-1       # so does gdt (not that that's any

    .long \_gdt      # magic number, but it works for me :^)

    .align 3

\_idt:   .fill 256,8,0       # idt is uninitialized

\_gdt:   .quad 0x0000000000000000    /\* NULL descriptor \*/

    .quad 0x00c09a0000000fff    /\* 16Mb \*/

    .quad 0x00c0920000000fff    /\* 16Mb \*/

    .quad 0x0000000000000000    /\* TEMPORARY - don't use \*/

    .fill 252,8,0           /\* space for LDT's and TSS's etc \*/

GDT 共设置了 4 个项，余下的 252 个项初始化为 0

gdt.png

(4)重新设置ds,es,fs,gs,ss

因为它们所指向的原描述符所指向的段的段限长为 8MB，所以当访问 8MB 以上的地址空间时，将会产生段限长超限报警。为了防止这类可能发生的情况，在这里需要对段寄存器（段选择子）重新设置：

movl $0x10,%eax     # reload all the segment registers

    mov %ax,%ds     # after changing gdt. CS was already

    mov %ax,%es     # reloaded in 'setup\_gdt'

    mov %ax,%fs

    mov %ax,%gs

    lss \_stack\_start,%esp

(5)检查A20是否打开

A20如果没有打开，计算机处于20位的寻址模式，超过0xFFFFF寻址必然会回滚，如0x100000会回滚到0x000000，也就是说地址0x100000存储的值会和0x000000一样。

所以通过在内存0x000000位置写入数据并和0x100000处比较来检查A20是否被打开。如果一直相同的话，就一直比较下去，即死循环、死机，表示 A20 线没有选通，结果内核就不能够使用 1MB 以上内存。

    xorl %eax,%eax

1:  incl %eax       # check that A20 really IS enabled

    movl %eax,0x000000  # loop forever if it isn't

    cmpl %eax,0x100000

    je 1b

(6)检查x87协处理器是否存在

为了弥补 x86 系列在进行浮点计算时的不足，Intel 于 1980 年推出了 x87 系列数学协处理器，那时是一个外置的、可选的芯片。1989 年，Intel 发布了 486 处理器。自从 486 开始，以后的 CPU 一般都内置了协处理器。这样，对于 486 以前的计算机而言，操作系统检验 x87 协处理器是否存在就非常有必要了。

/\*

 \* NOTE! 486 should set bit 16, to check for write-protect in supervisor

 \* mode. Then it would be unnecessary with the "verify\_area()"-calls.

 \* 486 users probably want to set the NE (#5) bit also, so as to use

 \* int 16 for math errors.

 \*/

    movl %cr0,%eax      # check math chip

    andl $0x80000011,%eax   # Save PG,PE,ET

/\* "orl $0x10020,%eax" here for 486 might be good \*/

    orl $2,%eax     # set MP

    movl %eax,%cr0

    call check\_x87

    jmp after\_page\_tables

/\*

 \* We depend on ET to be correct. This checks for 287/387.

 \*/

check\_x87:

    fninit

    fstsw %ax

    cmpb $0,%al

    je 1f           /\* no coprocessor: have to set bits \*/

    movl %cr0,%eax

    xorl $6,%eax        /\* reset MP, set EM \*/

    movl %eax,%cr0

    ret

.align 2

1:  .byte 0xDB,0xE4     /\* fsetpm for 287, ignored by 387 \*/

    ret

(7)构建分页管理机制

startup\_32:

    ……

    call check\_x87

    jmp after\_page\_tables

after\_page\_tables:

    #先将main函数参数，L6标号和main函数入口地址压栈

    pushl $0        # These are the parameters to main :-)

    pushl $0

    pushl $0

    pushl $L6       # return address for main, if it decides to.

    pushl $\_main

    jmp setup\_paging

L6:

    jmp L6          # main should never return here, but

                # just in case, we know what happens.

完成后跳转到setup\_paging，开始创建分页机制

1.先将页目录表和4个页表放在物理内存的起始位置，从内存起始位置开始的 5页空间内从全部清零，每页4KB。

.align 2

setup\_paging:

    movl $1024\*5,%ecx       /\* 5 pages - pg\_dir+4 page tables \*/

    xorl %eax,%eax

    xorl %edi,%edi          /\* pg\_dir is at 0x000 \*/

    cld;rep;stosl

stosl：每次保存的是 4 个字节。

ecx 控制循环次数

每次循环将 eax 的值保存到 es:edi （es 为段选择子）指向的内存

若 EFLAGS 中的方向标志位 DF=0 （使用 cld 指令），则 edi 自增 4 （因为比较的是 Long，所以递增 4）；若 DF=1（使用 std 指令），则 edi 自减 4

rep 表示当 ecx>0 时，循环继续；反之停止

在这个程序中，每循环 1024 次，清零的内存范围是 1024\*4=4096 字节，恰好是一个页。

2.设置页目录表的前四项，使之分别指向4个页表

/\*

 \* I put the kernel page tables right after the page directory,

 \* using 4 of them to span 16 Mb of physical memory. People with

 \* more than 16MB will have to expand this.

 \*/

.org 0x1000

pg0:

.org 0x2000

pg1:

.org 0x3000

pg2:

.org 0x4000

pg3:

setup\_paging:

    movl $1024\*5,%ecx       /\* 5 pages - pg\_dir+4 page tables \*/

    xorl %eax,%eax

    xorl %edi,%edi          /\* pg\_dir is at 0x000 \*/

    cld;rep;stosl

    /\*

        7看成二进制的 111，代表页属性，u/s、r/w、present

        111 代表：用户 u、读写 rw、存在 p

        000 代表：内核 s、只读 r、不存在

    \*/

    movl $pg0+7,\_pg\_dir     /\* set present bit/user r/w \*/

    movl $pg1+7,\_pg\_dir+4       /\*  --------- " " --------- \*/

    movl $pg2+7,\_pg\_dir+8       /\*  --------- " " --------- \*/

    movl $pg3+7,\_pg\_dir+12      /\*  --------- " " --------- \*/

3.页表填充

从高地址向低地址填写4个页表

.align 2

setup\_paging:

    ……

    movl $pg3+4092,%edi  #第4个页表的最后一个页表项的起始位置

    movl $0xfff007,%eax     /\*  16Mb - 4096 + 7 (r/w user,p) \*/

    #存储到第 4 个页表的最后一个页表项的内容。这里 7 表示页属性，0xfff000 为该页表项所指向的页基址（也称为页号）。该地址刚好是16MB内存的最后一页的地址

    std

#eax（初始值为 0xfff007，7为页属性）递减0x1000 （4KB，一个页大小），

#edi（初始值为 $pg3+4092） 按 4 (std，表示 4 个字节)递减

#将 eax 内容（即页表项）存储到内存 edi 处

#这样直到循环结束，刚好能将 4 个页表填满。每个页表有 4KB/4B=1024 个页表项。4 个页表支持的寻址范围为4 \* 1024 \*4KB = 16MB，恰好是 Linux 0.11 支持的寻址范围

1:  stosl           /\* fill pages backwards - more efficient :-) \*/

    subl $0x1000,%eax

    jge 1b

page.png

4.设置CR3和CR0

CR0:选择微处理器的工作方式和存储器的管理模式。其中第31位是PG标志，是分页机制控制位。当CPU的CR0的第0位PE（保护模式）置为1时，可以设置PG位为开启。开启后，地址映射模式采取分页机制。当PE（保护模式）置为0时，设置PG位会发生异常。

CR3:页目录表基址寄存器，保存"页表目录"的起始物理地址，CR3 的高 20 位提供页表目录的基地址，低 12 位“不用”（这里的“不用”并不是指真的不用，而是从 CR3 取 32 位数据时，低 12 位全“取”为 0。当CR0的PG标志位置位时，CPU使用CR3指向的页目录表和页表进行虚拟地址到物理地址的映射。

.align 2

setup\_paging:

    ……

    xorl %eax,%eax      /\* pg\_dir is at 0x0000 \*/

    movl %eax,%cr3      /\* cr3 - page directory start 将CR3指向页目录表 0x0000就是页目录表的起始位置\*/

    movl %cr0,%eax

    orl $0x80000000,%eax

    movl %eax,%cr0      /\* set paging (PG) bit 启动PG标志置位\*/

    ret         /\* this also flushes prefetch-queue  -----去了main函数---\*/

system\_model.png

5.跳转至main函数

.align 2

setup\_paging:

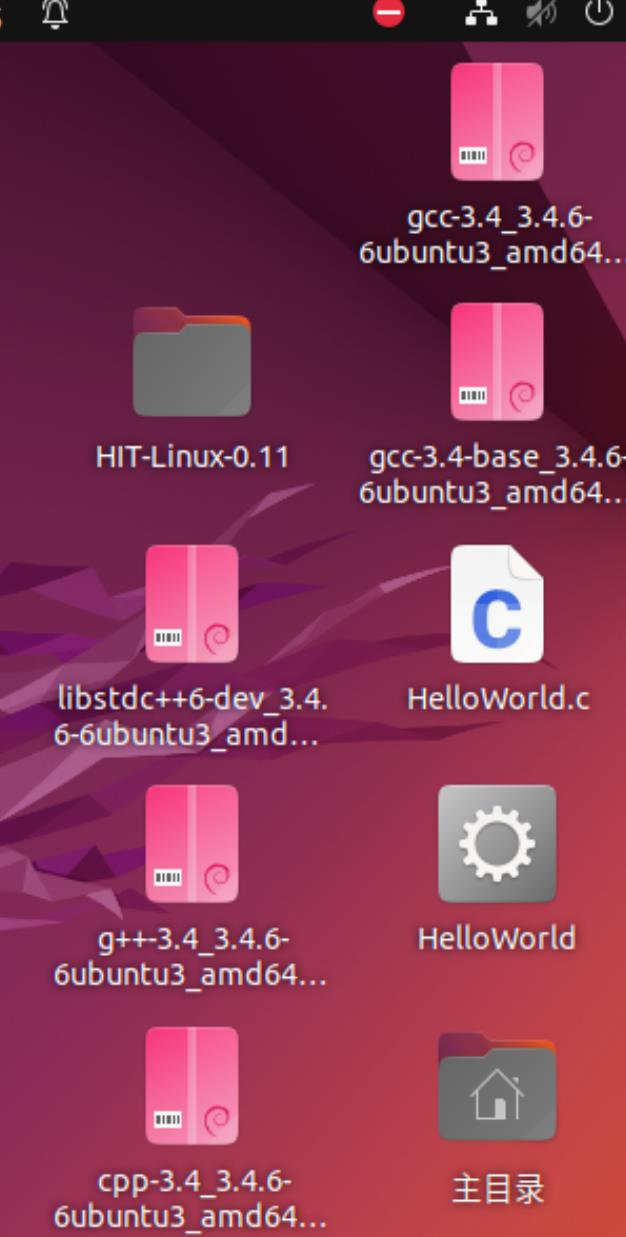
    ……

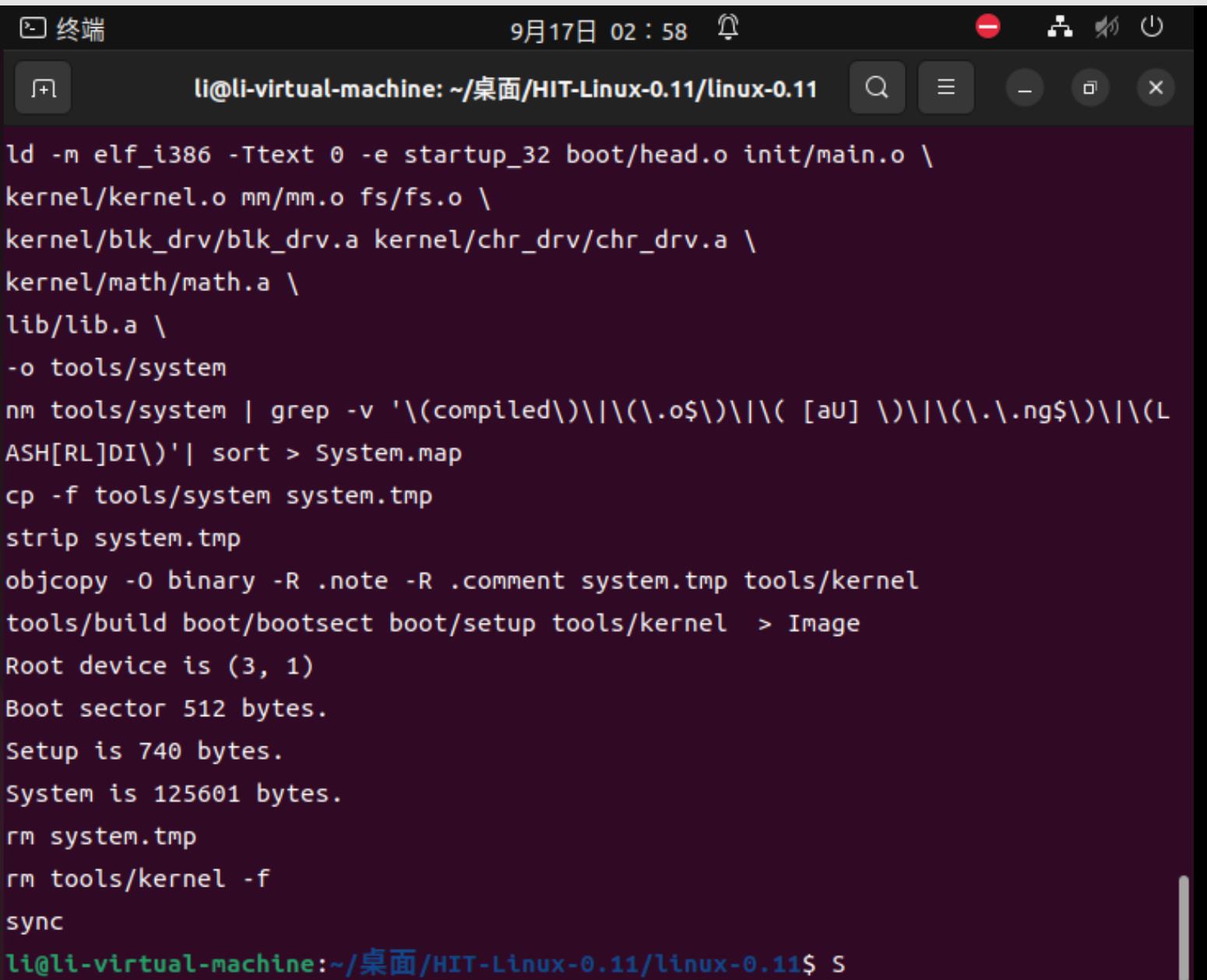
    ret         /\* this also flushes prefetch-queue  -----去了main函数---\*/

**六、Linux内核源码在虚拟机上的编译启动运行和测试验证**

1. 编译

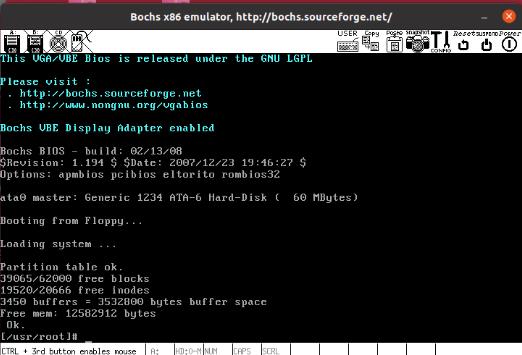
由于linux-0.11版本过于低，导致目前虚拟机上使用的并不是32位环境，首先需要建立32位环境。然后安装汇编工具，安装gcc3.4，这样就可以吧基础的环境配置完成。最后效果如图：



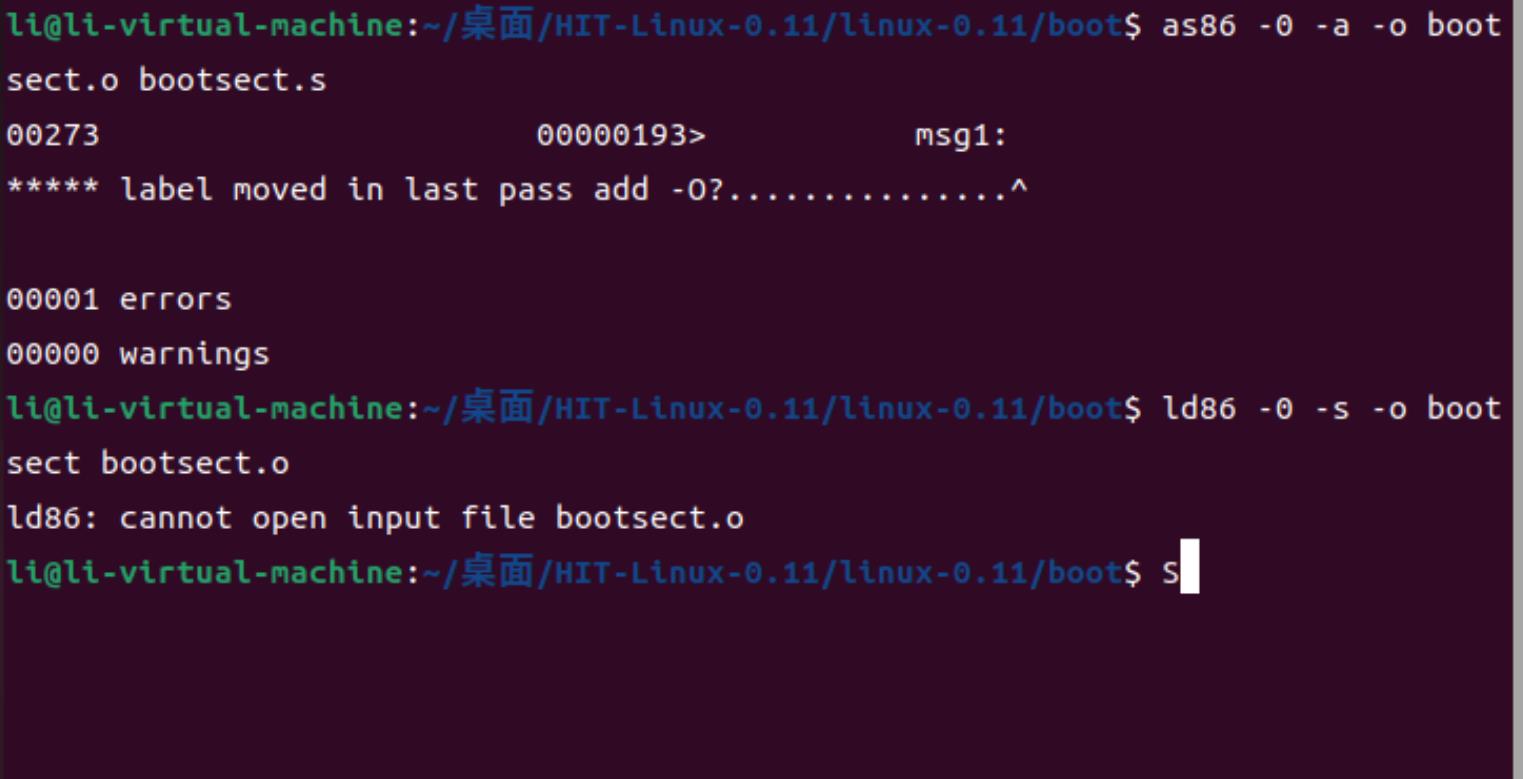
2.在linux-0.11下直接make，结果如图：

可以看出，我们已经将linux-0.11编译成功。

（3）启动，输入../run之后，弹出bochs界面如图：

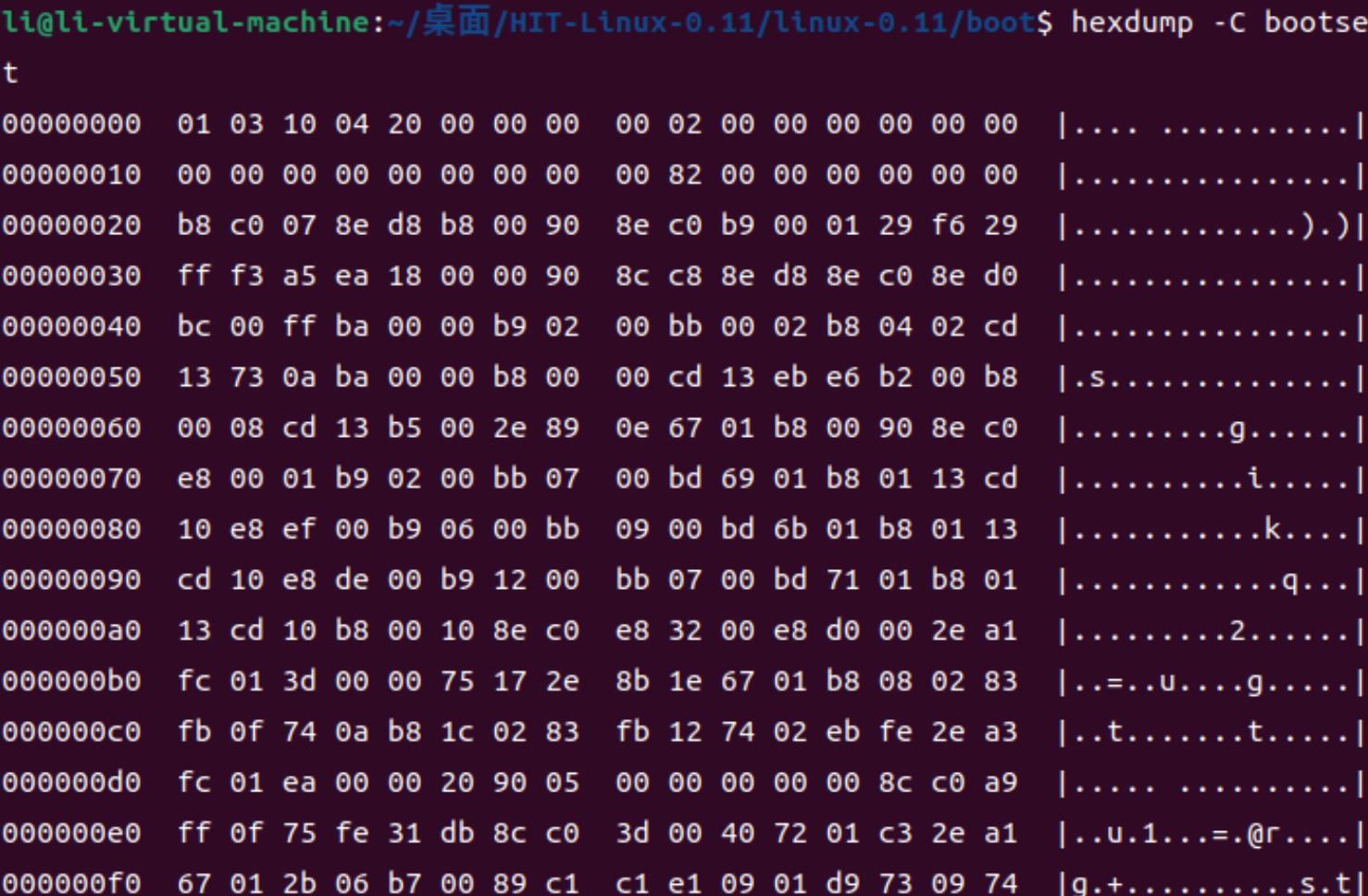


(4)测试与验证：

先编译与链接boot下的内容，如图：

再使用ls -s显示信息：

最后通过hexdump -C bootsect核对头文件信息,发现前几位确实是

01 03 10 04：

**七、总结与体会**

这次实验是探究Linux内核初始化和启动的过程。在完成这些之前需要对Linux源码进行研读，在研读的过程中，我体会到了Linux设计者构思的巧妙，更加深入的了解了linux系统底层的架构和原理。也在过程中发现了linux是如何进行我们上课所讲到的计算机启动过程中的每一步的。

在最后的编译启动测试环节，需要安装低版本的工具，这花费了我大量的时间，最后看到成功编译和启动并且测试验证支撑杆之后，我也有着极大地成就感。

这次实验我对Linux源码有了更深刻的理解，这将对我以后课程的学习打下一个良好的基础。