Linux0.01 内核 boot.s 文件注解

Fuall

在读内核源代码前，假定你已经具有了一定的汇编语言和操作系统原理等方面的基本知识。在机器加电前，操作系统映像是驻留在外部存储器（hard disk or floppy)中，因此，机器加电后必须先把操作系统映像读入内存。将操作系统读入内存的工作成为引导。操作系统被引导进内存以后，将接管机器的控制权，但在对整个系统实施管理前，还必须完成许多初始化工作（硬件初始化和软件初始化）。

32 位 Intel 处理器在机器加电或 Reset 时，处理器处于实模式。因此处理器所使用的地址和操作数的长度为 16 位；CS 的初值为 0xF000H,指令寄存器的初值 EIP=0xFFF0H。

下面是开机过程的简单描述：

1、 机器加电，处理器完成自检和初始化，设置各寄存器的初值，而后开始执行指令。
2、 处理器执行的第一条指令地址为 CS*16+IP=0xF000H*16+0xFFF0H=0xFFFF0H,距离实模式下处理器的最大可编地址 0xFFFFFH 只有 16Bytes,该地址是 ROM 中 BIOS 的入口地址，处理器从此开始执行第一条指令，因此，最先获得机器控制权的是 BIOS。
3、 BIOS 完成对整个机器系统的检测，并将有关系统配置的基本信息记录在内存的 BIOS 数据区，然后，从引导盘（floppy、hard disk 、cdrom）上将一个引导扇区，读入到内存的 0x7C00H(绝对地址 07C0:0000)处，最后转到 0x7C00H，将对机器的控制权交给引导程序。
4、 引导程序将操作系统内核读入内存，在把对机器的控制权交给操作系统内核。
5、 操作系统内核完成必要的初始化设置，创建必要的数据结构，装入并启动必要的程序后，整个机器就进入了正常的执行状态。其后，操作系统一直控制着机器，直到关机。

我们所要研究的 bootsect.S 就是实现上述将操作系统内核从引导盘上读到内存中，并将它放在内存的适当位置。Bootsect 读入的内容包括 setup 程序和一个经过压缩的内核映像。Bootsect.S 编译后的二进制代码就放在磁盘第 0 道的第一个扇区中，是 Linux 系统自己写入的。

以下是/boot/boot.S

```
|
| boot.s
|
| boot.s is loaded at 0x7c00 by the bios-startup routines, and moves itself
| out of the way to address 0x90000, and jumps there.
|
| It then loads the system at 0x10000, using BIOS interrupts. Thereafter
| it disables all interrupts, moves the system down to 0x0000, changes
| to protected mode, and calls the start of system. System then must
| RE-initialize the protected mode in it's own tables, and enable
| interrupts as needed.
```

```
|
| NOTE! currently system is at most 8*65536 bytes long. This should be no
| problem, even in the future. I want to keep it simple. This 512 kB
| kernel size should be enough - in fact more would mean we'd have to move
| not just these start-up routines, but also do something about the cache-
| memory (block IO devices). The area left over in the lower 640 kB is meant
| for these. No other memory is assumed to be "physical", ie all memory
| over 1Mb is demand-paging. All addresses under 1Mb are guaranteed to match
| their physical addresses.
|
| NOTE1 abouve is no longer valid in it's entirety. cache-memory is allocated
| above the 1Mb mark as well as below. Otherwise it is mainly correct.
|
| NOTE 2! The boot disk type must be set at compile-time, by setting
| the following equ. Having the boot-up procedure hunt for the right
| disk type is severe brain-damage.
| The loader has been made as simple as possible (had to, to get it
| in 512 bytes with the code to move to protected mode), and continuos
| read errors will result in a unbreakable loop. Reboot by hand. It
| loads pretty fast by getting whole sectors at a time whenever possible.

| 1.44Mb disks:                    /*对大内核（大于 400K）,会在磁盘上占很多磁道，在内
sectors = 18                       /*存中占多个数据段。为加快读写速度，最好是每次读
| 1.2Mb disks:                     /*一个磁道，为此要测试磁盘参数，linux 采用了枚举的
| sectors = 15                     /*方法，得到磁盘参数。且只有软盘的 sectors/track 有所
| 720kB disks:                     /*不同，所以对硬盘、光驱引导，则不需要以上参数。
| sectors = 9

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text
BOOTSEG = 0x07c0
INITSEG = 0x9000
SYSSEG    = 0x1000                 | system loaded at 0x10000 (65536).
ENDSEG  = SYSSEG + SYSSIZE    /*SYSSIZE 在 Makefile 中定义的。
entry start                        /*boot.S 所对应的二进制代码，被 BIOS 装到了物理内存
start:                             /* 0x7C00H 处。之后从此处开始执行。
/*将自己移到 0x90000H 处，而后从新位置开始执行*/
/*下面很简单，是用汇编语言实现 copy.*/
```

```
  mov ax,#BOOTSEG              /*在串操作中，一般假定源串在 DS 中，目的串在 ES 中
  mov ds,ax                    /*DS=0x07C0H
  mov ax,#INITSEG
  mov es,ax                    /*ES=0x9000H
  mov cx,#256                  /*设置循环次数 256，每次 copy 一个字，共 512 字节
  sub  si,si                   /*si 针对源串寻址，sub 将 si 清 0
  sub  di,di                   /*di 针对目的串寻址，sub 将 di 清 0
  rep                          /*si,di 方向由 EFLAGS 中第 11 位 DF 决定，在系统加电时
  movw                         /*EFLAGS=00000002H,DF=0，si,di 方向为加
  jmpi go,INITSEG              /*移动后，转到段 0x9000H 处的 go 位置执行，即下一条指令

go:   mov ax,cs                /*因为有段转移，故现在的 cs=0x9000H
  mov ds,ax                    /*数据段 ds=0x9000H
  mov es,ax                    /*附加段  es=0x9000H
  mov ss,ax                    /*堆栈段  ss=0x9000H
  mov sp,#0x400                /*因为代码段、数据段、附加段、堆栈段基址重合
                               /*且 9000:0000 至 9000:0200 的 512 字节为我们刚 copy 过来的
                               /*boot 的二进制代码，所以堆栈指针应远大于 9000：0200。
                               /*我们选择 9000:0400,即 sp=0400。


  mov ah,#0x03                 /*利用 int 10H 第 3h 号软中断读光标的位置。
  xor  bh,bh                   /*我在最后面附有详细讲解
  int   0x10

  mov cx,#24
  mov bx,#0x0007    | page 0, attribute 7 (normal)
  mov bp,#msg1
  mov ax,#0x1301    | write string, move cursor
  int   0x10                   /*利用 int 10H 第 13h 号软中断写字符串，最后有详细讲解
                               /*在萤幕上输出字串"Loading system…，眼熟吧。

| ok, we've written the message, now
| we want to load the system (at 0x10000)
```

/*下面将要 load 系统内核了*/

/*Linux 内核的大小记录在变量 SYSSIZE 中，该变量位于引导程序尾部的参数区内。引导程序装入内存后，变量 syssize 在地址 0x901f4 处。该变量占两个字节，表示范围 0~65535。内核的真正大小为 syssize*16,所以引导程序所能装入的内核的最大长度为 1MB。变量 syssize 的值是由内核创建程序预先写入的。

由于引导程序运行在实模式下，他所能使用的最大内存实际上只有 640KB (在实模式下的地址空间 00000---fffff=1M,因为从 0xa000:0000 开始是显示缓存，再往上是 BIOS 地址。 故可分配空间地址为 0000:0000--→a000:0000 = 0xa0000 = 640K)。由于一般情况下 Linux 内核

都是小内核（小于 508K）所以引导程序直接将其装入到内存中从 0x10000 处开始的位置（大内核的情况比较复杂，当研究高版本内核时是必须要了解的）。由于内核大小不超过 508KB(0x7F000),因此读入后的内核在 0x10000 到 0x8F000 之间，离引导程序（boot,0x90000）至少还有 4KB 的距离。

```
    mov ax,#SYSSEG
    mov es,ax           | segment of 0x010000     /*内核将被 copy 到 1000：0000
    call  read_it                                  /*调用  read_it  函数读入内核到内存指定地点
    call  kill_motor                               /*调用 kill_motor 关闭软驱


  | if the read went well we get current cursor position ans save it for
  | posterity.

    mov ah,#0x03 | read cursor pos                 /*熟悉吧，读显示器光标位置并存到 0x90510
    xor  bh,bh
    int   0x10           | save it in known place, con_init fetches
    mov [510],dx  | it from 0x90510.



  /*下面将要切换到保护模式，在切换前还要做一些工作*/
    cli                | no interrupts allowed !        /*在准备切换到保护模式前关掉中断


  /*移动系统到 0x0000:0000 开始的地方，是不是感到很奇怪为什么又搬移*/
    mov ax,#0x0000
    cld                                            /*置串操作的方向控制 DF=0，si,di 做加

  do_move:
    mov es,ax           | destination segment
    add  ax,#0x1000
    cmp ax,#0x9000
    jz    end_move
    mov ds,ax           | source segment
    sub  di,di
    sub  si,si
    mov       cx,#0x8000
    rep
    movsw
    j      do_move                                  /*将位于 0x1000:0000 的内核移到内存
                                                    /*0x0000:0000,覆盖了绝对地址 0x00000 里面
                                                    /*.实模式中断向量表
```

```
| then we load the segment descriptors

end_move:

  mov ax,cs             | right, forgot this at first. didn't work :-)
  mov ds,ax
  lidt   idt_48                         /* idt_48 和 gdt_48 都是一个 3 个 word 长的数据结构
  lgdt  gdt_48                          /*第一个字说明(Global || Interrupt) Descript
                                        /* Table 有多长,因为每个 Table 是四个字长,所以
                                        /*可以得出整个 DescriptorTable 的 entries.(见后面)
                                        /*后两个字指出 DT 的具体位置.
                                        /* idt_48 是 0,0,0;应表示没有中断描述符 entries.
                                        /* gdt_48 有 256 个入口,第一个是个空入口,然后
                                        /*定义了一个 code 段和一个 data 段.基址都是
                                        /*0x00000000,    *** 0x00000000 != 0x0000:0000 ***
| that was painless, now we enable A20

  call  empty_8042
  mov al,#0xD1        | command write
  out   #0x64,al
  call  empty_8042
  mov al,#0xDF        | A20 on
  out   #0x60,al
  call  empty_8042

| well, that went ok, I hope. Now we have to reprogram the interrupts :-(
| we put them right after the intel-reserved hardware interrupts, at
| int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
| messed this up with the original PC, and they haven't been able to
| rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
| which is used for the internal hardware interrupts as well. We just
| have to reprogram the 8259's, and it isn't fun.
|初始化中断处理器 8259i
|初始化顺序为:                  1.  向主 8259A 写 ICW1, 0x20
|                               2.  向第二块 8259A 写 ICW1, 0xA0
|                               3.  向主 8259A 写 ICW2, 0x21
|                               4.  向第二块 8259A 写 ICW2, 0xA1
|                               5.  如果 ICW1 指示有级联中断处理器,则初始化 Master&Slave
|                               (在下例中只有 IR2 有级联 8259A), 0x21, 0xA1
```

```
|                                   6. 向两块 8259 写 ICW4,指定工作模式.
| 输入了适当的初始化命令之后, 8259 已经准备好接收中断请求.
| 现在向他输入工作
| 命令字以规定其工作方式. 8259A 共有三个工作命令字,但下例中只用过 OCW1.
| OCW1 将所有的中断都屏蔽掉, OCW2&OCW3 也就没什么意义了.
| ** ICW stands for Initialization Command Word;
|       OCW for Operation Command Word.

  mov al,#0x11          | initialization sequence
  out   #0x20,al        | send it to 8259A-1
  .word       0x00eb,0x00eb             | jmp $+2, jmp $+2

  out   #0xA0,al        | and to 8259A-2
  .word       0x00eb,0x00eb

  mov al,#0x20          | start of hardware int's (0x20)    /*向主 8259A 写入 ICW2.
  out   #0x21,al                                            /*硬件中断入口地址 0x20, 并由 ICW1
                                                            /*得知中断向量长度 = 8 bytes.
  .word       0x00eb,0x00eb

mov   al,#0x28          | start of hardware int's 2 (0x28)
  out   #0xA1,al                                             /*第二块 8259A 的中断入口是 0x28
  .word       0x00eb,0x00eb

  mov al,#0x04          | 8259-1 is master
  out   #0x21,al                                            /* Interrupt Request 2 有级联处理
  .word       0x00eb,0x00eb

  mov al,#0x02          | 8259-2 is slave
  out   #0xA1,al                                           /*于上面对应,告诉大家我就是 IR2 对应
                                                           /*级联处理器
.word 0x00eb,0x00eb

  mov al,#0x01          | 8086 mode for both
  out   #0x21,al
  .word       0x00eb,0x00eb

  out   #0xA1,al
  .word       0x00eb,0x00eb

mov   al,#0xFF          | mask off all interrupts for now
  out   #0x21,al
  .word       0x00eb,0x00eb
```

```
  out   #0xA1,al

| well, that certainly wasn't fun :-(. Hopefully it works, and we don't
| need no steenking BIOS anyway (except for the initial loading :-).
| The BIOS-routine wants lots of unnecessary data, and it's less
| "interesting" anyway. This is how REAL programmers do it.
|
| Well, now's the time to actually move into protected mode. To make
| things as simple as possible, we do no register set-up or anything,
| we let the gnu-compiled 32-bit programs do that. We just jump to
| absolute address 0x00000, in 32-bit protected mode.

  mov ax,#0x0001     | protected mode (PE) bit            /*切换到保护模式了，累死了
                                                         /*到后面可不能这样读了。。呵呵
  lmsw       ax           | This is it!
  jmpi 0,8          | jmp offset 0 of segment 8 (cs)
|在整个初始化过程完毕后,系统 jump: jmpi       0,8  这是个长跳转    cs=8 eip=0
| cs=8 不是实模式的段，而是 gdt 表中第一 （0 开始），就是你定义的初始的两个 GDT
|中的第一项，所以，现在系统跳到绝对 0,即 head.s 的 startup.
|我简单介绍一下 head.S，在 0x1000(小内核)或（0x100000(大内核)处的内核映像有两部
|分组成。主要部分是真正的内核映像，但它已被打包压缩了，因此在使用以前需要先解
|压缩

| This routine checks that the keyboard command queue is empty
| No timeout is used - if this hangs there is something wrong with
| the machine, and we probably couldn't proceed anyway.
empty_8042:
  .word      0x00eb,0x00eb
  in     al,#0x64  | 8042 status port
  test   al,#2        | is input buffer full?
  jnz    empty_8042    | yes - loop
  ret




/*函数 read_it*/
| This routine loads the system at address 0x10000, making sure
| no 64kB boundaries are crossed. We try to load it as fast as
| possible, loading whole tracks whenever we can.
|
| in:    es - starting address segment (normally 0x1000)
|
```

```
| This routine has to be recompiled to fit another drive type,
| just change the "sectors" variable at the start of the file
| (originally 18, for a 1.44Mb drive)
|
sread: .word 1                  | sectors read of current track
head: .word 0                   | current head
track: .word 0                  | current track
read_it:
  mov ax,es                              /*当前 es=0x1000
  test ax,#0x0fff                        /*必需确保 ES 处在 64KB 段边界上,即 0x?000:XXXX.
                                         /*要不你就会收到一个"DMA..."什么什么的 ERR.
die:    jne die                          /*jne：不等于转移
  xor bx,bx         | bx is starting address within segment        /*对 bx 清 0


rp_read:                                 /*循环入口
  mov ax,es
  cmp ax,#ENDSEG                        /*判断是否已经装入完全
  jb ok1_read                            /*没有完全装入转移到 ok1_read
  ret


ok1_read:
  mov ax,#sectors                        /*1.44M 软盘, sectors=18,linux 的后续版本
                                         /*中已改成由操作系统来探测 sectors 的值
  sub ax,sread                           /*AX 内记载需要读的扇区数,初始 sread 为 1
                                         /*即跳过第一道的第一扇区(BOOT 区)
  mov cx,ax
  shl cx,#9                             /*CX 算出需要读出的扇区的字节数, ax*512.
  add cx,bx                              /* BX 是当前段内偏移.
                                        /*下面连续的两个转移指令开始还真让人莫名其妙
  jnc ok2_read                           /*这里先检查当前段内的空间够不够装 ax 个扇区
                                        /*cx 算出字节数,加上当前偏移试试,够了的话,就
                                        /*跳到 ok2_read 去读吧!
  je ok2_read                            /*这么巧的事也有,刚刚够！读!
                                        /*如果到了这里就确认溢出了,看下面的
  xor ax,ax                             /*这的代码很精妙，见识到 linux 内核的巧妙了吗?
  sub ax,bx                             /*它主要目的就是如果当前段内空间不够的话
  shr ax,#9                             /*那么反算出剩余空间最多能装多少个扇区
                                        /*就读出多少个.(Hint,段内空间是扇区的整数倍)


ok2_read:
  call read_track                       /*读取当前磁道.
```

```
  mov cx,ax                    /*别忙,这里暂时不关 cx 什么事
  add ax,sread                 /* AX 是这次读出的扇区数, sread 是该磁道已
                               /*读出的扇区,相加更新 AX 的值.
  cmp ax,#sectors              /*判断该磁道所有的扇区都读出了吗
  jne ok3_read                 /*尚未读完,还不能移到下个磁道。转到 ok3_read
  mov ax,#1
  sub ax,head                  /* head 对应软盘来说只能是 0,1
  jne ok4_read                 /*0,1 head 都读过了才准往下走!
  inc track                    /*终于可以读下个磁道了,真累!

ok4_read:
  mov head,ax                /*head 置一
  xor ax,ax

ok3_read:
  mov sread,ax                  /*如果是由于还没读完所有的磁道
                                /*那么 ax 记载当前磁道已读出的扇区,更新 sread
                               /*如果已读完 18 个扇区,ax 被上一行代码置零
  shl cx,#9                     /* cx 记载最近一次读的扇区数,*512 算成字节
  add bx,cx                    /* bx 是缓冲区的偏移.往前移
  jnc rp_read                   /*看看当前段(64K)是不是已经装满了?
                                /*这里是不会超出当前段的,(见上的代码)
                                /*最多也就是刚刚装满. :-)
  mov ax,es                     /*装满了!移到下一段!!!
  add ax,#0x1000
  mov es,ax

  xor bx,bx                     /*偏移置零!
  jmp rp_read


/*函数 read_track 读取当前磁道*/
read_track:
  push ax
  push bx
  push cx
  push dx
  mov dx,track
  mov cx,sread
  inc cx                       /*CL 的低位 0-5 指示扇区号
  mov ch,dl                    /*磁道号(0-1023)由 10 位 bits 组成:CH 8 位,CL 为 2 位
  mov dx,head
  mov dh,dl                     /* DH=磁头号
  mov dl,#0                     /* DL=驱动器号,0 代表 A:, 0x80 代表 C
```

```
  and dx,#0x0100                    /*为了确保万无一失，在检验一次
  mov ah,#2                         /* AL 的值,即扇区数为当前剩余未读的 sectors
  int 0x13
  jc bad_rt                         /*出错处理,见下
  pop dx
  pop cx
  pop bx
  pop ax
  ret


bad_rt:     mov ax,#0               /*int 13h 的磁盘复位功能
  mov dx,#0                         /*（DL）=0,1 软盘（A,B）or  80,81 硬盘
  int 0x13                          /*软盘复位
  pop dx
  pop cx
  pop bx
  pop ax
  jmp read_track



/*
  * This procedure turns off the floppy drive motor, so
  * that we enter the kernel in a known state, and
  * don't have to worry about it later.
  */
kill_motor:                                 /*关闭马达
  push dx
  mov dx,#0x3f2
  mov al,#0
  outb                                      /*  向 Floppy Controller 端口写零,STOP!
  pop dx
  ret


gdt:
  .word     0,0,0,0         | dummy

  .word     0x07FF          | 8Mb - limit=2047 (2048*4096=8Mb)
  .word     0x0000          | base address=0
  .word     0x9A00          | code read/exec
  .word     0x00C0          | granularity=4096, 386
```

```
  .word   0x07FF        | 8Mb - limit=2047 (2048*4096=8Mb)
  .word   0x0000        | base address=0
  .word   0x9200        | data read/write
  .word   0x00C0        | granularity=4096, 386

idt_48:
  .word   0             | idt limit=0
  .word   0,0           | idt base=0L

gdt_48:
  .word   0x800         | gdt limit=2048, 256 GDT entries
  .word   gdt,0x9       | gdt base = 0X9xxxx

msg1:
  .byte 13,10                   /*13 回车的 ascii 码，10 换行的 ascii 码
  .ascii "Loading system ..."
  .byte 13,10,13,10

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

附录：int 10H 软中断

功　能	入口参数	出口参数
读当前光标位置	（AH）=3 (BH) =页号（图形模式为 0）	(DH)=行号，(DL)=列号 (CX)=当前光标大小
写字符串	(AH)=19（0x13H） (ES:BP)=指向字符串 (CX)=字符串的长度 (DX)=起始的光标位置 (BH)=页号 (AL)=0,(BL)=属性 （字符，字符，。。。），光标不移动 (AL)=1,(BL)=属性 （字符，字符，。。。），光标移动 (AL)=2,(BL)=属性 （字符，字符，。。。），光标移动 (AL)=3,(BL)=属性 （字符，字符，。。。），光标移动	无

Int 13H

功能	入口参数	出口参数
(AH)=2 读指定扇区	（DL）=驱动器号（0—3） (DH)=面号（0—1） （CH）=道号（0—39） （CL）=扇区号（1—9） （AL）=扇区数（1—8） （ES,BX）=欲读/写 数据的地址	（AH）=磁盘状态 CF=0(成功) or 1(出错) （AL）=读出的扇区数