3.1 Introduction

This chapter mainly describes the three assembly code files in boot/ directory listed in List 3-1. As mentioned in previous chapter, although these three files all use assembly language, they use two different syntaxes. bootsect.s was written in 086 assembly language something like Intel assembly language format and will be compiled & linked using as86 and ld86. The other two files use GNU-format assembler and its syntax is like AT&T syntax.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Size</th>
<th>Last Modified Time(GMT)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootsect.s</td>
<td>5052 bytes</td>
<td>1991-12-05 22:47:58</td>
<td></td>
</tr>
<tr>
<td>head.s</td>
<td>5938 bytes</td>
<td>1991-11-18 15:05:09</td>
<td></td>
</tr>
<tr>
<td>setup.s</td>
<td>5364 bytes</td>
<td>1991-12-05 22:48:10</td>
<td></td>
</tr>
</tbody>
</table>

In order to read these code, you should have some knowledge of 8086 assembly language and know the architecture of 80x86 and how to program in the x86 protected mode. That is to say, before start reading the code, you had better first glance the knowledge mentioned. When reading, we will detailed some of it in comments.

3.2 Main Functions

Let’s first describe the main executing flowchart of Linux booting. When your PC powers on, x86 CPU will automatically enter into real mode and start running the code at address 0xffff0. This address is usually in the ROM-BIOS. The PC’s BIOS will do some detection of the machine and start initialize the interrupt vectors at physical address 0. It, then, reads the first sector from a bootable device into memory at absolute address 0x7c00 and jump to this place. Generally, the booting device is floppy drive or harddisk drive.

The foremost part of Linux (boot/bootsect.s) is written in 8086 assembly language and stored in the first sector of booting device. It will be loaded at absolute address 0x7c00 (31Kb) in memory by BIOS and moves to physical address 0x90000 (576Kb) by itself when executing. Then it reads 2Kb code that followed (boot/setup.s) into memory at address 0x902000 and the other parts of the kernel (system module) to address 0x10000 (64Kb). Because the old Linux kernel is no more than 0x80000 (512Kb) in size, it will never lay over bootsect and setup module begin at 0x90000. The setup program, afterwards, will move the system module to the beginning of memory. Thus, the code address of system module is equal to the real physical address. This way, it is easy to operate for the kernel code and data. Figure 3-1 clearly shows the dynamic locations
of programs or modules in memory when booting. In the figure, virtual frames represent maps of each program in memory at a time. At the loading period, the kernel will display “Loading...” and transport control to the code in boot/setup.s. This is another real mode program.

3.3 bootsect.s

3.3.1 Function description

bootsect.s is disk booting code resided at the first sector of booting diskette (booting sector, 0 cylinder, 0 head, first sector). After PC power on and ROM BIOS detection, the booting
sector was loaded at address 0x7c00 in memory. And then be moved to the address 0x90000 by itself. The main purpose of this program is first load the setup module (compiled from setup.s) from disk to the location 0x90200, just after bootsect code. Then it obtains the parameters from disk parameter table for the current booting disk by using BIOS interrupt 0x13 and displays the message “Loading system...” on screen. Afterwards, it loads the system module from disk to the starting address 0x10000 in memory. Next, it will identify the device ID of root filesystem. If not specified, it will recognize the type of disk by the sector number per track and store the device number to root_dev. At last, it jump to the beginning of setup program.

3.3.2 Code Comments

Program 3-1linux/boot/bootsect.s

```assembly
.globl begtext, begdata, begbss, endtext, enddata, endbss  ! 定义了 6 个全局标识符;
.text                         ! Code Segment.
begtext:
.data                         ! Data Segment.
begdata:
.bss                          ! Uninitialized data Segment (Block Started by Symbol).
begbss:
.text                         ! Code Segment begins.

SETUPLEN = 4                   ! nr of setup-sectors
BOOTSEG  = 0x07c0              ! original address of boot-sector. Seg address.
```

! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
! versions of linux
! The size of system module after compiled. Ref. to the Program 2-1, L92. Here a maximen
! size is given.
SYSSIZE = 0x3000    
! bootsect.s              (C) 1991 Linus Torvalds
! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
! iself out of the way to address 0x90000, and jumps there.
! It then loads 'setup' directly after itself (0x90200), and the system
! at 0x10000, using BIOS interrupts.
! 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, especially as this doesn't contain the
! buffer cache as in minix
! The loader has been made as simple as possible, 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.
3.3 bootsect.s

36 INITSEG = 0x9000             ! we move boot here - out of the way
37 SETUPSEG = 0x9020            ! setup starts here
38 SYSSEG = 0x1000              ! system loaded at 0x10000 (65536).
39 ENDSEG = SYSSEG + SYSSIZE    ! where to stop loading
40
41 ! ROOT_DEV:     0x000 - same type of floppy as boot.
42 !       0x301 - first partition on first drive etc
43 ROOT_DEV = 0x306             ! Specifiy device occupied by root filesystem: Second hd, 1st partition.
        ! This is hd naming method used by old Linux. The detail values is:
        ! dev_no = (major<<8) + minor
        ! (Main devNo: 1-mem;2-floppy;3-hd;4-ttyx;5-tty;6-paral;7-unnamed pipe)
        ! 0x300 - /dev/hd0 - represents the first whole hd;
        ! 0x301 - /dev/hd1 - 1st hd, 1st partition;
        ! 0x304 - /dev/hd4 - 1st hd, 4th partition;
        ! 0x305 - /dev/hd5 - represents 2ed whole hd;
        ! 0x306 - /dev/hd6 - 2ed hd, 1st partition;
        ! 0x309 - /dev/hd9 - 2ed hd, 4th partition.
        ! The naming method was changed after Linux kernel 0.95.
44
45 entry start             ! Tail the linker, the program starts here.
46 start:                   ! L47-56 are used to move itself(bootsect) from current location
        ! 0x07c00(31k) to 0x90000(576Kb), 256 words totally. Then jump
        ! to the label 'go' to continue executing program.
47         mov     ax,#BOOTSEG   ! Set seg ds = 0x7c0;
48         mov     ds,ax
49         mov     ax,#INITSEG   ! Set seg es = 0x9000;
50         mov     es,ax
51         mov     cx,#256     ! Move conuter = 256 words;
52         sub     si,si      ! Source addr. ds:si = 0x07C0:0x0000
53         sub     di,di      ! Destination addr. es:di = 0x9000:0x0000
54         rep     movw       ! Move one word
55         jmpi    go,INITSEG ! Indirect jumping. INITSEG is destination seg addr.
        ! From here on, CPU start executes code at 0x9000 seg.
56         go:                  ! Set ds, es, ss to new seg value (0x9000). As there are
57         mov     ax,cs       ! stack operating (push,pop,call), stack must be setup.
58         mov     ds,ax
59         mov     es,ax
60         mov stack at 0x9ff00. ! sp point to 0x9ff00 (0x9000:0xff00)
61         mov     ss,ax
62         mov     sp,#0xFF00  ! arbitrary value >>512
        ! Because the code has been moved, we must set the location
        ! of stack again. sp can be any arbitrary value >>512 (0x90200).
        ! As the starting place 0x902000 will lay setup code and
        ! it occupies about 4 sectors. So the stack pointer sp must
        ! great than (0x200 + 0x200 * 4 + stack size)
63
64 ! load the setup-sectors directly after the bootblock.
65 ! Note that 'es' is already set up.
66
67 load_setup:             ! L68-77 are used to load setup code from the 2ed sector on disk to the location 0x90200
3.3 bootsect.s

! in memory, 4 sectors totally. If read encounter error, reset drive and try again. No return.
! INT 0x13 using method:
! Read sector:
! ah = 0x02 - read sector to mem; al = nr of sectors;
! ch = low 8 bits of cyl nr; cl = start sector(0-5 bits), high 2 bits of cyl(6-7);
! dh = head no.; dl = drive no. (bit 7 must be set if devie is hd)
! es:bx -->points to data buf. If error encountered, set CF flag.

68    mov     dx,#0x0000          ! drive 0, head 0
69    mov     cx,#0x0002          ! sector 2, track 0
70    mov     bx,#0x0200          ! address = 512, in INITSEG
71    mov     ax,#0x0200+SETUPLEN   ! service 2, nr of sectors
72    int     0x13                ! read it
73    jnc     ok_load_setup      ! ok - continue
74    mov     dx,#0x0000
75    mov     ax,#0x0000          ! reset the diskette
76    int     0x13
77    j       load_setup
78
79    ok_load_setup:
80
81    ! Get disk drive parameters, specifically nr of sectors/track
82    ! The following is the format of INT 0x13 for obtaining drive parameters:
83    ! ah = 0x08   dl = drive no. (bit 7 must be set if devie is hd)
84    ! Return info:
85    ! If error encountered, set CF flag and ah = status code.
86    ! ah = 0, al = 0, bl = drive type (AT/PS2)
87    ! ch = low 8 bits of max track nr.
88    ! cl = max sector nr/track(bit0-5), high 2bits of max track nr.(bit6-7)
89    ! dh = max head nr.    dl = nr of drives
90    ! es:di --> disk parameter table.
91
92    mov     dl,#0x00
93    mov     ax,#0x0800          ! AH=8 is get drive parameters
94    int     0x13
95    mov     ch,#0x00
96    seg cs                    ! The operand of next instruction is in cs seg.
97    mov     sectors,cx        ! Save sector nr per track
98    mov     ax,#INITSEG       ! The es was changed when call int 0x13, here just re-set it
99
100   ! Print some inane message ! Display message('Loading system ...'cr,lf. Total 24 chars)
101    mov     ah,#0x03           ! read cursor pos
102    xor     bh,bh
103    int     0x10
104    mov     cx,#24            ! 24 chars totally.
105    mov     bx,#0x0007         ! page 0, attribute 7 (normal)
106    mov     bp,#msg1          ! point to string.
107    mov     ax,#0x1301        ! write string, move cursor
108    int     0x10
109
110    ! ok, we've written the message, now
! we want to load the system (at 0x10000)

mov     ax,#SYSSEG
mov     es,ax           ! segment of 0x010000   ! es = seg of system.
call    read_it         ! Read system module from disk. es is input parameter.
call    kill_motor      ! shut down drive motor, then we know the stats of drive.

! After that we check which root-device to use. If the device is
defined (!= 0), nothing is done and the given device is used.
Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
on the number of sectors that the BIOS reports currently.

! The meaning of two device file above is:
! The main device nr of floppy is 2, sub nr equals to TYPE*4 + NR. where NR is 0-3 coresponding
to floppy drive A, B, C or D separately. TYPE is drive type (2-->1.2Mb or 7-->1.44Mb, etc.)
As 7*4 + 0 = 28, so /dev/PS0 (2,28) indicates the first drive with 1.44Mb type. Its device
number is 0x021c. In the same way, /dev/at0 (2,8) indicates that the first drive is a 1.2M
drive and device number is 0x0208.

seg cs
mov     ax,root_dev
cmp     ax,#0
jne     root_defined
seg cs
! Load the sectors/track (SPT) saved at L88 into register. If SPT = 15, it indicates that
! the drive type is a 1.2Mb. If SPT = 18, then it means the drive type is 1.44Mb. The drive
! is the first drive of the machine because it is bootable.

mov     bx,sectors
mov     ax,#0x0208         ! /dev/ps0 - 1.2Mb
cmp     bx,#15
je      root_defined
mov     ax,#0x021c         ! /dev/PS0 - 1.44Mb
cmp     bx,#18
je      root_defined

undef_root:                        ! If we can’t identify the drive type, we are dead!
jmp undef_root
root_defined:
seg cs
mov     root_dev,ax        ! Save the checked root device.

! after that (everything loaded), we jump to
! the setup-routine loaded directly after
! the bootblock:
jmp     0,SETUPSEG        ! Jump to 0x9020:0000 (setup.s)
!!!! This ends of the program !!!!

! Here are two subroutines

! 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)
sread: .word 1+SETUPLEN        ! sectors read of current track. we have already loaded
! 5 sectors. The bootsect occupied 1 sector, setup has 4.
head: .word 0                 ! current head
track: .word 0                 ! current track

read_it:
! Check the segment value. Data loaded from floppy must be in 64Kb aligment in memory.
! The bx register is used to point at the writing location within current segment

mov ax,es
mov ax, #0x0fff
jne die                 ! es must be at 64kB boundary
xor bx, bx               ! bx is starting address within segment

rp_read:
! Check to see if we have read in all data. Compare the current segment with #ENDSEG. Jump to
! label ok1_read if not equal, else return.

mov ax, es
cmp ax, #ENDSEG          ! have we loaded all yet? ! 是否已经加载了全部数据？
jb ok1_read
ret

ok1_read:
! Calculate & verify sector number we should get from current floppy track.
! Check the total bytes read with 64Kb segment limitation to see if we can read the sectors
! in current track all together. If exceeded, then do a reverse calculate base on the maximum
! bytes we could read at this time (64Kb - offset in segment).

seg cs
mov ax, sectors          ! Get sector number per track.
sub ax, sread            ! minus sectors we already read.
mov cx, ax               ! cx = ax = sectors not yet read in current track.
shl cx, #9               ! cx = cx * 512 bytes.
add cx, bx               ! cx = cx + offset in current segment (bx)
                      ! = total bytes read after the processing of this time.
jnc ok2_read            ! Check 64Kb limit. If not exceed, then jump to ok2_read.
je ok2_read
xor ax, ax               ! If so, then counting the maximum bytes we can read at this
sub ax, bx               ! time (64Kb - offset in segment) and convert it to sector
shr ax, #9               ! number.

ok2_read:
    call read_track
    mov cx, ax               ! cx = sector number we have read.
    add ax, sread            ! sector number we have read in current track.
    seg cs
    cmp ax, sectors          ! If there are unread sectors in current track, then jump
    jne ok3_read            ! to label ok3_read.

! Read data under next disk head (head 1). If finished, then goto read the next track.

mov ax, #1
sub ax, head             ! Check the current head.
jne ok4_read            ! If is head 0, then get data under head 1
inc track               ! else goto next track.

ok4_read:
    mov head, ax             ! Save the current head
    xor ax, ax               ! Clear sectors number already read within current track.

ok3_read:
3.3 bootsect.s

188 mov sread,ax          ! Store the sector nr already read within current track.
189 shl cx,#9              ! Multiply 512 bytes
190 add bx,cx              ! Modify the start point in current segment.
191 jnc rp_read            ! If less than 64KB limit, then jump to rp_read (L156).
192 mov ax,es              ! otherwise modify the current segment to read next segment.
193 add ax,#0x1000          ! Modify segment to point at the start location of next 64KB.
194 mov es,ax
195 xor bx,bx              ! Clear the offset value in the segment.
196 jmp rp_read            ! Jump to rp_read (L156), continue reading.
197
198 ! Read data in current track to the buffer pointed at by es:bx.
199     ! al – sector numbers; es:bx – buffer location
sread:
200     push ax
201     push bx
202     push cx
203     push dx
204     mov dx,track        ! Get current track.
205     mov cx,sread        ! Get sector number already read in current track.
206     inc cx              ! cl = start sector.
207     mov ch,dl           ! ch = current track.
208     mov dx,head         !
209     mov dh,dl           ! dh = current head.
210     mov dl,#0           ! dl = drive number (0 indicates first drive).
211     and dx,#0x0100       ! Disk head number will no more than 1.
212     mov ah,#2           ! ah = 2, read disk sector.
213     int 0x13            !
214     jc bad_rt          ! If error, jump to bad_rt.
215     pop dx
216     pop cx
217     pop bx
218     pop ax
219     ret

220 ! Execute drive reset operation (function 0 of int 0x13), then jump to read_track to try again.
bad_rt: mov ax,#0
221     mov dx,#0
222     int 0x13
223     pop dx
224     pop cx
225     pop bx
226     jmp read_track
227
228 /*
229 * This procedure turns off the floppy drive motor, so
230 * that we enter the kernel in a known state, and
231 * don’t have to worry about it later.
232 */
233 kill_motor:
234     push dx
235     mov dx,#0x3f2         ! Drive port of floppy adapter, write only.
236     mov al,#0            ! First drive, close FDC, disable DMA and int, close moter.
237     outb                 ! output al to port dx.
3.4 setup.s

3.4.1 Function

The main function of setup program is to obtain the machine parameters by using ROM BIOS interrupts and store them at begin address 0x90000 in memory (thus overlay the bootsect program), see Table 3-1. These data will be used by many routines in kernel, especially by the terminal programs, e.g. ttyio.c.

<table>
<thead>
<tr>
<th>Address</th>
<th>Size (bytes)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x90000</td>
<td>2</td>
<td>Cursor Location</td>
<td>Column (0x00–left most), Row (0x00 – top most)</td>
</tr>
<tr>
<td>0x90002</td>
<td>2</td>
<td>Extended Memory</td>
<td>The size of extended memory begin from address 1MB (in Kb).</td>
</tr>
<tr>
<td>0x90004</td>
<td>2</td>
<td>Display Page</td>
<td>Current display page.</td>
</tr>
</tbody>
</table>

3.3.3 Information

There are numerous materials about the description of bootsect.s on the Internet. A paper written by Alessandro Rubini (rubini@linux.it) describe the steps to boot the Linux kernel in a detailed way.

bootsect.s program is executed in 386 real mode, so its relatively easy to understand. If you still meet some issues here, then I suggest you to review the 80x86 assembly language and 80x86 computer architectures first.

3.4 setup.s
Then, this program moves the system module from address 0x10000-0x8ffff (Linus thought the kernel size will never exceed 512Kb at that time) down to the absolute physical address 0x00000. Next, the interrupt descriptor table register (IDTR) and global descriptor table register (GDTR) are loaded, A20 address line is opened, the two programmable interrupt controller (PIC) chips, Intel 8259A, is reconfigured. The hardware interrupt identifier number is allocated within 0x20 - 0x2f. The last step is to enter into 80x86 protected mode by setting the CPU control register CR0 (also called Machine Status Word - MSW) and passed the executing to the head.s program at the start of system module.

In order to run head.s in 32 bits protected mode, this program setups a temporary interrupt descriptor table (IDT) and global descriptor table (GDT) at the end of the program. In the GDT, kernel code segment descriptor and data segment descriptor are configured. These two kernel code descriptors will be re-configured in head.s program.

As the protected running mode is vital for understanding the head.s program, now, let's first have a simple introduction to it by comparing the different addressing methods between real-address mode and 32 bits protected mode. The follow chapters will continue describing the mechanism of it.

In real-address mode, memory addressing use a pair of segment and offset values. The segment value (or called segment selector) is stored in segment register like ds, es. Each segment has fixed size in 64Kb. The offset in the segment is store in any register (or in memory) that could be used for addressing. Thus, we can identify the location in memory by using the value pair in segment and offset register: the 16-bit value in a segment selector is shifted left by four bits to form the base address of a segment. The effective address is extended with four high order zeros and added to the base to form a linear address as shown in Figure 3-2 (a). In 80x86 protected mode, the segment register is no longer contains the base address of a addressing segment, but a segment selector that indexes a segment descriptor in a segment descriptor table (e.g. GDT). The indexed segment descriptor provides the processor with the data it needs to map a logical address into a linear address. The base field in a descriptor defines the segment base address, that is the location of the segment within the 4 gigabyte linear address space. Unlike in real-address mode, the size of the segment can be changed through
the limit field of the descriptor, which defines the size of the segment. The addressing location in memory, thus, indicated by the base address of a descriptor that is indexed by segment selector in the segment register and the offset in a register as shown in Figure 3-2 (b). There are many other fields in a descriptor that describe the property of the segment.

By comparing with the real-address mode, we know that in protected mode, the value of a segment register is changed to a descriptor selector, but the offset has the same meaning in both modes. Therefore addressing in protected mode need a additional step to select the descriptor in a descriptor table.

![Figure 3-2 Addressing methods comparing between real-address mode and protected mode.](image)

Thereby, before entering protected mode, we must first setup the descriptor tables like GDT and then using the instruction lgdt to tell processor the base address of the GDT table. The base address of GDT is stored in register GDTR. Finally, set the protected mode flag in machine status word to enter into protected mode.

The format of a segment selector is shown in Figure 3-3 and the general format of segment descriptor is shown in Figure 3-4 for quick reference. The format of interrupt gate descriptor is shown in Figure 3-5

![Figure 3-3 The format of segment selector](image)
3.4 setup.s

3.4.2 Code Comments

Program 3-2 linux/boot/setup.s

```
1 !
2 !     setup.s         (C) 1991 Linus Torvalds
3 !
4 !     setup.s is responsible for getting the system data from the BIOS,
5 !     and putting them into the appropriate places in system memory.
6 !     both setup.s and system has been loaded by the bootblock.
7 !
8 !     This code asks the bios for memory/disk/other parameters, and
9 !     puts them in a “safe” place: 0x90000-0x901FF, ie where the
10 !    boot-block used to be. It is then up to the protected mode
11 !    system to read them from there before the area is overwritten
12 !    for buffer-blocks.
13 !
14 !     NOTE! These had better be the same as in bootsect.s!
15 !
16 INITSEG = 0x9000    ! we move boot here - out of the way
```
3.4 setup.s

18 SYSSEG = 0x1000 ! system loaded at 0x10000 (65536).
19 SETUPSEG = 0x9020 ! this is the current segment

21 .globl begtext, begdata, begbss, endtext, enddata, endbss
22 .text
23 begtext:
24 .data
25 begdata:
26 .bss
27 begbss:
28 .text
29
30 entry start
31 start:

32 ! ok, the read went well so we get current cursor position and save it for
33 ! posterity.

34 ! Use BIOS int 0x10 function 0x03 to read cursor position and save it to address 0x90000.
35 ! input: bh = page number;
36 ! return: ch = start scan line; cl = end scan line;
37 ! dh = row number (0x00 is top); dl = col number ( 0x00 is left).
38 mov ax,#INITSEG ! this is done in bootsect already, but...
39 mov ds,ax
40 mov ah,#0x03        ! read cursor pos
41 xor bh,bh
42 int 0x10            ! save it in known place, con_init fetches
43 mov [0],dx

44 ! Get memory size (extended mem, kB)
45 ! Use BIOS int 0x15 function 0x88 to get extended memory size and save the word to 0x90002.
46 ! Return: CF clear if successful
47 ! AX = number of contiguous KB starting at absolute address 0x10000(1Mb)
48 mov ah,#0x88
49 int 0x15
50 mov [2],ax

51 ! Get video-card data:
52 ! Get current video mode by using BIOS int 0x10 function 0x0f and store it.
53 ! Return: Ah = number of char columns; al = display mode; bh = active page
54 ! Active page number is stored at 0x90004 (one word), display mode is store at 0x90006,
55 ! char columns is stored at 0x90007.
56 mov ah,#0x0f
57 int 0x10
58 mov [4],bx ! bh = display page
59 mov [6],ax ! al = video mode, ah = window width

60 ! check for EGA/VGA and some config parameters
61 ! Use BIOS int 0x10 alternate function: ah = 0x12, bl = 0x10 to get EGA/VGA info.
62 ! Return:
63 ! bh = video state
! 0x00 - color mode in effect, I/O port 0x3dx
! 0x01 - mono mode in effect, I/O port 0x3bx
! bl = installed memory
! 0x00 - 64Kb; 0x01 - 128Kb; 0x02 - 192Kb; 0x03 - 256Kb
! ch = feature connector bits
! cl = switch settings (see the description at the end of the program)

57
58     mov     ah,#0x12
59     mov     bl,#0x10
60     int     0x10
61     mov     [8],ax    ! 0x90008 = ??
62     mov     [10],bx   ! 0x9000A = display mem; 0x9000B = display mode
63     mov     [12],cx   ! 0x9000C = display adapter properties.
64
65     Get hd0 data
! Get hd0 parameter table info pointed at by BIOS int 0x41. Table for hd1 is immediately
! following the parameter table of hd0. BIOS int 0x46 also point at hd1's table. Table len
! is 16 bytes. These two table are stored at location 0x90080 and 0x90090 seperately.
66
67     mov     ax,#0x0000
68     mov     ds,ax
69     lds     si,[4*0x41]  ! get vector value of int 0x41 --> ds:si
70     mov     ax,#INITSEG
71     mov     es,ax
72     mov     di,#0x0080   ! transfer to address: 0x9000:0x0080 --> es:di
73     mov     cx,#0x10     ! total bytes transferred (16 bytes)
74     rep
75     movsb
76
77     Get hd1 data
78
79     mov     ax,#0x0000
80     mov     ds,ax
81     lds     si,[4*0x46]  ! get vector value of int 0x46 --> ds:si
82     mov     ax,#INITSEG
83     mov     es,ax
84     mov     di,#0x0090   ! transfer to address: 0x9000:0x0090 --> es:di
85     mov     cx,#0x10
86     rep
87     movsb
88
89     Check that there IS a hd1 :-) If not so, then clear the second table.
! Use BIOS int 0x15 function 0x15 (Get disk type) to check.
! Input: dl = drive number (0x8X for harddisk. 0x80 - hd0; 0x81 - hd1)
! Output: ah = type code.
! 0x00 - no such drive; 0x01 - floppy without change-line support;
! 0x02 - floppy (or other removable drive) with change-line support. 0x03 - harddisk.
90
91     mov     ax,#0x01500
92     mov     dl,#0x81
93     int     0x13
94     jc      no_disk1
95     cmp     ah,#3             ! It's a harddisk? (type =3 ?)
There is no hd1, so clear the table for hd1

$ax \leftarrow \#\text{INITSEG}$

$\text{mov es},ax$

$\text{mov di},\#0x0090$

$\text{mov cx},\#0x10$

$\text{mov ax},\#0x00$

$\text{rept}$

$\text{stosb}$

There is no hd1, so clear the table for hd1

! now we want to move to protected mode ...

$\text{cli}$

! no interrupts allowed!

! first we move the system to it's rightful place

The system module has been loaded by bootsect program from disk into memory start at address

$0x10000(64Kb)$. Now we move it down to begin from physical address zero. The range to move

is from $0x10000$ to $0x8ffff$, as Linus assume that the length of system module would not exceed

$0x80000(512Kb)$ at that time.

$\text{mov ax},\#0x0000$

$\text{cld}$

! 'direction'=0, movs moves forward

$\text{do_move:}$

$\text{mov es},ax$

! destination segment! es:di=target addr (init to $0x0000:0x0$)

$\text{add ax},\#0x1000$

$\text{cmp ax},\#0x9000$

! Finished moving 64Kb start from seg $0x8000$?

$\text{jz end_move}$

$\text{mov ds},ax$

! source segment! ds:si=source addr (init to $0x1000:0x0$)

$\text{sub di,di}$

$\text{sub si,si}$

$\text{mov cx},\#0x8000$

! mve $0x8000$ words (64bytes)

$\text{rep}$

$\text{movsw}$

$\text{jmp do_move}$

! then we load the segment descriptors

! From here, we start encounter the operating of protected mode. Before entering protected

! mode, we must first setup two segment descriptor tables: Glose descriptor table and

! Interrupt descriptor table.

! Instruction LIDT is used to load interrupt descriptor table register(IDTR). It's operand is

! 6 bytes long. Bytes 0-1 is the length of descriptor table; Bytes 2-5 is the base address

! in linear address space. Refer to the description in L219-220 and L223-224 below. Each

! item ( 8 bytes) in interrupt descriptor table indicates the code info be called when a

! corresponding interrupt request happened, something like a int vector in real mode but

! contains more information.

! Instruction LGDT is used to load global descriptor table register (GDTR). It's operand

! format is same as LIDT. Each item ( 8 bytes ) in the table describes the information of

! a code or data ( or TSS LDT etc) segment under protected mode, including the segment size

! limit (16 bits), base address (32 bits), priority level, existence, read/write permission

! and some other flags. Ref to L205-216.
end_move:
    mov ax,#SETUPSEG  ! right, forgot this at first. didn't work :-) 
    mov ds,ax         ! ds points to the segment of setup program.
    lidt idt_48       ! load idt with 0,0
    lgdt gdt_48       ! load gdt with whatever appropriate

! that was painless, now we enable A20
! Refer to the description about A20 signal line after this program. The I/O ports and
! commands used here are described in section related to kernel/chr_drv/keyboard.S

    call empty_8042  ! wait for empty of input buffer.
    mov al,#0xD1     ! command write ! code 0xD1 indicates writing to 8042
    out #0x64,al     ! port P2. Bit 1 on P2 is used to control A20.
    call empty_8042  ! wait for buffer empty to see if the previous
                     ! command has been accepted.
    mov al,#0xDF     ! A20 on paras.
    out #0x60,al     ! data needs to be write to port 0x60.
    call empty_8042  ! wait for A20 be enabled.

! 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.

! 0x11 below indicates the start of initialize. It is a ICW1 command word represents
! edge trigger, multi 8259 chips and will send ICW4 at the end of the sequence.
    mov al,#0x11     ! initialization sequence
    out #0x20,al     ! send it to 8259A-1

! The two words defined below are instructions represented in machine code, used for delay.
    .word 0x00eb,0x00eb  ! jmp $+2, jmp $+2 ! '$' represents current addr.
    .word 0x00eb,0x00eb
    out #0x20,al      ! and to 8259A-2
    out #0x21,al      ! start of hardware int’s (0x20)
    .word 0x00eb,0x00eb
    out #0x21,al      ! Send to 8259A-1 chip, odd port. (ICW2)
    .word 0x00eb,0x00eb
    mov al,#0x28      ! start of hardware int’s 2 (0x28)
    out #0xA1,al      ! Send to 8259A-2 chip. (ICW2)
    .word 0x00eb,0x00eb

    mov al,#0x04      ! 8259-1 is master. The IRQ2 of master chip is
    out #0x21,al      ! connected to the int line of of slave chip. (ICW3)
    .word 0x00eb,0x00eb
    mov al,#0x02      ! 8259-2 is slave. The INT line is connected to
    out #0xA1,al      ! the IRQ2 of master chip. (ICW3)
    .word 0x00eb,0x00eb
    mov al,#0x01      ! 8086 mode for both. Send ICW4 command word: 8086
    out #0x21,al      ! mode, normal EOI, need instruction to reset int.
    .word 0x00eb,0x00eb
    out #0xA1,al      ! Send ICW4 to slave chip.
    .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 stinking 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 0x000000, in 32-bit protected mode.

mov ax,#0x0001      ! protected mode (PE) bit !
lmsw ax              ! This is it! !
jmpi 0,8             ! jmp offset 0 of segment 8 (cs)
!! As we have moved the system module begin at the absolute address 0x0000, the jump
!! instruction above will pass the CPU controls to the head.s code at the begin of
!! system module. The segment selector ( 8 ) is used to select a segment descriptor
!! in an identified descriptor table. Segment selector has 16 bits, bit 0-1 represents
!! requested privilege level (RPL), bit 2 is a table indicator used to specify the
!! referred descriptor table ( 0 - GDT, 1 - LDT), bit 3-15 is INDEX field used to index
!! a descriptor in the table. A specified descriptor address is:
!! Table Base Address + INDEX * 8
!! So segment selector 8 (0b0000,0000,0000,1000) means RPL=0, selects item 1 (INDEX=1)
!! within globle descriptor table (GDT). This specified descriptor indicates that the
!! segment base address is 0 (see L209), so the instruction of L193 will jump to the
!! beginning of system module.

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.
!! Only when the input buffer is empty (bit2 = 0 of status reg.), could we write to it.

empty_8042:
.in al,#0x64        ! Read keyboard (8042) status port
.test al,#2           ! is input buffer full?
jnz empty_8042      ! yes - loop
ret

Here begins the global descriptor table. It, now, contains 3 descriptors.
The first one is not used (4 zero words) but must be existed. The second one is kernel
code segment descriptor (L208-211). The third one is kernel data segment descriptor.

! The offset at this location is 0x08 within GDT, and is used when load code segment reg.

.word 0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
.word 0x00000        ! base address=0
.word 0x9A00        ! code read/exec
.word 0x00C0        ! granularity=4096, 386
! The offset at this location is 0x10 within GDT, and is used when load data segment reg.
.word   0x07FF       ! 8Mb - limit=2047 (2048*4096=8Mb)
.word   0x0000       ! base address=0
.word   0x09200      ! data read/write
.word   0x00C0       ! granularity=4096, 386

! Here is the operands for instruction LIDT (Load Interrupt Descriptor Table Register).
.idt_48:
.word   0                ! idt limit=0
.word   0,0              ! idt base=0L

! Here is the operands for instruction LGDT (Load Global Descriptor Table Register).
.gdt_48:
.word   0x800           ! gdt limit=2048, total 256 GDT entries
.word   512+gdt,0x9     ! gdt base = 0x9xxxx
    ! Two words above form a leaner address: 0x0009<<16 + 0x0200+gdt
    ! that is: 0x90200 + gdt( the offset of GDT table in this program, L205)
.text
.endtext:
data
.enddata:
.bss
.endbss:

3.4.3 Information

In order to gathering the basic parameters of the machine, program setup invokes the BIOS interrupt calls many times and start doing some I/O operating to the hardware ports. Below we briefly describe these BIOS interrupts used and give a simple description about the A20 issue. At the end of this section, we mentioned the Intel 32 bits protected mode again.

3.4.3.1 Current memory map

After setup.s finished execution, the system module has been moved to the beginning of the physical address 0x00000 in memory. Some machine parameters are stored starting at address 0x90000. The memory map at this moment is depicted in Figure 3-6.
At this moment, the global descriptor table contains three descriptors. The first one is not used. The other two are kernel code and data segment descriptors separately. The base address in these two descriptors is point to the start of the memory (0x0000). Therefore, when setup.s executes its last instruction: "jmpi 0,8", the CPU control is passed to the head.s code at the beginning of system module. The operand value '8' is a segment selector indicating the segment descriptor used. It indicates the code segment descriptor in global descriptor table in this case. The operand value '0' is the offset in code segment.

### 3.4.3.2 BIOS display interrupt 0x10

In Table 3-2 we list the details for function ah = 0x12 (obtaining display adapter info).

<table>
<thead>
<tr>
<th>In/Out Info</th>
<th>Register</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input info</td>
<td>ah</td>
<td>function number= 0x12, Get adapter information</td>
</tr>
<tr>
<td></td>
<td>bh</td>
<td>sub function number = 0x10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Video state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x00 - color mode in effect (I/O port is 0x3dx);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01 - mono mode in effect (I/O port is 0x3bx);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that the value of x in port address is between 0x0 - 0xf.</td>
</tr>
<tr>
<td>Return info</td>
<td>bh</td>
<td>Installed memory:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K</td>
</tr>
<tr>
<td></td>
<td>bl</td>
<td>Feature connector bits:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit(s) Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0  Feat 1 line, state 2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  Feat 0 line, stat 2;</td>
</tr>
</tbody>
</table>
3.4.3.3 Hard disk parameter table ("INT 0x41")

The vector corresponding to BIOS interrupt 0x41 is the address of first hard disk in the machine. This address may be overridden by the hard disk controller’s BIOS to support drive formats unknown to the ROM BIOS. BIOSes which support four hard drives may store the parameter tables for drives 0x81—0x83 immediately following the parameter table pointed at by int 0x41. The second drive (0x81) hard disk parameter table is also pointed at by int 0x46. The format of fixed hard disk parameters is shown in Table 3-3.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>word</td>
<td>cyl</td>
<td>number of cylinders.</td>
</tr>
<tr>
<td>0x02</td>
<td>byte</td>
<td>head</td>
<td>number of heads.</td>
</tr>
<tr>
<td>0x03</td>
<td>word</td>
<td>head</td>
<td>start reducing write current cylinder (XT only, 0 for others).</td>
</tr>
<tr>
<td>0x05</td>
<td>word</td>
<td>wpcom</td>
<td>start write precompensation cylinder number (multiply 4)</td>
</tr>
<tr>
<td>0x07</td>
<td>byte</td>
<td>ctl</td>
<td>maximum ECC burst length (XT only).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>control byte (drive steps selection)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 0: unused;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 1: reserved (0) (disable IRQ);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 2: reserved (0) (no reset)</td>
</tr>
<tr>
<td>0x08</td>
<td>byte</td>
<td>ctl</td>
<td>bit 3: set if more than 8 heads;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: always 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: set if manufacturer’s defect map on max cylinder+1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 6: disable ECC retries;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 7: disable access retries.</td>
</tr>
</tbody>
</table>
### 3.4.3.4 A20 Address Line

In August 1981, IBM delivered the first IBM PC. It contains a Intel 8088 processor and has only 20 address lines (A0 - A19). Therefore, its addressing range is in 0x000000-0x10ffef (0xffff:0xffff) and was suitable for memory size only within several hundred kerabytes at that time. When program access the address greater than 0x100000, for example 0x10ffef, it would wrap to 0x00ffef silently and some programs did use this property for running. When IBM introduced the PC/AT in 1985, it uses a Intel 80286 CPU which has 24 address lines and can address 16Mb physical memory. In order to be 100% compatible with the original PC in real mode, IBM invented a switch to enable/disable the 0x100000 address bit. Since the keyboard controller chip 8042 happened to have a spare pin (output port P2, bit 1, pin P21), that was used to control the AND gate that disables this address bit. The signal is called A20, and if it is zero, bit 20 of all addresses is cleared.

Since the default status of A20 address line is disabled at boot time, the operating system has to do something to enable it. The enabling procedures are variant depending on the chipset used for implementing this property by compatible PC manufacturer. The classical method used by most software is via the keyboard controller as used in setup.s program (L138-144).

Some operating systems use the switching off and on of A20 as part of the standard procedure to switch between real address mode and protected mode. Since the keyboard controller is relatively very slow, a Fast Gate A20 Option was introduced, where I/O port 0x92 is used to handle A20. Another way used by some systems is to read I/O port 0xee to enable A20, and write to it to disable A20.

### 3.4.3.5 8259A Interrupt Controller Chip

Intel 8259A is a programmable interrupt control chip. Each chip can handle 8 interrupt request sources. On PC/AT and compatibles, two 8259A chips are used to handle 16 possible interrupt sources IRQ0 -- IRQ15. One 8259A is called the “Master” with I/O address ranging from 0x20 to 0x3f, the other is called the “Slave” with I/O address ranges of 0xa0—0xbf. The Master’s INT is connected to CPU, the Slave’s INT pin is connected to the Master’s IRQ2 as shown in Figure 3-7.
Under the controlling bus controller, 8259A can be at programming or operating status. Before normal operation can begin, each 8259A must be initialized to be ready to accept interrupt requests from peripheral devices (IRQ0 -- IRQ15) using Initialization Command Words (ICWs). During the operation, CPU can command the 8259A to operate in various modes through the Operation Command Words (OCWs). From the selection of priority, 8259A will select the interrupt request that has highest priority to serve and passed the request to CPU through the INT pin. When CPU responding to the request, a interrupt number is sent to CPU from the data bus D7--D0. CPU, thus, obtains the current interrupt vector corresponding to the interrupt number and executing the related interrupt routine.

Once the CPU has an identifying vector, it uses this number to select one of 256 programmer-defined interrupt handling routines. So we should write code to deal with each of these cases. To choose the correct handler in protected mode, CPU will consults the IDT – the Interrupt Descriptor Table. The IDT contains exactly 256 interrupt descriptors. After the CPU acquires a interrupt vector, it uses that number as a direct index into the IDT. If a hardware device passes a vector of 32, the 32th descriptor is used.

In the Linux kernel code, the interrupt number corresponding to the peripheral device interrupt request signal is programmed to start from int 32 to int 47 (0x20 -- 0x2f). Interrupt number ranging from int 0 to int 31 is reserved to be used by CPU internally.

### 3.4.3.6 Intel CPU 32 bits protected mode

Intel 32-bit CPU may runs under two different mode: the real-address mode and protected mode. The protected mode is the natural 32-bit environment of the Intel 80X86 and support multitasking operating system, virtual memory management. see the reference materials for detailed information.
3.5 head.s

3.5.1 Function

The head.s is the first part of system module and this is the reason it gets its name. From here, the kernel is totally running under the protected mode. The assembly language syntax of head.s uses AT&T's format and compiled by GNU assembler. This is different from the Intel’s format.

The most significant difference is that, while Intel’s syntax uses ‘op dest, src’, AT&T syntax uses ‘op src, dest’. The easy way to distinguish these two kind of assembly syntax is to look for the '%' symbols. AT&T names the registers like %eax, %ebx, etc. The other major difference is that the operand size in AT&T syntax is clear from the instruction. For a 'inc' instruction in Intel format, you have incb, incw and incl to use 8, 16 or 32 bits operands. If the size is clear from the operands, you can just write 'inc', but if it’s memory operand, then you should indicates the operand size by instruction, for example, 'incl foo'.

Immediate values are written with a leading symbol '$'. Thus, 'movl foo, %eax' copies the contents of memory location foo into %eax. 'movl $foo, %eax' copies the value of foo. For example, 'movl 55, %eax' is to fetch from an absolute address, while 'movl $55, %eax' is to load an immediate value to %eax.

Addressing modes are written 'offset(base, index, scale)'. You may leave out anything irrelevant. So '(%eax)' is legal, as is '-44(%ebx, %eax)', which is equivalent to '-44(%ebx, %eax, 1)'. Legal scales are 1, 2, 4 and 8.

Actually, head.s is located at beginning of physical memory and its function is simple. It first loads each data segment registers, reconfigures the interrupt descriptor table (IDT) with 256 descriptor items. Each item point to a same dummy interrupt routine by default. Then, reinstalls the global descriptor table (GDT), checks whether the A20 address line is been enabled by circlely comparing the containts of the location begin from physical address 0x000000 and 0x100000. If all values are same, it means the A20 line is not enabled and the system enter dead cycling.

Afterwards, the code checks to see if there is a math coprocessor in the system and set relating flag bits in the control register CR0. Then, it sets up one page directory table and 4 page tables used for memory management system of CPU begin from physical address 0x000000, this will overlay the head.s program after it finishes its function. The page directory table (PDT) occupies 1 memory page (4096 bytes) and the 4 page tables occupy 16Kb memory used for addressing 16Mb physical memory by the kernel code. At last, it sets up /init/main.c’s entry point in the stack and passed the control to it using a ret instruction.

3.5.2 Code Comments

Program 3-3 linux/boot/head.s

1 /*
2 *  linux/boot/head.s
* (C) 1991 Linus Torvalds
*/

/*
* head.s contains the 32-bit startup code.
*
* NOTE!!! Startup happens at absolute address 0x00000000, which is also where
* the page directory will exist. The startup code will be overwritten by
* the page directory.
*/

.text
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area

.pg_dir:                  # page directory will begin from here
    # L18-22 are used to setup each data segment register.
    # Notice that here is in protected mode, so the immediate value operand '$0x10' of the following
    # instruction is a segment descriptor's selector. It indexed into GDT table's item 2, which
    # is a data segment descriptor of the kernel.
    # The following instructions are used to setup segment register to point at the kernel data
    # segment and setup the kernel stack area within the user_stack array which is pointed to by
    # stack_start at the end of the array. Then, reload IDTR and GDTR with new table pointers defined
    # at the end of this program. The contents of the two new descriptor tables are almost the
    # same except the limit size changed from 8Mb to 16Mb. So, Afterwards, we need reload those
    # data segment registers again.
    # stack_start is a long pointer defined at L69 in kernel/sched.c.
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
    lss _stack_start,%esp   # _stack_start --> ss:esp, setup kernel stack.
call setup_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

    # L32-36 are used to check if A20 line is enabled. The adopted method is to write a arbitrary
    # value begin at the location 0x000000++ and compared it with the value at location 0x100000.
    # If they are the same then continue the comparisons until the find a difference or system
    # crashes. If A20 is not enabled, then we can not use the memory above address 0x100000.
    xorl %eax,%eax
    incl %eax               # check that A20 really IS enabled
    movl %eax,0x00000000    # loop forever if it isn't
    cmpl %eax,0x100000
    je 1b
    jmp 5f

    /*
    * 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.
*
# The following lines is used to check the existence of math coprocessor. The method used is
# to modify control register CR0 first, set MP flag (math present, bit 1) on assumption. Then
# executes a math instruction. If an error occurred, means no math presented. So we have to
# reset MP flag and setup EM (Emulate Math, bit 2) flag in the CR0.
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
orl $2,%eax             # set EM
movl %eax,%cr0
jmp after_page_tables   # Jump to L135

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

# The following is a assembler directive used to pad the assembler's location counter to a
# particular memory boundary. The number is the low-order zero bits the location counter
# must have after advancement for a.out format. Here, '.align 2' means to advances the location
# counter until it a multiple of 4.
.align 2
1:      .byte 0xDB,0xE4         /* fsetpm for 287, ignored by 387 */  # 287协处理器码。
ret

* setup_idt
* setup a idt with 256 entries pointing to
* ignore_int, interrupt gates. It then loads
* idt. Everything that wants to install itself
* in the idt-table may do so themselves. Interrupts
* are enabled elsewhere, when we can be relatively
* sure everything is ok. This routine will be over-
* written by the page tables.
*
# Although the descriptor in the IDT is 8 bytes same as in GDT, its format is different,
# and be called gate descriptor. Bytes 0–1, 6–7 are offset in the segment, bytes 2–3 are
# segment selector, bytes 4–5 are some of flags.
setup_idt:
lea ignore_int,%edx      # Get effective address of ignore_int to edx.
movl $0x00080000,%eax   # Set selector 0x0008 into the high 16-bit of eax.
movw %dx,%ax            /* selector = 0x0008 = cs */
3.5 head.s

# Set the low 16-bit of offset into eax. Now, eax contains # the low 4 bytes of a gate descriptor.
mov $0x8E00,%dx /* interrupt gate - dpl=0, present */
lea _idt,%edi # Now, edx contains high 4 bytes of a gate descriptor.
mov $256,%ecx

rp_sidt:
movl %eax,(%edi) # Store the dummy gate descriptor into the talbe.
movl %edx,4(%edi) # edi points to next item in the table.
addl $8,%edi
dec %ecx
jne rp_sidt

lidt idt_descr # Load the IDT talbe register.
ret

setup_gdt:
lgdt gdt_descr # See L234-238
ret

*/

* This routines sets up a new gdt and loads it.
* Only two entries are currently built, the same
* ones that were built in init.s. The routine
* is VERY complicated at two whole lines, so this
* rather long comment is certainly needed :-).
* This routine will be overwritten by the page tables.
*/

setup_gdt:
lidt gdt_descr # See L234-238
ret

*/

* 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.
*/

# A page table is an array of 32-bit page specifiers. A page table is itself a page, and # therefore contains 4 Kb memory or at most 1K entries. Two levels of tables are used to # address a page of memory. At the higher level is a page directory. The page directory # addresses up to 1K page tables of the second level. A page table of the second level # addresses up to 1K pages. All the tables addressed by on page directory table, therefore, # can address 1M pages or 4G memory for entire physical address space of the 80386. # The physical address of the page directory table is stored in the register CR3, also # called page directory base register (PDBR).
# Entries in either level of page tables have the same format. Bits 31-12 is the page frame # address specifies the physical starting address of a page. In a page directory, the page # frame address is the address of a page table. Bits 11-0 indicates the property of a page. # Bit 0, for example, is the present flag indicates whether a page table entry can be used # in address translation. Bits 2-1 are used for page-level protection (Read/Write,Supervisor).

.org 0x1000 # From the offset of the program, we reserve 4Kb space for page table 0
pg0:

.org 0x2000
pg1:
.org 0x3000

pg2:

.org 0x4000

pg3:

.org 0x5000    # Start offset address for floppy transfer area.

/*
* tmp_floppy_area is used by the floppy-driver when DMA cannot
* reach to a buffer-block. It needs to be aligned, so that it isn’t
* on a 64kB border.
*/

tmp_floppy_area:

.fill 1024,1,0          # Reserve 1024 items, 1 byte per item, filled with 0.

# Several pushes bellow are used to prepare an environment for passing controls to the
# kernel/main.c program. The first three pushes are used to push values cooresponding to
# arguments envp, argv and argc for main(). $L6 is the return address for main.c, $_main
# is the address of main. When execute instruction ’ret’ at L218, the address of main
# is popped and the control will be transfer to main.c program.

after_page_tables:

pushl $0                # These are the parameters to main :-) 136
pushl $0
pushl $0
pushl $L6               # return address for main, if it decides to.
pushl $_main
jmp setup_paging        # Jump to L198.

L6:

jmp L6                  # main should never return here, but
# just in case, we know what happens.

/* This is the default interrupt “handler” :-) */

text_msg:

.asciz “Unknown interrupt\n\r”

.align 2

ignore_int:

pushl %eax
pushl %ecx
pushl %edx
push %ds          # Notice that although the ds, es, etc are 16-bit register,
# they still occupy 4 bytes each in stack space.

push %es
push %fs
movl $0x10,%eax    # Set to point to kernel data segment.

mov %ax,%ds
mov %ax,%es
mov %ax,%fs
pushl $text_msg    # Push the argument for calling printk() function.
call printk      # printk() is in /kernel/printk.c

popl %eax
pop %fs
pop %es
pop %ds
3.5 head.s

167    popl %edx
168    popl %ecx
169    popl %eax
170    iret
171
172 /*
173 * Setup_paging
174 *
175 * This routine sets up paging by setting the page bit
176 * in cr0. The page tables are set up, identity-mapping
177 * the first 16MB. The pager assumes that no illegal
178 * addresses are produced (ie >4Mb on a 4Mb machine).
179 *
180 * NOTE! Although all physical memory should be identity
181 * mapped by this routine, only the kernel page functions
182 * use the >1Mb addresses directly. All "normal" functions
183 * use just the lower 1Mb, or the local data space, which
184 * will be mapped to some other place - mm keeps track of
185 * that.
186 *
187 *
188 * For those with more memory than 16Mb - tough luck. I've
189 * not got it, why should you :-) The source is here. Change
190 * it. (Seriously - it shouldn't be too difficult. Mostly
191 * change some constants etc. I left it at 16Mb, as my machine
192 * even cannot be extended past that (ok, but it was cheap :-)
193 * I've tried to show which constants to change by having
194 * some kind of marker at them (search for "16Mb"), but I
195 * won't guarantee that's all :-(
196 */
# One page directory table and 4 second level page table are stored begin from physical
# memory 0x000. The page directory table are shared by all tasks (processes) in the
# system, but the 4 page tables are used by kernel code only. Page tables used by new
# process is to be created within the main memory area (MMA) of the system.
197 .align 2
198 setup_paging:                           # First clear the area for 5 pages
199    movl $1024*5,%ecx               /* 5 pages - pg_dir+4 page tables */
200    xorl %eax,%eax
201    xorl %edi,%edi                  /* pg_dir is at 0x000 */
202    cld;rep;stosl
203    # L203-206 are used to setup items in the page directory table for the kernel code. As the
204    # kernel has only 4 page tables, we need only setup 4 items in page directory table now.
205    # '$pg0+7' represents 0x00001007, is the first entry in pg_dir table. The address of it, thus,
206    # is 0x00001007 & 0xfffff000 = 0x1000. The property of it is 0x00001007 & 0x00000fff = 0x7,
207    # means the page is exist, user can read/write.
208    movl $pg0+7,.pg_dir             /* set present bit/user r/w */
209    movl $pg1+7,.pg_dir+4          /* * --------- * */
210    movl $pg2+7,.pg_dir+8          /* * --------- * */
211    movl $pg3+7,.pg_dir+12         /* * --------- * */
212    # L207-212 are used to fill up all entries in 4 kernel page tables.
213    # Total entries are 4 * 1024 = 4096 ( 0x0 - 0xffff ). They can map 4096 * 4Kb = 16 Mb physical
214    # memory. The contents of each entry is: The physical address mapped + flags of this page (0x7).
215    # The processing method is backward fill in the entry begin from the last entry in the last
# page table. The location of last entry in a table is at offset 1023 * 4 = 4092 in the table.
# So, the last entry location in the last page table is $pg3 + 4092 and its corresponding
# physical address is 0xff000, plus property flags we get 0xff007.

movl $pg3+4092,%edi          # edi points to last entry in last page table.
movl $0xfff007,%eax          /* 16Mb - 4096 + 7 (r/w user,p) */
std                              # Set direction flag. edi decreases 4 bytes each.
subl $0x1000,%eax              # Decrease address 0x1000 after fill one entry
jge 1b                           # If less than 0, then all finished.

# Set the page directory base address register CR3.
xorl %eax,%eax                  /* pg_dir is at 0x0000 */
movl %eax,%cr3                  /* cr3 - page directory start */

# Set PG flag to use paging (set bit 31 of CR0).
movl %cr0,%eax
orl $0x80000000,%eax             # add PG flag.
movl %eax,%cr0                  /* set paging (PG) bit */
ret                             /* this also flushes prefetch-queue */

# At this point, the CPU passed the control to init/main.c program. This program is end.

.idt_descr:                      # 6 bytes operand for lidt instruction (limit, base)
   .word 256*8-1           # idt contains 256 entries
   .long _idt
.align 2
.word 0

.gdt_descr:                      # 6 bytes operand for lgdt instruction (limit, base)
   .word 256*8-1           # so does gdt (note that that's any
   .long _gdt              # magic number, but it works for me :^)
.align 2
.word 0

.idt:   .fill 256,8,0           # idt is uninitialized. 256 items, 8 bytes per, filled 0

# The following is the contents of global descriptor table (GDT). The first descriptor (L234)
# is a NULL descriptor (reserved and not used). The second (L235) is for kernel code segment.
# The third (L236) is for kernel data segment. The fourth (L237) is for system segment but
# not used in this kernel. The remaining 252 descriptors is used for each process or task.
# (0-nul, 1-cs, 2-ds, 3-sys, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc ...)

.gdt:   .quad 0x0000000000000000        /* NULL descriptor */
   .quad 0x00c09a0000000fff        /* code seg, 0x08, limit 16Mb */
   .quad 0x00c0920000000fff        /* data seg, 0x10, limit 16Mb */
   .quad 0x0000000000000000        /* TEMPORARY - don’t use */
   .fill 252,8,0                   /* space for LDT’s and TSS’s etc */

3.5.3 Information

3.5.3.1 Memory map after end of head.s

After the end of execution of head.s, the page directory table and second level page tables
is setted up. The IDT and GDT table are replaced with new one which is slightly modified. In
addition, a buffer for floppy is also installed. At this moment, the memory map of system module
is shown in Figure 3-8.
3.5 head.s

3.5.3.2 Intel 32-bit protected mode

The key to the understanding of head.s program is the familiar with protected mode running mechanism which is also the important part for the remaining programs. In order to be compatible with 8086 CPU, the protected mode of 80x86 has been handled complicatedly. When runs in protected mode, CPU uses the value in a segment register as an selector to a descriptor table. The base address of the descriptor is kept in its base register like, for example, GDTR for GDT and IDTR for IDT.

For the different usage aspect, the descriptor tables have three types: global, local, and interrupt. There must be exactly one GDT and one IDT while the CPU is running in protected mode. The system programmer can, if wanted, have others defined in memory, on reserve, but only one of each can be considered active at any moment — the ones pointed to by the GDTR and IDTR registers. Although we can also define maximum 8192 local descriptor tables, but only one is considered active at a time.

Figure 3-9 demonstrate the descriptor table used in Linux kernel. Each task occupies two descriptors in GDT. LDT0 is the segment descriptor occupied by local descriptor table (LDT) for the first task. TSS0 is the segment descriptor occupied by the Task Status Segment for the first task. Each LDT contains three descriptors used to describe the code and data segment of the task. As with the GDT, the first descriptor is not used. When the DS register contains the selector of first task, DS:ESI will point to the location in the data segment of the task.
3.6 Summary

The bootsect program is mainly used to load the setup code and system module from disk into memory. The head program is located at the very beginning of system module. When moves itself and setup code to the upper location ox 0x90000 and 0x90200, bootsect transfer the executing control to setup code.

The setup program code is used to get some parameters of the machine by using ROM BIOS interrupt and store them at the location begin from 0x90000 in memory, overwritten the bootsect code. Then, it moves the system module to the physical memory begin from the absolute 0x0000. Afterwards, it loads the base address of GDT into GDTR, reconfigured the interrupt chips and, at last, enabling the running in protected mode and transfer the controls to the head code located at the start of system module.

The head code in the system module is used to initialize the interrupt descriptor items in IDT, check the enableing of A20 line, test if there is a math copprocessor in machine and then setup the page directory table and second level page tables for the kernel. At the end, it jumps to the main code in the system module to continue the kernel initializations.