Chapter 3 Booting System

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 santaxes. 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.

List 3-1 linux/boot/ directory

Filename	Size	Last Modified Time(GMT)	Description
bootsect.s	5052 bytes	1991-12-05 22:47:58	
head.s	5938 bytes	1991-11-18 15:05:09	
setup.s	5364 bytes	1991-12-05 22:48:10	

In order to read these code, you should have some knowledge of 8086 assembly language and know the architecture of 80x86 and howto programing 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, virtical frames represents 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.

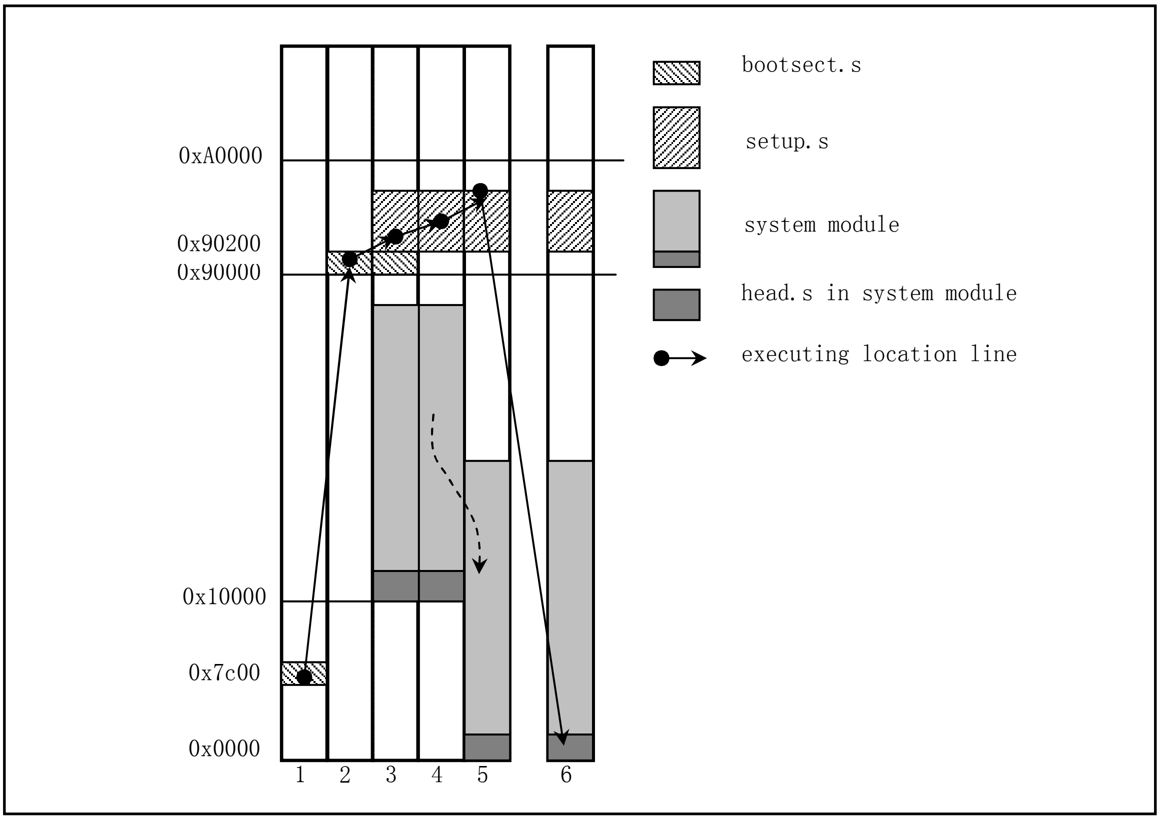


Figure 3-1 The location of kernel in memory when booting

The booting parts identify some properties and the type of VGA adapter in machine. If need, it will request user to select display mode for the console and then move the whole system from address 0x10000 to 0x0000. Entering protect mode and jump to the rest of the system at 0x0000. The begin part of system module contains the head.s code. At this moment, all the running methods have been setup: IDT and GDT were loaded. The processor and coprocessor was identified. Paging method is also settled. The kernel calls the init/main.c program at last.

Someone may ask, why not load the system module to absolute physical address zero directory? This is because the setup program will use the BIOS interrupts for obtaining machine parameters and the BIOS interrupt's vectors are just stored at the start area of the physical memory by BIOS. So we have to move down the system module after we finished using BIOS interrupts to overlay this area of the memory.

3.3 bootsect.s

3.3.1 Function description

bootsect.s is disk booting code residented 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 loaction 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

```
!
! 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.

.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                            ! Uninitialzed data Segment (Block Started by Symbol).
begbss:
.text                           ! Code Segment begins.

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

```
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 progrom.
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                     ! Repeat until cx = 0
55         movw                    ! Move one word
56         jmpi    go,INITSEG      ! Indirect jumping. INITSEG is destination seg addr.
                                   ! From here on, CPU start executes code at 0x9000 seg.
57 go:     mov     ax,cs           ! Set ds, es, ss to new seg value (0x9000). As there are
58         mov     ds,ax           ! stack operating (push,pop,call), stack must be setup.
59         mov     es,ax
60 ! put 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
```

```
    ! 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.
        mov     dx,#0x0000              ! drive 0, head 0
        mov     cx,#0x0002              ! sector 2, track 0
        mov     bx,#0x0200              ! address = 512, in INITSEG
        mov     ax,#0x0200+SETUPLEN     ! service 2, nr of sectors
        int     0x13                    ! read it
        jnc     ok_load_setup           ! ok - continue
        mov     dx,#0x0000
        mov     ax,#0x0000              ! reset the diskette
        int     0x13
        j       load_setup

ok_load_setup:

! Get disk drive parameters, specifically nr of sectors/track
! The following is the format of INT 0x13 for obtaining drive parameters:
! ah = 0x08     dl = drive no.( bit 7 must be set if devie is hd)
! Return info:
! If error encountered, set CF flag and ah = status code.
! ah = 0,  al = 0,         bl = drive type (AT/PS2)
! ch = low 8 bits of max track nr.
! cl = max sector nr/track(bit0-5), high 2bits of max track nr.(bit6-7)
! dh = max head nr.         dl = nr of drives
! es:di --> disk parameter table.

        mov     dl,#0x00
        mov     ax,#0x0800              ! AH=8 is get drive parameters
        int     0x13
        mov     ch,#0x00
        seg cs                  ! The operand of next instruction is in cs seg.
        mov     sectors,cx      ! Save sector nr per track
        mov     ax,#INITSEG
        mov     es,ax           ! The es was changed when call int 0x13. here just re-set it

! Print some inane message    ! Display message('Loading system ...'cr,lf. Total 24 chars)

        mov     ah,#0x03                ! read cursor pos
        xor     bh,bh
        int     0x10

        mov     cx,#24                  ! 24 chars totally.
        mov     bx,#0x0007              ! page 0, attribute 7 (normal)
        mov     bp,#msg1                ! point to string.
        mov     ax,#0x1301              ! write string, move cursor
        int     0x10                    !

! ok, we've written the message, now
```

```
105 ! we want to load the system (at 0x10000)
106
107         mov     ax,#SYSSEG
108         mov     es,ax           ! segment of 0x010000   ! es = seg of system.
109         call    read_it         ! Read system module from disk. es is input parameter.
110         call    kill_motor      ! shut down drive motor, then we know the stats of drive.
111
112 ! After that we check which root-device to use. If the device is
113 ! defined (!= 0), nothing is done and the given device is used.
114 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
115 ! 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 separatly. 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.
116
117         seg cs
118         mov     ax,root_dev
119         cmp     ax,#0
120         jne     root_defined
121         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.
122         mov     bx,sectors
123         mov     ax,#0x0208         ! /dev/ps0 - 1.2Mb
124         cmp     bx,#15
125         je      root_defined
126         mov     ax,#0x021c         ! /dev/PS0 - 1.44Mb
127         cmp     bx,#18
128         je      root_defined
129 undef_root:                        ! If we cann't identify the drive type, we are dead!
130         jmp undef_root
131 root_defined:
132         seg cs
133         mov     root_dev,ax        ! Save the checked root device.
134
135 ! after that (everyting loaded), we jump to
136 ! the setup-routine loaded directly after
137 ! the bootblock:
138
139         jmpi    0,SETUPSEG        ! Jump to 0x9020:0000 (setup.s)
                                      !!!! This ends of the program !!!!
    ! Here are two subroutines
140
141 ! This routine loads the system at address 0x10000, making sure
142 ! no 64kB boundaries are crossed. We try to load it as fast as
143 ! possible, loading whole tracks whenever we can.
144 !
145 ! 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
        test ax,#0x0fff
die:    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:
```

```
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
    ! Read data in current track to the buffer pointed at by es:bx.
    ! al - sector numbers; es:bx - buffer location
198 read_track:
199         push ax
200         push bx
201         push cx
202         push dx
203         mov dx,track          ! Get current track.
204         mov cx,sread          ! Get sector number already read in current track.
205         inc cx                ! cl = start sector.
206         mov ch,dl             ! ch = current track.
207         mov dx,head           !
208         mov dh,dl             ! dh = current head.
209         mov dl,#0             ! dl = drive number (0 indicates first drive).
210         and dx,#0x0100        ! Disk head number will no more than 1.
211         mov ah,#2             ! ah = 2, read disk sector.
212         int 0x13
213         jc bad_rt             ! If error, jump to bad_rt.
214         pop dx
215         pop cx
216         pop bx
217         pop ax
218         ret
    ! Execute drive reset operation (function 0 of int 0x13), then jump to read_track to try again.
219 bad_rt: mov ax,#0
220         mov dx,#0
221         int 0x13
222         pop dx
223         pop cx
224         pop bx
225         pop ax
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.
```

```
        pop dx
        ret

sectors:
        .word 0                     ! Place used to store sector number per track (SPT).

msg1:
        .byte 13,10                 ! cr, lf
        .ascii "Loading system ..."
        .byte 13,10,13,10           ! total 24 charactors.

.org 508                            ! Indicates the next instruction begins at addr 508.
root_dev:
        .word ROOT_DEV              ! Device nr of root filesystem (used in init/main.c).
boot_flag:
        .word 0xAA55                ! Identifer of a valid hd.

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

3.3.3 Infomation

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

3.4.1 Function

The main function of setup progam 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 3-1 Parameters obtained by setupprogram

Address	Size(bytes)	Name	Description
0x90000	2	Cursor Location	Column (0x00-left most), Row (0x00 - top most)
0x90002	2	Extended Memory	The size of extended memory begin from address 1Mb (in Kb).
0x90004	2	Display Page	Current display page.

0x90006	1	Display Mode	
0x90007	1	Char Columns	
0x90008	2	??	
0x9000A	1	Display Memory	Display memory (0x00-64k, 0x01-128k, 0x02-192k, 0x03=256k)
0x9000B	1	Display Status	0x00-Color, I/O=0x3dX; 0x11-Mono, I/O=0x3bX
0x9000C	2	Property. Paras	Property parameters of display adapter.
...			
0x90080	16	Hd Paras Table	Harddisk parameter table for first hd.
0x90090	16	Hd Paras Table	Hardisk parameter table for second hd(zero if none)
0x901FC	2	Root Device	Root filesystem device number.

Then, this program moves the system module from address 0x10000-0x8ffff (Linus thought the kernel size will never execcds 512Kb at that time) down to the absolute physical address 0x00000. Next, the interrupt descripter table register (IDTR) and globle descripter table register (GDTR) are loaded, A20 address line is opened, the two programmable interrupt controller (PIC) chips, Intel 8259A, is reconfigured. The hardware interrupt identifer number is allocated within 0x20 - 0x2f. The last step is 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 descripter table (IDT) and globle descripter table (GDT) at the end of the program. In the GDT, kernel code segment descripter and data segment descripter are configured. These two kernel code descripters will be re-configured in head.s program.

As the protected running mode is vital for understanding the head.s program, now, lets 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 rigister: 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 descripter in a segment descripter table (e.g. GDT). The indexed segment descripter provides the processor with the data it needs to map a logical address into a linear address. The base field in a descripter 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 descripter, which defines the size of the segment. The addressing location in memory, thus, indicated by the base address of a descripter 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 descripter that describe the property of the segment.

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

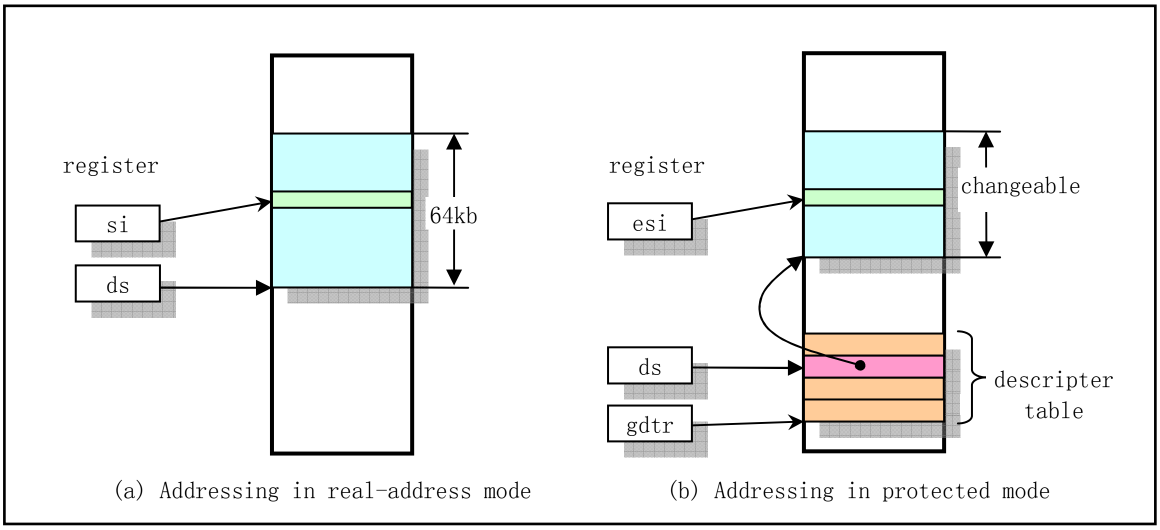


Figure 3-2 Addressing methods comparing between real-address mode and protected mode.

Thereby, before entering protected mode, we must first setup the descripter talbes 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

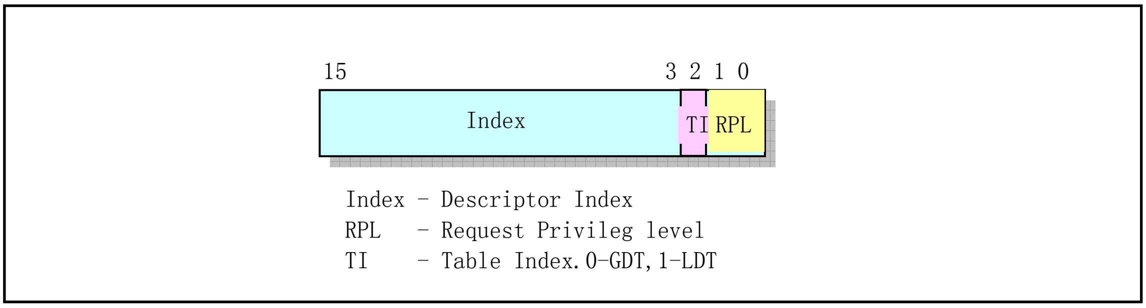
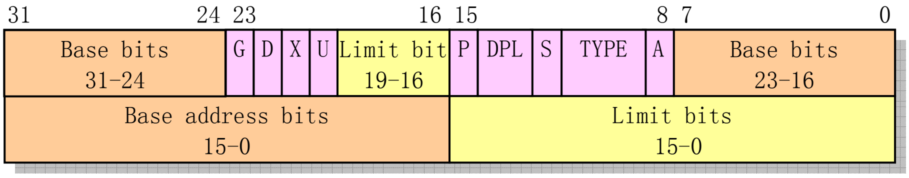


Figure 3-3 The format of segment selector




```
A    - Accessed.
S    - System descriptor if set, else code, data or stack descriptor.
DPL  - Descriptor Privilege Level.
P    - Present.
U    - User can use this bit.
X    - Intel Reserved bit.
D    - Default size. If set, operans are 32 bits, else 16 bits.
G    - Granularity. If set, limit is measured by 4Kb, else 1 byte.
TYPE - Segment type.
       000  Data, read only;     100  Code, execute only;
       001  Data, read/write;    101  Code, execute/read;
       010  Stack, read only;    110  Code, execute only, confirming;
       011  Stack, read/write;   111  Code, execute/read, confirming.
```

Figure 3-4 Segment descriptor format

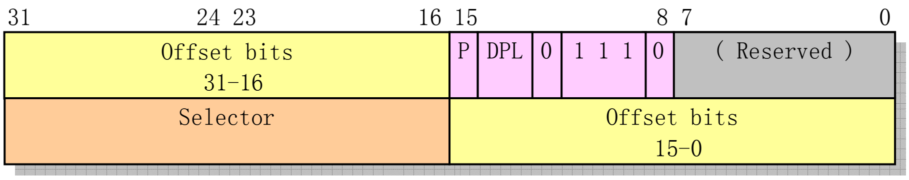


Figure 3-5 The format of interrupt gate descriptor

3.4.2 Code Comments

Program 3-2 linux/boot/setup.s

```
!
!       setup.s         (C) 1991 Linus Torvalds
!
! setup.s is responsible for getting the system data from the BIOS,
! and putting them into the appropriate places in system memory.
! both setup.s and system has been loaded by the bootblock.
!
! This code asks the bios for memory/disk/other parameters, and
! puts them in a "safe" place: 0x90000-0x901FF, ie where the
! boot-block used to be. It is then up to the protected mode
! system to read them from there before the area is overwritten
! for buffer-blocks.
!

! NOTE! These had better be the same as in bootsect.s!

INITSEG  = 0x9000        ! we move boot here - out of the way
```

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

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

entry start
start:

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

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

! Get memory size (extended mem, kB)
! Use BIOS int 0x15 function 0x88 to get extended memory size and save the word to 0x90002.
! Return: CF clear if successful
! AX = number of contiguous KB starting at absolute address 0x100000(1Mb)

        mov     ah,#0x88
        int     0x15
        mov     [2],ax

! Get video-card data:
! Get current video mode by using BIOS int 0x10 function 0x0f and store it.
! Return: Ah = number of char columns; al = display mode; bh = active page
! Active page number is stored at 0x90004 (one word), display mode is store at 0x90006,
! char columns is stored at 0x90007.

        mov     ah,#0x0f
        int     0x10
        mov     [4],bx          ! bh = display page
        mov     [6],ax          ! al = video mode, ah = window width

! check for EGA/VGA and some config parameters
! Use BIOS int 0x10 alternate function: ah = 0x12, bl = 0x10 to get EGA/VGA info.
! Return:
! 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 )

        mov     ah,#0x12
        mov     bl,#0x10
        int     0x10
        mov     [8],ax      ! 0x90008 = ??
        mov     [10],bx     ! 0x9000A = display mem; 0x9000B = display mode
        mov     [12],cx     ! 0x9000C = display adapter properties.

! 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.

        mov     ax,#0x0000
        mov     ds,ax
        lds     si,[4*0x41]   ! get vector value of int 0x41 --> ds:si
        mov     ax,#INITSEG
        mov     es,ax
        mov     di,#0x0080    ! transfer to address: 0x9000:0x0080 --> es:di
        mov     cx,#0x10      ! total bytes transferred (16 bytes)
        rep
        movsb

! Get hd1 data

        mov     ax,#0x0000
        mov     ds,ax
        lds     si,[4*0x46]   ! get vector value of int 0x46 --> ds:si
        mov     ax,#INITSEG
        mov     es,ax
        mov     di,#0x0090    ! transfer to address: 0x9000:0x0090 --> es:di
        mov     cx,#0x10
        rep
        movsb

! Check that there IS a hd1 :-) If not so, then clear the second table.
! Use BIOS int 0x13 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 - hardisk.

        mov     ax,#0x01500
        mov     dl,#0x81
        int     0x13
        jc      no_disk1
        cmp     ah,#3              ! It's a harddisk? (type =3 ?)
```

```
96          je      is_disk1
97  no_disk1:
98          mov     ax,#INITSEG        ! There is no hd1, so clear the table for hd1
99          mov     es,ax
100         mov     di,#0x0090
101         mov     cx,#0x10
102         mov     ax,#0x00
103         rep
104         stosb
105 is_disk1:
106
107 ! now we want to move to protected mode ...
108
109         cli                        ! no interrupts allowed !
110
111 ! 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.
112
113         mov     ax,#0x0000
114         cld                        ! 'direction'=0, movs moves forward
115 do_move:
116         mov     es,ax              ! destination segment ! es:di=target addr (init to 0x0000:0x0)
117         add     ax,#0x1000
118         cmp     ax,#0x9000         ! Finished moving 64Kb start from seg 0x8000?
119         jz      end_move
120         mov     ds,ax              ! source segment  ! ds:si=source addr (init to 0x1000:0x0)
121         sub     di,di
122         sub     si,si
123         mov     cx,#0x8000         ! mve 0x8000 words (64kbytes)
124         rep
125         movsw
126         jmp     do_move
127
128 ! then we load the segment descriptors
    ! From here, we start enconter the operating of protected mode. Before entering protected
    ! mode, we must first setup two segment descripter tables: Globle descripter table and
    ! Interrupt descripter table.
    ! Instruction LIDT is used to load interrupt descripter table register(IDTR). It's operand is
    ! 6 bytes long. Bytes 0-1 is the length of descripter 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 descripter table indicats 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 descripter 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.
    !
129
```

```
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.
                                        ! we can write port only when the input buffer is empty.
        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 privious
                                        ! 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.
        out     #0xA0,al                ! and to 8259A-2
        .word   0x00eb,0x00eb
        mov     al,#0x20                ! start of hardware int's (0x20)
        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
```

```
176         mov     al,#0xFF                ! mask off all interrupts for now
177         out     #0x21,al
178         .word   0x00eb,0x00eb
179         out     #0xA1,al
180
181 ! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
182 ! need no steenking BIOS anyway (except for the initial loading :-).
183 ! The BIOS-routine wants lots of unnecessary data, and it's less
184 ! "interesting" anyway. This is how REAL programmers do it.
185 !
186 ! Well, now's the time to actually move into protected mode. To make
187 ! things as simple as possible, we do no register set-up or anything,
188 ! we let the gnu-compiled 32-bit programs do that. We just jump to
189 ! absolute address 0x00000, in 32-bit protected mode.
190
191         mov     ax,#0x0001      ! protected mode (PE) bit !
192         lmsw    ax              ! This is it! !
193         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 descripter
    !! in an identified descripter table. Segment selector has 16 bits, bit 0-1 represents
    !! requested privilege level (RPL), bit 2 is a table indicator used to specify the
    !! refered descriptor table ( 0 - GDT, 1- LDT), bit 3-15 is INDEX field used to index
    !! a descripter in the talbe. A sepecified descripter address is:
    !!     Table Base Address + INDEX * 8
    !! So segment selector 8 (0b0000,0000,0000,1000) means RPL=0, selects item 1 (INDEX=1)
    !1 within globle descripter table (GDT). This specified descripter indicates that the
    !! segment base address is 0 (see L209), so the instruction of L193 will jump to the
    !! beginning of system module.
194
195 ! This routine checks that the keyboard command queue is empty
196 ! No timeout is used - if this hangs there is something wrong with
197 ! 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.
198 empty_8042:
199         .word   0x00eb,0x00eb   ! Delay a little while.
200         in      al,#0x64        ! Read keyboard (8042) status port
201         test    al,#2           ! is input buffer full?
202         jnz     empty_8042      ! yes - loop
203         ret
204
    ! Here begins the global descripter table. It, now, contains 3 descripters.
    ! The first one is not used (4 zero words) but must be existed. The second one is kernel
    ! code segment descripter (L208-211). The third one is kernel data segment descripter.
205 gdt:
206         .word   0,0,0,0         ! dummy
207
    ! The offset at this location is 0x08 within GDT, and is used when load code segment reg.
208         .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
209         .word   0x0000          ! base address=0
210         .word   0x9A00          ! code read/exec
211         .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	0x9200		! 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 Infomation

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 0x0000 in memory. Some machine parameters are stored starting at address 0x90000. The memory map at this moment is depicted in Figure 3-6.

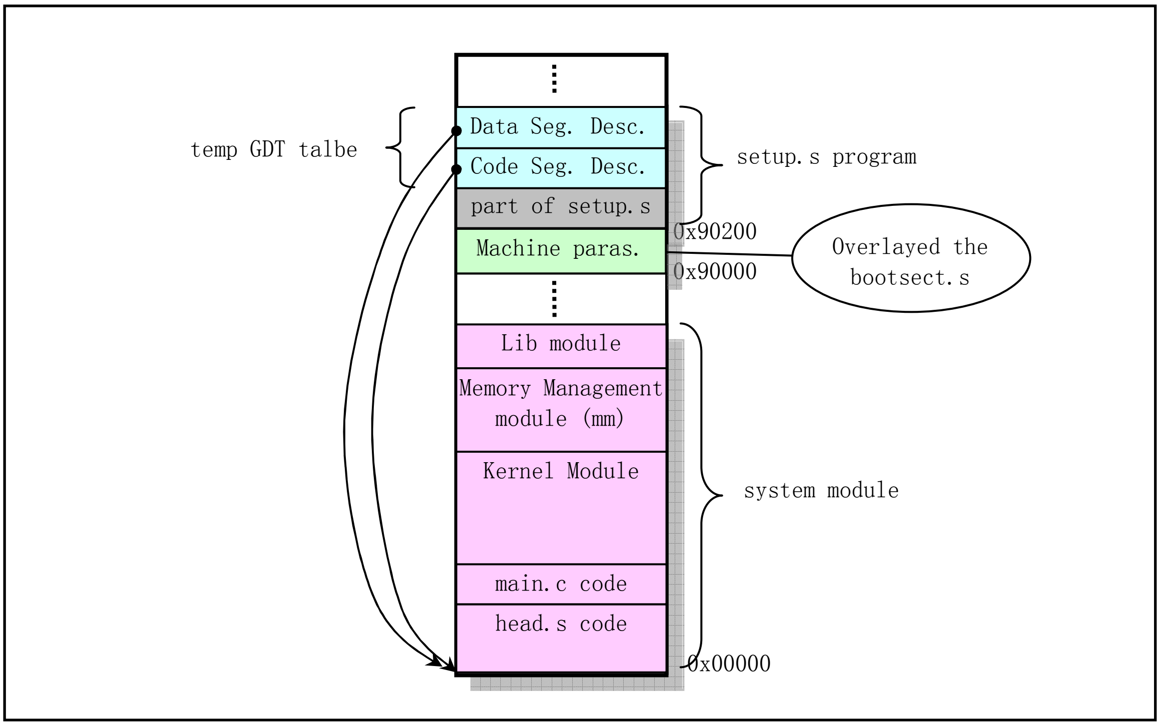


Figure 3-6 The memory map after end of setup.s execution

At this moment, the golbal descriptor table contains three descriptors. The first one is not used. The other two are kernel code and data segment descripters 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 3-2 Get vedio adapter infoamtion(function: ah = 0x12，bh = 0x10）

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

		2 Feat 1 line, stat 1; 3 Feat 0 line, stat 1; 4-7 unused(0). Video switch settings: bit(s) Description 0 switch 1 off; 1 switch 2 off; 2 switch 3 off; 3 switch 4 off; 4-7 unused. Values for switch settings on original EGA/VGA: 00h MDA/HGC; 01h-03h MDA/HGC; 04h CGA 40x25; 05h CGA 80x25; 06h EGA+ 40x25; 07h-09h EGA+ 80x25; 0Ah EGA+ 80x25 mono; 0Bh EGA+ 80x25 mono;
	cl	

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 had 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 3-3 Format of hard disk parameters

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

0x09	byte		standard timeout (XT only, 0 for others).
0x0A	byte		formatting timeout (XT and WD1002 only, 0 for others).
0x0B	byte		timeout for checking drive (XT and WD1002 only, 0 for others).
0x0C	word	lzone	cylinder number of landing zone.
0x0E	byte	sect	number of sectors per track.
0x0F	byte		reserved.

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 kerobytes 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 contaol 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 enableing procedures are varient depending on the chipset used for implementing this property by compatible PC manufactory. 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 posible 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.

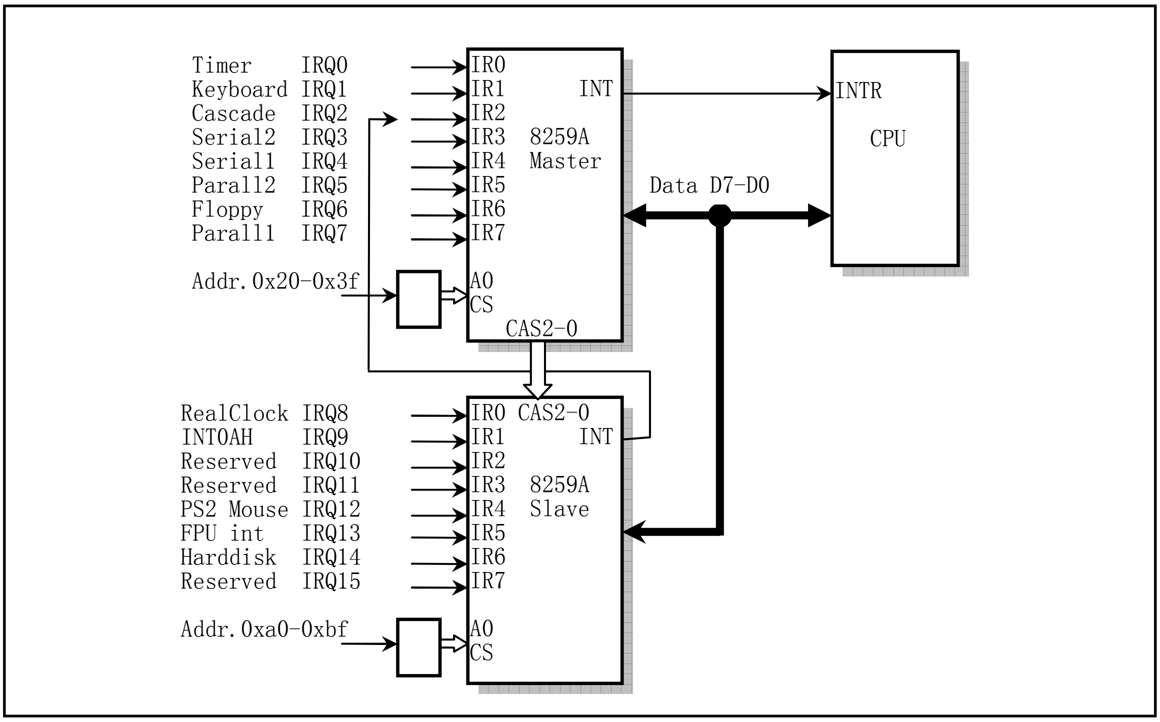


Figure 3-7 PC/AT nested connection 8259A controller system

Under the controling 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 operatin 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 gots 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 0x0000, 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

```
/*
 *  linux/boot/head.s
```

```
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  *  head.s contains the 32-bit startup code.
9  *
10  * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11  * the page directory will exist. The startup code will be overwritten by
12  * the page directory.
13  */
14 .text
15 .globl _idt,_gdt,_pg_dir,_tmp_floppy_area
16 _pg_dir:                  # page directory will begin from here
17 startup_32:
   # 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.
18         movl $0x10,%eax
19         mov %ax,%ds
20         mov %ax,%es
21         mov %ax,%fs
22         mov %ax,%gs
23         lss _stack_start,%esp   # _stack_start --> ss:esp, setup kernel stack.
24         call setup_idt
25         call setup_gdt
26         movl $0x10,%eax         # reload all the segment registers
27         mov %ax,%ds             # after changing gdt. CS was already
28         mov %ax,%es             # reloaded in 'setup_gdt'
29         mov %ax,%fs
30         mov %ax,%gs
31         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.
32         xorl %eax,%eax
33 1:      incl %eax               # check that A20 really IS enabled
34         movl %eax,0x000000      # loop forever if it isn't
35         cmpl %eax,0x100000
36         je 1b                   # '1b' means jump backward to the label 1 (L33)
                                   # If it's a '5f', then it means jump forward to label 5.
37 /*
38  * NOTE! 486 should set bit 16, to check for write-protect in supervisor
39  * mode. Then it would be unnecessary with the "verify_area()"-calls.
```

```
40  * 486 users probably want to set the NE (#5) bit also, so as to use
41  * int 16 for math errors.
42  */
   # 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.
43         movl %cr0,%eax          # check math chip
44         andl $0x80000011,%eax   # Save PG,PE,ET
45 /* "orl $0x10020,%eax" here for 486 might be good */
46         orl $2,%eax             # set MP
47         movl %eax,%cr0
48         call check_x87
49         jmp after_page_tables   # Jump to L135
50
51 /*
52  * We depend on ET to be correct. This checks for 287/387.
53  */
54 check_x87:
55         fninit
56         fstsw %ax
57         cmpb $0,%al
58         je 1f                   /* no coprocessor: have to set bits */
59         movl %cr0,%eax
60         xorl $6,%eax            /* reset MP, set EM */
61         movl %eax,%cr0
62         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.
63 .align 2
64 1:      .byte 0xDB,0xE4         /* fsetpm for 287, ignored by 387 */  # 287 协处理器码。
65         ret
66
67 /*
68  *  setup_idt
69  *
70  *  sets up a idt with 256 entries pointing to
71  *  ignore_int, interrupt gates. It then loads
72  *  idt. Everything that wants to install itself
73  *  in the idt-table may do so themselves. Interrupts
74  *  are enabled elsewhere, when we can be relatively
75  *  sure everything is ok. This routine will be over-
76  *  written by the page tables.
77  */
   # 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.
78 setup_idt:
79         lea ignore_int,%edx      # Get effective address of ignore_int to edx.
80         movl $0x00080000,%eax    # Set selector 0x0008 into the high 16-bit of eax.
81         movw %dx,%ax            /* selector = 0x0008 = cs */
```

```
                                        # Set the low 16-bit of offset into eax. Now, eax contains
                                        # the low 4 bytes of a gate descriptor.
        movw $0x8E00,%dx               /* interrupt gate - dpl=0, present */
                                        # Now, edx contains high 4 bytes of a gate descriptor.
        lea _idt,%edi                   # _idt is the address of IDT talbe.
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)                # Store the dummy gate descriptor into the talbe.
        movl %edx,4(%edi)
        addl $8,%edi                    # edi points to next item in the table.
        dec %ecx
        jne rp_sidt
        lidt idt_descr                  # Load the IDT talbe register.
        ret

/*
 *  setup_gdt
 *
 *  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 beoverwritten by the page tables.
 */
setup_gdt:
        lgdt 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
# thereofre 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 talbes 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:

```

```
120 .org 0x3000
121 pg2:
122
123 .org 0x4000
124 pg3:
125
126 .org 0x5000        # Start offset address for floppy transfer erea.
127 /*
128  * tmp_floppy_area is used by the floppy-driver when DMA cannot
129  * reach to a buffer-block. It needs to be aligned, so that it isn't
130  * on a 64kB border.
131  */
132 _tmp_floppy_area:
133         .fill 1024,1,0          # Reserve 1024 items, 1 byte per item, filled with 0.
134
    # 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 coorespending 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.
135 after_page_tables:
136         pushl $0                # These are the parameters to main :-)
137         pushl $0
138         pushl $0
139         pushl $L6               # return address for main, if it decides to.
140         pushl $_main
141         jmp setup_paging        # Jump to L198.
142 L6:
143         jmp L6                  # main should never return here, but
144                                 # just in case, we know what happens.
145
146 /* This is the default interrupt "handler" :-) */
147 int_msg:
148         .asciz "Unknown interrupt\n\r"
149 .align 2
150 ignore_int:
151         pushl %eax
152         pushl %ecx
153         pushl %edx
154         push %ds          # Notice that although the ds, es, etc are 16-bit register,
                          # they still occupy 4 bytes each in stack space.
155         push %es
156         push %fs
157         movl $0x10,%eax   # Set to point to kernel data segment.
158         mov %ax,%ds
159         mov %ax,%es
160         mov %ax,%fs
161         pushl $int_msg    #  Push the argument for calling printk() function.
162         call _printk      # printk() is in /kernel/printk.c
163         popl %eax
164         pop %fs
165         pop %es
166         pop %ds
```

```
167         popl %edx
168         popl %ecx
169         popl %eax
170         iret
171
172
173 /*
174  * Setup_paging
175  *
176  * This routine sets up paging by setting the page bit
177  * in cr0. The page tables are set up, identity-mapping
178  * the first 16MB. The pager assumes that no illegal
179  * addresses are produced (ie >4Mb on a 4Mb machine).
180  *
181  * NOTE! Although all physical memory should be identity
182  * mapped by this routine, only the kernel page functions
183  * use the >1Mb addresses directly. All "normal" functions
184  * use just the lower 1Mb, or the local data space, which
185  * will be mapped to some other place - mm keeps track of
186  * that.
187  *
188  * For those with more memory than 16 Mb - 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 $1024*5,%ecx               /* 5 pages - pg_dir+4 page tables */
200         xorl %eax,%eax
201         xorl %edi,%edi                  /* pg_dir is at 0x000 */
202         cld;rep;stosl
    # L203-206 are used to setup items in the page directory table for the kernel code. As the
    # kernel has only 4 page tables, we need only setup 4 items in page directory table now.
    # '$pg0+7' represents 0x00001007, is the first entry in pg_dir table. The address of it, thus,
    # is 0x00001007 & 0xfffff000 = 0x1000. The property of it is 0x00001007 & 0x00000fff = 0x7,
    # means the page is exist, user can read/write.
203         movl $pg0+7,_pg_dir             /* set present bit/user r/w */
204         movl $pg1+7,_pg_dir+4           /*  --------- " " --------- */
205         movl $pg2+7,_pg_dir+8           /*  --------- " " --------- */
206         movl $pg3+7,_pg_dir+12          /*  --------- " " --------- */
    # L207-212 are used to fill up all entries in 4 kernel page tables.
    # Total entries are 4 * 1024 = 4096 ( 0x0 - 0xfff ). They can map 4096 * 4Kb = 16 Mb physical
    # memory. The contents of each entry is: The physical address mapped + flags of this page (0x7).
    # 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 0xfff000, plus property flags we get 0xfff007.
207         movl $pg3+4092,%edi              # edi points to last entry in last page table.
208         movl $0xfff007,%eax             /* 16Mb - 4096 + 7 (r/w user,p) */
209         std                              # Set direction flag. edi decreases 4 bytes each.
210 1:      stosl                           /* fill pages backwards - more efficient :-) */
211         subl $0x1000,%eax                # Decrease address 0x1000 after fill one entry
212         jge 1b                           # If less than 0, then all finished.
    # Set the page directory base address register CR3.
213         xorl %eax,%eax                  /* pg_dir is at 0x0000 */
214         movl %eax,%cr3                  /* cr3 - page directory start */
    # Set PG flag to use paging (set bit 31 of CR0).
215         movl %cr0,%eax
216         orl $0x80000000,%eax             # add PG flag.
217         movl %eax,%cr0                  /* set paging (PG) bit */
218         ret                             /* this also flushes prefetch-queue */
    # At this point, the CPU passed the control to init/main.c program. This program is end.
219
220 .align 2
221 .word 0
222 idt_descr:                      # 6 bytes operand for lidt instruction (limit, base)
223         .word 256*8-1           # idt contains 256 entries
224         .long _idt
225 .align 2
226 .word 0
227 gdt_descr:                      # 6 bytes operand for lgdt instruction (limit, base)
228         .word 256*8-1           # so does gdt (note that that's any
229         .long _gdt              # magic number, but it works for me :^)
230
231         .align 3
232 _idt:   .fill 256,8,0           # idt is uninitialized. 256 items, 8 bytes per, filled 0
233
    # 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 ...)
234 _gdt:   .quad 0x0000000000000000        /* NULL descriptor */
235         .quad 0x00c09a0000000fff        /* 16Mb */       # code seg, 0x08, limit 16Mb
236         .quad 0x00c0920000000fff        /* 16Mb */       # data seg, 0x10, limit 16Mb
237         .quad 0x0000000000000000        /* TEMPORARY - don't use */
238         .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.

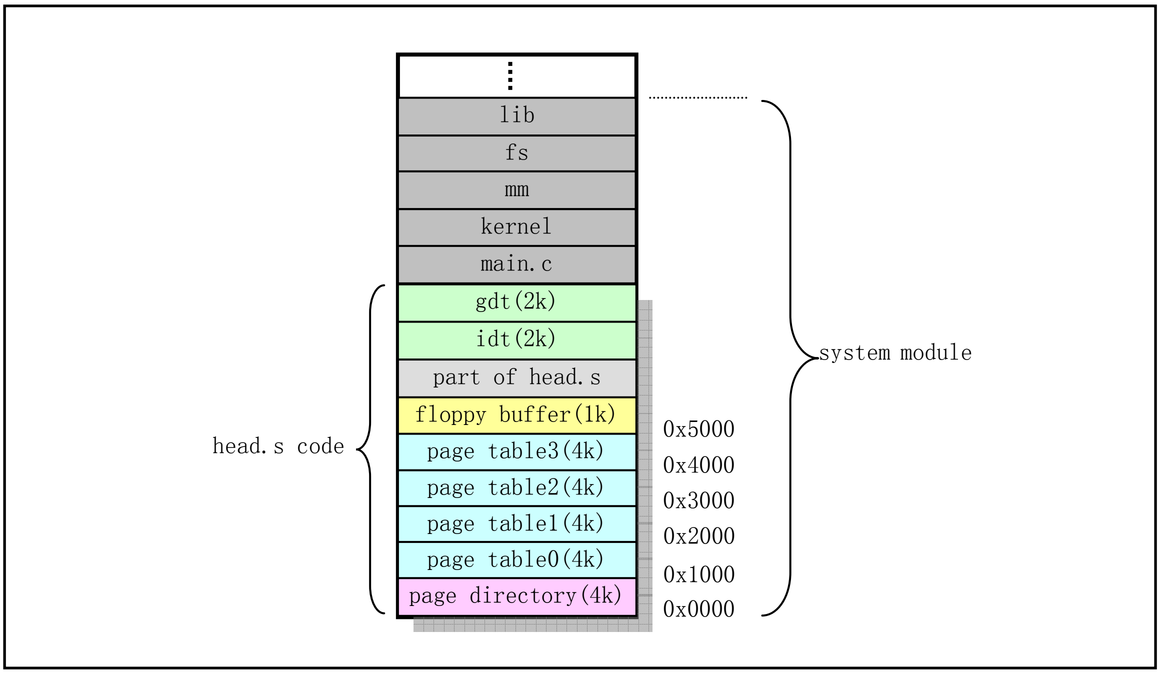


Figure 3-8 Memory map of system module after the end of head program

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 comapatible 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 mement -- the ones pointed to by the GDTR and IDTR registers. Although we can also define maxmum 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.

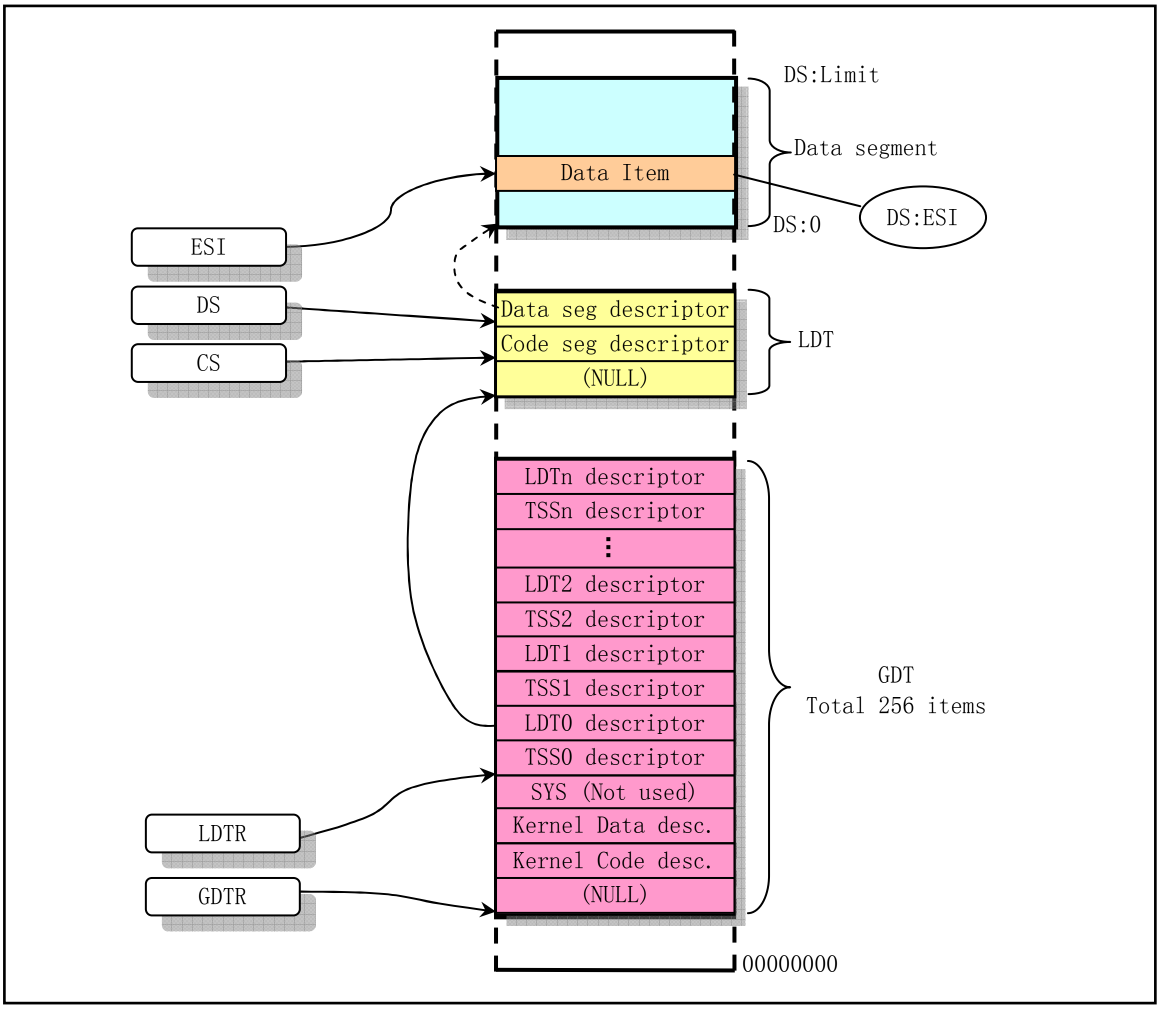


Figure 3-9 Descriptor tables used in Linux kernel

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 ath the very beginning of system module. When moves itself and setup code to the upper location ox 0x90000 and 0x90200, bootsect tranfer 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.